Progress DataDirect for JDBC for Oracle Service Cloud Driver

User's Guide

Release 5.1.4

Copyright

Visit the following page online to see Progress Software Corporation's current Product Documentation Copyright Notice/Trademark Legend: https://www.progress.com/legal/documentation-copyright.

Updated: 2025/10/23

Table of Contents

Welcome to the Progress DataDirect for JDBC for Oracle Service Cloud Driver....9
What's New in this Release?....10
Requirements....11
Data Source and Driver Classes....11
Connection Properties....11
Version String Information....12
Data Types....12
getTypeInfo()....13
Mapping Objects to Tables....16
Standard and Custom Objects....17
Primary Objects, Sub-objects, and Lists of Sub-objects....17
Many-to-many Relationships....18
DataDirect tools....18
Troubleshooting....19
Contacting Technical Support....19

Getting started....21
Connecting with the JDBC Driver Manager....21
Setting the Classpath....22
Passing the Connection URL....22
Testing the Connection....23

Using the driver....27
Required permissions for Java SE with the standard Security Manager enabled....28
Permissions for establishing connections....28
Granting access to Java properties....29
Granting access to temporary files....29
Connecting from an Application....29
Connecting Using the JDBC Driver Manager....30
Connecting Using Data Sources....31
Testing the Connection....32
Connecting Through a Proxy Server....35
Using Connection Properties....36
Required Properties....36
Embedded Database Properties....36
Proxy Server Properties....37

Web Service Properties..37
Statement Pooling Properties..38
Additional Properties..39
Performance Considerations...40
Using the Database Configuration File...41
<Database>...42
<User>...42
<UseSchema>...42
<Schema>...43
<ConfigOptions>..43
<SessionOptions>..44
Data Encryption...44
FIPS (Federal Information Processing Standard)..44
Views and Remote/Local Tables...45
Client-Side Caches..45
Creating a Cache...46
Modifying a Cache Definition...46
Disabling and Enabling a Cache..46
Refreshing Cache Data..46
Dropping a Cache..47
Cache MetaData..47
Catalog Tables...47
SYSTEM_CACHES Catalog Table..47
SYSTEM_CACHE_REFERENCES Catalog Table..49
SYSTEM_REMOTE_SESSIONS Catalog Table..49
SYSTEM_SESSIONS Catalog Table..50
Reports..51
Identifiers...53
IP Addresses..54
Auto-Generated Keys Support...54
Rowset Support..55
ResultSet MetaData Support..55
Parameter Metadata Support...55
Unicode support...56
Error Handling..56
Timeouts..56
Scrollable Cursors..57
Large Objects (LOBs)...57
"Poor Performing Query" Error Message..57

Connection Property Descriptions..61

ConfigOptions...64
ConvertNull...67
CreateDB..67

DatabaseName....68
EnableAccessRequestHeader....69
EnablePagingWithOrderById....69
FetchSize....70
ImportStatementPool....71
InitializationString....71
InsensitiveResultSetBufferSize....72
InterfaceName....73
JavaDoubleToString....74
LogConfigFile....74
LoginHost....75
LoginTimeout....75
ManagedObjects....76
MaxPooledStatements....76
OracleServiceCloudDatabase....77
Password....78
ProxyHost....79
ProxyPassword....79
ProxyPort....80
ProxyUser....80
RefreshDirtyCache....81
RegisterStatementPoolMonitorMBean....81
SpyAttributes....82
StmtCallLimit....84
StmtCallLimitBehavior....85
TransactionMode....85
User....86
WSCompressData....86
WSFetchSize....87
WSRetryCount....88
WSTimeout....89

Supported SQL Statements and Extensions 91

Alter Cache (EXT)....92
Alter Index....93
Alter Sequence....94
Alter Session (EXT)....94
Alter Table....96
    Add Clause: Columns....97
    Add Clause: Constraints....98
    Drop Clause: Columns....98
    Drop Clause: Constraints....99
    Rename Clause....99
Checkpoint....100

Create Cache (EXT)....100
- Refresh Interval Clause....102
- Initial Check Clause....102
- Persist Clause....103
- Enabled Clause....104
- Call Limit Clause....104
- Filter Clause....105

Create Index....106
Create Sequence....106
- Next Value For Clause....107

Create Table....107
- Column Definition for Local Tables....109
- Constraint Definition for Local Tables....110

Create View....114
Drop Cache (EXT)....115
Drop Index....116
Drop Sequence....116
Drop Table....117
Drop View....118
Explain Plan....118
Refresh Cache (EXT)....118
Set Checkpoint Defrag....119
Set Logsize....120
Select....120
- Select Clause....122

SQL Expressions....131
- Column Names....132
- Literals....132
- Operators....134
- Functions....138
- Conditions....138

Subqueries....139
- IN Predicate....139
- EXISTS Predicate....140
- UNIQUE Predicate....140
- Correlated Subqueries....140

1

Welcome to the Progress DataDirect for JDBC for Oracle Service Cloud Driver

The Progress® DataDirect® *for* JDBC™ Oracle Service Cloud™ driver supports standard SQL query language to provide read-only access to data managed by the Oracle Service Cloud Web service. The driver maps the Oracle Service Cloud data model to a set of relational tables and uses a client-side data cache for improved performance. The driver supports the Oracle RightNow CX API version 1.2 for sites using Oracle Service Cloud February 2014 or later.

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-service-cloud.

For details, see the following topics:

- What's New in this Release?
- Requirements
- Data Source and Driver Classes
- Connection Properties
- Version String Information
- Data Types
- Mapping Objects to Tables
- DataDirect tools

- Troubleshooting
- Contacting Technical Support

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 5.1.4 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 44 for details.
  - The new ManagedObjects connection property allows you to specify the managed objects that you want the driver to fetch metadata for. For details, see ManagedObjects on page 76
  - The new EnableAccessRequestHeader connection property allows you to enable the access request header for a connection to improve its performance and quality of service. For details, see EnableAccessRequestHeader on page 69
  - The new NamedIDBehavior config option allows you to determine whether the Name attribute of NamedID fields is exposed in the relational map. This option can be used to avoid "poor performing query" errors by reducing the size of result sets. For details, see "Poor Performing Query" Error Message on page 57 and ConfigOptions on page 64.
  - The driver no longer registers the Statement Pool Monitor as a JMX MBean by default. To register the Statement Pool Monitor and manage statement pooling with standard JMX API calls, the new RegisterStatementPoolMonitorMBean connection property must be set to true. For details, see RegisterStatementPoolMonitorMBean on page 81.
- **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 82 for details.
  - The Interface Name connection property has been deprecated. The driver will continue to provide backward compatibility for it, but the current and newer versions of the driver will not use it for connections.

Release Highlights for 5.1.4 GA

- Support for Oracle RightNow CX API version 1.2 against sites using Oracle Service Cloud February 2014 or later.
- Support for all JDBC Core functions.

- Support for core SQL-92 grammar.
- Maps the Oracle Service Cloud data model to a set of relational tables.
- Support for client-side data cache for improved performance.
- Support for SSL encryption and User ID/Password authentication.
- Support for statement pooling.
- Support for connecting through a proxy server.

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.

Note: To use the driver on a Java Platform with standard Security Manager enabled, certain permissions must be set in the security policy file of the Java Platform. See Required permissions for Java SE with the standard Security Manager enabled on page 28 for details.

Data Source and Driver Classes

The driver class for the driver is:

com.ddtek.jdbc.oracleservicecloud.OracleServiceCloudDriver

The driver provides the following data source class that supports the functionality for all JDBC specifications and Java SE 8 or higher:

com.ddtek.jdbcx.oracleservicecloud.OracleServiceCloudDataSource

Note: For compatibility purposes, the driver also provides a second data sources class, com.ddtek.jdbcx.oracleservicecloud.OracleServiceCloudDataSource40. This data source class functions identically to com.ddtek.jdbc.oracleservicecloud.OracleServiceCloudDriver; however, it is named after a data source class provided in earlier releases, allowing applications that rely on the older version to use the driver without changes.

Connection Properties

The driver includes thirty connection properties. You can use these connection properties to customize the driver for your environment. Connection properties can be used to accomplish different tasks, such as implementing driver functionality and optimizing performance. You can specify connection properties in a connection URL or within a JDBC data source object.

See Using Connection Properties on page 36 and Connection Property Descriptions on page 61 for more information.

Version String Information

The DatabaseMetaData.getDriverVersion() method returns a Driver Version string in the format:

```
M.m.s.bbbbbb(CXXXX.FYYYYYY.UZZZZZZ)
```

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:

```
5.1.0.000002(C0003.F000001.U000002)
      |____| |___| |______| |_____|
      Driver Cloud Frame     Utl
```

Data Types

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

Note: See getTypeInfo() on page 13 for getTypeInfo() results for Oracle Service Cloud data stores supported by the driver.

Table 1: Data Type Mapping

Oracle Service Cloud Data Type	JDBC Data Type
BASE_64_BINARY	LONGVARBINARY
BOOLEAN	BOOLEAN
DATE	DATE
DATETIME	TIMESTAMP
DECIMAL	DOUBLE
ID	BIGINT

Oracle Service Cloud Data Type	JDBC Data Type
INTEGER	INTEGER
LONG	BIGINT
LONGTEXT	LONGVARCHAR
STRING	VARCHAR

getTypeInfo()

The DatabaseMetaData.getTypeInfo() method returns information about data types. The following table provides getTypeInfo() results for all sources supported by the Oracle Service Cloud driver.

Table 2: getTypeInfo() Results

TYPE_NAME = BASE_64_BINARY AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = null DATA_TYPE = -4 FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = BASE_64_BINARY MAXIMUM_SCALE = null	MINIMUM_SCALE = null NULLABLE = 1 NUM_PREC_RADIX = null PRECISION = 2147483647 SEARCHABLE = 0 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 FIXED_PREC_SCALE = false LITERAL_PREFIX = null LITERAL_SUFFIX = null LOCAL_TYPE_NAME = BOOLEAN MAXIMUM_SCALE = null	MINIMUM_SCALE = null NULLABLE = 1 NUM_PREC_RADIX = null PRECISION = 1 SEARCHABLE = 3 SQL_DATA_TYPE = null SQL_DATETIME_SUB = null UNSIGNED_ATTRIBUTE = null

TYPE_NAME = DATE AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = null DATA_TYPE = 91 FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = DATE MAXIMUM_SCALE = null	MINIMUM_SCALE = null NULLABLE = 1 NUM_PREC_RADIX = null PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = null SQL_DATETIME_SUB = null UNSIGNED_ATTRIBUTE = null
TYPE_NAME = DATETIME AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = null DATA_TYPE = 93 FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = DATETIME MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = null PRECISION = 19 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 = 8 FIXED_PREC_SCALE = false LITERAL_PREFIX = null LITERAL_SUFFIX = null LOCAL_TYPE_NAME = DECIMAL MAXIMUM_SCALE = 18	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 18 SEARCHABLE = 3 SQL_DATA_TYPE = null SQL_DATETIME_SUB = null UNSIGNED_ATTRIBUTE =false

TYPE_NAME = ID AUTO_INCREMENT = true CASE_SENSITIVE = false CREATE_PARAMS = null DATA_TYPE = -5 FIXED_PREC_SCALE = false LITERAL_PREFIX = null LITERAL_SUFFIX = null LOCAL_TYPE_NAME = ID 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 =INTEGER AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = null DATA_TYPE = 4 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 = LONG AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = null DATA_TYPE = -5 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 = LONGTEXT AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = null DATA_TYPE = -1 FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = LONGTEXT MAXIMUM_SCALE = null	MINIMUM_SCALE = null NULLABLE = 1 NUM_PREC_RADIX = null PRECISION = 1048576 SEARCHABLE = 0 SQL_DATA_TYPE = null SQL_DATETIME_SUB = null UNSIGNED_ATTRIBUTE = null
TYPE_NAME = STRING AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = null DATA_TYPE = 12 FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = STRING MAXIMUM_SCALE = null	MINIMUM_SCALE = null NULLABLE = 1 NUM_PREC_RADIX = null PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = null SQL_DATETIME_SUB = null UNSIGNED_ATTRIBUTE = null

Mapping Objects to Tables

Data mapping describes how to map elements between two distinct data models, usually distinguished as the source and the target. To support SQL access to Oracle Service Cloud data stores, the driver maps data source objects to target relational tables. The driver maps standard and custom objects and includes relationships defined between objects. The first time it connects to an Oracle Service Cloud instance the driver automatically maps source objects and fields to tables and columns. On subsequent connections, the Oracle Service Cloud driver detects when the underlying schema has changed and automatically updates the relational map.

The driver uses a local embedded database to instantiate the mapping of the remote data source objects to tables and the metadata associated with those tables. The driver creates one database for each user. The embedded database is created in the directory from which the application is run. The CreateDB connection property allows you to update or re-create the embedded database that defines and handles the object-to-table mapping. The embedded database uses the user ID specified for the connection as the name of the database. 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 database. You can set the DatabaseName connection property to override the default setting for the name and location of the database.

See the following topics for more information on configuring embedded databases:

- Using Connection Properties on page 36
- Embedded Database Properties on page 36
- ConfigOptions on page 64
- Using the Database Configuration File on page 41

Standard and Custom Objects

The driver automatically maps Oracle Service Cloud data source objects and fields to tables and columns the first time it connects to an Oracle Service Cloud instance. The driver maps standard and custom objects and includes relationships defined between objects. You can use the getPrimaryKey(), getExportedKeys(), and getImportedKeys() methods to report relationships among objects.

Standard and custom objects are mapped to corresponding standard and custom tables. Every standard and custom object comes with a set of fields by default. These fields are collectively referred to as audit columns. By default, the driver includes the audit columns in table definitions when mapping objects to tables. If you do not want your application to see the audit columns, then you may use the ConfigOptions connection property to exclude audit columns. For details, see ConfigOptions on page 64.

The driver maps standard objects into the RIGHTNOW schema; however, custom objects are handled differently. Oracle Service Cloud custom objects are created in packages through the Oracle Service Cloud user interface. When mapping custom objects from the data model to the relational model, the driver exposes the different namespaces for the packages as different schemas within the relational model. For example, if you have created an object called `bar` in the `foo` namespace, the driver will expose a schema called `FOO` that contains a table called `BAR`. To access this table, the application would have to fully qualify it as `FOO.BAR` or change the current schema using `ALTER SESSION SET CURRENT_SCHEMA = FOO`.

The driver can map system fields to columns within a table so that they can be easily identified. If you want the driver to change the names of system columns, make sure the MapSystemColumnNames key of the ConfigOptions connection property is set to `1`. For details, see ConfigOptions on page 64.

Primary Objects, Sub-objects, and Lists of Sub-objects

To map the Oracle Service Cloud object data model to a relational data model, the driver decomposes primary objects into parent-child tables. The fields of a primary object and the fields of its associated sub-objects are exposed as columns in a parent table, while lists of sub-objects and their fields are exposed as a child table.

For example, an Oracle Service Cloud data store has a primary object called `EMPLOYEE` which takes the following form:

```
EMPLOYEE (ID, NAME, EMAIL, PHONES, HIREDATE)
```

The driver decomposes this primary object into two separate but related tables. Where the `NAME` sub-object contains the fields `FIRST` and `LAST`, the driver exposes corresponding columns in an `EMPLOYEE` parent table using the `<objectname>_<fieldname>` pattern:

```
EMPLOYEE (ID, NAME_FIRST, NAME_LAST, EMAIL, HIREDATE, PRIMARY KEY (ID))
```

In turn, where `PHONES` is a list of sub-objects with `NUMBER` and `TYPE` fields, the driver exposes a distinct yet related child table:

```
EMPLOYEE_PHONE (EMPLOYEE_ID, NUMBER, TYPE, FOREIGN KEY (EMPLOYEE_ID)
     REFERENCES EMPLOYEE(ID) ON DELETE CASCADE)
```

As the example shows, the name of the child table is concatenation of the parent table name and the name of the list of sub-objects. The example further shows that the driver maintains the relationship between parent and child tables by exposing the parent table's primary key as a foreign key in the child table. The name of this referential column is a concatenation of the parent table's name and the name of the parent table's primary key column joined by an underscore (_) separator.

Many-to-many Relationships

When mapping associated objects to a relational data model, the driver exposes one-to-one and one-to-many relationships as foreign key relationships. However, many-to-many relationships are handled differently. For many-to-many relationships, the driver exposes a join table to show associations between objects. For example, suppose the source data model contains an `EMPLOYEE` object and a `DEPARTMENT` object that are related.

```
EMPLOYEE (ID, NAME, HIREDATE)
```

The `DEPARTMENT` object contains a list of fields and takes the form:

```
DEPARTMENT (ID, NAME, PROJECTS)
```

Suppose there is an association between the `EMPLOYEE` object and the `DEPARTMENT` object. `EMPLOYEES` can belong to multiple `DEPARTMENTS` and `DEPARTMENTS` can contain multiple `EMPLOYEES`. In this instance, the driver exposes the following join table using the `<objectname>_<associationname>` pattern:

```
EMPLOYEE_DEPARTMENT (EMPLOYEE_ID, DEPARTMENT_ID)
```

The following query retrieves the name and ID of all the employees who work in the Mailroom:

```
SELECT EMPLOYEE.ID, EMPLOYEE.NAME
     FROM EMPLOYEE, EMPLOYEE_DEPARTMENT, DEPARTMENT
     WHERE EMPLOYEE.ID = EMPLOYEE_DEPARTMENT.EMPLOYEE_ID
     AND EMPLOYEE_DEPARTMENT.DEPARTMENT_ID = DEPARTMENT.ID
     AND DEPARTMENT.NAME = 'Mailroom'
```

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.

