Progress DataDirect for JDBC for Oracle Eloqua

User's Guide

6.0.0 Release

Copyright

Table of Contents

1

Welcome to the Progress DataDirect for JDBC for Oracle Eloqua Driver

The Progress® DataDirect® *for* JDBC™ for Oracle Eloqua™ driver supports SQL read-write access to Oracle Eloqua. To support SQL access to Oracle Eloqua, the driver creates a relational map of the Oracle Eloqua data model and translates SQL statements provided by the application to Oracle Eloqua queries and web service calls. The driver optimizes performance when executing joins by leveraging data relationships among Oracle Eloqua objects to minimize the amount of data that needs to be fetched over the network. The driver recognizes the relationships among standard objects and custom objects and can access and update each. Relationships among objects can be reported with the metadata methods `getBestRowIdentifier()`, `getColumns()`, `getExportedKeys()`, `getImportedKeys()`, `getPrimaryKey()`, `getTables()`, and `getTypeInfo()`.

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-oracle-eloqua.

For details, see the following topics:

What's New in this Release?

Support and certification

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

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

Changes Since 6.0.0 GA

- **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 40 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 67 for details.

Highlights of the 6.0.0 release

- Supports SQL read-write access to Oracle Eloqua. See Supported SQL functionality on page 73 for details.
- The driver supports JDBC core functions. For details, refer to "JDBC support" in the *Progress DataDirect for JDBC Drivers Reference*.
- Supports Oracle Eloqua data types. See Data Types on page 14 and getTypeInfo() on page 15 for details.
- Supports HTTP proxy. See Proxy server support on page 39 for details.
- Uses a combination of REST and BULK APIs for optimized fetch operations. See Bulk operations on page 40 for details.
- Supports retrieving LONGVARCHAR data, using JDBC methods designed for Character Large Object (CLOB) data. See Large object (LOB) support on page 46 for details.
- Supports timeout functionality. See Timeouts on page 46 for details.

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.

Driver and DataSource classes

Driver class

The `Driver` class is:

`com.ddtek.jdbc.eloqua.EloquaDriver`

DataSource class

The `DataSource` class is:

`com.ddtek.jdbcx.eloqua.EloquaDataSource`

Connection URL

The connection URL takes the following form:

```
jdbc:datadirect:eloqua:Company=company_id;
   User=user_id;Password=password[;property=value[;...]]
```

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}}`.

---

where:

company_id

specifies the company identifier issued by Oracle Eloqua during the registration process.

user_id

specifies the user name that used to connect to the Oracle Eloqua instance.

password

specifies the password for the specified `User`.

property=value

specifies additional connection property settings.

The following is an example of a connection URL to an Oracle Eloqua instance.

```
jdbc:datadirect:eloqua:Company=ABCcorp;User=123@abccorp.com;Password=secret
```

Version string information

The `DatabaseMetaData.getDriverVersion()` method returns a driver version string in the format:

M.m.s.bbbbbb(C*XXXX*.F*YYYYYY*.U*ZZZZZZ*)

where:

M is the major version number.

m is the minor version number.

s is the service pack number.

bbbbbb is the driver build number.

XXXX is the cloud adapter build number.

YYYYYY is the framework build number.

ZZZZZZ is the utl build number.

For example:

```
6.0.0.000002(C0003.F000001.U000002)
      |____| |___| |_____| |_____|
      Driver Cloud Frame    Utl
```

Connection properties and configuration options

The driver includes over 20 connection properties and configuration options. You can use these properties and options to customize the driver for your environment. These properties and options can be used to accomplish different tasks, such as controlling driver functionality and optimizing performance.

See Using connection properties on page 35 and Connection property descriptions on page 51 for more information.

SQL escape sequences

The driver supports the following SQL escape sequences.

- Date, Time, and Timestamp Escape Sequences
- Scalar Functions

- Outer Join Escape Sequences
- LIKE Escape Character Sequence for Wildcards

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

Supported scalar functions

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

`{fn scalar-function}`

where:

scalar-function

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

Example:

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

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

Table 1: Scalar functions

String Functions	Numeric Functions	Timedate Functions	System Functions
ASCII	ABS	CURDATE	CURSESSIONID
BIT_LENGTH	ACOS	CURTIME	DATABASE
CHAR	ASIN	DATEDIFF	IDENTITY
CHAR_LENGTH	ATAN	DAY	USER
CHARACTER_LENGTH	ATAN2	DAYNAME	IFNULL
CONCAT	CEILING	DAYOFMONTH	
DIFFERENCE	BITAND	DAYOFWEEK	
HEXTORAW	BITOR	DAYOFYEAR	
INSERT	BITXOR	EXTRACT	
LCASE	COS	HOUR	
LEFT	COT	MINUTE	
LENGTH	DEGREES	MONTH	
LOCATE	EXP	MONTHNAME	
LOCATE_2	FLOOR	NOW	

String Functions	Numeric Functions	Timedate Functions	System Functions
LOWER	LOG	SECOND	
LTRIM	LOG10	TO_CHAR	
OCTET_LENGTH	MOD	WEEK	
RAWTOHEX	PI	YEAR	
REPEAT	POWER		
REPLACE	RADIANS		
RIGHT	RAND		
RTRIM	ROUND		
SOUNDEX	SIGN		
SPACE	SIN		
SUBSTR	SQRT		
SUBSTRING	TAN		
UCASE	TRUNCATE		
UPPER	ROUNDMAGIC		

Data Types

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

See getTypeInfo() on page 15 for `getTypeInfo()` results of data types supported by the driver.

Table 2: Oracle Eloqua data types

Oracle Eloqua data type	JDBC data type
ARRAY	VARCHAR
BOOLEAN	BOOLEAN
DATETIME	TIMESTAMP
DECIMAL	DECIMAL
DURATION	VARCHAR

Oracle Eloqua data type	JDBC data type
INTEGER	INTEGER
LARGETEXT	LONGVARCHAR
LONG	BIGINT
TEXT	VARCHAR
URL	VARCHAR

getTypeInfo()

The `DatabaseMetaData.getTypeInfo()` method returns information about data types. The following table provides `getTypeInfo()` results for supported Oracle Eloqua data types.

Table 3: getTypeInfo()

TYPE_NAME = array AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = ARRAY MAXIMUM_SCALE = 0	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 4000 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = boolean AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 16 (BOOLEAN) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = BOOLEAN MAXIMUM_SCALE = 0	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = datetime AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = TIMESTAMP ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = DATETIME MAXIMUM_SCALE = 3	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 23 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = decimal AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = DECIMAL MAXIMUM_SCALE = 4	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 = duration AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = DURATION MAXIMUM_SCALE = 0	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 50 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

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 = largetext AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = LARGETEXT MAXIMUM_SCALE = 0	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32000 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = long AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -5 (BIGINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = LONG MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 19 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = text AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = TEXT MAXIMUM_SCALE = 0	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 4000 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = url AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = URL MAXIMUM_SCALE = 0	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 4000 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

Mapping objects to tables

The driver automatically maps Oracle Eloqua objects and fields to tables and columns when it first connects to an Oracle Eloqua instance. The driver maps standard as well as custom objects, including relationships defined among objects. Relationships among objects can be reported with the metadata methods `getBestRowIdentifier()`, `getColumns()`, `getExportedKeys()`, `getImportedKeys()`, `getPrimaryKey()`, `getTables()`, and `getTypeInfo()`.

The driver uses a local schema map to instantiate the mapping of the remote data source objects to tables and the metadata associated with those tables. The driver creates a schema map for each user. By default, the schema map is created in the directory from which the application is run. The CreateMap connection property allows you to update or re-create the schema map. The schema map uses the user ID specified for the connection as the name of the schema map configuration file. If the user ID contains punctuation or other non-alphanumeric characters, the driver strips those characters from the user ID to form the name of the schema map configuration file. You can set the SchemaMap connection property to override the default setting for the name and location of the database.

Note that the Refresh Map SQL extension can also be used to update the relational map of your data.

See also

CreateMap on page 60
SchemaMap on page 65
Refresh Map (EXT) on page 77
Mapping properties on page 36

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

Getting started

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

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

For details, see the following topics:

Driver and DataSource classes

Driver class

The `Driver` class is:

`com.ddtek.jdbc.eloqua.EloquaDriver`

DataSource class

The `DataSource` class is:

```
com.ddtek.jdbcx.eloqua.EloquaDataSource
```

Connecting using the DriverManager

You can connect to an Oracle Eloqua instance using the JDBC `DriverManager` with the `DriverManager.getConnection()` method. As the following example shows, this method specifies a string containing a connection URL.

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:eloqua:Company=ABCcorp;User=123@abccorp.com;Password=secret")
```

Setting the class path

The driver must be defined on your class path before you can connect. The class path 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 class path, you will receive a `class not found` exception when trying to load the driver. Set your system class path to include the `eloqua.jar` file as shown, where *`install_dir`* is the path to your product installation directory.

```
install_dir/lib/60/eloqua.jar
```

Windows Example

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

UNIX Example

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

Passing the connection URL

After setting the class path, the required connection information needs to be passed in the form of a connection URL. The connection URL takes the form:

```
jdbc:datadirect:eloqua:Company=company_id;
   User=user_id;Password=password[;property=value[;...]]
```

where:

`company_id`

specifies the company identifier issued by Oracle Eloqua during the registration process.

`user_id`

specifies the user name that used to connect to the Oracle Eloqua instance.

`password`

specifies the password for the specified `User`.

property=value

specifies additional connection property settings.

The following is an example of a connection URL to an Oracle Eloqua instance.

