Progress® DataDirect®

Progress DataDirect for JDBC for Salesforce Driver

User's Guide

Release 6.0.0

Copyright

Table of Contents

1

Welcome to the Progress DataDirect for JDBC for Salesforce Driver

The Progress® DataDirect® *for* JDBC™ for Salesforce™ driver supports standard SQL query language to select, insert, update, and delete data from Salesforce.com and other Web service data stores that use the Salesforce API such as Force.com, and Veeva CRM. To support SQL access to Salesforce, the driver creates a map of the Salesforce data model and translates SQL statements provided by the application to Salesforce queries (SOQL) and Web service calls.

The driver optimizes performance when executing joins by leveraging data relationships among Salesforce objects to minimize the amount of data that needs to be fetched over the network. The Salesforce driver recognizes the relationships among standard objects and custom objects and can access and update both. Relationships among objects can be reported with the metadata methods `getBestRowIdentifier()`, `getColumns()`, `getExportedKeys()`, `getImportedKeys()`, `getPrimaryKey()`, `getTables()`, and `getTypeInfo()`.

The driver also provides a client-side data cache. You can improve performance for some queries by defining rules that specify which data to cache on the client as well as when the cached data becomes invalid and needs to be refreshed.

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-salesforce.

For details, see the following topics:

What's New in this Release?

Support and certification

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

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

Changes Since 6.0.0

- **Driver Enhancements**
  - The driver has been enhanced to comply with FIPS standards for data encryption. As part of this enhancement, the driver was tested with FIPS 140-3 enabled using a Red Hat OpenJDK 21 on a Red Hat Universal Base Image 9 instance. See FIPS (Federal Information Processing Standard) on page 88 for details.
  - The driver has been enhanced to allow you to specify the timezone to be used when translating data store time and timestamp values between the client and driver. You can configure this behavior using the new ClientTimeZone connection property. See ClientTimeZone on page 126 for details.
  - The InValuesLimit connection property has been added to the driver. It determines the maximum number of values in an `IN` clause subquery to be materialized. See InValuesLimit on page 141 for details.
  - The driver has been enhanced to support JSON web token (JWT) grant authentication. See Configuring the driver to use OAuth 2.0 with JWT grant on page 87 for details.
  - The driver has been enhanced to support the Salesforce Bulk API V2 for bulk load operations. This functionality can be enabled and configured with the BulkLoadVersion and BulkLoadJobSize connection properties. See Bulk API Properties on page 63 for details.
  - The ApplyToLabel connection property has been added to the driver. It determines whether the driver applies the toLabel() function to the picklist fields when executing queries. See ApplyToLabel on page 115 for details.
  - `yes`, `no`, `on` and `off` have been added as valid values for the connection properties that accept boolean values.

- The driver has been enhanced to support OAuth 2.0 authentication. See Configuring OAuth 2.0 authentication on page 83 for details.

- **Changed Behavior**
  - The connection property `SpyAttributes` has been updated to exclude the attribute `load=classname`, which was previously used to load the driver specified by the given class name. See SpyAttributes on page 157 for details.
  - The configuration options have been deprecated. However, the driver will continue to support them until the next major release of the driver.
  - The precision for the Integer data type has been changed from `10` to `9`.
  - The NUM_PREC_RADIX value for the Double data type has been changed from `10` to `2`.

Changes for 6.0.0

- **Driver Enhancements**
  - The driver has been enhanced to support the Salesforce Bulk API V1, including PK chunking, for bulk fetch operations. This functionality can be enabled and configured with the EnableBulkFetch, BulkFetchThreshold, EnablePKChunking, and PKChunkSize connection properties. See Bulk API Properties on page 63 for details.
  - The driver has been enhanced to support multiple simultaneous sessions. The number of active sessions should not exceed the number permitted by your Salesforce account and can be limited by the setting of the WSPoolSize connection property.
  - The new WSPoolSize connection property allows you to specify the maximum number of sessions the driver uses. This allows the driver to have multiple Web service requests active simultaneously when multiple JDBC connections are open, thereby improving throughput and performance. See WSPoolSize on page 164 for details.
  - The Refresh Map SQL extension has been added to the driver. REFRESH MAP discovers native objects that have been added to the native data store since connection or since the last refresh and maps the objects into your relational view of native data. It also incorporates any configuration changes made to your relational view by reloading the schema map and associated files. See Refresh Map (EXT) on page 218 for details.