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:

- Connecting with the JDBC Driver Manager
- Setting the Classpath
- Passing the Connection URL
- Testing the Connection

Connecting with the JDBC Driver Manager

Once the driver is installed, you can immediately connect from an application to your database with the JDBC Driver Manager by specifying the connection URL in the `DriverManager.getConnection()` method.

For information on connecting using data sources, see Connecting Using Data Sources on page 31.

Setting the Classpath

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

```
install_dir/lib/51/rightnow.jar
```

Windows Example

```
CLASSPATH=.;C:\Program Files\Progress\DataDirect\JDBC\lib\51\rightnow.jar
```

UNIX Example

```
CLASSPATH=.:/opt/Progress/DataDirect/JDBC/lib/51/rightnow.jar
```

Passing the Connection URL

After registering the driver, the connection information needs to be passed in the form of a connection URL. The connection URL takes the form:

```
jdbc:datadirect:oracleservicecloud:loginHost=host;
[property=value[;...]]
```

where:

`LoginHost`

is the host name of the Oracle Service Cloud site to which you want to connect, for example, mysite.custhelp.com. This value should not include an internet protocol such as http:// or https://.

`property=value`

specifies connection property settings. Multiple properties are separated by a semi-colon. For more information on connection properties, see Using Connection Properties on page 36.

This example shows how to establish a connection to an Oracle Service Cloud data source and include user ID and password information:

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:oracleservicecloud:loginHost=mysite.custhelp.com;
 User=test;Password=secret");
```

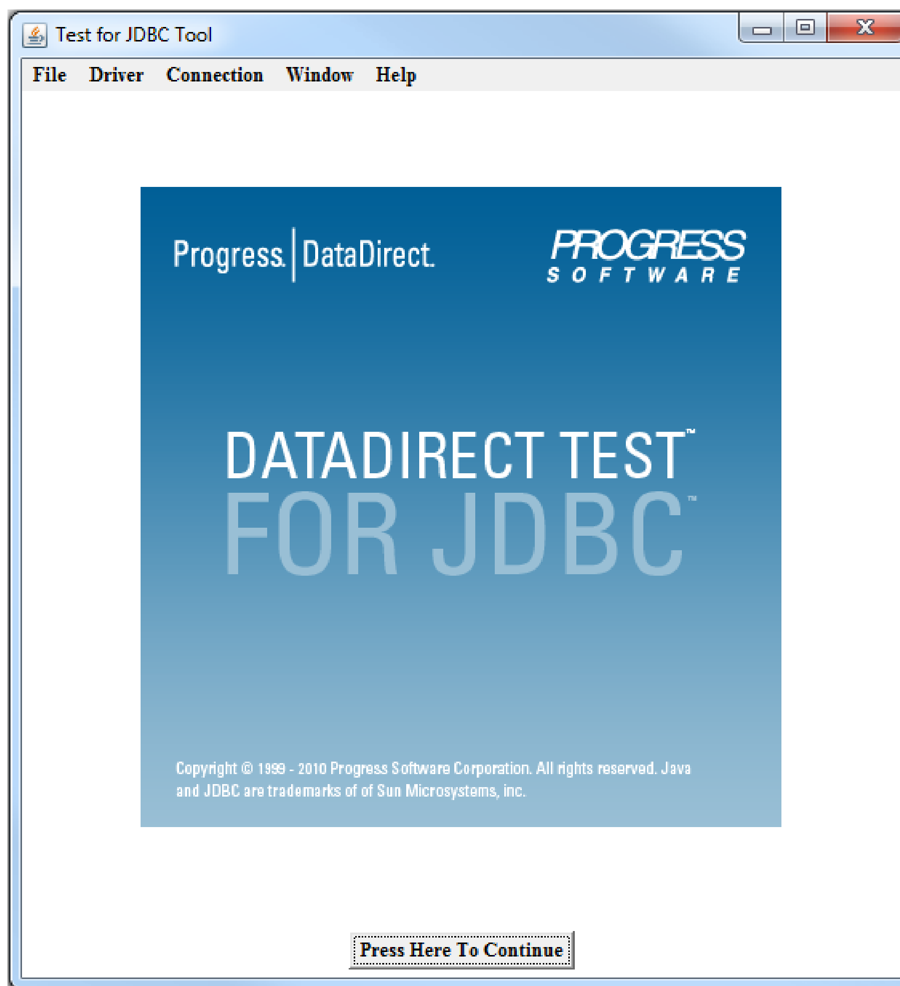
Testing the Connection

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

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

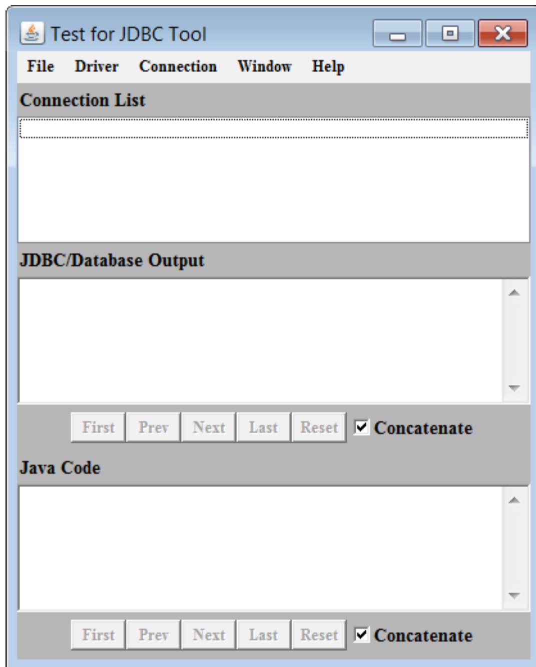
1. Navigate to your installation directory.
2. From the `testforjdbc` folder, run the platform-specific tool:
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

   The Test for JDBC Tool window appears:



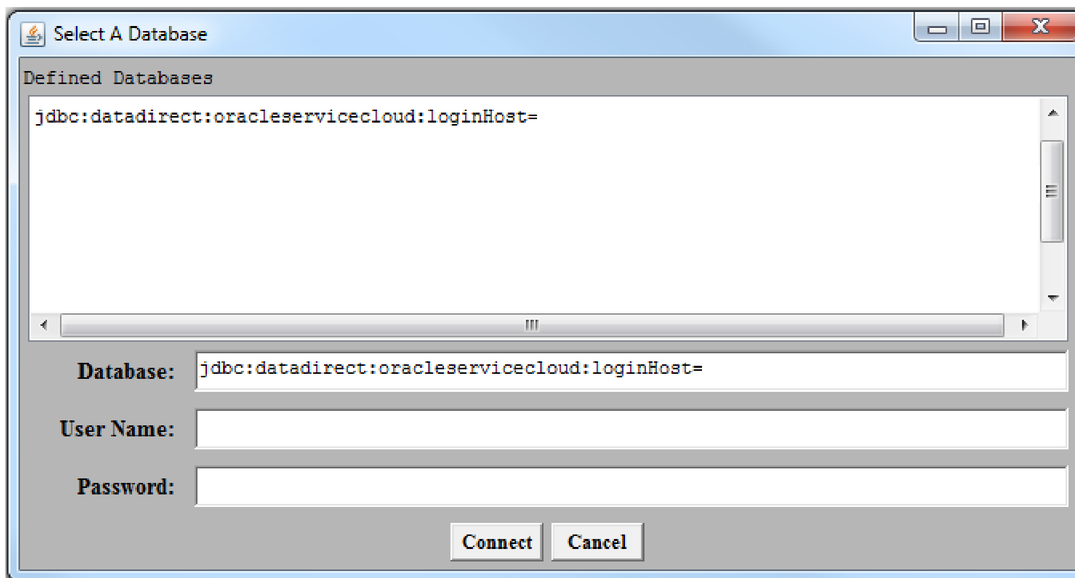
3. Click **Press Here to Continue**.

   The main dialog appears:



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

   The Select A Database dialog appears:



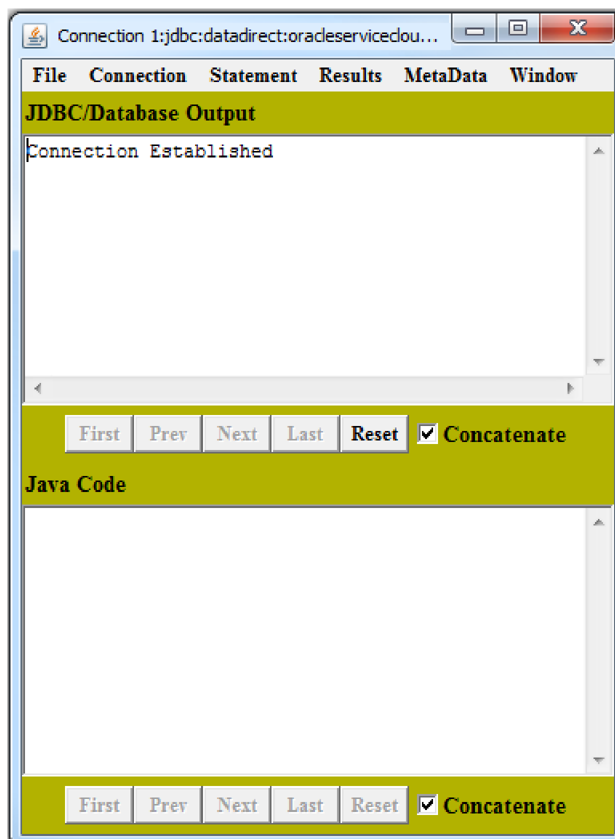
5. Select the appropriate database template from the **Defined Databases** field.
6. In the **Database** field, specify the correct LoginHost for your Oracle Service Cloud data source.

   For example:

```
jdbc:datadirect:oracleservicecloud:loginHost=mysite.custhelp.com;
```

7. If you did not specify your user name and password in the connection URL, enter this information 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.

3

Using the driver

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

For details, see the following topics:

- Required permissions for Java SE with the standard Security Manager enabled
- Connecting from an Application
- Connecting Through a Proxy Server
- Using Connection Properties
- Performance Considerations
- Using the Database Configuration File
- Data Encryption
- Views and Remote/Local Tables
- Client-Side Caches
- Catalog Tables
- Reports
- Identifiers
- IP Addresses
- Auto-Generated Keys Support

- Rowset Support
- ResultSet MetaData Support
- Parameter Metadata Support
- Unicode support
- Error Handling
- Timeouts
- Scrollable Cursors
- Large Objects (LOBs)
- "Poor Performing Query" Error Message

Required permissions for Java SE with the standard Security Manager enabled

Using the driver on a Java platform with the standard Security Manager enabled requires certain permissions to be set in the Java SE security policy file `java.policy`. The default location of this file is *`java_install_dir`*`/jre/lib/security`.

Note: Security manager may be enabled by default in certain scenarios, such as running on an application server or in a Web browser applet.

To run an application on a Java platform with the standard Security Manager, use the following command:

```
"java -Djava.security.manager application_class_name"
```

where *`application_class_name`* is the class name of the application.

Refer to your Java documentation for more information about setting permissions in the security policy file.

Permissions for establishing connections

To establish a connection to the database server, the driver must be granted the permissions as shown in the following example:

```
grant codeBase "file:/install_dir/lib/60/-" {
   permission java.net.SocketPermission "*", "connect";
};
```

where:

`install_dir`

is the product installation directory.

Granting access to Java properties

To allow the driver to read the value of various Java properties to perform certain operations, permissions must be granted as shown in the following example:

```
grant codeBase "file:/install_dir/lib/60/-" {
   permission java.util.PropertyPermission "*", "read, write";
};
```

where:

install_dir

is the product installation directory.

Granting access to temporary files

Access to the temporary directory specified by the JVM configuration must be granted in the Java SE security policy file to use insensitive scrollable cursors or to perform client-side sorting of DatabaseMetaData result sets. The following example shows permissions that have been granted for the `C:\TEMP` directory:

```
grant codeBase "file:/install_dir/lib/60/-" {
// Permission to create and delete temporary files.
// Adjust the temporary directory for your environment.
   permission java.io.FilePermission "C:\\TEMP\\-", "read,write,delete";
};
```

where:

install_dir

is the product installation directory.

Connecting from an Application

Once the driver is installed and configured, you can connect from your application to your database in either of the following ways:

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

Note: For details on connecting through a proxy server, see Connecting Through a Proxy Server on page 35 and Proxy Server Properties on page 37.

Connecting Using the JDBC Driver Manager

One way to connect to a database is through the JDBC Driver Manager using the DriverManager.getConnection() method. This method specifies a string containing a connection URL. This example shows how to establish a connection to an Oracle Service Cloud data source:

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:oracleservicecloud:loginHost=mysite.custhelp.com;
 User=test;Password=secret");
```

Registering the Driver

Registering the driver with the JDBC Driver Manager allows the driver manager to load the driver. The class name for the driver is:

- com.ddtek.jdbc.oracleservicecloud.OracleServiceCloudDriver

You can register the driver with the JDBC Driver Manager using any of the following methods:

- **Method 1**: Set the Java system property jdbc.drivers using the Java -D option. The jdbc.drivers property is defined as a colon-separated list of driver class names. For example:

  ```
  java -Djdbc.drivers=com.ddtek.jdbc.oracleservicecloud.OracleServiceCloudDriver
  ```

- **Method 2**: Set the Java property jdbc.drivers from within your Java application or applet. Include the following code fragment in your Java application or applet, and call DriverManager.getConnection(). For example:

  ```
  Properties p = System.getProperties();
  p.put ("jdbc.drivers",
  "com.ddtek.jdbc.oracleservicecloud.OracleServiceCloudDriver");
  System.setProperties (p);
  ```

- **Method 3**: Explicitly load the driver class using the standard Class.forName() method. Include the following code fragment in your application or applet and call DriverManager.getConnection(). For example:

  ```
  Class.forName("com.ddtek.jdbc.oracleservicecloud.OracleServiceCloudDriver");
  ```

Passing Connection URLs

This section shows the correct format for specifying a connection URL.

Syntax

```
jdbc:datadirect:oracleservicecloud:loginHost=host;
[property=value[;...]]
```

where:

LoginHost

is the host name of the Oracle Service Cloud site to which you want to connect, for example, mysite.custhelp.com. This value should not include an internet protocol such as http:// or https://.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon. For more information on connection properties, see Using Connection Properties on page 36.

The `install_dir/samples` directory, where `install_dir` is your product installation directory, contains a sample application that illustrates the steps of connecting to an Oracle Service Cloud instance.

This example shows how to establish a connection to an Oracle Service Cloud data source and include user ID and password information:

```
jdbc:datadirect:oracleservicecloud:loginHost=mysite.custhelp.com;
User=test;Password=secret
```

Connecting Using Data Sources

A *JDBC data source* is a Java object, specifically a DataSource object, that defines connection information required for a JDBC driver to connect to the database. Each JDBC driver vendor provides their own data source implementation for this purpose. A DataDirect Connect Series *for* JDBC 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 data source (DataSource object). The applications using the database do not need to change because they only refer to the name of the data source.

How Data Sources Are Implemented

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

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

See Data Source and Driver Classes on page 11 for data source class information.

Creating Data Sources

The following examples show how to create and use DataDirect Connect Series *for* JDBC data sources:

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

Note: To connect using a data source, the driver needs to access a JNDI data store to persist the data source information. To download the JNDI File System Service Provider, go to the Oracle Technology Network Java SE Support downloads page and make sure that the `fscontext.jar` and `providerutil.jar` files from the download are on your classpath.

You can use these examples as templates to create your own data sources. These examples are in the *install_dir*/examples/JNDI directory, where *install_dir* is your product installation directory.

Note: You must include the javax.sql.* and javax.naming.* classes to create and use DataDirect Connect Series *for* JDBC data sources. The driver provides the necessary JAR files, which contain the required classes and interfaces. If you plan to connect using a JDBC data source, the fscontext.jar and providerutil.jar files, which are shipped with the JNDI File System Service Provider, must be on your classpath. To download the JNDI File System Service Provider, go to the Oracle Technology Network Java SE Support downloads page and select a JNDI version.