```
jdbc:datadirect:eloqua:Company=ABCcorp;User=123@abccorp.com;Password=secret
```

Testing the connection

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

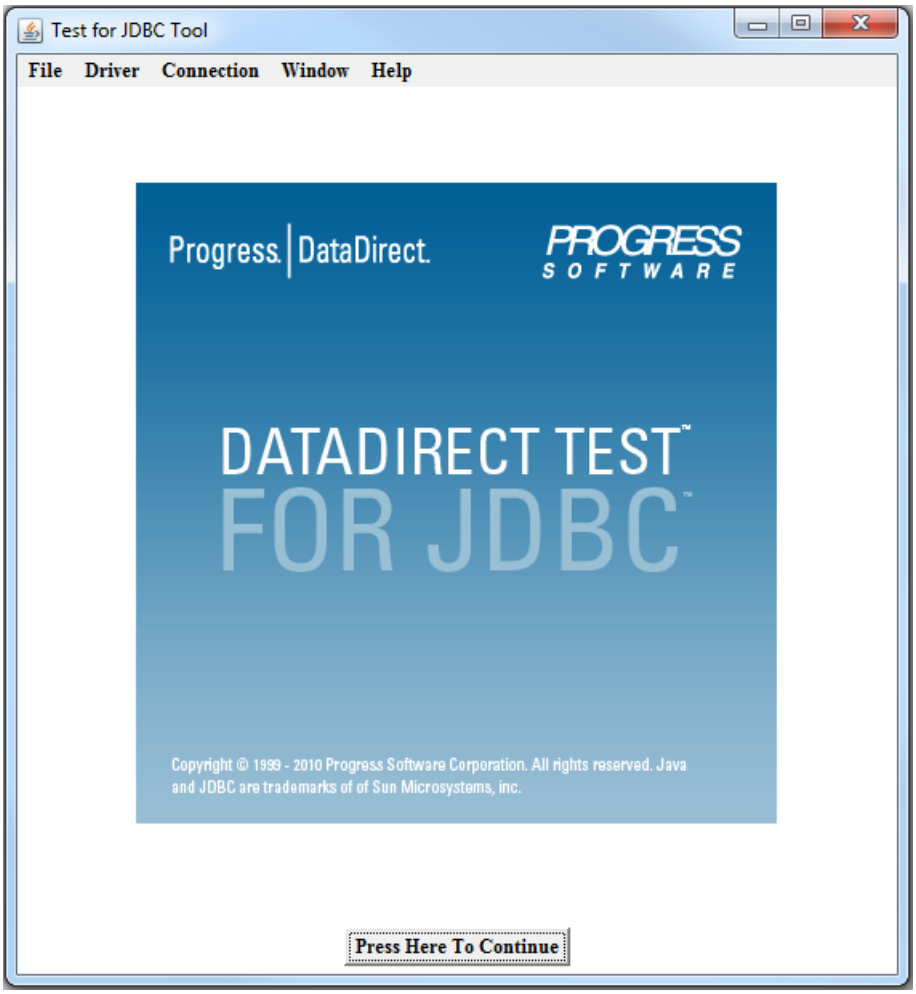
Take the following steps to establish a connection.

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.

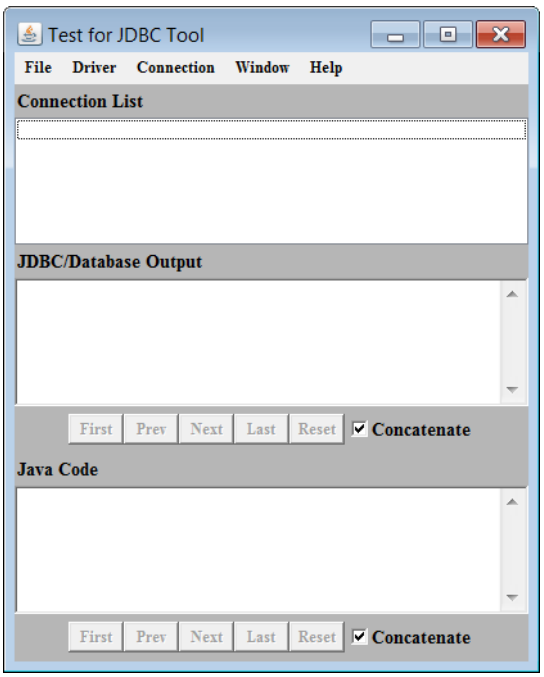
2. From the `testforjdbc` folder, run the platform-specific tool:
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

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



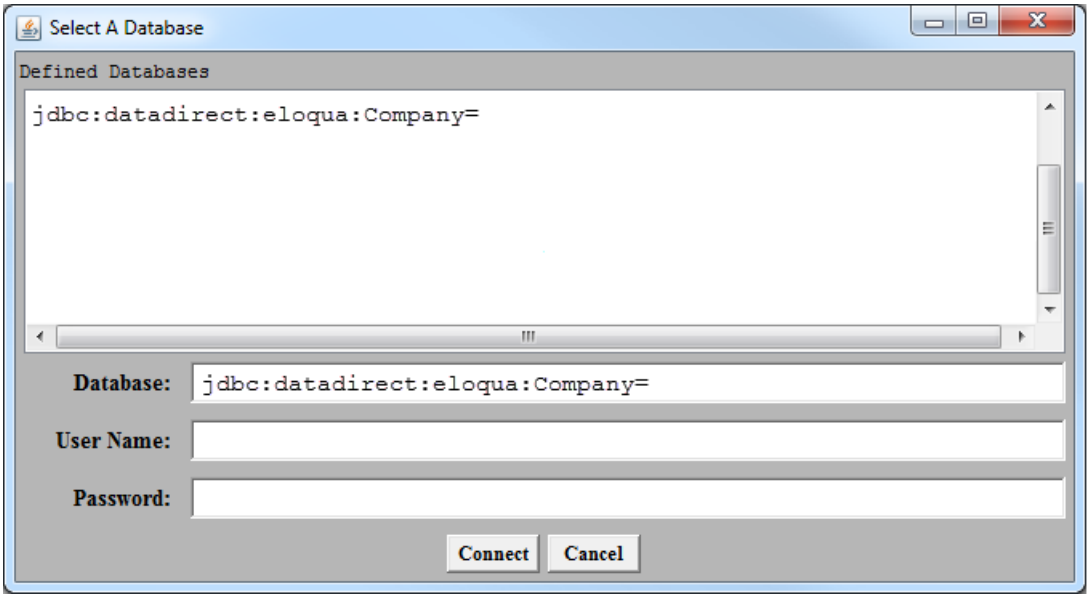
3. Click **Press Here to Continue**.

   The main dialog appears:



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

   The **Select A Database** dialog appears:



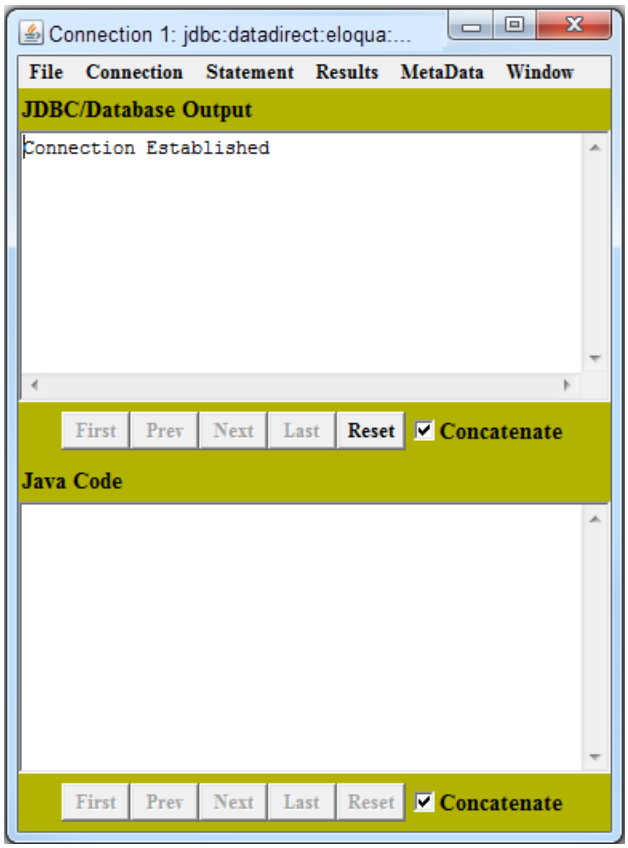
5. Select the appropriate database template from the **Defined Databases** field.
6. In the **Database** field, specify the company identifier issued by Oracle Eloqua during the registration process.

   For example:

   ```
   jdbc:datadirect:eloqua:Company=ABCcorp
   ```

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.)



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 `DataSource` is a Java 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 `DataSource` class implements the `javax.sql.DataSource` interface.

See also

Driver and DataSource classes on page 11

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 DataSource 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 DataSource and save it in your local file system using the File System JNDI Provider.

To connect using a JNDI DataSource, 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 class path. These steps are not required for LDAP implementations because the LDAP Service Provider is included with supported versions of Java SE.

Calling a DataSource in an application

Applications can call a Progress DataDirect DataSource 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 DataSource 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.

3

Using the driver

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

For details, see the following topics:

- Connecting from an application
- Using connection properties
- Performance considerations
- Proxy server support
- Unicode support
- Data encryption
- Bulk operations
- Catalog tables
- Identifiers
- Timeouts
- Scrollable cursors
- Large object (LOB) support
- Parameter metadata
- ResultSet metadata

- Rowset support
- Auto-generated keys
- Error handling

Connecting from an application

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

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

Driver and DataSource classes

Driver class

The `Driver` class is:

`com.ddtek.jdbc.eloqua.EloquaDriver`

DataSource class

The `DataSource` class is:

`com.ddtek.jdbcx.eloqua.EloquaDataSource`

Connecting using the DriverManager

You can connect to an Oracle Eloqua instance using the JDBC `DriverManager` with the `DriverManager.getConnection()` method. As the following example shows, this method specifies a string containing a connection URL.

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:eloqua:Company=ABCcorp;User=123@abccorp.com;Password=secret")
```