- **Changed Behavior**
  - The data source class `com.ddtek.jdbcx.sforce.SForceDataSource40` has been deprecated. The data source class `com.ddtek.jdbcx.sforce.SForceDataSource` should be used for data source connections. The data source class `com.ddtek.jdbcx.sforce.SForceDataSource` now supports all JDBC specifications. See also Data Source and Driver Classes on page 15.
  - In addition to the information listed here, refer to this compatibility FAQ for guidance on upgrading from the Progress DataDirect *for* JDBC for Salesforce 5.1 driver to the 6.0 driver.
  - The driver's SQL engine was upgraded for this release. Consequently, there are differences in how the driver handles some SQL queries. Refer to this SQL engine upgrade document for details. See also Supported SQL Statements and Extensions on page 181.
  - The 6.0 driver pushes queries to Salesforce whenever possible. Queries that cannot be pushed to Salesforce with the 6.0 driver may be slower than comparable queries made with previous versions of the driver because data may be paged to disk while completing an operation. If you experience slow performance, please contact Technical Support. Our team will quickly address any performance issues you encounter.
  - Bulk load operations are no longer restricted to 10,000 rows for evaluation installations of the driver.

- The native CURRENCY and PERCENTAGE data types now map to the DECIMAL JDBC data type. In earlier releases, these data types mapped to the DOUBLE data type. See Data Types on page 17 details.
- The DatabaseName property has been deprecated. The SchemaMap property should now be used to specify the fully qualified path of the configuration file where the map of the Salesforce data model is written. See SchemaMap on page 153 for details.
- The CreateDB and RefreshSchema properties have been deprecated. The CreateMap property should now be used to specify whether the driver creates a new schema map when establishing the connection. See CreateMap on page 135 for details.
- The RefreshDirtyCache property has been deprecated. Now, for every fetch operation, the driver refreshes the cached object to pick up changes made to tables and rows.
- The following connection properties and configuration options now have new default settings.
  - The default value of the EnableBulkLoad connection property has been updated to `true`. By default, the bulk load protocol can be used for inserts, updates, and deletes based on the BulkLoadThreshold property. See EnableBulkLoad on page 136 for details.
  - The default value for the StmtCallLimit connection property has been updated to `100`. By default, the driver can make a maximum of 100 Web service calls when executing any single SQL statement or metadata query. See StmtCallLimit on page 159 for details.
  - The default value for the AuditColumns configuration option has been updated to `all` (`AuditColumns=all`). By default, the driver includes all audit columns and the master record id column in its table definitions. See ConfigOptions on page 127 for details.
  - The default value for the CustomSuffix configuration option has been updated to `include` (`CustomSuffix=include`). By default, the driver includes the "__c" suffix table and column names when mapping the Salesforce data model. See ConfigOptions on page 127 for details.
  - The default value for the MapSystemColumnName configuration option has been updated to `0` (`MapSystemColumnNames=0`). By default, the driver does not change the names of the Salesforce system columns when mapping the Salesforce data model. See ConfigOptions on page 127 for details.

Requirements

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

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

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 44 for details.

Version String Information

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

M.m.s.bbbbbb(C*XXXX*.FY*YYYYY*.U*ZZZZZZ*)

where:

M is the major version number.

m is the minor version number.

s is the service pack number.

bbbbbb is the driver build number.

XXXX is the cloud adapter build number.

YYYYYY is the framework build number.

ZZZZZZ is the utl build number.

For example:

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

Data Source and Driver Classes

The `Driver` class for the driver is:

`com.ddtek.jdbc.sforce.SForceDriver`

The `DataSource` class for the driver is:

`com.ddtek.jdbcx.sforce.SForceDataSource`

Connection URL

After setting the CLASSPATH, the required connection information needs to be passed in the form of a connection URL.

```
jdbc:datadirect:sforce://servername;User=username;Password=password;
  SecurityToken=value[;property=value[;...]]
```

Note:

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

---

where:

servername

specifies the base Salesforce URL to use for logging in. The default is `login.salesforce.com`.

`User`

specifies the user name that is used to connect to the Salesforce instance.

`Password`

specifies the password to use to connect to your Salesforce instance. A security token may be appended to the password. See "Password" for details.

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

`SecurityToken`

specifies the security token required to make a connection to a Salesforce instance that is configured for a security token. If a security token is required and you do not supply one, the driver returns an error indicating that an invalid user or password was supplied. A security token may be appended to the password. See "SecurityToken" for details.

Important: If setting the security token using a data source, be aware that the SecurityToken property, like all data source properties, is persisted in clear text.

The following examples show how to establish a connection to a Salesforce instance.