Calling a Data Source in an Application

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

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

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

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

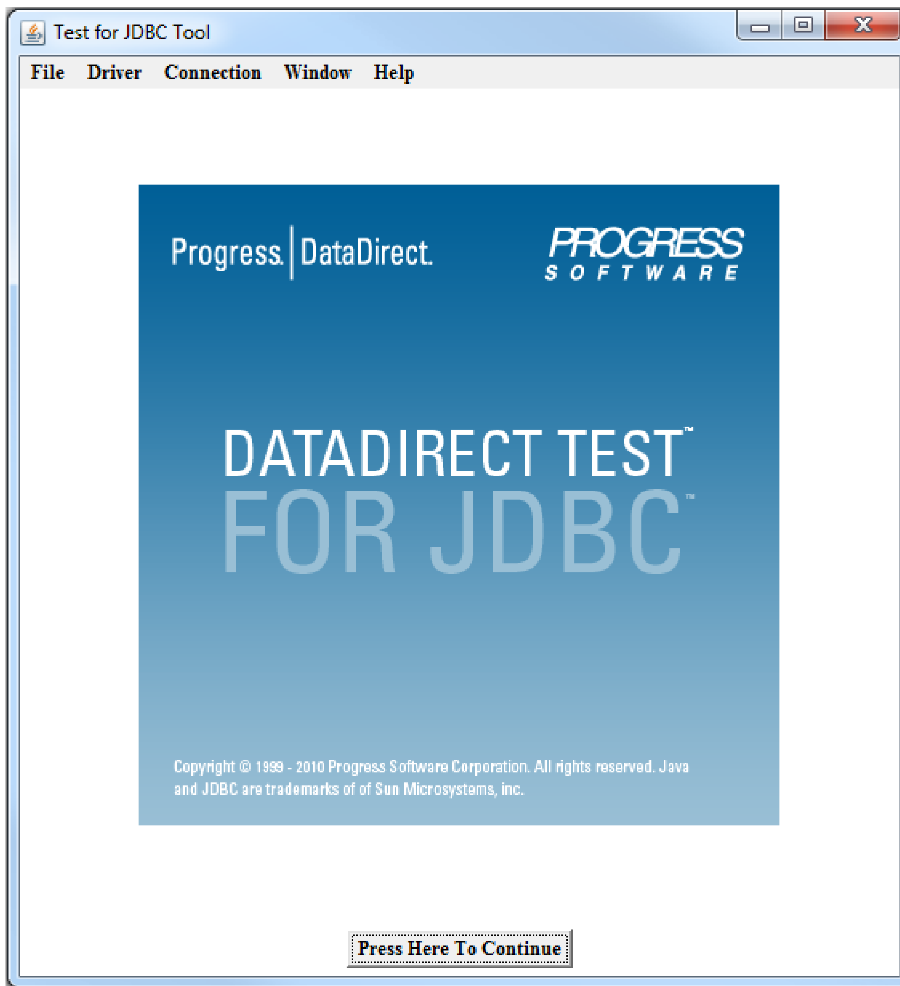
Testing the Connection

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

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

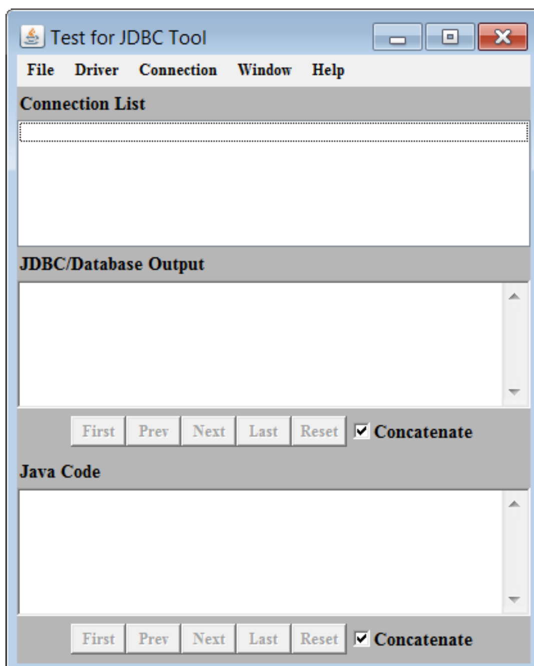
1. Navigate to your installation directory.
2. From the testforjdbc folder, run the platform-specific tool:
   - testforjdbc.bat (on Windows systems)
   - testforjdbc.sh (on UNIX and Linux systems)

   The Test for JDBC Tool window appears:



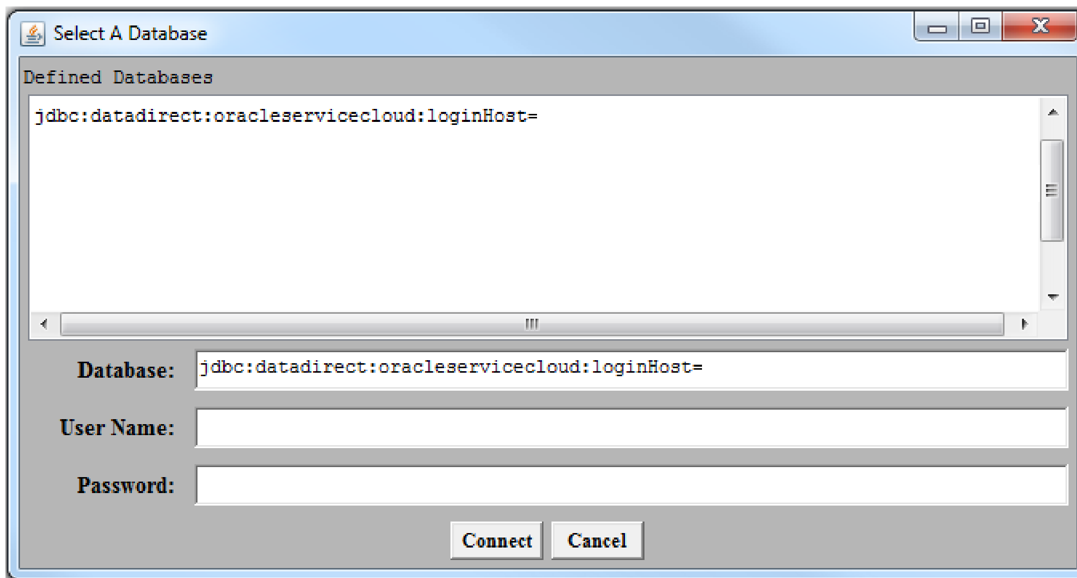
3. Click **Press Here to Continue**.

   The main dialog appears:



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

   The Select A Database dialog appears:

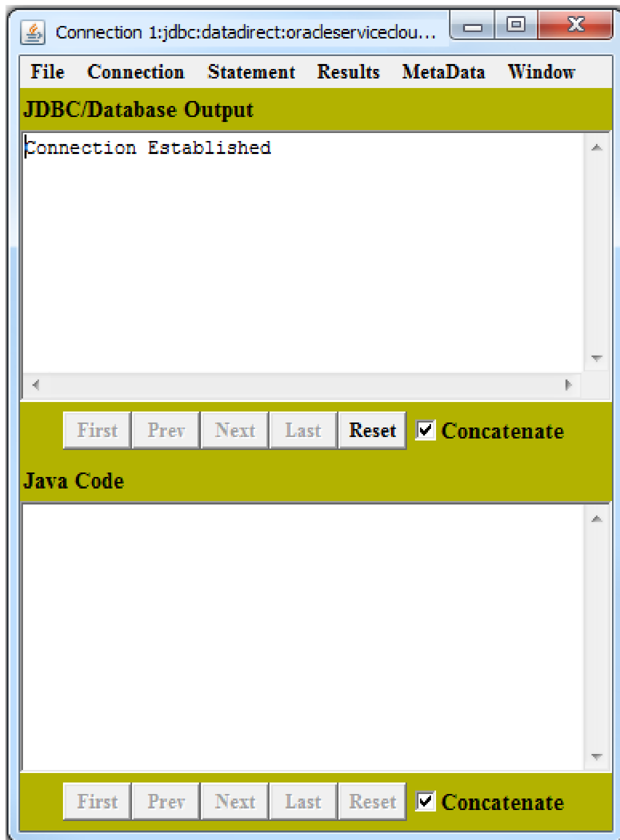
5. Select the appropriate database template from the **Defined Databases** field.
6. In the **Database** field, specify the correct LoginHost for your Oracle Service Cloud data source.

   For example:

```
jdbc:datadirect:oracleservicecloud:loginHost=mysite.custhelp.com;
```

7. If you did not specify your user name and password in the connection URL, enter this information 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 Through a Proxy Server

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

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 `808`, and provides a user ID and password for authentication:

```
java -Dhttp.proxyHost=pserver -Dhttp.proxyPort=808 -Dhttp.proxyUser=smith
-Dhttp.proxyPassword=secret -cp rightnow.jar com.acme.myapp.Main
```

Alternatively, you can use the ProxyHost, ProxyPort, ProxyUser, and ProxyPassword connection properties, but these properties are applied only for the first connection. See Using Connection Properties on page 36 and Proxy Server Properties on page 37 for details.

Using Connection Properties

You can use connection properties to customize the driver for your environment. Connection properties can be used to accomplish different tasks, such as implementing driver functionality or optimizing performance.

You can specify connection properties using either of the following methods:

- JDBC Driver Manager (see Connecting Using the JDBC Driver Manager on page 30)
- JDBC data sources (see Connecting Using Data Sources on page 31)

Connection properties take the form `property=value`. All connection property names are case-insensitive. For example, `Password` is the same as `password`.

See Connection Property Descriptions on page 61 for an alphabetical list of connection properties and their descriptions.

Required Properties

The following table summarizes connection properties which are required to connect to a database.

Table 3: Required Properties

Property	Characteristic
LoginHost on page 75	Specifies the base Oracle Service Cloud URL to use for logging in.
Password on page 78	Specifies a password that is used to connect to your Oracle Service Cloud instance.
User on page 86	Specifies the user name that is used to connect to the Oracle Service Cloud instance.

See also

- Connection Property Descriptions on page 61
- Connecting from an Application on page 29

Embedded Database Properties

The following table summarizes connection properties which are used in the configuration of embedded databases and in connecting to embedded databases.

Table 4: Embedded Database Properties

Property	Characteristic
ConfigOptions on page 64	Determines how the embedded database and the mapping of the remote data model to the relational data model is configured, customized, and updated.

Property	Characteristic
CreateDB on page 67	Determines whether the driver creates a new embedded database when establishing the connection.
DatabaseName on page 68	Specifies the file name prefix the driver uses to create or locate the set of files that define the embedded database per connection.

See also

- Connection Property Descriptions on page 61
- Mapping Objects to Tables on page 16
- Using the Database Configuration File on page 41

Proxy Server Properties

The following table summarizes proxy server connection properties.

Table 5: Proxy Server Properties

Property	Characteristic
ProxyHost on page 79	Identifies a proxy server to use for the first connection.
ProxyPassword on page 79	Specifies the password needed to connect to a proxy server for the first connection.
ProxyPort on page 80	Specifies the port number where the proxy server is listening for HTTP or HTTPS requests for the first connection.
ProxyUser on page 80	Specifies the user name needed to connect to a proxy server for the first connection.

See also

- Connection Property Descriptions on page 61
- Connecting Through a Proxy Server on page 35

Web Service Properties

The following table summarizes Web service connection properties, including those related to timeouts.

Table 6: Web Service Properties

Property	Characteristic
OracleServiceCloudDatabase on page 77	Controls how the driver instructs the Oracle Service Cloud web service to resolve queries. Oracle Service Cloud can satisfy queries against the production or the reporting database that backs the service. This property instructs the driver to indicate against which database the queries should be resolved. The default is `report`.
LoginTimeout on page 75	The amount of time, in seconds, that the driver waits for a connection to be established before timing out the connection request. The default is `0` (no timeout).
StmtCallLimit on page 84	Specifies the maximum number of Web service calls the driver can make when executing any single SQL statement or metadata query. The default behavior depends on which version of Oracle Service Cloud the site is using.
StmtCallLimitBehavior on page 85	Specifies the behavior of the driver when the maximum Web service call limit specified by the StmtCallLimit property is exceeded. The default is `errorAlways`.
WSCompressData on page 86	Specifies whether the driver compresses data it sends to or receives from the Web server. The default is `compress`.
WSFetchSize on page 87	Specifies the number of rows of data the driver attempts to fetch for each JDBC call. The default behavior depends on which version of Oracle Service Cloud the site is using.
WSRetryCount on page 88	The number of times the driver retries a timed-out Select request. Insert, Update, and Delete requests are never retried. By default, the driver does not retry timed-out requests after an initial, unsuccessful attempt.
WSTimeout on page 89	Specifies the time, in seconds, that the driver waits for a response to a Web service request. The default is `120`.

See also

- Connection Property Descriptions on page 61
- Performance Considerations on page 40
- Timeouts on page 56

Statement Pooling Properties

The following table summarizes statement pooling connection properties.

Table 7: Statement Pooling Properties

Property	Characteristic
ImportStatementPool on page 71	Specifies the path and file name of the file to be used to load the contents of the statement pool. When this property is specified, statements are imported into the statement pool from the specified file.
MaxPooledStatements on page 76	The maximum number of pooled prepared statements for this connection. If set to `0` (default), the driver's internal prepared statement pooling is not enabled. If set to a value greater than zero, the driver's internal prepared statement pooling is enabled. Enabling is useful when the driver is not running from within an application server or another application that provides its own prepared statement pooling. By default, the driver's statement pooling is not enabled.

See also

- Connection Property Descriptions on page 61
- Refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference* for further details.

Additional Properties

The following table summarizes connection properties not included in the preceding topics.

Table 8: Additional Properties

Property	Characteristic
ConvertNull on page 67	Controls how data conversions are handled for null values. If set to `1` (default), a data type check is performed for null column values.
EnableAccessRequestHeader on page 69	Determines whether the driver enables the access request header for a connection. The access request header contains a token value that, when included in a SOAP request, identifies the current connection among the competing connections to optimize its performance and quality of service.
EnablePagingWithOrderById on page 69	Specifies whether the driver can inject the Order By clause in the Select query for each JDBC call. If set to `true` (default), this property provides a stable paging mechanism for retrieving result sets that are larger than the site configured maximum number of rows.
FetchSize on page 70	Specifies the number of rows that the driver processes before returning data to the application. Smaller fetch sizes can improve the initial response time of the query. Larger fetch sizes improve overall fetch times at the cost of additional memory.
InitializationString on page 71	Specifies one or multiple SQL commands to be executed by the driver after it has established the connection to the database and has performed all initialization for the connection. If the execution of a SQL command fails, the connection attempt also fails and the driver throws an exception indicating which SQL command or commands failed.

Property	Characteristic
InsensitiveResultSetBufferSize on page 72	Determines the amount of memory that is used by the driver to cache insensitive result set data. The default is `2048` (KB of memory).
JavaDoubleToString on page 74	Determines which algorithm the driver uses when converting a double or float value to a string value. By default, the driver uses its own internal conversion algorithm, which improves performance. The default is `false`.
LogConfigFile on page 74	Specifies the file name, and optionally, the path of the properties file used to initialize driver logging.
ManagedObjects on page 76	Specifies a comma-separated list of managed objects that you want the driver to fetch metadata for. The driver may not fetch metadata for the newly added managed objects by default. Therefore, the Managed Objects connection option must be used to fetch metadata for them.
RefreshDirtyCache on page 81	Specifies whether the driver refreshes a dirty cache on the next fetch operation from the cache. A cache is marked as dirty when a row is inserted into or deleted from a cached table or a row in the cached table is updated.
TransactionMode on page 85	Specifies how the driver handles manual transactions.

See also

- Connection Property Descriptions on page 61
- Performance Considerations on page 40

Performance Considerations

You can optimize performance for your application by adjusting connection properties in the scenarios described below.

EnablePagingWithOrderByID

When enabled, this connection property provides a stable paging mechanism when retrieving large result sets by injecting the Order By clause in Select queries; however, this behavior may diminish performance. If your application does not retrieve large result sets, you should consider disabling this feature to improve performance.

FetchSize/WSFetchSize

The connection properties FetchSize and WSFetchSize can be used to adjust the trade-off between throughput and response time. In general, setting larger values for WSFetchSize and FetchSize will improve throughput, but can reduce response time.

For example, if an application attempts to fetch 100,000 rows from the remote data source and WSFetchSize is set to `500`, the driver must make 200 Web service calls to get the 100,000 rows. If, however, WSFetchSize is set to `10000` (the maximum), the driver only needs to make 10 Web service calls to retrieve 100,000 rows. Web service calls are expensive, so generally, minimizing Web service calls increases throughput. In addition, many Cloud data sources impose limits on the number of Web service calls that can be made in a given period of time. Minimizing the number of Web service calls used to fetch data also can help prevent exceeding the data source call limits.

For many applications, throughput is the primary performance measure, but for interactive applications, such as Web applications, response time (how fast the first set of data is returned) is more important than throughput. For example, suppose that you have a Web application that displays data 50 rows to a page and that, on average, you view three or four pages. Response time can be improved by setting FetchSize to `50` (the number of rows displayed on a page) and WSFetchSize to `200`. With these settings, the driver fetches all of the rows from the remote data source that you would typically view in a single Web service call and only processes the rows needed to display the first page.

InsensitiveResultSetBufferSize

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

Using the Database Configuration File

You can configure an embedded database and data mapping using a database configuration file in XML format (see Mapping Objects to Tables on page 16 for an explanation of embedded databases). Some values can be set either in the database configuration file or using the ConfigOptions connection property (see ConfigOptions on page 64). Some values can be set using only a database configuration file.