Setting the class path

The driver must be defined on your class path before you can connect. The class path 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 class path, you will receive a `class not found` exception when trying to load the driver. Set your system class path to include the `eloqua.jar` file as shown, where *`install_dir`* is the path to your product installation directory.

```
install_dir/lib/60/eloqua.jar
```

Windows Example

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

UNIX Example

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

Passing the connection URL

After setting the class path, the required connection information needs to be passed in the form of a connection URL. The connection URL takes the form:

```
jdbc:datadirect:eloqua:Company=company_id;
   User=user_id;Password=password[;property=value[;...]]
```

where:

company_id

specifies the company identifier issued by Oracle Eloqua during the registration process.

user_id

specifies the user name that used to connect to the Oracle Eloqua instance.

password

specifies the password for the specified `User`.

property=value

specifies additional connection property settings.

The following is an example of a connection URL to an Oracle Eloqua instance.

```
jdbc:datadirect:eloqua:Company=ABCcorp;User=123@abccorp.com;Password=secret
```

Testing the connection

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

Take the following steps to establish a connection.

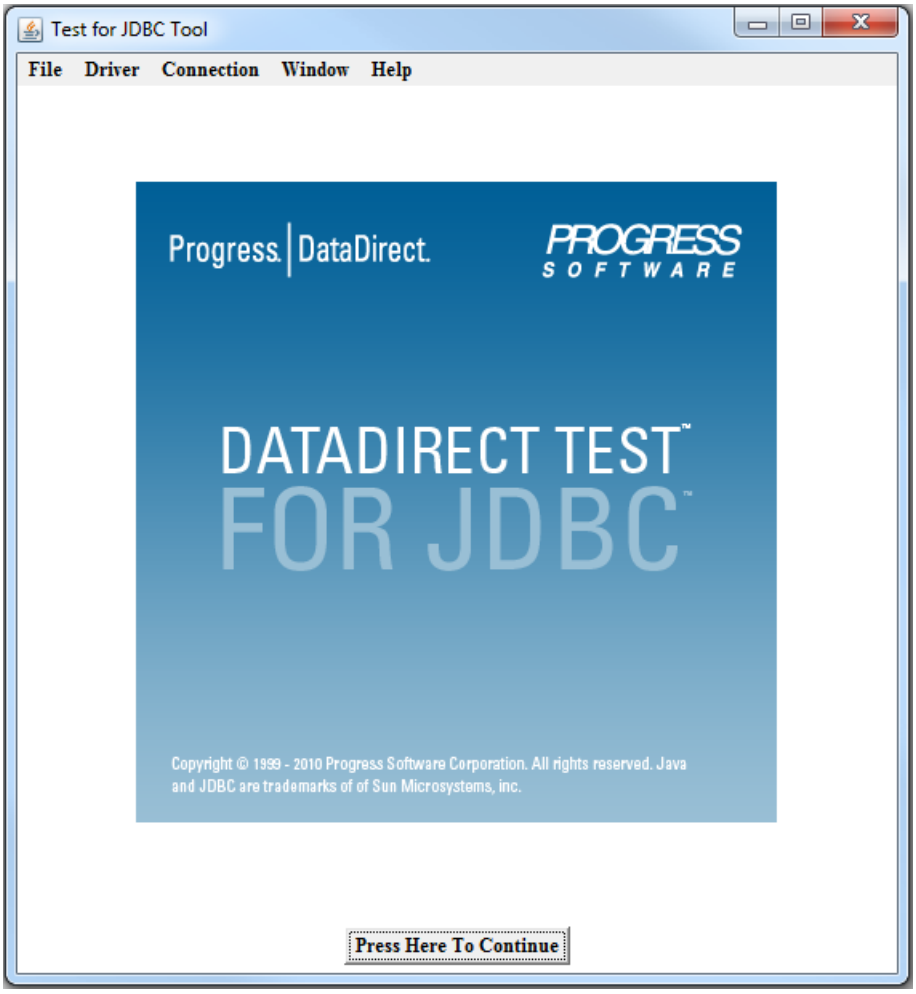
1. Navigate to the installation directory. The default location is:
   - Windows systems: `Program Files\Progress\DataDirect\JDBC\testforjdbc`
   - UNIX and Linux systems: `/opt/Progress/DataDirect/JDBC/testforjdbc`

   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:

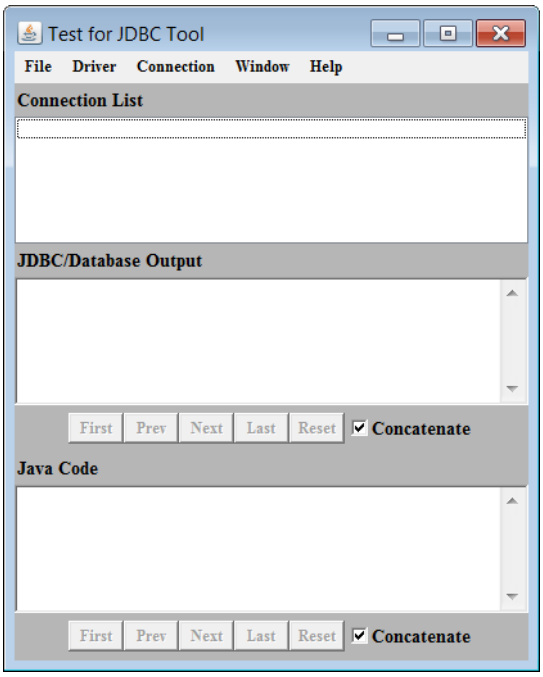
- `testforjdbc.bat` (on Windows systems)
- `testforjdbc.sh` (on UNIX and Linux systems)

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



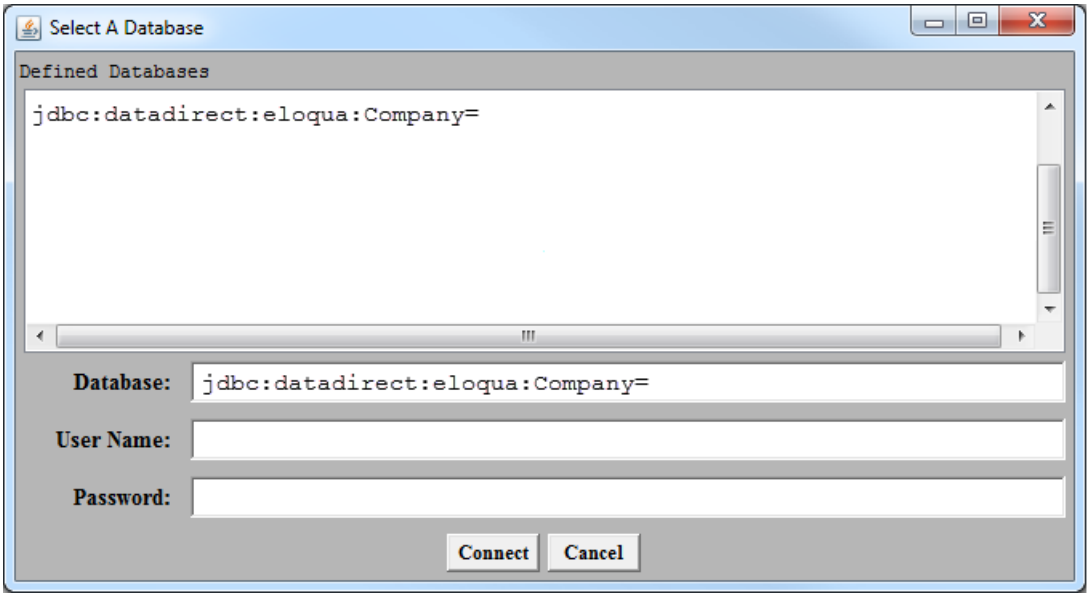
3. Click **Press Here to Continue**.