The name of the database configuration file has the format:

```
databasename.config
```

where:

`databasename`

is the name of the database to be configured.

For example, assuming your database is named *mydb* or you have a database configuration file named *mydb.config*, when the driver establishes a connection, it performs the following steps:

1. Checks to see if an embedded database named *mydb* exists (or a database using the default *`databasename`* if one is not specified). If *mydb* exists, the driver connects to the remote data source using the *mydb* database.
2. If *mydb* does not exist and the driver is configured to create a database, the driver looks for a database configuration file named *mydb.config*. If the database configuration file exists, the driver creates the database and mapping using the properties specified in the database configuration file.
3. If *mydb.config* does not exist, the driver generates a database configuration file with default settings and uses those settings to create the database and its mapping.

Example: Database Configuration File

```
<?xml version='1.0' encoding='UTF-8'?>
<Database xmlns="http//datadirect.com/cloud/config" version="1">
```

```
    <User name="CONNECT2" defaultSchema="RIGHTNOW">
      <UseSchema name="RIGHTNOW"/>
      <UseSchema name="PUBLIC"/>
    </User>
    <Schema name="RIGHTNOW" type="RightNow">
      <ConfigOptions>uppercaseidentifiers=true;auditcolumns=none;
       mapsystemcolumnnames=1;</ConfigOptions>
      <SessionOptions>loginHost=mysite.custhelp.com;
       User=test;Password=secret</SessionOptions>
    </Schema>
    <Schema name="PUBLIC" type="local">
    </Schema>
</Database>
```

The following sections describe the elements in the database configuration file.

<Database>

Child Elements

<User>, <Schema>

Description

This element is the root element of the database configuration file. It contains all the elements that define the database configuration. Only a single <Database> element can exist.

<User>

Parent Element

<Database>

Child Element

<UseSchema>

Description

This element specifies the user ID used by the driver. At least one <User> element must exist.

Attributes

name [required]: The user name is a string with a maximum length of 128 characters. The default is `name=`*`userid`*, where *`userid`* is the user ID used by the driver.

defaultSchema: The name of the schema used for unqualified table and column identifiers. If this attribute is not specified, the schema specified in the first <useSchema> child element is used as the default schema. The default value is `RIGHTNOW`.

<UseSchema>

Parent Element

<User>

Child Element

None

Description

This element specifies a schema that is visible to the user of the element. A schema contains the mapping between the remote data model and the relational tables the driver exposes. Multiple schemas can be associated with a user. At least one <UseSchema> element must exist.

A basic <User> definition typically has two UseSchema elements: one that specifies the mapping to the remote data source and one for the local schema.

Attributes

name [required]: The name of the schema to associate with the user. The schema name is a string with a maximum length of 128 characters. The default value is `RIGHTNOW` for the remote data source and `PUBLIC` for the local database.

<Schema>

Parent Element

<Database>

Child Elements

<ConfigOptions>, <SessionOptions>

Description

This element defines the schema that contains the mapping for a remote data source. The database configuration file can contain multiple schema definitions, but it must contain at least one. Each schema definition defines the type of data source to which the schema maps, the information to connect to the remote database (except password), and the information needed to configure the remote data model to relational table mapping. At least one <Schema> element must exist.

Attributes

name [required]: The name of the schema that defines the data model to relational mapping. This attribute can be any valid identifier name. The default value is `RIGHTNOW` for the remote data source and `PUBLIC` for the local database.

type [required]: The type of remote data source for which the schema defines mapping. The value must be `RightNow` for the remote data source and `local` for the local database.

<ConfigOptions>

Parent Element

<Schema>

Child Element

None

Description

This element is a string that specifies the configuration options used to define how the remote data source data model is mapped to relational tables. The ConfigOption string has the same keys, values, and syntax as the ConfigOptions connection option (see ConfigOptions on page 64) except that the enclosing parentheses are not required. The default is an empty string.

Attributes

None

<SessionOptions>

Parent Element

<Schema>

Child Element

None

Description

This element is a string of key=value pairs that specifies the information needed to connect to the remote data source. For example:

```
loginHost=mysite.custhelp.com
User=test;Password=secret
```

Attributes

None

Data Encryption

All communication between the driver and Oracle Service Cloud, 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 36 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.

Views and Remote/Local Tables

You can create views with the Create View statement. A view is like a named query. The view can refer to any combination of remote and local tables as well as other views.

You can create a remote or local table using the Create Table statement. A remote table is an Oracle Service Cloud object and is exposed in the ORACLESERVICECLOUD schema. A local table is maintained by the driver and is local to the machine on which the driver is running. A local table is exposed in the PUBLIC schema.

See Supported SQL Statements and Extensions on page 91 for details on the Create View and Create Table statements and other SQL statements supported by the driver.

Client-Side Caches

The driver can implement a client-side data cache for improved performance. Data is cached from the remote data source to the local machine on which the driver is located.

The driver caches data on a per-table basis, as opposed to caching the result of a particular query. Caching data on a table level allows the caches to be queried, filtered, and sorted in other queries. Once a cache is created, its use is transparent to the application. For example, if a cache is created on the Account table, then all subsequent queries that reference Account access the Account cache. Disabling or dropping the cache gives references to the Account table access to the remote data again. Because the use of the cache is transparent, no changes to the application are required to take advantage of the cache.

You must specifically create a cache before it can be populated; caches are not created automatically. After you have created a cache on a table, the cache will be populated as a result of the next operation on the table. For example, after creating a cache on Account, data is returned from the data source and stored locally in the cache when you first execute the following statement:

```
SELECT ROWID, SYS_LOOKUPNAME FROM Account
```

Any subsequent queries against the Account table return data from the cache, which reduces response time. SQL queries can access both cached data and remote data (data stored in the data source that has not been assigned to a cache) in the same statement.

To create, modify, refresh, or delete client-side data caches, use the following SQL statement extensions:

- Create Cache
- Alter Cache
- Refresh Cache
- Drop Cache

See the following sections for overviews of each extension. See Supported SQL Statements and Extensions on page 91 for descriptions of the syntax of these extensions.

Creating a Cache

You create a cache using the Create Cache statement (see Create Cache (EXT) on page 100). A cache can only be created from a single table. When creating a cache, you specify the name of the table to cache and can optionally specify a filter for the table. The filter determines whether the cache holds all of the data in the remote table or a subset of the data that matches the filter. You can also specify attributes for the Create Cache statement that determine:

- Whether the cache data is held on disk or in memory
- How often the cache data is refreshed
- Whether the cache is initially enabled
- Whether the driver checks to see if a refresh is needed at connect time

Modifying a Cache Definition

Once a cache has been created, you can modify the definition of the cache with the Alter Cache statement (see Alter Cache (EXT) on page 92). Only the attributes of the cache can be modified through the Alter Cache statement.

Warning: Changing the attributes of a cache may cause the current data in the cache to be discarded and refetched from the remote data source.

Disabling and Enabling a Cache

When a cache is defined on a table, all fetch operations performed on that table access the cache, essentially hiding the remote table from the application. At times, you may want an application to query the remote data instead of the cached data. For example, assume that a cache was created on Account with a filter set to cache accounts that have had activity in the past year. You may want to run a query to get information about an account that has not been active for two years. One alternative would be to drop the Account cache, run the query, and then recreate the cache on Account, but this can be problematic. First, you must recreate the cache and make sure it had the same attributes as before. Second, the data in the cache is discarded and needs to be refetched when the cache is recreated. Depending on the amount of cached data, this could take a significant amount of time. To address this type of issue, the driver can temporarily disable a cache. When a cache is disabled, its definition and data are maintained. Any queries that reference a table with a disabled cache access the remote table. When you want to access cached data again, the cache can be enabled.

Refreshing Cache Data

To prevent the data in a cache from becoming out of date, the driver must periodically refresh the cache data with data from the remote data source. When the driver refreshes the cache, it discards all of the data from the cache and repopulates it with data from the remote data source. You or the application can refresh the cache manually or the driver can refresh the cache automatically.

You can refresh a cache manually at any time by using the Refresh Cache statement (see Refresh Cache (EXT) on page 118 for additional information).

The driver can refresh a cache automatically by setting the refresh interval attribute when creating a cache (see Refresh Interval Clause on page 102 for additional information). During each cache query, the driver checks to see whether the time elapsed since the last refresh exceeds the refresh interval for the cache. If it has, the driver refreshes the cache before satisfying the query.

Dropping a Cache

You can drop an existing cache using the Drop Cache statement (see Drop Cache (EXT) on page 115 for additional information). When a cache is dropped, all of the data in that cache is discarded.

Cache MetaData

The driver maintains information about the caches that have been created. The driver provides a system table to expose the cache information, the SYSTEM_CACHES table.

The SYSTEM_CACHES table exists in the INFORMATION_SCHEMA schema. See Catalog Tables on page 47 for a complete description of the contents of the system tables.

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_CACHES Catalog Table

The SYSTEM_CACHES catalog table stores the definitions of the caches created on remote tables. The data in the SYSTEM_CACHES table provides the name, type (single table or relational), status, and other information for each defined cache. The table name returned for a remote relational cache is the name of the primary table of the relational cache; however, its type is REMOTE RELATIONAL. You can query SYSTEM_CACHES to determine the caches currently defined by the driver. The values in the SYSTEM_CACHES table are read-only. The referenced tables of a relational cache can be determined by querying the SYSTEM_CACHE_REFERENCES catalog table (see SYSTEM_CACHE_REFERENCES Catalog Table on page 49).

The following table describes the columns of the SYSTEM_CACHES table, which is sorted on the following columns: CACHE_TYPE, TABLE_SCHEMA, and TABLE_NAME.

Table 9: SYSTEM_CACHES Catalog Table

Column Name	Data Type	Description
TABLE_CAT	VARCHAR (128),NULLABLE	The catalog that contains the remote table on which the cache is defined. It is NULL for this driver.
TABLE_SCHEM	VARCHAR (128),NULLABLE	The schema that contains the remote table on which the cache is defined.

Column Name	Data Type	Description
TABLE_NAME	VARCHAR (128),NOT NULL	The name of the remote table on which the cache is defined.
CACHE_TYPE	VARCHAR (20),NOT NULL	The type cache, which can be either `REMOTE TABLE` or `REMOTE RELATIONAL`.
REFRESH_INTERVAL	INTEGER,NOT NULL	The refresh interval (in minutes).
INITIAL_CHECK	VARCHAR (20),NOT NULL	The value that defines when the initial refresh check is performed: `ONFIRSTCONNECT` or `FIRSTUSE`.
PERSIST	VARCHAR (20),NOT NULL	The value that defines whether the data in the cache is persisted past the lifetime of the connection: `TEMPORARY`, `MEMORY`, or `DISK`.
ENABLED	BOOLEAN,NOT NULL	The value that defines whether the cache is enabled for use with SQL statements: `TRUE` or `FALSE`.
CALL_LIMIT	INTEGER,NOT NULL	The maximum number of Web service calls that can be made when refreshing the cache. The value `0` indicates no call limit.
REFRESH_MODE	INTEGER,NOT NULL	For internal use only.
FILTER	VARCHAR (128),NULLABLE	The Where clause used to filter the rows that are cached.
LAST_REFRESH	DATETIME, NULLABLE	The time, in Coordinated Universal Time (UTC), the cache was last refreshed.
STATUS	VARCHAR (30)	The Cache status. Valid values are: • `New`: The cache has been created, but the data has not been populated. • `Initialized`: The cache has been created and the data has been populated. • `Load aborted`: The cache has been created, but the last attempt to populate the data failed. The cache is still valid. The next access attempts to populate the data again. • `Invalid`: The cache is invalid. The second attempt to populate the data failed. • `Dirty`: An insert or update operation has been performed on the cache and the cache has not been refreshed.

SYSTEM_CACHE_REFERENCES Catalog Table

The referenced tables in a relational cache can be determined by querying the SYSTEM_CACHE_REFERENCES system table. This table contains the names of the referenced tables as well as the name of the primary table with which they are associated.

The following table defines the columns of the SYSTEM_CACHE_REFERENCES table, which is sorted on the following columns: TABLE_SCHEMA, TABLE_NAME, and REF_TABLE_NAME.

Table 10: SYSTEM_CACHE_REFERENCES Catalog Table

Column Name	Data Type	Description
PRIMARY_TABLE_CAT	VARCHAR (128), NULLABLE	The catalog that contains the primary table of the relational cache. It is NULL for this driver.
PRIMARY_TABLE_SCHEM	VARCHAR (128), NULLABLE	The schema that contains the primary table of the relational cache.
PRIMARY_TABLE_NAME	VARCHAR (128), NOT NULL	The primary table of the relational cache.
REF_TABLE_NAME	VARCHAR (128), NOT NULL	The name of the referenced table.
RELATIONSHIP_NAME	VARCHAR (128), NOT NULL	The name of the foreign key relationship used to relate this table to the primary table or one of the other tables in the relational cache.

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 RightNow.

Column Name	Data Type	Description
INSTANCE	VARCHAR(128)	The remote session instance name or null if the remote data source does not have multiple instances. The Oracle Service Cloud value for INSTANCE has the following form: *Organization_Name* where *Organization_Name* is the organization name of the Oracle Service Cloud instance to which the connection is established.
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 Service Cloud.
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.
CONNECTED	DATETIME, NOT NULL	The date and time the session was established.
USERNAME	VARCHAR (128), NOT NULL	The name of the embedded database that the session is using.
IS_ADMIN	BOOLEAN	For internal use only.
AUTOCOMMIT	BOOLEAN, NOT NULL	For future use.
READONLY	BOOLEAN, NOT NULL	`True` if the connection is in read-only mode. The READONLY status is based on whether the connection has been explicitly set to read-only mode by the ReadOnly connection option or the Connection.setReadOnly() method.
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.

Reports

The driver exposes reports defined on an Oracle Service Cloud instance as stored procedures. An application can obtain a list of the reports defined on an Oracle Service Cloud instance by calling the DatabaseMetaData.getProcedures method. The names of the reports that can be invoked through the driver are listed in the PROCEDURE_NAME name column of the getProcedures() results.

The driver incorporates report name and unique report ID into the procedure name reported by getProcedures(). The driver creates the reported procedure name by appending the report ID to the report name using an underscore (_) to join them. Additionally, any spaces in the report name are replaced with an underscore character. The report name is also prepended with `RN` if the report starts with a digit. Like all identifier name metadata returned by the driver, the procedure name is uppercase. For example, a report named Contact Lookups would be modified as follows:

```
CONTACT_LOOKUPS_14001
```

An application invokes a report using the standard Call escape syntax:

```
CALL ReportName (FilterName operator FilterValue;FilterName operator
FilterValue...)
```

And JDBC mechanisms are used for calling a stored procedure that returns a result set. The following example shows one way to invoke the Contact Lookups report:

```
String sql = "{call CONTACT_LOOKUPS_14001('[Contact ID] < 10')}";
CallableStatement callStmt = con.prepareCall(sql);
boolean isResultSet = callStmt.execute();
if (isResultSet)
{ resultSet = callStmt.getResultSet(); // process the resultset }
```

In this example, the standard Call escape syntax is employed for the Opportunities report:

```
CALL Opportunities_13029 ([Opportunity ID] > 10;[Organization ID] < 20;
      Name [NOT LIKE] [Imaging Equipment])
```

The following rules for filters apply when writing applications which use reports:

- Filters are separated by semicolons.
- The FilterName must be specified to the left of the operator; the FilterValue must be specified to the right of the operator.
- FilterName or FilterValue must be delimited by brackets ([]) if it contains the following special characters:

  ```
  ';', '.', '?', '/', '(', ')', '{', '}', '[', ']', ',', whitespace,
  operators,  single quote, double quote
  ```

- If a singlequote (') is part of the Filter Name or Filter Value, it must be doubled.
- If single bracket (]) is part of the Filter Name or Filter Value, it must be doubled.
- The following multiple-word operators must be delimited by []:

  ```
  NOT LIKE, IN LIST, NOT IN LIST, NOT REGEX
  ```

- *null* is considered a NULL value. For example, `Name = null`. In contrast, if 'null' is the string literal, it must be delimited by [].
- An empty string is represented as []. For example, `Name = []`.

The following table shows the operators available for column comparison.

Table 13: Operators for Column Comparison

Name	Operator
Equal	=

Name	Operator
Not Equal	<>
Less Than	<
Less Than or Equal To	<=
Greater Than	>
Greater Than or Equal To	>=
Is Like	LIKE
Is Not Like	NOT LIKE
Is Between	RANGE
Is In List	IN LIST
Is Not In List	NOT IN LIST
Not Equal To or NULL	NLIKE_OR_NULL
Not Like or NULL	NLIKE_OR_NULL
Regular Expression	REGEX
Not Regular Expression	NOT REGEX

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 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.

IP Addresses

The driver supports Internet Protocol (IP) addresses in IPv4 and IPv6 format when connecting through a proxy server. See Connecting Through a Proxy Server on page 35 for details.