   The main dialog appears:



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

   The **Select A Database** dialog appears:



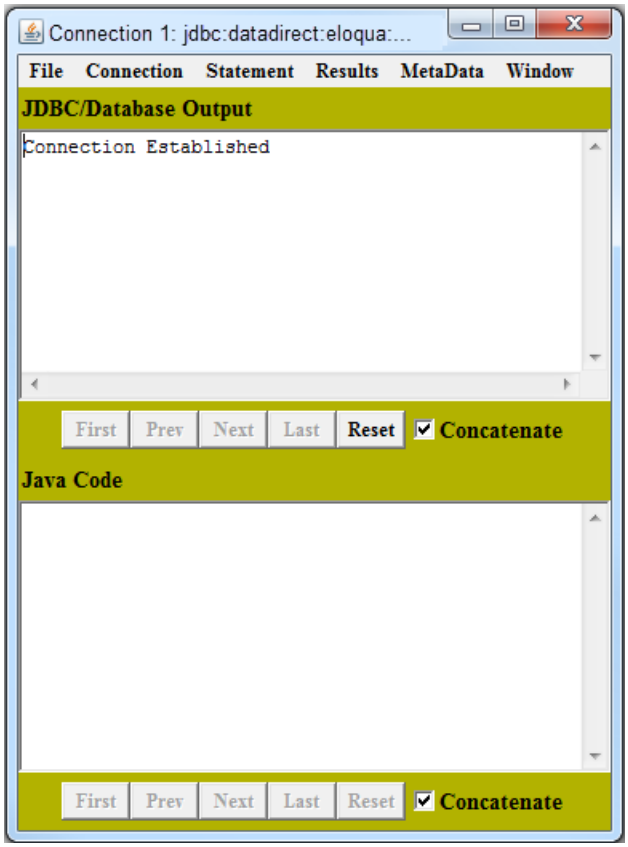
5. Select the appropriate database template from the **Defined Databases** field.
6. In the **Database** field, specify the company identifier issued by Oracle Eloqua during the registration process.

   For example:

   ```
   jdbc:datadirect:eloqua:Company=ABCcorp
   ```

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.)



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 `DataSource` is a Java 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 `DataSource` class implements the `javax.sql.DataSource` interface.

See also

Driver and DataSource classes on page 11

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 DataSource 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 DataSource and save it in your local file system using the File System JNDI Provider.

To connect using a JNDI DataSource, 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 class path. These steps are not required for LDAP implementations because the LDAP Service Provider is included with supported versions of Java SE.

Calling a DataSource in an application

Applications can call a Progress DataDirect DataSource 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 DataSource 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.

Using connection properties

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 DataSource. For a DriverManager connection, a property is expressed as a key value pair and takes the form *property*=*value*. For a DataSource connection, a property is expressed as a JDBC method and takes the form set*property*(*value*). Connection property names are case-insensitive. For example, Password is the same as password.

Note: In a JDBC data source, string values must be enclosed in double quotation marks, for example, setCompany("ABCCorp").

See Connection property descriptions on page 51 for an alphabetical list of connection properties and their descriptions.

Required properties

The following table summarizes connection properties which are required to connect to an Oracle Eloqua instance.

Table 4: Required properties

Property	Characteristic
Company on page 57	Specifies the company identifier issued by Oracle Eloqua during the registration process.
Password on page 62	Specifies the password used to connect to your Oracle Eloqua instance for user ID/password authentication.
User on page 69	Specifies the user ID used to connect to your Oracle Eloqua instance for user ID/password authentication.

See also

Connection property descriptions on page 51

Mapping properties

The following table summarizes connection properties which are used for the relational mapping of Oracle Eloqua data.

Table 5: Mapping properties

Property	Characteristic
ConfigOptions on page 57	Determines how the mapping of the remote data model to the relational data model is configured, customized, and updated.
CreateMap on page 60	Determines whether the driver creates the internal files required for a relational map of the native data when establishing a connection.
SchemaMap on page 65	Specifies the name and location of the configuration file used to create the relational map of native data. The driver looks for this file when connecting to an Oracle Eloqua instance. By default, if the file does not exist, the driver creates one.

See also

Connection property descriptions on page 51

Web service properties

The following table summarizes web service properties.

Table 6: Web service properties

Property	Characteristic
WSFetchSize on page 70	Specifies the maximum number of rows of data the driver can attempt to fetch for each call. The default is `1000` (rows).
WSRetryCount on page 71	Specifies the number of times the driver retries a timed-out Select request. Insert, Update, and Delete requests are never retried. The timeout period is specified by the WSTimeout connection property. The default is `0`.
WSTimeout on page 72	Specifies the time, in seconds, that the driver waits for a response to a web service request. The default is `120` seconds.

See also

Connection property descriptions on page 51
Performance considerations on page 38

Bulk load properties

The following table summarizes connection properties related to bulk load operations.

Table 7: Bulk load properties

Property	Characteristic
BulkActivityPageSize on page 53	Specifies the maximum number of rows to be fetched from `Activity_XXX` tables in a single request. The default is `50000`.
BulkPageSize on page 54	Specifies the maximum number of rows to be fetched from Oracle Eloqua in a single request. The default is `5000`.
BulkTimeout on page 55	Specifies the timeout duration for a bulk call in seconds. The default is `1800`.
BulkTopThreshold on page 56	Specifies a threshold for determining whether bulk operations are used to process qualifying Select queries that include a Top *n* clause. The default is `1000`. (See Bulk operations on page 40 for more information on qualifying Select queries.)

See also

Bulk operations on page 40
Connection property descriptions on page 51
Performance considerations on page 38

Proxy server properties

The following table summarizes connection properties used in the implementation of a proxy server.

Table 8: Proxy server properties

Property	Characteristic
ProxyPassword on page 63	Specifies the password needed to connect to a proxy server.
ProxyPort on page 64	Specifies the port number where the proxy server is listening for HTTP or HTTPS requests.
ProxyHost on page 63	Specifies a proxy server.
ProxyUser on page 65	Specifies the user name needed to connect to a proxy server.

See also

Connection property descriptions on page 51

Additional properties

The following table summarizes additional connection properties.

Table 9: Additional properties

Property	Characteristic
FailOnIncompleteData on page 61	Determines how the driver processes a query when no data is returned for some columns. For these columns, which together form incomplete data, the driver can either return NULL values or throw an exception. The default is `0` (disabled).
LogConfigFile on page 61	Specifies the filename of the configuration file used to initialize driver logging. The default filename is `ddlogging.properties`.
SpyAttributes on page 67	Enables DataDirect Spy to log detailed information about calls that are issued by the driver on behalf of the application. DataDirect Spy is not enabled by default.

See also

Connection property descriptions on page 51

Performance considerations

The connection properties described in this topic have direct impact on driver performance.

BulkActivityPageSize on page 53. The BulkActivityPageSize connection property specifies the maximum number of rows to be fetched from `Activity_XXX` tables in a single request. Generally, higher page sizes return results more quickly. However, Oracle Eloqua imposes a 32 MB limit on response package size. If queries return large records, too many records within a single page will exceed that limit, causing the query to fail. In addition, all of the objects returned within a page must be materialized as the page is retrieved, so sufficient Java heap space is necessary with large page sizes containing many small columns.

BulkPageSize on page 54. The BulkPageSize connection property specifies the maximum number of records to be fetched in a single request. Generally, higher page sizes return results more quickly. However, Oracle Eloqua imposes a 32 MB limit on response package size. If queries return large records, too many records within a single page will exceed that limit, causing the query to fail. In addition, all of the objects returned within a page must be materialized as the page is retrieved, so sufficient Java heap space is necessary with large page sizes containing many small columns.

BulkTopThreshold on page 56. The BulkTopThreshold connection property can be used to determine the point at which bulk operations can be used to execute qualifying Select queries. See Bulk operations on page 40 for more information on qualifying Select queries.

WSFetchSize on page 70. The WSFetchSize connection property allows you to specify the maximum number of rows allowed for a fetch on each call to the web service. The default value of 1000 rows typically provides the maximum throughput. However, for interactive applications, you can reduce response time by setting WSFetchSize at a lower value.

Proxy server support

In some environments, your application may need to connect through a proxy server, for example, if your application accesses an external resource such as a Web service. At a minimum, your application needs to provide the following connection information when you invoke the JVM if the application connects through a proxy server:

- Server name or IP address of the proxy server
- Port number on which the proxy server is listening for HTTP/HTTPS requests

The driver supports proxy server connections with the ProxyHost, ProxyPort, ProxyUser, and ProxyPassword connection properties. See Using connection properties on page 35 and Proxy server properties on page 38 for details.

In addition, if authentication is required, your application may need to provide a valid user ID and password for the proxy server. Consult with your system administrator for the required information.

For example, the following command invokes the JVM while specifying a proxy server named `pserver`, a port of `8888`, and provides a user ID and password for authentication:

```
java -Dhttp.proxyHost=pserver -Dhttp.proxyPort=8888 -Dhttp.proxyUser=smith
-Dhttp.proxyPassword=secret -cp eloqua.jar com.acme.myapp.Main
```

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.

Data encryption

All communication between the driver and Oracle Eloqua, including user ID/password authentication, is encrypted using TLS/SSL.

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

See Using connection properties on page 35 for information on specifying a user ID and password.

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.

Bulk operations

The driver supports Oracle Eloqua bulk operations with some limitations.

Note: Bulk operations are most efficient for queries that return large amounts of data for a relatively small set of columns. For example, `SELECT folderid,name,country,c_website FROM Account WHERE country='Switzerland'` has only four columns but many rows.

Bulk operations for some Select queries with a Top *n* clause are not supported because it is usually faster to use a standard query to fetch more columns for a few rows than to use a bulk operation. Nevertheless, you can use the BulkTopThreshold connection property to control, in part, how the driver handles queries with a Top *n* clause. The default value of BulkTopThreshold is 1000.

Queries on Account and Contact tables have additional limitations. The following criteria must be met for queries on Account and Contact tables.

- The result must have multiple rows.
- The result cannot have more than 250 columns.
- The result must include at least one user-defined column.
- The query must either have no Top *n* clause, or the value of *n* in the Top *n* clause must be greater than the value specified in the BulkTopThreshold custom property.
- The query can only include columns that the bulk interface supports. For more information, see the table below.

Table 10: Columns that cannot be retrieved using bulk operations

Account table	Contact table
accessedAt	accessedAt
createdBy	bouncebackDate
currentStatus	currentStatus
Description	createdBy
folderId	Description
Permissions	folderId
scheduledFor	Permissions
sourceTemplateId	scheduledFor
updatedBy	sourceTemplateId
	subscriptionDate
	updatedBy
	unsubscriptionDate

The following table provides some query examples and describes why they would not take advantage of bulk operations and offers suggestions for modifying them. However, there will obviously be use cases where an application will use queries that cannot be returned using bulk operations.

Query	Description
SELECT * FROM Contact WHERE Country='Switzerland'	There are more than 250 columns in the Contact table. To take advantage of bulk operations, constrain the SELECT statement to a set of less than 250 columns.
SELECT * FROM ContactList WHERE Region='East'	The ContactList table is not supported for bulk operations.
SELECT Id, C_Website FROM Account WHERE Id=17	This returns one row.
SELECT Id, C_Website FROM Contact WHERE Country='Switzerland' AND Region='EAST'	More than one criterion or comparison operator is used in the WHERE clause. **Tips:** • If a query has zero or one comparison operators (meaning one of =, <=, <, >, >=, <>) in the WHERE clause, the query will usually be processed using the bulk operations. • If a query contains the LIKE operator in the WHERE clause, the query is not processed using the bulk operations.

See also

Bulk load properties on page 37
Performance considerations on page 38
BulkActivityPageSize on page 53
BulkPageSize on page 54
BulkTimeout on page 55
BulkTopThreshold on page 56

Efficient queries

The driver supports queries based on the definition of Oracle Eloqua minimal, partial, and complete column sets. If you know which type of column you are querying, you can optimize queries by following these guidelines.

Note: Refer to your Oracle Eloqua documentation for details on minimal, partial, and complete column sets.

- A minimal column set provides the best performance. For example: `SELECT name,description,createdBy FROM Account WHERE scheduledFor='May'`
- A partial column set provides the next best performance. For example: `SELECT name, description, createdBy, country FROM Account WHERE scheduledFor='May'` is processed faster than: `SELECT name, description, createdBy, country, c_website FROM Account WHERE scheduledFor='May'`, which contains columns from Complete Column Set.
- A complete column set is processed faster than a query that uses the bulk interface only when the required number of records is less than the actual number of records that the bulk query would return. For example, the following query would return all rows: `SELECT name, country, c_website FROM Account`

`WHERE scheduledFor='May'`. However, if you wanted to return the first 500 of 10,000 records, the query `SELECT TOP 500 name, country, c_website FROM Account WHERE scheduledFor='May'` would probably be faster, even though it would not be fetched in a bulk operation.

See also

Bulk load properties on page 37
Performance considerations on page 38
BulkActivityPageSize on page 53
BulkPageSize on page 54
BulkTimeout on page 55
BulkTopThreshold on page 56

Catalog tables

The driver provides a standard set of catalog tables that maintain the information returned by the methods of the JDBC `DatabaseMetaData`, `ParameterMetaData`, and `ResultSetMetaData` interfaces. If possible, use JDBC metadata methods to obtain this information instead of querying the catalog tables directly.

The driver also provides additional catalog tables that maintain metadata specific to the driver. This section defines the catalog tables that provide driver-specific information. The catalog tables are defined in the `INFORMATION_SCHEMA` schema.

SYSTEM_REMOTE_SESSIONS catalog table

The system table named `SYSTEM_REMOTE_SESSIONS` stores information about the each of the remote sessions that are active for a given database. The values in the `SYSTEM_REMOTE_SESSION` table are read-only.

The following table defines the columns of the `SYSTEM_REMOTE_SESSIONS` table, which is sorted on the following columns: `SESSION_ID` and `SCHEMA`.

Table 11: SYSTEM_REMOTE_SESSIONS catalog table

Column Name	Data Type	Description
`SESSION_ID`	INTEGER, NOT NULL	The connection (session) id with which the remote session is associated.
`SCHEMA`	VARCHAR(128), NOT NULL	The schema name that is mapped to the remote session.
`TYPE`	VARCHAR(30), NOT NULL	The remote session type. The current valid type is `Eloqua`.
`INSTANCE`	NULL	The value for `INSTANCE` is null.
`VERSION`	VARCHAR(30), NOT NULL	The version of the remote data source to which the session is connected. This is the version of the web service API the driver is using to connect to Oracle Eloqua.

Column Name	Data Type	Description
CONFIG_OPTIONS	LONGVARCHAR, NOT NULL	The configuration options used to define the remote data model to relational data model mapping.
SESSION_OPTIONS	LONGVARCHAR, NOT NULL	The options used to establish the remote connection. This typically is information needed to log into the remote data source. The password value is not displayed.
WS_CALL_COUNT	INTEGER, NOT NULL	The number of Web service calls made through this remote session. The value of the WS_CALL_COUNT column can be reset using the ALTER SESSION statement.
WS_AGGREGATE_CALL_COUNT	INTEGER, NOT NULL	The total of all of the web service calls made to the same remote data source by all active connections using the same server name and user ID.
REST_AGGREGATE_CALL_COUNT	INTEGER, NOT NULL	The number of REST calls made by this connection. REST calls are used for bulk operations, invoking reports, and describing report parameters.

SYSTEM_SESSIONS catalog table

The system table named SYSTEM_SESSIONS stores information about current system sessions. The values in the SYSTEM_SESSIONS table are read-only.

The following table defines the columns of the SYSTEM_SESSIONS table.

Table 12: SYSTEM_SESSIONS catalog table

Column	Data Type	Description
SESSION_ID	INTEGER, NOT NULL	A unique ID that identifies this session. The system function CURSESSIONID( ) returns the session ID associated with the connection. See Supported scalar functions on page 13 for more information about the CURSESSIONID( ) system function.
CONNECTED	DATETIME, NOT NULL	The date and time the session was established.
USERNAME	VARCHAR (128), NOT NULL	The name of the schema map that the session is using.
IS_ADMIN	BOOLEAN	For internal use only.
AUTOCOMMIT	BOOLEAN, NOT NULL	For future use.

Column	Data Type	Description
READONLY	BOOLEAN, NOT NULL	The value for READONLY is False.
MAX_ROWS	INTEGER, NOT NULL	For future use.
LAST_IDENTITY	BIGINT, NULLABLE	For future use.
TRANSACTION_SIZE	INTEGER, NOT NULL	For future use.
CURRENT_SCHEMA	VARCHAR (128), NOT NULL	The current schema for the session. The current schema may be changed using the ALTER SESSION SET CURRENT_SCHEMA statement.
STMT_CALL_LIMIT	INTEGER, NOT NULL	The maximum number of Web service calls that the driver uses in attempting to execute a query to a remote data source. The statement call limit for the session may be changed via the ALTER SESSION SET STMT_CALL_LIMIT statement.

Identifiers

Identifiers are used to refer to objects exposed by the driver, such as tables, columns, or caches. The driver supports both unquoted and quoted identifiers for naming objects. An unquoted identifier must start with an ASCII alpha character and can be followed by zero or more ASCII alpha or numeric characters. Unquoted identifiers are converted to uppercase before being used.

Quoted identifiers must be enclosed in double quotation marks (""). A quoted identifier can contain any Unicode character, including the space character, and is case-sensitive. The Oracle Eloqua 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.

Note: When object names are passed as arguments to catalog functions, the case of the value must match the case of the name in the database. If an unquoted identifier name was used when the object was created, the value passed to the catalog function must be uppercase because unquoted identifiers are converted to uppercase before being used. If a quoted identifier name was used when the object was created, the value passed to the catalog function must match the case of the name as it was defined. Object names in results returned from catalog functions are returned in the case that they are stored in the database.

KeywordConflictSuffix

The driver also includes the KeywordConflictSuffix configuration option. KeywordConflictSuffix allows you to avoid naming conflicts when the name of an object corresponds to the name of a SQL engine keyword. KeywordConflictSuffix specifies a string of up to five alphanumeric characters that the driver appends to any object or field name that conflicts with a SQL engine keyword. For example, if you specify KeywordConflictSuffix=tab, the driver maps the case object to the casetab column.

Timeouts

Most remote data sources impose a limit on the duration of active sessions, meaning a session can fail with a session timeout error if the session extends past the limit. The following scenarios show how the driver handles timeouts.

Session timeouts

If the driver receives a session timeout error from a data source, the driver automatically attempts to re-establish a new session. The driver uses the initial server name, port (if appropriate), remote user ID, and remote password (encrypted) to re-establish the session. If the attempt fails, the driver returns an error indicating that the session timed out and the attempt to re-establish the session failed.

Web service request timeouts

You can configure the driver to never time out while waiting for a response to a Web service request or to wait for a specified interval before timing out by setting the WSTimeout connection property for fetch requests. Additionally, in a case where requests might fail, you can configure the driver to retry the request a specified number of times by setting the WSRetryCount connection property. If all subsequent attempts to retry a request fail, the driver will return an error indicating that the service request timed out and that the subsequent requests failed.

Bulk operation timeouts

The BulkTimeout connection property can be used to specify the duration allowed for bulk calls. If the query takes longer than the specified duration, the query could be aborted. The default value is 18000 seconds.

See Using connection properties on page 35 and Web service properties on page 37 for details.

Scrollable cursors

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.