Auto-Generated Keys Support

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.

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.

Rowset Support

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

- CachedRowSets
- FilteredRowSets
- WebRowSets
- JoinRowSets
- JDBCRowSets

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

ResultSet MetaData Support

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.

Parameter Metadata Support

The driver supports returning parameter metadata for all types of SQL statements and stored procedure arguments.

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.

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][OracleServiceCloud 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][OracleServiceCloud JDBC Driver][OracleServiceCloud]Invalid Object Name.
```

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

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.

See Using Connection Properties on page 36 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 Objects (LOBs)

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

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

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

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

"Poor Performing Query" Error Message

Oracle Service Cloud returns a "Poor performing query" error when executing certain queries with the driver. This typically occurs when executing a query that requires the Oracle Service Cloud service to perform join operations on tables containing a large number of rows. You can work around this issue by doing the following:

1. Configure the following connection properties to the values provided.

Option	Value
ConfigOptions **Note:** In addition to specifying the following recommended value, include any other key-value pairs that you currently specify for ConfigOptions. These key-value pairs should be specified within parentheses in a comma-separated list.	`(NamedIDBehavior=2)`
CreateDB	`forceNew`
EnablePagingWithOrderByID	`1` (Enabled) **Note:** This is the default value for this option.

2. Connect to your Oracle Service Cloud instance. At connection, the driver deletes the current local map files and creates new files using the updated config option settings.
3. Change the value of CreateDB to your preferred setting before your next connection. We recommend specifying a value of `2` (NotExist).
4. Test your settings by executing a query that previously returned the error. Based on the results, choose one of the following:
   - If you receive the "Poor Performing Query" error, proceed to the next step.
   - If your query is successfully returned, skip to the end of the procedure.
5. Tune the driver's fetching behavior. Using the WSFetchSize option, decrease the number of rows the driver attempts to fetch until the "Poor Performing Query" error is no longer returned. For optimal performance, configure WSFetchSize so that the driver returns the maximum number of rows possible without returning the error. For example, if you receive the error fetching 500 rows, but not 499 rows, set WSFetchSize=499 for maximum throughput.

   Be aware that decreasing the fetch size can increase the number of web service calls the driver makes to the Oracle Service Cloud service. If your query exceeds the statement call limit, increase the number of rows fetched using WSFetchSize or increase the statement call limit using the StatementCallLimit option to work around this issue.

   Note: You will need to disconnect and reconnect for changes to connection option values to have effect.

   Note: In the default setting, `WSFetchSize=0`, the driver uses the maximum page size allowed by the database to determine the number of rows it attempts to fetch for Oracle Service Cloud versions August 2014 or later. For versions earlier than August 2014, the driver attempts to fetch a maximum of 10,000 rows.

This configuration will allow you to avoid the "Poor performing query" error in subsequent connections. If you begin receiving the error again, tune the WSFetchSize connection option according to the guidelines provided in step 4 on page 58.

See also

WSFetchSize on page 87
ConfigOptions on page 64
EnablePagingWithOrderById on page 69

4

Connection Property Descriptions

The following table is an alphabetical list of connection properties with their default values. Each property is hyperlinked to its full description.

Note: The data type listed in each description is the Java data type used for the property value in a JDBC data source.

Table 14: Connection Properties with Defaults

Property	Default
ConfigOptions on page 64	AuditColumns=all; MapCustomNamedIDFields=1; MapSystemColumnNames=0; UppercaseIdentifiers=true
ConvertNull on page 67	1 (data type check is performed if column value is null)
CreateDB on page 67	notExist
DatabaseName on page 68	user ID specified for the connection
EnableAccessRequestHeader on page 69	true
EnablePagingWithOrderById on page 69	true
FetchSize on page 70	100 (rows)

Property	Default
ImportStatementPool on page 71	empty string
InitializationString on page 71	none
InsensitiveResultSetBufferSize on page 72	2048 (KB of memory)
InterfaceName on page 73	empty string
JavaDoubleToString on page 74	false
LogConfigFile on page 74	ddlogging.properties
LoginHost on page 75	none
LoginTimeout on page 75	0 (no timeout)
ManagedObjects on page 76	none
MaxPooledStatements on page 76	0 (driver's internal prepared statement pooling is not enabled)
OracleServiceCloudDatabase on page 77	report
Password on page 78	none
ProxyHost on page 79	empty string
ProxyPassword on page 79	empty string
ProxyPort on page 80	0
ProxyUser on page 80	empty string
RefreshDirtyCache on page 81	1
SpyAttributes on page 82	none
StmtCallLimit on page 84	-1
StmtCallLimitBehavior on page 85	errorAlways
TransactionMode on page 85	noTransactions
User on page 86	none
WSCompressData on page 86	compress
WSFetchSize on page 87	0

Property	Default
WSRetryCount on page 88	0 (no retries for timed-out requests)
WSTimeout on page 89	120 (seconds)

For details, see the following topics:

- ConfigOptions
- ConvertNull
- CreateDB
- DatabaseName
- EnableAccessRequestHeader
- EnablePagingWithOrderById
- FetchSize
- ImportStatementPool
- InitializationString
- InsensitiveResultSetBufferSize
- InterfaceName
- JavaDoubleToString
- LogConfigFile
- LoginHost
- LoginTimeout
- ManagedObjects
- MaxPooledStatements
- OracleServiceCloudDatabase
- Password
- ProxyHost
- ProxyPassword
- ProxyPort
- ProxyUser
- RefreshDirtyCache
- RegisterStatementPoolMonitorMBean
- SpyAttributes
- StmtCallLimit

- StmtCallLimitBehavior
- TransactionMode
- User
- WSCompressData
- WSFetchSize
- WSRetryCount
- WSTimeout

ConfigOptions

Purpose

Determines how the embedded database and the mapping of the remote data model to the relational data model is configured, customized, and updated.

Note: 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. The preferred method for setting the configuration options for a particular user is through the database configuration file. See Using the Database Configuration File on page 41 for details.

Valid Values

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

where:

`key`

is one of the following values: `AuditColumns`, `MapCustomNamedIDFields`, `MapSystemColumnNames`, `NamedIDBehavior`, or `UppercaseIdentifiers`.

The value is a set of key value pairs separated by a semicolon (;). The value must be enclosed in parentheses. For example:

```
(AuditColumns=none;UppercaseIdentifiers=false)
```

AuditColumns: Determines whether the driver includes audit columns when mapping the remote data model to the relational data model.

The driver supports the following audit columns for standard objects and custom objects:

- CreatedTime
- UpdatedTime
- CreatedByAccountId
- UpdatedByAccountId

Note: Although the CreatedTime and UpdatedTime fields on custom objects map to the relational view, you cannot query against them unless you enable the fields within the Object Designer in the Oracle Service Cloud admin tool.

Valid values for AuditColumns are:

Value	Description
`all`	The driver includes audit columns in all objects in the relational map.
`standard`	The driver includes audit columns only for standard objects in the relational map.
`custom`	The driver includes audit columns only for custom objects in the relational map.
`none`	The driver removes audit columns from all objects in the relational map.

The default value for AuditColumns is `all`.

MapCustomNamedIDFields: Codetermines whether the driver exposes the NameID fields on custom objects and Name ID fields that are custom attributes on standard objects. When these fields are exposed, the driver uses the DESCRIBE command to discover how they are exposed. This process can significantly affect the time it takes the driver to build the relational map the first time it connects to Oracle Service Cloud. Valid values for MapCustomNamedIDFields are:

Value	Description
`0`	The driver does not expose the NameID fields on custom objects and Name ID fields that are custom fields on standard objects.
`1`	The driver exposes the NameID fields on custom objects and Name ID fields that are custom fields on standard objects.

The default value for MapCustomNamedIDFields is `1`.

MapSystemColumnNames: Determines how the driver maps Oracle Service Cloud system columns. Valid values for MapSystemColumnNames are:

Value	Description
`0`	The driver does not change the names of any Oracle Service Cloud system columns.

`1`	The driver changes the names of Oracle Service Cloud system columns as follows:
	`Field Name              MappedName`
	`----------              ----------`
	`Id                      ROWID`
	`LookUpName              SYS_LOOKUPNAME`
	`CreatedTime             SYS_CREATEDTIME`
	`UpdatedTime             SYS_UPDATEDTIME`
	`CreatedByAccountId      SYS_CREATEDBYACCOUNT_ID`
	`UpdatedByAccountId      SYS_UPDATEDBYACCOUNT_ID`

The default value for MapSystemColumnNames is `0`.

NamedIDBehavior: Determines whether the Name attribute of NamedID fields is exposed in the relational map.

Valid values for NamedIDBehavior are:

Value	Description
1	The driver exposes the Name attribute of the NamedID fields in the relational model. Specify this setting if your application needs to access the Name strings associated with NamedID columns.
2	The driver does not expose the Name attribute of the NamedID fields in the relational model. This setting can be used to avoid "poor performing query" errors received when executing SELECT * queries against tables containing a large number of rows.

Note: If you are receiving "poor performing query" errors, you can work around this issue by setting `NamedIDBehavior=2`. This reduces the size of the result set, allowing the Oracle Service Cloud service to successfully return the query. For this change to have effect, you will need recreate the local mapping files by setting `CreateDB=1`; then, connecting to your Oracle Service Cloud instance. After creating your new local mapping files, you should reconfigure CreateDB to your preferred setting. We recommend a setting of `CreateDB=2`. For more information, see "Poor Performing Query errors."

The default value for NamedIDBehavior is `1`.

UppercaseIdentifiers: Defines how the driver maps identifiers. By default, the driver maps all identifier names to uppercase.

Note: Do not change the value of UppercaseIdentifiers unless the data source you are connecting to has objects with names that differ only by case.

Valid values for UppercaseIdentifiers are:

Value	Description
true	The driver maps identifiers to uppercase.
false	If set to false, the driver maps identifiers in mixed case as they exist in the object model. If mixed case identifiers are used, then they must be quoted in SQL statements because the identifier case in the driver's SQL engine is SQL_IC_UPPER.

The default value for UppercaseIdentifiers is `true`.

Defaults

```
AuditColumns=all;
MapCustomNamedIDFields=1;
MapSystemColumnNames=0;
NamedIDBehavior=1;
UppercaseIdentifiers=true
```

Data Type

String

ConvertNull

Purpose

Controls how data conversions are handled for null values.

Valid Values

`0` | `1`

Behavior

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

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

Default

`1`

Data Type

Int

CreateDB

Purpose

Determines whether the driver creates a new embedded database when establishing the connection.

Valid Values

`forceNew` | `notExist` | `no`

Behavior

If set to `forceNew`, the driver deletes the current embedded database specified by DatabaseName and creates a new one at the same location.

---

warning: This causes all views, data caches, and map customizations defined in the current database to be lost.

If set to `notExist`, the driver uses the current embedded database specified by DatabaseName. If one does not exist, the driver creates one.

If set to `no`, the driver uses the current embedded database specified by DatabaseName. If one does not exist, the connection fails.

Default

`notExist`

Data Type

String

DatabaseName

Purpose

Specifies the file name prefix the driver uses to create or locate the set of files that define the embedded database per connection. See Mapping Objects to Tables on page 16 for an explanation of embedded databases.

Valid Values

`prefix` | *`path+prefix`*

where:

`prefix`

is the file name prefix for the embedded database. For example, if DatabaseName is set to a value of `JohnQPublic`, the embedded database files that are created or loaded have the form `johnqpublic.`xxx.

`path+prefix`

is a relative or absolute path appended to the file name prefix. The path defines the directory the driver uses to store the newly created database files or locate the existing database files. For example, if DatabaseName is set to a value of `C:\data\db\johnqpublic`, the driver either creates or looks for the database johnqpublic.xxx in the directory `C:\data\db`. If you do not specify a path, the current working directory is used.

Notes

- The driver parses the UserId value and removes all non-alphanumeric characters. For example, if the User ID is `John.Q.Public`, the value used for DatabaseName is `JohnQPublic`.

Default

user ID specified for the connection

Data Type

String

Alias

Database property. If both the Database and DatabaseName properties are specified in a connection URL, the last property that is positioned in the connection URL takes precedence.

EnableAccessRequestHeader

Purpose

Determines whether the driver enables the access request header for a connection. The access request header contains a token value that, when included in a SOAP request, identifies the current connection among the competing connections to optimize its performance and quality of service.

Valid Values

`true | false`

Behavior

If set to `true`, the driver enables the access request header.

If set to `false`, the driver does not enable the access request header.

Default

`true`

Data Type

String

EnablePagingWithOrderById

Purpose

Specifies whether the driver can inject the Order By clause in the Select query for each JDBC call. When enabled, this property provides a stable paging mechanism for retrieving result sets that are larger than the site configured maximum number of rows.

Note: If your application does not retrieve large result sets, consider disabling this feature because adding the Order By clause can have a negative performance impact on queries.

Valid Values

`true | false`

Behavior

If set to `true`, the driver can inject the Order By clause in the Select query.

If set to `false`, the driver cannot inject the Order By clause in the Select query.

Default

`true`

Data Type

Boolean

FetchSize

Purpose

Specifies the number of rows that the driver processes before returning data to the application. Smaller fetch sizes can improve the initial response time of the query. Larger fetch sizes improve overall fetch times at the cost of additional memory.

Valid Values

`0` | *x*

where:

x

is a positive integer.

Behavior

If set to `0`, the driver fetches and processes all of the rows of the result before returning control to the application.

If set to *x*, the driver fetches and processes the specified number of rows before returning data to the application.

Notes

FetchSize is related to, but different than, WSFetchSize. WSFetchSize specifies the number of rows of raw data that the driver fetches from the remote data source, while FetchSize specifies how many of these raw data rows the driver processes before returning data to the application. Processing the data includes converting from the remote data source data type to the driver SQL data type used by the application. If FetchSize is greater than WSFetchSize, the driver makes multiple round trips to the data source to get the requested number of rows before returning control to the application.

Default

`100` (rows)

Data Type

Int

See also

- Performance Considerations on page 40
- WSFetchSize on page 87

ImportStatementPool

Purpose

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

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

Valid Values

string

where:

string

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

Default

empty string

Data Type

String

See also

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

InitializationString

Purpose

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

Valid Values

command [[; *command*]...]

where:

command

is a SQL command.

Notes

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

Example

When mapping custom objects to a relational model, the driver exposes the different namespaces for the packages as different schemas within the relational model. For example, if you create an object called `bar` in the `foo` namespace, the driver exposes a schema called `FOO` that contains a table called `BAR`. To access this table using the single part naming convention, the application can change the current schema by passing the SQL command `ALTER SESSION SET CURRENT_SCHEMA = FOO` in the InitializationString property. For example:

```
jdbc:datadirect:oracleservicecloud:loginHost=mysite.custhelp.com;
User=test;Password=secret;
InitializationString=(ALTER SESSION SET CURRENT_SCHEMA = FOO)
```

Default

none

Data Type

String

InsensitiveResultSetBufferSize

Purpose

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

Valid Values

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

where:

x

is a positive integer that represents the amount of memory.

Behavior

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

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

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

Default

`2048`

Data Type

Int

InterfaceName

Important: The InterfaceName connection property has been deprecated. The driver will continue to provide backward compatibility for it, but the current and newer versions of the driver will not use it for connections.

Purpose

Specifies the name of the Oracle Service Cloud interface to which the driver will connect.

Valid Values

string

where:

string

is the name of the Oracle Service Cloud interface to which you want to connect.

Notes

The driver builds the SOAP endpoint using the value provided for interfaceName in conjunction with the value provided for the LoginHost connection property.

Default

None. However, on successful connection, it will be updated with the LoginHost value.

Data Type

String

JavaDoubleToString

Purpose

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

Valid Values

`true | false`

Behavior

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

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

Default

`false`

Data Type

Boolean

LogConfigFile

Purpose

Specifies the file name, and optionally, the path of the properties file used to initialize driver logging.

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.

Default

`ddlogging.properties`

Data Type

String

See also

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

LoginHost

Purpose

The host name for the Oracle Service Cloud site to which the driver will connect.

Valid Values

string

where:

string

is the host name of the Oracle Service Cloud site to which you want to connect, for example, mysite.custhelp.com. This value should not include an internet protocol such as http:// or https://.

Default

empty string

Data Type

String

LoginTimeout

Purpose

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

Valid Values

`0` | *x*

where:

x

is a positive integer that represents a number of seconds.

Behavior

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

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

Default

`0`

Data Type

Int

ManagedObjects

Purpose

Specifies a comma-separated list of managed objects that you want the driver to fetch metadata for. For example, if you specify `ManagedObjects=object1,object2,object3`, the driver fetches metadata for objects 1, 2, and 3.

The driver may not fetch metadata for the newly added managed objects by default. Therefore, the Managed Objects connection option must be used to fetch metadata for them.

Valid Values

`string`

where:

`string`

is a comma-separated list of managed objects.

Default

None

Data Type

String

MaxPooledStatements

Purpose

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

Valid Values

`0` | *x*

where:

x

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

Behavior

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

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

Notes

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

Example

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

Default

`0`

Data Type

Int

OracleServiceCloudDatabase

Purpose

Controls how the driver instructs the Oracle Service Cloud Web service to resolve queries. Oracle Service Cloud can satisfy queries against the production or the reporting database that backs the service. This property instructs the driver to indicate against which database the queries should be resolved.

Valid Values

`none` | `report` | `operational`

Behavior

If set to `none`, the driver sends the base ROQL command directly. This results in the default database behavior.

If set to `report`, the driver prepends a "USE REPORT; " statement to the base ROQL command. This results in the reporting database being used for subsequent queries. The report database is a copy of the operational database. The report database is only up to date once the data from the site has propagated to it. When your queries do not depend on using live data, you should set the OracleServiceCloudDatabase property to `report` to reduce the load a system is under.

If set to `operational`, the driver prepends a "USE OPERATIONAL; " statement to the base ROQL command. This results in the production database being used for subsequent queries. The operational database contains live, up-to-date data. When your queries require live data, you should set the OracleServiceCloudDatabase property to `operational`. However, there is an increased likelihood that queries made to the operational database may fail.

Notes

Depending on the load a system is under, some queries made to the Web service can fail, independent of the driver. Queries made to the report database are generally less likely to fail than queries made to the operational database.

Default

`report`

Data Type

String

See also

Using Connection Properties on page 36

Password

Description

Specifies the password to use to connect to your Oracle Service Cloud instance. A password is required. Contact your system administrator to obtain your password.

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

Valid Values

`string`

where:

`string`

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

Default

none

Data Type

String

ProxyHost

Description

Identifies a proxy server to use for the first connection.

Valid Values

server_name | *IP_address*

where:

server_name

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

IP_address

is an IP address, specified in either IPv4 or IPv6 format, or a combination of the two. See IP Addresses on page 54 for details about using these formats.

Default

empty string

See also

Connecting Through a Proxy Server on page 35

ProxyPassword

Purpose

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

Valid Values

password

where:

password

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

Default

empty string

See also

Connecting Through a Proxy Server on page 35

ProxyPort

Purpose

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

Valid Values

port

where:

port

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

Default

0

See also

Connecting Through a Proxy Server on page 35

ProxyUser

Purpose

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

Valid Values

user_name

where:

user_name

is a valid user ID for the proxy server.

Default

empty string

See also

Connecting Through a Proxy Server on page 35

RefreshDirtyCache

Purpose

Specifies whether the driver refreshes a dirty cache on the next fetch operation from the cache. A cache is marked as dirty when a row is inserted into or deleted from a cached table or a row in the cached table is updated.

Valid Values

`1` | `0`

Behavior

If set to `1`, a dirty cache is refreshed when the cache is referenced in a fetch operation. The cache state is set to initialized if the refresh succeeds.

If set to `0`, a dirty cache is not refreshed when the cache is referenced in a fetch operation.

Default

`1`

Data Type

Int

RegisterStatementPoolMonitorMBean

Purpose

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

Valid Values

`true` | `false`

Behavior

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

If set to `false`,the driver does not register an MBean for the Statement Pool Monitor for any statement pool

Notes

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

Default

`false`

Data Type

Boolean

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.

Behavior

Attribute	Description
`linelimit=numberofchars`	Sets the maximum number of characters that DataDirect Spy logs on a single line. The default is 0 (no maximum limit).
`log=(file)filename`	Directs logging to the file specified by *filename*. For Windows, if coding a path to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: `log=(file)C:\\temp\\spy.log;logIS=yes;logTName=yes`.

Attribute	Description

`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.
`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 `no`.

Example

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

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

Notes

- If coding a path on Windows to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: `log=(file)C:\\temp\\spy.log`.
- If a log file name does not include the `.log` extension, the driver automatically appends it. For example, a file named `spy.jsp` is renamed to `spy.jsp.log` by the driver.
- Refer to "Tracking JDBC Calls with DataDirect Spy" in the *Progress DataDirect for JDBC Drivers Reference* for a list of supported attributes.

Default

none

Data Type

String

StmtCallLimit

Purpose

Specifies the maximum number of Web service calls the driver can make when executing any single SQL statement or metadata query.

Valid Values

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

where:

`x`

is a positive integer that defines the maximum number of Web service calls the driver can make when executing any single SQL statement or metadata query.

Behavior

If set to `-1`, the driver uses the default value that is configured in the service when connected to a site whose version is August 2014 or later. When connected to sites whose version is prior to August 2014, the driver sets the maximum number of calls to 100.

If set to `0`, the driver uses the maximum number of calls allowed by the service when connected to a site whose version is August 2014 or later. When connected to sites whose version is prior to August 2014, there is no limit.

If set to *`x`*, the driver uses this value to set the maximum number of Web service calls that can be made when executing a SQL statement or metadata query.

The value specified for this connection property can be overridden by changing the STMT_CALL_LIMIT session attribute using the ALTER SESSION statement. For example, the following statement sets the statement call limit to 10 Web service calls:

```
ALTER SESSION SET STMT_CALL_LIMIT=10
```

If the Web service call limit is exceeded, the behavior of the driver depends on the value specified for the StmtCallLimitBehavior property.

Additionally, if you specify a value that is greater than the maximum number of calls allowed when connected to a site whose version is August 2014 or later, the driver returns a warning that the setting exceeds the maximum value and uses the maximum value permitted.

Default

`-1`

Data Type

Int

StmtCallLimitBehavior

Purpose

Specifies the behavior of the driver when the maximum Web service call limit specified by the StmtCallLimit property is exceeded.

Valid Values

`errorAlways` | `returnResults`

Behavior

If set to `errorAlways`, the driver generates an exception if the maximum Web service call limit is exceed.

If set to `returnResults`, the driver returns any partial results it received prior to the call limit being exceeded. The driver generates a warning that not all of the results were fetched.

Default

`errorAlways`

Data Type

String

TransactionMode

Purpose

Specifies how the driver handles manual transactions.

Valid Values

`ignore` | `noTransactions`

Behavior

If set to `ignore`, the data source does not support transactions and the driver always operates in auto-commit mode. Calls to set the driver to manual commit mode and to commit transactions are ignored. Calls to rollback a transaction cause the driver to throw an exception indicating that no transaction is started. Metadata indicates that the driver supports transactions and the ReadUncommitted transaction isolation level.

If set to `noTransactions`, the data source and the driver do not support transactions. Metadata indicates that the driver does not support transactions.

Default

`noTransactions`

Data Type

String

User

Purpose

Specifies the user name that is used to connect to the Oracle Service Cloud instance. A user name is required.

Valid Values

`string`

where:

`string`

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

Default

none

Data Type

String

WSCompressData

Purpose

Specifies whether the driver compresses data it sends to or receives from the Web server.

Valid Values

`none` | `compress`

Behavior

If set to `none`, the driver sends and receives uncompressed data to and from the Web server.

If set to `compress`, the driver sends and receives compressed data to and from the Web server.

Notes

- Setting the WSCompressData property to `none` can significantly degrade performance.

Default

`compress`

Data Type

String

WSFetchSize

Purpose

Specifies the number of rows of data the driver attempts to fetch for each JDBC call.

Valid Values

`0` | *`x`*

where:

`x`

is a positive integer that defines the number of rows.

Behavior

If set to `0`, the driver uses the maximum page size for the Oracle Service Cloud database to which it is connecting (Operational or Report) for sites whose version is August 2014 or later. When connecting to sites whose version is prior to August 2014, the driver attempts to fetch up to a maximum of 10,000 rows. This value typically provides the maximum throughput.

If set to *`x`*, the driver attempts to fetch up to a maximum of the specified number of rows. Setting the value lower than the maximum page size allowed can reduce the response time for returning the initial data. However, using a smaller WSFetchSize could improve response time for interactive applications.

Notes

- If the value specified is greater than the value allowed by the server, the driver returns a warning that the setting exceeds the maximum value and uses the maximum value permitted. For servers prior to version August 2014, the maximum is 10,000 rows. For versions August 2014 or later, the maximum is server dependent.

- FetchSize is related to, but different than, WSFetchSize. WSFetchSize specifies the number of rows of raw data that the driver fetches from the remote data source, while FetchSize specifies how many of these raw data rows the driver processes before returning data to the application. Processing the data includes converting from the remote data source data type to the driver SQL data type used by the application. If FetchSize is greater than WSFetchSize, the driver makes multiple round trips to the data source to get the requested number of rows before returning control to the application.

Default

`0`

Data Type

Int

See also

- Performance Considerations on page 40
- FetchSize on page 70

WSRetryCount

Description

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.

Default

`0`

Data Type

Int

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.

Default

`120` (seconds)

Data Type

Int

5

Supported SQL Statements and Extensions

The Oracle Service Cloud driver provides support for standard SQL (primarily SQL-92). In addition, the product supports a set of SQL extensions. For example, the product supports extensions that allow you to change the default schema or set the maximum number of Web service calls the driver can make when executing a SQL statement.

This chapter describes both the standard SQL statements and the SQL extensions. SQL extensions are denoted by an (EXT) in the topic title.

For details, see the following topics:

- Alter Cache (EXT)
- Alter Index
- Alter Sequence
- Alter Session (EXT)
- Alter Table
- Checkpoint
- Create Cache (EXT)
- Create Index
- Create Sequence
- Create Table
- Create View
- Drop Cache (EXT)

- Drop Index
- Drop Sequence
- Drop Table
- Drop View
- Explain Plan
- Refresh Cache (EXT)
- Set Checkpoint Defrag
- Set Logsize
- Select
- SQL Expressions
- Subqueries

Alter Cache (EXT)

Purpose

The Alter Cache statement changes the definition of a cache on a remote table or view. An error is returned if the remote table or view specified does not exist.

Syntax

```
ALTER CACHE ON {remote_table | view}
  [REFRESH_INTERVAL {0 | -1 | interval_value [{M, H, D}]}]
  [INITIAL_CHECK [ONFIRSTCONNECT | FIRSTUSE | DEFAULT}]
  [PERSIST {TEMPORARY | MEMORY | DISK | DEFAULT}]
  [ENABLED {YES | TRUE | NO | FALSE}]
  [CALL_LIMIT {0 | -1 | max_calls}]
  [FILTER (expression)]
```

where:

remote_table

is the name of the remote table cache definition to be modified. The remote table name can be a two-part name: *schemaname.tablename*. When specifying a two-part name, the specified remote table must be defined in the specified schema, and you must have the privilege to alter objects in the specified schema.

view

is the name of the view cache definition to be modified. The view name can be a two-part name: *schemaname.viewname*. When specifying a two-part name, the specified view must be defined in the specified schema, and you must have the privilege to alter objects in the specified schema. Caches on views are not currently supported in the product.

`REFRESH_INTERVAL`

is an optional clause that specifies the length of time the data in the cached table can be used before being refreshed. See "Refresh Interval Clause" for a complete explanation.

`INITIAL_CHECK`

is an optional clause that specifies when the driver initially checks whether the data in the cache needs refreshed. See "Initial Check Clause" for a complete explanation.

`PERSIST`

is an optional clause that specifies the life span of the data in the cached table or view. See "Persist Clause" for a complete explanation.

`ENABLED`

is an optional clause that specifies whether the cache is enabled or disabled for use with SQL statements. See "Enabled Clause" for a complete explanation.

`CALL_LIMIT`

is an optional clause that specifies the maximum number of Web service calls that can be used to populate or refresh the cache. See "Call Limit Clause" for a complete explanation.

`FILTER`

is an optional clause that specifies a filter for the primary table to limit the number of rows that are cached in the primary table. See "Filter Clause" for a complete explanation.

Notes

- At least one of the optional clauses must be used. If two or more are specified, they must be specified in the order shown in the syntax description.

See also

Refresh Interval Clause on page 102
Initial Check Clause on page 102
Persist Clause on page 103
Enabled Clause on page 104
Call Limit Clause on page 104
Filter Clause on page 105

Alter Index

Purpose

The Alter Index statement changes the name of an existing index. Index names must not conflict with other user-defined or system-defined names.

Syntax

```
ALTER INDEX index_name RENAME TO new_name
```

where:

index_name

specifies an existing index name.

new_name

specifies the new index name.

Alter Sequence

Purpose

The Alter Sequence statement resets the next value of an existing sequence.

Syntax

```
ALTER SEQUENCE sequence_name RESTART WITH value
```

where:

sequence_name

specifies an existing sequence.

value

specifies the next value to be returned through the Next Value For clause (see "Next Value for Clause").

Notes

- Indexes on remote tables cannot be created, altered or dropped. Indexes can only be defined on local tables by the driver.

See also

Next Value For Clause on page 107

Alter Session (EXT)

Purpose

The Alter Session statement changes various attributes of a database session or a remote session. A database session maintains the state of the overall connection. A remote session maintains the state that pertains to a particular remote data source connection.

Syntax

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

where:

attribute_name

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

value

refers to the specific value setting for that attribute.

The following table lists the database and remote session attributes, and provides their descriptions.

Table 15: 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 database session. For example: `ALTER SESSION SET CURRENT_SCHEMA=RIGHTNOW`

Attribute Name	Session Type	Description
Stmt_Call_Limit	Database	Sets the maximum number of Web service calls the driver can make when executing a statement. Setting the Stmt_Call_Limit attribute has the same effect as setting the StmtCallLimit connection option. It sets the default Web service call limit used by any statement on the connection. Executing this command on a statement overrides the previously set StmtCallLimit for the connection. The value specified for this attribute can be `-1`, `0`, or a positive integer: If `-1` is specified, the limit is the default value configured in the service when connected to a site whose version is August 2014 or later. When connected to a site whose version is prior to August 2014, the driver sets the maximum number of calls to 100. If `0` is specified, the limit is the maximum number of calls allowed by the service when connected to sites whose version is August 2014 or later. For versions prior to August 2014, there is no limit. For example, the following statement sets the statement call limit to 10 Web service calls: `ALTER SESSION SET STMT_CALL_LIMIT=10` **Note:** If you specify a value that is greater than the maximum value allowed by the service when connecting to a site whose version is August 2014 or later, the driver returns a warning indicating that it will use the service defined maximum instead of the value specified.

Ws_Call_Count	Remote	Resets the Web service call count of a remote 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 remote data source instance for the current session. For example:
		
		`ALTER SESSION SET servicecloud.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()`

Alter Table

Purpose

The Alter Table statement adds a column, removes a column, or redefines a column in a local table. A local table is maintained by the driver and is local to the machine on which the driver is running. A local table is exposed in the PUBLIC schema.

Syntax

```
ALTER TABLE table_name [add_clause] [drop_clause] [rename_clause]
```

where:

table_name

specifies an existing local table.

add_clause

specifies a column or constraint to be added to the table. See "Add Clause: Columns" and "Add Clause: Constraints" for a complete explanation.

drop_clause

specifies a column or constraint to be dropped from the table. See "Drop Clause: Columns" and "Drop Clause: Constraints" for a complete explanation.

rename_clause

specifies a new name for the table. See "Rename Clause" for a complete explanation.

Notes

- Alter Table for remote tables is not supported.

See also

Add Clause: Columns on page 97
Add Clause: Constraints on page 98
Drop Clause: Columns on page 98
Drop Clause: Constraints on page 99
Rename Clause on page 99

Add Clause: Columns

Purpose

Use the Add clause to add a column to an existing table. It is optional.

This clause adds a column to the end of the column list. It defines a column with the same syntax as the Create Table command (see "Column Definition for Local Tables"). If `NOT NULL` is specified and the table is not empty, a default value must be specified. In all other respects, this command is the equivalent of a column definition in a Create Table statement.

Syntax

```
ADD [COLUMN] column_nameDatatype ... [BEFORE existing_column]
```

Notes

- The optional `Before` *existing_column* can be used to specify the name of an existing column so that the new column is inserted in a position just before the existing column.
- The optional `Before` *existing_column* can be used to specify the name of an existing column so that the new column is inserted in a position just before the existing column.

- If a SQL view includes `SELECT * FROM` for the table to which the column was added in the view's Select statement, the new column is added to the view.

Example A

Assuming the current schema is PUBLIC, this example adds the `status` column with a default value of `ACTIVE` to the `test` table.

```
ALTER TABLE test ADD COLUMN status VARCHAR(30) DEFAULT 'ACTIVE'
```

Example B

Assuming the current schema is PUBLIC, this example adds a `deptId` column that can be used as a foreign key column.

```
ALTER TABLE test ADD COLUMN deptId VARCHAR(18)
```

See also

Column Definition for Local Tables on page 109

Add Clause: Constraints

Purpose

Use the Add clause to add a constraint to an existing table. It is optional.

This command adds a constraint using the same syntax as the Create Table command (see Constraint Definition for Local Tables on page 110).

Syntax

```
ADD [CONSTRAINT constraint_name] ...
```

Notes

- You cannot add a Unique constraint if one is already assigned to the same column list. A Unique constraint works only if the values of the columns in the constraint columns list for the existing rows are unique or include a Null value.
- Adding a foreign key constraint to the table fails if, for each existing row in the referring table, a matching row (with equal values for the column list) is not found in the referenced table.

Example A

Assuming the current schema is PUBLIC, this example adds a foreign key constraint to the `deptId` column of the `test` table that references the rowId of the `dept` table.

```
ALTER TABLE test ADD CONSTRAINT test_fk FOREIGN KEY (deptId) REFERENCES dept(id)
```

Drop Clause: Columns

Purpose

Use the Drop clause to drop a column from an existing table. It is optional.

Syntax

```
DROP {[COLUMN] column_name}
```

where:

column_name

specifies an existing column in an existing table.

Notes

- Drop fails if a SQL view includes the column.

Example A

This example drops the `status` column. For the operation to succeed, the status column cannot have a constraint defined on it and cannot be used in a SQL view.

```
ALTER TABLE test DROP COLUMN status
```

Drop Clause: Constraints

Purpose

Use the Drop clause to drop a constraint from an existing table. It is optional.

Syntax

```
DROP {[CONSTRAINT] constraint_name}
```

where:

constraint_name

specifies an existing constraint.

Notes

- The specified constraint cannot be a primary key constraint or unique constraint.

Example A

This example drops the `test_fk` constraint.

```
ALTER TABLE test DROP CONSTRAINT test_fk
```

Rename Clause

Purpose

Use the Rename clause to rename an existing table. It is optional.

Syntax

```
RENAME TO new_name
```

where:

new_name

specifies the new name for the table.

Example A

This example renames the table to `test2`.

```
ALTER TABLE test RENAME TO test2
```

Checkpoint

Purpose

The Checkpoint statement ensures that all database changes in memory are committed to disk. Executing the Checkpoint statement closes the database files, rewrites the script file, deletes the log file, and reopens the database.

Syntax

```
CHECKPOINT [DEFRAG]
```

Notes

- If `DEFRAG` is specified, this statement evaluates abandoned space in the database data (.data) file and shrinks the data file to its minimum size.

Create Cache (EXT)

Purpose

The Create Cache statement creates a cache that holds the data of a remote table. The data is not loaded into the cache when the Create Cache statement is executed; the data is loaded the first time that the remote table is executed or when a Refresh Cache statement on the remote table is executed. An error is returned if the remote table specified does not exist.

Syntax

```
CREATE CACHE ON {remote_table}
  [REFRESH_INTERVAL {0 | -1 | interval_value [{M, H, D}]}]
  [INITIAL_CHECK [{ONFIRSTCONNECT | FIRSTUSE | DEFAULT}]
  [PERSIST {TEMPORARY | MEMORY | DISK | DEFAULT}]
  [ENABLED {YES | TRUE | NO | FALSE}]
  [CALL_LIMIT {0 | -1 | max_calls}]
  [FILTER (expression)]
```

where:

remote_table

is the name of the remote table from which data is to be cached on the client. The name of the cached table is the same as the name of the remote table. When the table name is specified in a query, the cached table is accessed, not the remote table.

The remote table name can be a two-part name: *schemaname.tablename*. When specifying a two-part name, the specified remote table must be defined in the specified schema, and you must have the privilege to create objects in the specified schema.

`REFRESH_INTERVAL`

is an optional clause that specifies the length of time the data in the cached table can be used before being refreshed. See "Refresh Interval clause" for a complete explanation.

`INITIAL_CHECK`

is an optional clause that specifies when the driver initially checks whether the data in the cache needs refreshed. See "Initial Check clause" for a complete explanation.

`PERSIST`

is an optional clause that specifies the life span of the data in the cached table or view. See "Persist clause" for a complete explanation.

`ENABLED`

is an optional clause that specifies whether the cache is enabled or disabled for use with SQL statements. See "Enabled clause" for a complete explanation.

`CALL_LIMIT`

is an optional clause that specifies the maximum number of Web service calls that can be used to populate or refresh the cache. See "Call Limit clause" for a complete explanation.

`FILTER`

is an optional clause that specifies a filter for the primary table to limit the number of rows that are cached in the primary table. See "Filter clause" for a complete explanation.

Notes

- Caches on views are not supported.
- If two or more optional clauses are specified, they must be specified in the order shown in the syntax description.

See also

Refresh Interval Clause on page 102
Initial Check Clause on page 102
Persist Clause on page 103
Enabled Clause on page 104
Call Limit Clause on page 104
Filter Clause on page 105

Refresh Interval Clause

Purpose

The Refresh Interval clause specifies the length of time the data in the cached table can be used before being refreshed; it is optional. The driver maintains a timestamp of when the data in a table was last refreshed. When a cached table is used in a query, the driver checks if the current time is greater than the last refresh time plus the value of Refresh_Interval. If it is, the driver refreshes the data in the cached table before processing the query.

Syntax

`[REFRESH_INTERVAL {0 | -1 | interval_value [{M, H, D}]}]`

where:

`0`

specifies that the cache is refreshed manually. You can use the Refresh Cache statement to refresh the cache manually.

`-1`

resets the refresh interval to the default value of 12 hours.

`interval_value`

is a positive integer that specifies the amount of time between refreshes. The default unit of time is hours (`H`). You can also specify `M` for minutes or `D` for days. For example, `60M` would set the time between refreshes to 60 minutes.The default refresh interval is 12 hours.

Initial Check Clause

Purpose

The Initial Check clause specifies when the driver performs its initial check of the data in the cache to determine whether it needs to be refreshed; it is optional.

Syntax

`[INITIAL_CHECK [ONFIRSTCONNECT | FIRSTUSE | DEFAULT}]`

where:

`ONFIRSTCONNECT`

specifies that the initial check is performed the first time a connection for a user is established. Subsequently, it is performed each time the table or view is used. A driver session begins on the first connection for a user and the session is active as long as at least one connection is open for the user.

`FIRSTUSE`

specifies that the initial check is performed the first time the table or view is used in a query. Subsequently, it is performed each time the table or view is used.

DEFAULT

resets the value back to its default, which is FIRSTUSE.

Persist Clause

Purpose

The Persist clause specifies the life span of the data in the cached table or view; it is optional.

Syntax

```
[PERSIST {TEMPORARY | MEMORY | DISK | DEFAULT}]
```

where:

TEMPORARY

specifies that the data exists for the life of the driver session. When the driver session ends, the data is discarded. A driver session begins on the first connection for a user and the session is active if at least one connection is open for the user.

MEMORY

specifies that the data exists beyond the life of the connection. While the connection is active, the cached data is stored in memory. When the connection is closed, the cached data is persisted to disk. If the connection ends abnormally, changes to the cached data may not be persisted to disk. This is the default.

DISK

specifies that the data exists beyond the life of the connection. A portion of the cached data is stored in memory while the connection is active. If the size of the cached data exceeds the cache memory threshold, the remaining data is stored on disk. When the connection is closed, the portion of the cached data that is in memory is persisted to disk. If the connection ends abnormally, changes to the cached data held in memory may not be persisted to disk.

DEFAULT

resets the PERSIST value back to its default, which is MEMORY.

Notes

- You can design your application to force all cached data held in memory to be persisted to disk at any time by using the "Checkpoint" statement.
- If you specify a value of MEMORY or DISK for the Persist clause, the remote data remains on the client past the lifetime of the application.

See also

Checkpoint on page 100

Enabled Clause

Purpose

The Enabled clause specifies whether the cache is enabled or disabled for use with SQL statements; it is optional.

Syntax

`[ENABLED {YES | TRUE | NO | FALSE}]`

where:

`YES | TRUE`

specifies that the cache is enabled. When a cache is enabled, the driver accesses the cached data for the remote table or view when a query is executed.

The driver does not check whether the cache needs to be refreshed when the Alter Cache statement is used to enable the cache. The check occurs the next time that the cache is accessed.

`NO | FALSE`

specifies that the cache is disabled, which means that the driver accesses the data in the remote table or view rather than the cache when a query is executed. The driver does not update the cache when inserts, updates, and deletes are performed on a remote table or view. To use the cache, you must enable it.

All data in an existing cache is persisted on the client even when the cache is disabled, except for the case where `PERSIST` is set to `TEMPORARY`.

The default is `TRUE`.

Call Limit Clause

Purpose

The Call Limit clause specifies the maximum number of Web service calls that can be used to populate or refresh the cache; it is optional.

Syntax

`[CALL_LIMIT {0 | -1 | `*`max_calls`*`}]`

where:

`0`

specifies no call limit.

`-1`

resets the call limit back to its default, which is 0 (no call limit).

max_calls

is a positive integer that specifies the maximum number of Web service calls.

The default value is `0`.

Notes

- The call limit for a cache is independent of the Stmt_Call_Limit set on a database session. See "Alter Session (EXT)" for details.

  If the call limit of a cache is exceeded during the population or refresh of the cache, the cache is marked as partially initialized. At the next refresh opportunity, the driver attempts to complete the population or refresh of the cache. If the call limit (or other error) occurs during this second attempt, the cache becomes invalid and is disabled. All data in the cache is discarded after the second attempt to populate or refresh the cache fails. Before re-enabling the cache, consider altering the cache definition to allow more Web service calls or specify a more restrictive filter, or both.

See also

Alter Session (EXT) on page 94

Filter Clause

Purpose

Filter is an optional clause that specifies a filter for the primary table to limit the number of rows that are cached in the primary table. This clause is not supported for views.

Syntax

```
[FILTER (expression)]
```

where:

expression

is any valid Where clause. See "Where Clause" for details. Do not include the Where keyword in the clause. The filter for an existing cache can be removed by specifying an empty string for the filter expression, for example, `FILTER()`.

The default value is that cached data is not filtered.

Example A

The following example filters by last activity date.

```
FILTER (lastactivitydate >= {d'2010-01-01'})
```

Example B

The following example caches all rows of the account table with a refresh interval of 12 hours, checks whether data of the cached table needs to be refreshed on the first use, persists the data beyond the life of the connection, and stores the data in memory while the connection is active.

```
CREATE CACHE ON account
```

Example C

The following example caches all active accounts in the account table with a refresh interval of 1 day, checks whether data of the cached table needs to be refreshed when the connection is established, and discards the data when the connection is closed.

```
CREATE CACHE ON account REFRESH_INTERVAL 1d INITIAL_CHECK ONFIRSTCONNECT PERSIST
TEMPORARY FILTER(account.active = 'Yes')
```

See also

Where Clause on page 126

Create Index

Purpose

The Create Index statement creates an index on one or more columns in a local table.

Syntax

CREATE [UNIQUE] INDEX *index_name* ON *table_name* (*column_name* [, ...])

where:

UNIQUE

means that key columns cannot have duplicate values.

index_name

specifies the name of the index to be created.

table_name

specifies an existing local table.

column_name

specifies an existing column.

Notes

- The driver cannot create an index in a remote table; the driver returns an error indicating that the operation cannot be performed on a remote table.
- Creating a unique constraint is the preferred way to specify that the values of a column must be unique.

Create Sequence

Purpose

The Create Sequence statement creates an auto-incrementing sequence for a local table.

Syntax

```
CREATE SEQUENCE sequence_name [AS {INTEGER | BIGINT}] [START WITH start_value]
[INCREMENT BY increment_value]
```

where:

sequence_name

specifies the name of the sequence. By default, the sequence type is INTEGER.

start_value

specifies the starting value of the sequence. The default start value is 0.

increment_value

specifies the value of the increment; the value must be a positive integer. The default increment is 1.

Next Value For Clause

Purpose

Use the Next Value For clause to specify the next value for a sequence that is used in a Select, Insert, or Update statement.

Syntax

```
NEXT VALUE FOR sequence_name
```

where:

sequence_name

specifies the name of the sequence from which to retrieve the value.

Example

The following example retrieves the next value or set of values in Sequence1:

```
SELECT NEXT VALUE FOR Sequence1 FROM Account
```

Create Table

Purpose

Creates a new local table in the machine on which the driver is running. A local table is exposed in the PUBLIC schema.

Syntax

```
CREATE [{MEMORY | DISK | [GLOBAL] {TEMPORARY | TEMP}]
TABLE table_name (column_definition [, ...]
[, constraint_definition...]) [ON COMMIT {DELETE | PRESERVE} ROWS
```

where:

`MEMORY`

creates the new table in memory. The data for a memory table is held entirely in memory for the duration of the database session. When the database is closed, the data for the memory table is persisted to disk.

`DISK`

creates the new table on disk. A disk table caches a portion of its data in memory and the remaining data on disk.

`TEMPORARY and TEMP`

are equivalent and create the new table as a global temporary table. The GLOBAL qualifier is optional. The definition of a global temporary table is visible to all connections. The data written to a global temporary table is visible only to the connection used to write the data.

Note: If MEMORY, DISK, or TEMPORARY/TEMP is not specified, the new table is created in memory.

`table_name`

specifies the name of the new table.

`column_definition`

specifies the definition of a column in the new table. See "Column Definition for Local Tables" for a complete explanation.

`constraint_definition`

specifies constraints on the columns of the new table. See "Constraint Definition for Local Tables" for a complete explanation.

`ON COMMIT PRESERVE ROWS`

preserves row values in a temporary table while the connection is open; this is the default action.

`ON COMMIT DELETE ROWS`

empties row values on each commit or rollback.

See also

Column Definition for Local Tables on page 109
Constraint Definition for Local Tables on page 110

Column Definition for Local Tables

Purpose

Use the following syntax to define a column for local tables.

Syntax

```
column_name Datatype [(precision[,scale])]
[{DEFAULT default_value | GENERATED BY DEFAULT AS IDENTITY
(START WITH n[, INCREMENT BY m])}] | [[NOT] NULL]
[IDENTITY] [PRIMARY KEY]
```

where:

column_name

is the name to be assigned to the column.

Datatype

is the data type of the column to be created. See the "Data Types" topic for a list of supported data types.

precision

is the number characters for CHAR and VARCHAR columns, the number of bytes for BINARY and VARBINARY columns, and the total number of digits for DECIMAL columns.

scale

is the number of digits to the right of the decimal point for DECIMAL columns and the number of seconds for DATETIME columns.

default_value

is the default value to be assigned to the column. The following default values are allowed in column definitions for local tables:

- For character columns, a single-quoted string or NULL. The only SQL function that can be used is CURRENT_USER.
- For datetime columns, a single-quoted Date, Time, or Timestamp value or NULL. You can also use the following datetime SQL functions: CURRENT_DATE, CURRENT_TIME, CURRENT_ TIMESTAMP, TODAY, or NOW.
- For boolean columns, the literals FALSE, TRUE, NULL.
- For numeric columns, any valid number or NULL.
- For binary columns, any valid hexadecimal string or NULL.

`IDENTITY | GENERATED BY DEFAULT AS IDENTITY`

defines an auto-increment column. Either clause can be specified only on INTEGER or BIGINT columns. Identity columns are considered primary key columns, so a table can have only one Identity column.

The GENERATED BY DEFAULT AS IDENTITY clause is the standard SQL syntax for specifying an Identity column.

The IDENTITY operator is equivalent to GENERATED BY DEFAULT AS IDENTITY without the optional START WITH clause.

START WITH *n*[, INCREMENT BY *m*]

specifies the sequence of numbers generated for the Identity column. *n* and *m* are the starting and incrementing values, respectively, for an Identity column. The default start value is 0 and the default increment value is 1.

Example A

Assuming the current schema is PUBLIC, a local table is created. id is an identity column with a starting value of 0 and an increment value of 1 because no Start With and Increment By clauses are specified.

```
CREATE TABLE Test (id INTEGER GENERATED BY DEFAULT AS IDENTITY, name VARCHAR(30))
```

This example is equivalent to the previous example.

```
CREATE TABLE Test (id INTEGER IDENTITY, name VARCHAR(30))
```

Example B

Assuming the current schema is PUBLIC, a local table is created. id is an identity column with a starting value of 2 and an increment of 2.