Large object (LOB) support

The driver allows you to retrieve LONGVARCHAR data, using JDBC methods designed for Character Large Object (CLOB) data. When using these methods to update long data as CLOBs data, 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 CLOB data provides some of the same benefits as retrieving and updating CLOB data, such as:

- Provides random access to data
- Allows searching for patterns in the data

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

Parameter metadata

The driver supports returning parameter metadata as described in the following topics.

Insert and Update statements

The driver supports returning parameter metadata for the following forms of Insert and Update statements:

- `INSERT INTO employee VALUES(?, ?, ?)`
- `INSERT INTO department (col1, col2, col3) VALUES(?, ?, ?)`
- `UPDATE employee SET col1=?, col2=?, col3=? WHERE col1 operator ? [{AND | OR} col2 operator ?]`

where:

operator

is any of the following SQL operators:

=, <, >, <=, >=, and <>.

Select statements

The driver supports returning parameter metadata for Select statements that contain parameters in ANSI SQL-92 entry-level predicates, for example, such as COMPARISON, BETWEEN, IN, LIKE, and EXISTS predicate constructs. Refer to the ANSI SQL reference for detailed syntax.

Parameter metadata can be returned for a Select statement if one of the following conditions is true:

- The statement contains a predicate value expression that can be targeted against the source tables in the associated FROM clause. For example:

  ```
  SELECT * FROM foo WHERE bar > ?
  ```

  In this case, the value expression "bar" can be targeted against the table "foo" to determine the appropriate metadata for the parameter.

- The statement contains a predicate value expression part that is a nested query. The nested query's metadata must describe a single column. For example:

  ```
  SELECT * FROM foo WHERE (SELECT x FROM y    WHERE z = 1) < ?
  ```

The following Select statements show further examples for which parameter metadata can be returned:

```
SELECT col1, col2 FROM foo WHERE col1 = ? AND col2 > ?
SELECT ... WHERE colname = (SELECT col2 FROM t2 WHERE col3 = ?)
SELECT ... WHERE colname LIKE ?
SELECT ... WHERE colname BETWEEN ? and ?
```

```
SELECT ... WHERE colname IN (?, ?, ?)
SELECT ... WHERE EXISTS(SELECT ... FROM T2 WHERE col1 < ?)
```

ANSI SQL-92 entry-level predicates in a WHERE clause containing GROUP BY, HAVING, or ORDER BY statements are supported. For example:

```
SELECT * FROM t1 WHERE col = ? ORDER BY 1
```

Joins are supported. For example:

```
SELECT * FROM t1,t2 WHERE t1.col1 = ?
```

Fully qualified names and aliases are supported. For example:

```
SELECT a, b, c, d FROM T1 AS A, T2 AS B WHERE A.a = ? AND B.b = ?
```

ResultSet metadata

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

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

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

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

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

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

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

Rowset support

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

- `CachedRowSets`

- `FilteredRowSets`
- `WebRowSets`
- `JoinRowSets`
- `JDBCRowSets`

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

Auto-generated keys

The driver supports retrieving the values of auto-generated keys. An auto-generated key returned by the driver is the value of an auto-increment column.

An application can return values of auto-generated keys when it executes an Insert statement. How you return these values depends on whether you are using an Insert statement with a `Statement` object or with a `PreparedStatement` object, as outlined in the following scenarios:

- When using an Insert statement with a `Statement` object, the driver supports the following form of the `Statement.execute` and `Statement.executeUpdate` methods to instruct the driver to return values of auto-generated keys:
  - `Statement.execute(String sql, int autoGeneratedKeys)`
  - `Statement.execute(String sql, int[] columnIndexes)`
  - `Statement.execute(String sql, String[] columnNames)`
  - `Statement.executeUpdate(String sql, int autoGeneratedKeys)`
  - `Statement.executeUpdate(String sql, int[] columnIndexes)`
  - `Statement.executeUpdate(String sql, String[] columnNames)`
- When using an Insert statement with a `PreparedStatement` object, the driver supports the following form of the `Connection.prepareStatement` method to instruct the driver to return values of auto-generated keys:
  - `Connection.prepareStatement(String sql, int autoGeneratedKeys)`
  - `Connection.prepareStatement(String sql, int[] columnIndexes)`
  - `Connection.prepareStatement(String sql, String[] columnNames)`

An application can retrieve values of auto-generated keys using the `Statement.getGeneratedKeys()` method. This method returns a `ResultSet` object with a column for each auto-generated key.

See also

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

Error handling

SQLExceptions

The driver reports errors to the application by throwing SQLExceptions. Each SQLException contains the following information:

- Description of the probable cause of the error, prefixed by the component that generated the error
- Native error code (if applicable)
- String containing the XOPEN SQLstate

Driver Errors

An error generated by the driver has the format shown in the following example:

```
[DataDirect][Eloqua JDBC Driver]Timeout expired.
```

You may need to check the last JDBC call your application made and refer to the JDBC specification for the recommended action.

Database Errors

An error generated by the database has the format shown in the following example:

```
[DataDirect][Eloqua JDBC Driver][Eloqua]Invalid Object Name.
```

If you need additional information, use the native error code to look up details in your database documentation.

4

Connection property descriptions

You can use connection properties to customize the driver for your environment. This section lists the connection properties supported by the driver and describes each property. You can use these connection properties with either the JDBC `DriverManager` or a JDBC `DataSource`. 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 table provides a summary of the connection properties supported by the driver, their corresponding data source methods, and their default values.

Table 13: Driver properties

Connection String Property	Data Source Method	Default Value
BulkActivityPageSize on page 53	`setBulkActivityPageSize`	`50000` (rows)

Connection String Property	Data Source Method	Default Value
BulkPageSize on page 54	setBulkPageSize	5000 (rows)
BulkTimeout on page 55	setBulkTimeout	1800 (seconds)
BulkTopThreshold on page 56	setBulkTopThreshold	1000 (rows)
Company on page 57	setCompany	No default value
ConfigOptions on page 57	setConfigOptions	CheckBoxAsText=0; KeywordConflictSuffix=
CreateMap on page 60	setCreateMap	notExist
FailOnIncompleteData on page 61	setFailOnIncompleteData	0
LogConfigFile on page 61	setLogConfigFile	ddlogging.properties
Password on page 62	setPassword	No default value
ProxyHost on page 63	setProxyHost	No default value
ProxyPassword on page 63	setProxyPassword	No default value
ProxyPort on page 64	setProxyPort	No default value
ProxyUser on page 65	setProxyUser	No default value
SchemaMap on page 65	setSchemaMap	Default value depends on environment
SpyAttributes on page 67	setSpyAttributes	No default value
User on page 69	setUser	No default value
WSFetchSize on page 70	setWSFetchSize	1000 (rows)
WSRetryCount on page 71	setWSRetryCount	0
WSTimeout on page 72	setWSTimeout	120 (seconds)

For details, see the following topics:

- BulkActivityPageSize
- BulkPageSize
- BulkTimeout
- BulkTopThreshold

- Company
- ConfigOptions
- CreateMap
- FailOnIncompleteData
- LogConfigFile
- Password
- ProxyHost
- ProxyPassword
- ProxyPort
- ProxyUser
- SchemaMap
- SpyAttributes
- User
- WSFetchSize
- WSRetryCount
- WSTimeout

BulkActivityPageSize

Purpose

Specifies the maximum number of rows to be fetched from `Activity_XXX` tables in a single request.

Valid Values

x

where

x

is a positive integer between 2 and 50000.

Notes

- Generally, higher page sizes return results more quickly. However, Oracle Eloqua imposes a 32 MB limit on response package size. If queries return large records, too many records within a single page will exceed that limit, causing the query to fail.
- All of the objects returned within a page must be materialized as the page is retrieved, so sufficient Java heap space is necessary with large page sizes containing many small columns.

Data Source Method

`setBulkActivityPageSize`

Default

`50000`

Data Type

Int

See also

Bulk operations on page 40
Bulk load properties on page 37
BulkPageSize on page 54
BulkTimeout on page 55
BulkTopThreshold on page 56
Performance considerations on page 38

BulkPageSize

Purpose

Specifies the maximum number of records to be fetched from Oracle Eloqua in a single request.

Valid Values

x

where

x

is a positive integer between 2 and 50000.

Notes

- Generally, higher page sizes return results more quickly. However, Oracle Eloqua imposes a 32 MB limit on response package size. If queries return large records, too many records within a single page will exceed that limit, causing the query to fail.
- All of the objects returned within a page must be materialized as the page is retrieved, so sufficient Java heap space is necessary with large page sizes containing many small columns.

Data Source Method

`setBulkPageSize`

Default

`5000`

Data Type

Int

See also

Bulk operations on page 40
Bulk load properties on page 37
BulkActivityPageSize on page 53
BulkTimeout on page 55
BulkTopThreshold on page 56
Performance considerations on page 38

BulkTimeout

Purpose

Specifies the timeout duration for a bulk call in seconds.

Valid Values

x

where

x

is a positive integer between 3600 and 1209600.

Notes

Oracle Eloqua automatically clears out the bulk staging area after this timeout. If the query is large and the data takes more than this time to run, the driver may throw a timeout error and abort the query midstream.

Data Source Method

`setBulkTimeout`

Default

`1800`

Data Type

Int

See also

Bulk operations on page 40
Bulk load properties on page 37
BulkActivityPageSize on page 53
BulkPageSize on page 54
BulkTopThreshold on page 56

BulkTopThreshold

Purpose

Specifies a threshold for determining whether bulk operations are used to process qualifying Select queries that include a Top *n* clause. (See Bulk operations on page 40 for more information on qualifying Select queries.)