```
CREATE TABLE Test (id INTEGER GENERATED BY DEFAULT AS IDENTITY (START WITH 2,
INCREMENT BY 2), name VARCHAR(30))
```

Constraint Definition for Local Tables

Purpose

Defines the syntax to define a contraint for a local table.

Syntax

```
[CONSTRAINT [constraint_name]
  {unique_constraint |
  primary_key_constraint |
  foreign_key_constraint}]
```

where:

constraint_name

specifies a name for the constraint.

unique_constraint

specifies a constraint on a single column in the table. See Unique Clause for syntax.

Values in the constrained column cannot be repeated, except in the case of null values. For example:

```
ColA
```

```
1
2
NULL
4
5
NULL
```

A single table can have multiple columns with unique constraints.

primary_key_constraint

specifies a constraint on one or more columns in the table. See Primary Key Clause for syntax.

Values in a single column primary key column must be unique. Values across multiple constrained columns cannot be repeated, but values within a column can be repeated. Null values are not allowed. For example:

```
Col A  Col B
2       1
3       1
4       2
5       2
6       2
```

Only one primary key constraint is allowed in the table.

foreign_key_constraint

defines a link between related tables. See Foreign Key Clause for syntax.

A column defined as a foreign key in one table references a primary key in the related table. Only values that are valid in the primary key are valid in the foreign key. The following example is valid because the foreign key values of the dept id column in the EMP table match those of the id column in the referenced table DEPT:

Referenced Table	Main Table
DEPT	EMP (Foreign Key)

Referenced Table		Main Table		
id	name	id	name	dept id
1	Dev	1	Mark	1
2	Finance	1	Jim	3
3	Sales	1	Mike	2

The following example, however, is not valid. The value 4 in the dept id column does not match any value in the referenced id column of the DEPT table.

Referenced Table		Main Table		
DEPT		EMP		
				(Foreign Key)
id	name	id	name	dept id
1	Dev	1	Mark	1
2	Finance	1	Jim	3
3	Sales	1	Mike	4

Unique Clause

`UNIQUE (`*`column_name`*` [,`*`column_name`*`...])`

where:

column_name

specifies the column to which the constraint is applied. Multiple columns names must be separated by commas.

Primary Key Clause

`PRIMARY KEY (`*`column_name`*` [,`*`column_name`*`...])`

where:

column_name

specifies the primary key column to which the constraint is applied. Multiple column names must be separated by commas.

Foreign Key Clause

```
FOREIGN KEY (fcolumn_name [,fcolumn_name...])
  REFERENCES ref_table (pcolumn_name [,pcolumn_name...])
  [ON {DELETE | UPDATE}
  {CASCADE | SET DEFAULT | SET NULL}]
```

fcolumn_name

specifies the foreign key column to which the constraint is applied. Multiple column names must be separated by commas.

ref_table

specifies the table to which a foreign key refers.

pcolumn_name

specifies the primary key column or columns referenced in the referenced table. Multiple column names must be separated by commas.

`ON DELETE`

is a clause that defines the operation performed when a row in the table referenced by a foreign key constraint is deleted. One of the following operators must be specified in the On Delete clause:

where:

- `CASCADE` specifies that all rows in the foreign key table that reference the deleted row in the primary key table are also deleted.
- `SET DEFAULT` specifies that the value of the foreign key column is set to the column default value for all rows in the foreign key table that reference the deleted row in the primary key table.
- `SET NULL` specifies that the value of the foreign key column is set to NULL for all rows in the foreign key table that reference the deleted row in the primary key table.

`ON UPDATE`

is a clause that defines the operation performed when the primary key of a row in the table referenced by a foreign key constraint is updated. One of the following operators must be specified in the On Update clause:

- `CASCADE` specifies that the value of the foreign key column for all rows in the foreign key table that reference the row in the primary key table that had the primary key updated are updated with the new primary key value.
- `SET DEFAULT` specifies that the value of the foreign key column is set to the column default value for all rows in the foreign key table that reference the row that had the primary key updated in the primary key table.
- `SET NULL` specifies that the value of the foreign key column is set to NULL for all rows in the foreign key table that reference the row that had the primary key updated in the primary key table.

Notes

- Remote tables are not supported.
- You must specify at least one constraint.
- Both the On Delete and On Update clauses can be used in a single foreign key definition.

Example

Assuming the current schema is PUBLIC, the `emp` table is created with the `name`, `empId`, and `deptId` columns. The table contains a foreign key constraint on the `deptId` column that references the `id` column in the `dept` table. In addition, it sets the value of any rows in the `deptId` column to NULL that point to a deleted row in the referenced `dept` table.

```
CREATE TABLE emp (name VARCHAR(30), empId INTEGER, deptId INTEGER, FOREIGN
KEY(deptId) REFERENCES dept(id)) ON DELETE SET NULL)
```

Create View

Purpose

The Create View statement creates a new view. A view is analogous to a named query. The view's query can refer to any combination of remote and local tables as well as other views. Views are read-only; they cannot be updated.

Syntax

```
CREATE VIEW view_name[(view_column,...)] AS
SELECT ... FROM ... [WHERE Expression]
  [ORDER BY order_expression [, ...]]
  [LIMIT limit [OFFSET offset]];
```

where:

view_name

specifies the name of the view.

view_column

specifies the column associated with the view. Multiple column names must be separated by commas.

The other commands used for Create View are the same as those used for Select (see "Select").

Notes

- A view can be thought of as a virtual table. A Select statement is stored in the database; however, the data accessible through a view is not stored in the database. The result set of the Select statement forms the virtual table returned by the view. You can use this virtual table by referring to the view name in SQL statements the same way you refer to a table. A view is used to perform any or all of these functions:
  - Restrict a user to specific rows in a table.
  - Restrict a user to specific columns.
  - Join columns from multiple tables so that they function like a single table.
  - Aggregate information instead of supplying details. For example, the sum of a column, or the maximum or minimum value from a column can be presented.
- Views are created by defining the Select statement that retrieves the data to be presented by the view.

- The Select statement in a View definition must return columns with distinct names. If the names of two columns in the Select statement are the same, use a column alias to distinguish between them. Alternatively, you can define a list of new columns for a view.

Example A

This example creates a view named myOpportunities that selects data from three database tables to present a virtual table of data.

```
CREATE VIEW myOpportunities AS
SELECT a.name AS AccountName,
       o.name AS OpportunityName,
       o.amount AS Amount,
       o.description AS Description
FROM Opportunity o INNER JOIN Account a
       ON o.AccountId = a.id
       INNER JOIN User u
       ON o.OwnerId = u.id
WHERE u.name = 'MyName'
       AND o.isClosed = 'false'
ORDER BY Amount desc
```

You can then refer to the myOpportunities view in statements just as you would refer to a table. For example:

```
SELECT * FROM myOpportunities;
```

Example B

The myOpportunities view contains a detailed description for each opportunity, which may not be needed when only a summary is required. A view can be built that selects only specific myOpportunities columns as shown in the following example:

```
CREATE VIEW myOpps_NoDesc as
SELECT AccountName,
       OpportunityName,
       Amount
FROM myOpportunities
```

The view selects the name column from both the opportunity and account tables. These columns are assigned the alias OpportunityName and AccountName, respectively.

See also

Select on page 120

Drop Cache (EXT)

Purpose

The Drop Cache statement drops the cache defined on a remote table.

Syntax

DROP CACHE ON {*remote_table*} [IF EXISTS]

where:

remote_table

is the name of the remote table cache to be dropped. The remote table name can be a two-part name: *schemaname.tablename*. When specifying a two-part name, the specified remote table must be mapped in the specified schema, and you must have the privilege to drop objects in the specified schema.

`IF EXISTS`

specifies that an error is not to be returned if a cache for the remote table or view does not exist.

Notes

- Caches on views are not supported.

Drop Index

Purpose

The Drop Index statement drops an index for a local table.

Syntax

`DROP INDEX` *index_name* `[IF EXISTS]`

where:

index_name

specifies an existing index.

`IF EXISTS`

specifies that an error is not to be returned if the index does not exist. The Drop Index command generates an error if an index that is associated with a UNIQUE or FOREIGN KEY constraint is specified.

Notes

- Indexes on a remote table cannot be dropped. Only indexes on local tables can be created, altered, and dropped.

Drop Sequence

Purpose

The Drop Sequence statement drops a sequence for local tables.

Syntax

`DROP SEQUENCE` *sequence_name* `[IF EXISTS] [RESTRICT|CASCADE]`

where:

sequence_name

specifies the name of a sequence to drop.

IF EXISTS

specifies that an error is not to be returned if the sequence does not exist.

RESTRICT

is in effect by default, meaning that the drop fails if any view refers to the sequence.

CASCADE

silently drops all dependent database objects.

Drop Table

Purpose

The Drop Table statement drops (removes) a local table, its data, and its indexes.A local table is maintained by the driver and is local to the machine on which the driver is running. A local table is exposed in the PUBLIC schema. Dropping a table in the PUBLIC schema drops a local table.

Syntax

```
DROP TABLE table_name [IF EXISTS] [RESTRICT | CASCADE]
```

where:

table_name

specifies the name of an existing table to drop.

IF EXISTS

specifies that an error is not to be returned if the table does not exist.

RESTRICT

is in effect by default, meaning that the drop fails if any tables or views reference this table.

CASCADE

specifies that the drop extends to linked objects. If the specified table is a local table, it drops all dependent views and any foreign key constraints that link this table to other tables.

Notes

- Remote tables cannot be dropped.

Drop View

Purpose

The Drop View statement drops a view.

Syntax

```
DROP VIEW view_name [IF EXISTS] [RESTRICT | CASCADE]
```

where:

view_name

specifies the name of a view.

`IF EXISTS`

specifies that an error is not to be returned if the view does not exist.

`RESTRICT`

is in effect by default, meaning that the drop fails if any other view refers to this view.

`CASCADE`

silently drops all dependent views.

Explain Plan

Purpose

The Explain Plan statement can be used with any query to retrieve a detailed list of the elements in the execution plan. Explain Plan generates a result set with a single column named `OPERATION`. The individual elements that comprise the plan are returned as rows in the result set.

Syntax

```
EXPLAIN PLAN FOR {SELECT ... | DELETE ... | INSERT ... | UPDATE ...}
```

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

Refresh Cache (EXT)

Purpose

The Refresh Cache statement forces the data in the cache for the specified remote table to be refreshed.

Syntax

```
REFRESH CACHE ON {remote_table | ALL}
```

where:

remote_table

is the name of the remote table cache to be refreshed. The remote table name can be a two-part name: *schemaname.tablename*. When specifying a two-part name, the specified remote table must be mapped in the specified schema, and you must have the privilege to insert, update, and delete objects in the specified schema.

ALL

forces all caches to be refreshed.

Notes

- Caches on views are not supported.

Set Checkpoint Defrag

Purpose

The Set Checkpoint Defrag statement is used in conjunction with the Checkpoint statement. Set Checkpoint Defrag sets the threshold for triggering a Checkpoint Defrag.

Syntax

```
SET CHECKPOINT DEFRAG size
```

where:

size

specifies the threshold size.

Notes

- When a Checkpoint statement is performed, either as a result of the .log file reaching the limit set by Set Logsize or by the user issuing a Checkpoint statement, the amount of abandoned space in the database data (.data) file is checked. If it is larger than the value of *size*, a CHECKPOINT DEFRAG, which eliminates the abandoned space, is performed instead of CHECKPOINT.

See also

Checkpoint on page 100
Set Logsize on page 120

Set Logsize

Purpose

The Set Logsize statement sets the maximum size to which the driver's embedded database log file can grow before a Checkpoint statement is performed.

Syntax

```
SET LOGSIZE size
```

where:

size

specifies the maximum size in MB of the .log file. The default is 200 MB. A value of 0 means no limit is imposed on the size of the log file.

Notes

- When the log file exceeds the specified size, the Checkpoint statement closes and then reopens the database files, resetting the `.log` file.

See also

Checkpoint on page 100

Select

Purpose

Use the Select statement to fetch results from one or more tables. The Select statement can operate on local and remote tables in any combination.

Syntax

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

where:

select_clause

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

from_clause

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

where_clause

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

groupby_clause

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

having_clause

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

UNION

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

INTERSECT

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

EXCEPT | MINUS

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

orderby_clause

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

limit_clause

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

Note: Although the CreatedTime and UpdatedTime fields on custom objects map to the relational view, you cannot query against them unless you enable the fields within the Object Designer in the Oracle Service Cloud admin tool.

See also

Select Clause on page 122
From Clause on page 124
Where Clause on page 126
Group By Clause on page 127

Having Clause on page 127
Union Operator on page 128
Intersect Operator on page 129
Except and Minus Operators on page 129
Order By Clause on page 130
Limit Clause on page 131

Select Clause

Purpose

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

Syntax

```
SELECT [{LIMIT offsetnumber | TOP number}] [ALL | DISTINCT] {* | column_expression
[[AS] column_alias] [,column_expression [[AS] column_alias], ...]}
[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 offset0`.

`TOP number`

is equivalent to `LIMIT 0number`.

column_expression

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

column_alias

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

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

`DISTINCT`

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

```
SELECT DISTINCT dep FROM emp
```

INTO

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.

See also

SQL Expressions on page 131
Aggregate Functions on page 123

Aggregate Functions

Aggregate functions can also be a part of a Select clause. Aggregate functions return a single value from a set of rows. An aggregate can be used with a column name (for example, AVG(salary)) or in combination with a more complex column expression (for example, AVG(salary * 1.07)). The column expression can be preceded by the DISTINCT operator. The DISTINCT operator eliminates duplicate values from an aggregate expression.

The following table lists supported aggregate functions.

Table 16: Aggregate Functions

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

Example A

In the following example, only distinct last name values are counted. The default behavior is that all duplicate values be returned, which can be made explicit with ALL.

```
COUNT (DISTINCT last_name)
```

Example B

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

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

From Clause

Purpose

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

Syntax

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

where:

`table_name`

> is the name of a table or a subquery. Multiple tables define an implicit inner join among those tables. Multiple table names must be separated by a comma. For example:
>
> ```
> SELECT * FROM emp, dep
> ```
>
> Subqueries can be used instead of table names. Subqueries must be enclosed in parentheses. See "Subquery in a From Clause" for an example.

`table_alias`

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

Example

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

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

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

```
FROM emp E
```

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

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

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

See also

Subquery in a From Clause on page 126

Outer Join Escape Sequences

Purpose

SQL-92 left, right, and full outer join syntax is supported.

Syntax

```
{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 by Oracle Service Cloud data stores:

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

Join in a From Clause

Purpose

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

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

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

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

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*).

Example

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

See also

For more information about subqueries, see "Subqueries."

See also

Subqueries on page 139

Where Clause

Purpose

Specifies the conditions that rows must meet to be retrieved.

Syntax

`WHERE` *expr1rel_operatorexpr2*

where:

expr1

is either a column name, literal, or expression.

expr2

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

rel_operator

is the relational operator that links the two expressions.

Example

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

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

See also

Subqueries on page 139
SQL Expressions on page 131

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 139
SQL Expressions on page 131

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 expr1rel_operatorexpr2
```

where:

expr1 | expr2

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

rel_operator

is the relational operator that links the two expressions.

Example

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

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

See also

SQL Expressions on page 131
Subqueries on page 139

Union Operator

Purpose

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

Syntax

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

Notes

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

Example A

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

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

Example B

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

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

Intersect Operator

Purpose

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

Syntax

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

where:

`DISTINCT`

eliminates duplicate rows from the results.

Notes

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

Example A

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

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

Example B

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

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

Except and Minus Operators

Purpose

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

Syntax

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

where:

`DISTINCT`

eliminates duplicate rows from the results.

Notes

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

Example A

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

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

Example B

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

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

Order By Clause

Purpose

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

Syntax

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

where:

`sort_expression`

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

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

Example

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

`SELECT emp_id, last_name, first_name FROM emp`

`ORDER BY last_name, first_name`

or

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

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

See also

See "SQL Expressions" for details regarding SQL expressions.

See also

SQL Expressions on page 131

Limit Clause

Purpose

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

Syntax

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

where:

`number_of_rows`

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

`OFFSET`

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

Notes

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

Example

The following example returns a maximum of 20 rows.

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

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 Oracle Service Cloud 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 Oracle Service Cloud 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 17: Literal Syntax Examples

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

SQL Type	Literal Syntax	Example
BOOLEAN	Min Value: 0 Max Value: 1	`0` `1`
DATE	'yyyy-mm-dd'	'2010-05-21'
DATETIME	'yyyy-mm-dd hh:mm:ss.SSSSSS'	'2010-05-21 18:33:05.025'
DECIMAL	*n.f* where: *n* is the integral part *f* is the fractional part	0.25 3.1415 -7.48
DOUBLE	*n.f*E*x* where: *n* is the integral part *f* is the fractional part *x* is the exponent	1.2E0 or 2.5E40 or -3.45E2 or 5.67E-4
INTEGER	*n* where *n* is a valid integer value in the range of the INTEGER data type	12 or -34 or 0
LONGVARBINARY	'*hex_value*'	'000482ff'
LONGVARCHAR	'*value*'	'This is a string literal'
TIME	'hh:mm:ss'	'18:33:05'
VARCHAR	'*value*'	'This is a string literal'

Character String Literals

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

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

Example

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

Numeric Literals

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

Example

```
+1894.1204
```

Binary Literals

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

Example

```
'00af123d'
```

Date/Time Literals

Date and time literal values are:

- A Date literal is enclosed in single quotation marks (' '). The format is `yyyy-mm-dd`.
- A Time literal is enclosed in single quotation marks (' '). The format is `hh:mm:ss`.
- A Timestamp is enclosed in single quotation marks (' '). The format is `yyyy-mm-dd hh:mm:ss.SSSSSS`.

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 18: 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 19: 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 Oracle Service Cloud driver considers the UNKNOWN result as FALSE.

The following table lists the supported comparison operators.

Table 20: 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 21: 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 22: Operator Precedence

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

Example A

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

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

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

Example B

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

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

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

Functions

The Oracle Service Cloud driver supports a number of functions that you can use in expressions.

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

Conditions

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

Table 23: Conditions

Condition	Description
Simple comparison	Specifies a comparison with expressions or subquery results. `= , !=, <>, < , >, <=, <=`
Group comparison	Specifies a comparison with any or all members in a list or subquery. `[= , !=, <>, < , >, <=, <=] [ANY, ALL, SOME]`
Membership	Tests for membership in a list or subquery. `[NOT] IN`
Range	Tests for inclusion in a range. `[NOT] BETWEEN`

Condition	Description
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)
```