Valid Values

x

where

x

is a positive integer greater than 0.

Behavior

If *x* is less than or equal *n* (the number of rows specified in the Top *n* clause), the standard mechanism is used to process the query.

If *x* is greater than *n* (the number of rows specified in the Top *n* clause), bulk load is used to process the qualifying queries.

Notes

Oracle Eloqua automatically clears out the bulk staging area after this timeout, so if the query is large and the data takes more than this time to run, the query could be aborted midstream.

Data Source Method

`setBulkTopThreshold`

Default

`1000`

Data Type

Int

See also

Bulk operations on page 40
Bulk load properties on page 37
BulkActivityPageSize on page 53
BulkPageSize on page 54
BulkTimeout on page 55
Performance considerations on page 38

Company

Purpose

Specifies the company identifier issued by Oracle Eloqua during the registration process.

Valid Values

company_id

where:

company_id

is the company identifier issued by Oracle Eloqua during the registration process. For example, if your company name is ABC Corporation, Oracle Eloqua might issue the company identifier as ABCCorp.

Data Source Method

`setCompany`

Default

No default value

Data Type

String

See also

Required properties on page 36
Password on page 62
User on page 69

ConfigOptions

Purpose

Determines how the mapping of the remote data model to the relational data model is configured, customized, and updated. (See also CreateMap on page 60.

Notes

This property is primarily used for initial configuration of the driver for a particular user. It is not intended for use with every connection. By default, the driver configures itself and this option is normally not needed. If ConfigOptions is specified on a connection after the initial configuration, the values specified for ConfigOptions must match the values specified for the initial configuration.

Valid Values

("*key* = *value* [; *key* = *value*]")

where:

key

is one of the following configuration options:

- CheckBoxAsText
- KeywordConflictSuffix

value

specifies a setting for the configuration option.

When specifying configuration options in a connection string, key value pairs must be separated by a semicolon and enclosed in quotation marks and then in parentheses. For example:

```
ConfigOptions=("CheckBoxAsText=1;KeywordConflictSuffix=TAB")
```

Similarly, when specifying configuration options using a DataSource method, key value pairs must be enclosed in quotation marks and separated by a semicolon. For example:

```
setConfigOptions("CheckBoxAsText=1;KeywordConflictSuffix=TAB;RefreshMap=true")
```

Data Source Method

setConfigOptions

Default

```
CheckBoxAsText=0;
KeywordConflictSuffix=
```

Data Type

String

See also

Mapping objects to tables on page 18
Mapping properties on page 36
CreateMap on page 60
SchemaMap on page 65

CheckBoxAsText (configuration option)

Purpose

Specifies whether the check box values of user-defined columns should be returned as string or Boolean.

Valid Values

0 | 1

Behavior

If set to `0`, the check box value is returned as a Boolean. The Boolean is described as a Bit in the relational map, either `checkedValue` or `uncheckedValue`. When neither of these values is present, the check box value is returned as NULL.

If set to `1`, the stored literal value of the check box is returned as a string. The string is described as a Varchar in the relational map.

Default

`0`

Data Type

String

KeywordConflictSuffix (configuration option)

Purpose

Specifies a string of up to five alphanumeric characters that the driver appends to any object or field name that conflicts with a SQL engine keyword.

Valid Values

`string`

where:

`string`

is a string of up to five alphanumeric characters.

Example

A field called `CASE` exists in the native Oracle Eloqua data. To avoid a naming conflict with the SQL engine keyword CASE, you could set `KeywordConflictSuffix=TAB`. In this scenario, the driver maps the Case object to the CASETAB column.

Notes

Do not use a string that matches the suffix of a custom table, for example, `CASEOFICE`. If you specify `KeywordConflictSuffix=OFICE`, a name collision occurs with the Standard object `CASE` and the custom table `CASEOFICE`, or a table with a column called `CASEOFICE`. In this situation, the standard object `CASE` is returned. The custom object is ignored.

Default

No default value

Data Type

String

CreateMap

Purpose

Determines whether the driver creates the internal files required for a relational map of the native data when establishing a connection.

Valid Values

`forceNew` | `no` | `notExist`

Behavior

If set to `forceNew`, the driver deletes the group of internal files specified by SchemaMap and creates a new group of these files at the same location.

If set to `no`, the driver uses the current group of internal files specified by SchemaMap. If the files do not exist, the connection fails.

If set to `notExist`, the driver uses the current group of internal files specified by SchemaMap. If the files do not exist, the driver creates them.

Notes

- The internal files share the same directory as the schema map's configuration file. This directory is specified with the SchemaMap connection property.
- You can refresh the internal files related to an existing relational view of your data with the SQL extension Refresh Map. Refresh Map runs a discovery against your native data and updates your internal files accordingly.

Data Source Method

`setCreateMap`

Default

`notExist`

Data Type

String

See also

Mapping objects to tables on page 18
Mapping properties on page 36
ConfigOptions on page 57
SchemaMap on page 65

FailOnIncompleteData

Purpose

Determines how the driver processes a query when no data is returned for some columns. For these columns, which together form incomplete data, the driver can either return NULL values or throw an exception.

Valid Values

`false` | `true`

Behavior

If set to `false`, the driver returns NULL values for columns that return no data.

If set to `true`, the driver attempts to retrieve the complete data. The driver throws an exception if the data cannot be retrieved.

Data Source Method

`setFailOnIncompleteData`

Default

`false`

Data Type

Boolean

LogConfigFile

Purpose

Specifies the filename of the configuration file used to initialize driver logging. If the driver cannot locate the specified file when establishing the connection, the connection fails and the driver returns an error.

Valid Values

`string`

where:

`string`

is the relative or fully qualified path of the properties file to load to initialize driver logging. If you do not specify a path, the driver looks for this file in the current working directory. If the specified file does not exist, the driver continues searching for an appropriate properties file.

Data Source Method

`setLogConfigFile`

Default

`ddlogging.properties`

Data Type

String

See also

Refer to "Using Java logging" in the *Progress DataDirect for JDBC Drivers Reference*.

Password

Description

Specifies the password used to connect to your Oracle Eloqua instance for user ID/password authentication.

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

Valid Values

`password`

where:

`password`

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

Data Source Method

`setPassword`

Default

No default value

Data Type

String

See also

Required properties on page 36
Company on page 57
User on page 69

ProxyHost

Description

Specifies a proxy server.

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 Method

`setProxyHost`

Default

No default value

Data Type

String

See also

Proxy server properties on page 38
ProxyPassword on page 63
ProxyPort on page 64
ProxyUser on page 65

ProxyPassword

Purpose

Specifies the password needed to connect to a proxy server.

Valid Values

password

where:

`password`

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

Data Source Method

`setProxyPassword`

Default

No default value

Data Type

String

See also

Proxy server properties on page 38
ProxyHost on page 63
ProxyPort on page 64
ProxyUser on page 65

ProxyPort

Purpose

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

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 Method

`setProxyPort`

Default

`0`

Data Type

Int

See also

Proxy server properties on page 38

ProxyHost on page 63
ProxyPassword on page 63
ProxyUser on page 65

ProxyUser

Purpose

Specifies the user name needed to connect to a proxy server.

Valid Values

user_name

where:

user_name

is a valid user ID for the proxy server.

Data Source Method

setProxyUser

Default

No default value

Data Type

String

See also

Proxy server properties on page 38
ProxyHost on page 63
ProxyPassword on page 63
ProxyPort on page 64

SchemaMap

Purpose

Specifies the name and location of the configuration file used to create the relational map of native data. The driver looks for this file when connecting to an Oracle Eloqua instance. If the file does not exist, the driver creates one.

Valid Values

string

where:

`string`

is the absolute path and filename of the configuration file, including the `.config` extension. For example, if SchemaMap is set to a value of `C:\\Users\\Default\\AppData\\Local\\Progress\\DataDirect\\Eloqua_Schema\\testuser.config`, the driver either creates or looks for the configuration file `testuser.config` in the directory `C:\Users\Default\AppData\Local\Progress\DataDirect\Eloqua_Schema\`.

Notes

- When connecting to a server, the driver looks for the SchemaMap configuration file. If the configuration file does not exist, the driver creates a SchemaMap configuration file using the name and location you have provided. If you do not provide a name and location for a SchemaMap configuration file, the driver creates it using default values.
- The driver uses the path specified in this connection property to store additional internal files.
- You can refresh the internal files related to an existing relational view of your data by using the SQL extension Refresh Map. Refresh Map runs a discovery against your native data and updates your internal files accordingly.

Example

As the following examples show, escapes are needed when specifying SchemaMap for a `DataSource` but are not used when specifying SchemaMap in a `DriverManager` connection URL.

DriverManager example

```
jdbc:datadirect:eloqua:Company=ABCcorp;User=123@abccorp.com;Password=secret;
SchemaMap=C:\Users\Default\AppData\Local\Progress\DataDirect
   \Eloqua_Schema\testuser.config
```

DataSource example

```
EloquaDataSource mds = new EloquaDataSource();
mds.setDescription("My EloquaDataSource");
mds.setCompany("ABCCorp")
mds.setUser("123@abccorp.com")
mds.setPassword("secret")
mds.setSchemaMap("C:\\Users\\Default\\AppData\\Local\\Progress
   \\DataDirect\\Eloqua_Schema\\testuser.config")
```

Data Source Method

`setSchemaMap`

Default

The default is determined by the environment. The driver attempts to create the files in a subdirectory of the first available directory in the following order:

- Windows
  - DD_HOME environment variable
  - dd.home system property
  - LOCALAPPDATA environment variable
  - APPDATA environment variable
  - user.home system property

For Windows, the file path takes the following format:

available_location\Progress\DataDirect\Eloqua_Schema*user_name*.config

- UNIX/Linux
  - DD_HOME environment variable
  - dd.home system property
  - user.home system property

  For UNIX/Linux, the file path takes the following format:

  available_location/progress/datadirect/Eloqua_schema/*user_name*.config

Data Type

String

See also

Mapping objects to tables on page 18
Mapping properties on page 36
ConfigOptions on page 57
CreateMap on page 60

SpyAttributes

Purpose

Enables DataDirect Spy to log detailed information about calls that are 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.

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).

Attribute	Description
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.
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.

Attribute	Description
`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`.

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.

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)
```

Data Source Method

`setSpyAttributes`

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.

User

Purpose

Specifies the user ID used to connect to your Oracle Eloqua instance for user ID/password authentication.

Valid Values

userid

where:

userid

is a valid user ID used to connect to your Oracle Eloqua instance.

Data Source Method

`setUser`

Default

No default value

Data Type

String

See also

Required properties on page 36
Company on page 57
Password on page 62

WSFetchSize

Purpose

Specifies the maximum number of rows of data the driver can attempt to fetch for each call.

Valid Values

x

where

x

is a positive integer from 1 to 1000 that defines the maximum number of rows the driver can attempt to fetch for each call.

Data Source Method

`setWSFetchSize`

Default

`1000` (rows)

Data Type

Int

See also

Web service properties on page 37
Performance considerations on page 38
WSRetryCount on page 71
WSTimeout on page 72

WSRetryCount

Description

Specifies the number of times the driver retries a timed-out Select request. Insert, Update, and Delete requests are never retried. The timeout period is specified by the WSTimeout connection property.

Valid Values

`0` | *x*

where:

x

is a positive integer.

Behavior

If set to `0`, the driver does not retry timed-out requests after the initial unsuccessful attempt.

If set to *x*, the driver retries the timed-out request the specified number of times.

Data Source Method

`setWSRetryCount`

Default

`0`

Data Type

Int

See also

Web service properties on page 37
WSFetchSize on page 70
WSTimeout on page 72

WSTimeout

Purpose

Specifies the time, in seconds, that the driver waits for a response to a web service request.

Valid Values

`0` | *x*

where:

x

is a positive integer that defines the number of seconds the driver waits for a response to a web service request.

Behavior

If set to `0`, the driver waits indefinitely for a response; there is no timeout.

If set to *x*, the driver uses the value as the default timeout for any statement created by the connection. If a Select request times out and WSRetryCount is set to retry timed-out requests, the driver retries the request the specified number of times.

Data Source Method

`setWSTimeout`

Default

`120` (seconds)

Data Type

Int

See also

Web service properties on page 37
WSFetchSize on page 70
WSRetryCount on page 71

5

Supported SQL functionality

The driver provides support for SQL statements and 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)
- Delete
- Insert
- Refresh Map (EXT)
- Select
- Update
- SQL expressions
- Subqueries

Alter Session (EXT)

Purpose

The `Alter Session` statement allows you to change various attributes of a connection session.

Syntax

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

where:

attribute_name

Specifies the name of the attribute to be changed.

value

Refers to the specific value setting for that attribute.

The following table lists session attributes and describes them.

Table 14: Alter Session Attributes

Attribute Name	Session Type	Description
Current_Schema	Database	Sets the current schema for the database 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 session. For example: `ALTER SESSION SET CURRENT_SCHEMA=eloqua`

Ws_Call_Count	Remote	Resets the Web service call count of a session to the value specified. The value must be zero or a positive integer. Ws_Call_Count represents the total number of Web service calls made to the data store instance for the current session. For example:
		`ALTER SESSION SET eloqua.WS_CALL_COUNT=0`
		The current value of Ws_Call_Count can be obtained by referring to the System_Remote_Sessions system table. 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.

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 on page 82 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.

Insert

Purpose

The Insert statement is used to add new rows to a 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. See Specifying an external ID column on page 76 for more information.

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 on page 89 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 on page 77 for information about Select statements.

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 database 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)
```

Refresh Map (EXT)

Purpose

The REFRESH MAP statement adds newly discovered objects to your relational view of native data. It also incorporates any configuration changes made to your relational view by reloading the schema map configuration file.

Syntax

```
REFRESH MAP
```

Notes

REFRESH MAP is an expensive query since it involves the discovery of native data.

Select

Purpose

The `Select` statement can be used 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]
[orderby_clause]
[limit_clause]
```

where:

`select_clause`

specifies the columns from which results are to be returned by the query. See Select clause on page 78 for a complete explanation.

from_clause

specifies one or more tables on which the other clauses in the query operate. See From clause on page 80 for a complete explanation.

where_clause

is optional and restricts the results that are returned by the query. See Where clause on page 82 for a complete explanation.

groupby_clause

is optional and allows query results to be aggregated in terms of groups. See Group By clause on page 83 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 on page 83 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 on page 84 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 on page 85 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 on page 86 for a complete explanation.

orderby_clause

is optional and sorts the results that are returned by the query. See Order By clause on page 86 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 on page 87 for a complete explanation.

Select clause

The `Select` clause can be used to specify 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], ...]}
```

```
[INTO [DISK | TEMP] new_table]
SELECT [{LIMIT offsetlimit | TOP limit}][ALL | DISTINCT]
{select_expression | table.* | *} [, ...]
[INTO [DISK | TEMP] new_table]
```

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 0`*`number`*.

`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 on page 89 for details. *`column_expression`* can also include aggregate functions. See Aggregate functions on page 80 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
```

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`.

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

```
SELECT DISTINCT dep FROM 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`.

The `INTO` clause copies the result set into *`new_table`*. `INTO DISK` creates the new table in cached memory. `INTO TEMP` creates a temporary table.

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`.

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 field name (for example, `AVG(SALARY)`) or in combination with a more complex column expression (for example, `AVG(SALARY * 1.07)`). The column expression can be preceded by the `Distinct` operator. The `Distinct` operator eliminates duplicate values from an aggregate expression. For example:

```
COUNT (DISTINCT last_name)
```

In this example, only distinct last name values are counted.

The following table lists valid aggregate functions.

Table 15: Aggregate Functions

Aggregate	Returns
`SUM`	The total of the values in a numeric field expression. For example, `SUM(SALARY)` returns the sum of all salary field values.
`AVG`	The average of the values in a numeric field expression. For example, `AVG(SALARY)` returns the average of all salary field 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 field values. A special example is `COUNT(*)`, which returns the number of rows in the set, including rows with *`NULL`* values.
`MAX`	The maximum value in any field expression. For example, `MAX(SALARY)` returns the maximum salary field value.
`MIN`	The minimum value in any field expression. For example, `MIN(SALARY)` returns the minimum salary field value.

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 on page 82 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.

Outer join escape sequences

JDBC supports the SQL-92 left, right, and full outer join syntax. The escape sequence for outer joins is:

```
  {oj outer-join}
```

where *outer-join* is

```
table-reference {LEFT | RIGHT | FULL} OUTER JOIN {table-reference | outer-join} ON
search-condition
```

where *table-reference* is a database table name, and *search-condition* is the join condition you want to use for the tables.

```
Example: SELECT Customers.CustID, Customers.Name, Orders.OrderID, Orders.Status FROM {oj
 Customers LEFT OUTER JOIN Orders ON Customers.CustID=Orders.CustID} WHERE
Orders.Status='OPEN'
```

The following outer join escape sequences are supported:

- Left outer joins
- Right outer joins
- Full outer joins
- Nested outer joins

Join in a From clause

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
```

Syntax

```
FROM table_name {RIGHT OUTER | INNER | LEFT OUTER | CROSS} 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`*). For example:

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

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

This 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

Subqueries on page 96

SQL expressions on page 89

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

Subqueries on page 96

SQL expressions on page 89

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

is a column name, a constant value, or an expression. An expression does not have to match a column expression in the `Select` clause.

expr2

is a column name, a constant value, or an expression. An expression does not have to match a column expression in the `Select` clause.

rel_operator

is the relational operator that links the two expressions.

Example

This 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

Subqueries on page 96

SQL expressions on page 89

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

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
```

`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
UNION
SELECT salary, last_name FROM raises
```

Except and Minus operators

Purpose

Returns the rows from the left `Select` statement that are not included in the result of the right `Select` statement. These operators are synonymous.

Syntax

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

`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

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

Subqueries on page 96

SQL expressions on page 89

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_id
LIMIT 20
```

Update

Purpose

An `Update` statement changes the value of columns in 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.

Notes

- A `Where` clause can be used to restrict which rows are updated.

See also

Subqueries on page 96

Where clause on page 82

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'
```

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 or more ASCII alphanumeric characters. Unquoted identifiers are converted to uppercase before being used.

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 16: 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`
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	X'*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 and 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.

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 17: 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 18: 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 19: 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`

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`
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 20: 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 21: Operator precedence

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

Precedence	Operator
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, as listed and described in Supported scalar functions on page 13.

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 22: 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]`

Condition	Description
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`

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)
```