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:sforce://login.salesforce.com;User=abc@defcorp.com;
   Password=secret;SecurityToken=XaBARTsLZReM4Px47qPLOS");
```

See also

Using Connection Properties on page 56

Connection Properties

The driver includes over 40 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 and optimizing performance. You can specify connection properties in a connection URL or within a JDBC data source object.

See also

Using Connection Properties on page 56
Connection Property Descriptions on page 109

Mapping Objects to Tables

The driver automatically maps Salesforce objects and fields to tables and columns the first time it connects to a Salesforce instance. The driver maps both standard and custom objects and includes any relationships defined between objects. You can use the `getPrimaryKeys`, `getExportedKeys`, and `getImportedKeys` methods to report relationships among objects.

Schema Map

The driver uses a local schema map to instantiate the mapping of the remote Salesforce data model as tables and the metadata associated with those tables. By default, the driver creates a schema map for each user. The schema map is created in each user's application data folder and uses the user ID specified for the connection as the name of the schema map file. If the user ID contains punctuation or other non-alphanumeric characters, the driver strips those characters from the user ID to form the name of the schema map file. The driver includes a SchemaMap connection property that allows you to override the default setting for the name and location of the schema map file.

Identifiers

When mapping the remote data model, the driver converts unquoted identifiers to uppercase by default. You can use the UppercaseIdentifiers configuration option to map identifiers to use the case of the remote object.

Audit Columns

Salesforce adds audit fields to all the objects defined in a Salesforce instance. By default, the driver includes corresponding audit columns in table definitions when mapping the remote data model. The AuditColumns configuration option can be used to exclude or limit audit columns.

Custom Objects and Fields

Salesforce appends custom objects and fields with a standard "__c" suffix. By default, the driver includes the standard "__c" suffix when mapping the remote data model. You can use the CustomSuffix configuration option to strip the "__c" suffix.

System Fields

Salesforce includes system fields in a Salesforce instance. By default, the driver uses the name of the system field when mapping the system fields as columns. You can use the MapSystemColumnNames configuration option to make it evident that the columns in a table correspond to system fields.

See also

SchemaMap on page 153
Using Identifiers on page 88
ConfigOptions on page 127
UppercaseIdentifiers (Configuration Option) on page 132
AuditColumns (Configuration Option) on page 128
CustomSuffix (Configuration Option) on page 129
MapSystemColumnNames (Configuration Option) on page 130

Data Types

The following table lists the data types supported by the driver, how the Salesforce data types are exposed in the Salesforce Web Service API, and how these data types are mapped to JDBC data types.

Table 1: Salesforce Data Types

Salesforce Data Type	Web Service API Data Type	JDBC Data Type
ANYTYPE[1]	anytype	VARCHAR
AUTONUMBER	string	VARCHAR
BINARY[1]	binary	LONGVARBINARY
CHECKBOX	boolean	BOOLEAN
COMBOBOX[1]	combobox	VARCHAR
CURRENCY Formula (CURRENCY)	currency	DECIMAL
DATACATEGORYGROUPREFERENCE	DataCategoryGroupReference	VARCHAR
DATE Formula (DATE)	date	DATE
DATETIME Formula (DATETIME)	datetime	DATETIME
EMAIL	email	VARCHAR
ENCRYPTEDTEXT	encryptedtext	VARCHAR
HTML	html	VARCHAR
ID	id	LONGVARCHAR
INT[2]	double	INTEGER or DOUBLE
LONGTEXTAREA	longtextarea	LONGVARCHAR
MULTISELECTPICKLIST	multipicklist	VARCHAR
NUMBER[3]	double	INTEGER or DOUBLE
PERCENT Formula (PERCENT)	percent	DECIMAL
PHONE	phone	VARCHAR
PICKLIST	picklist	VARCHAR
REFERENCE	reference	VARCHAR
TEXT Formula (TEXT)	string	VARCHAR

[1] You cannot create columns with this data type using the Create Table and AlterTable statements.

[2] If the NumberFieldMapping key of the ConfigOptions property is set to `emulateInteger`, this data type maps to INTEGER. If set to `alwaysDouble`, this data type maps to DOUBLE.

[3] If scale = 0 and precision <= 9 and the NumberFieldMapping key of the ConfigOptions property is set to `emulateInteger`, this data type maps to INTEGER. If scale does not = 0, precision > 9, or the NumberFieldMapping key of the ConfigOptions property is set to `alwaysDouble`, this data type maps to DOUBLE.

Salesforce Data Type	Web Service API Data Type	JDBC Data Type
TEXTAREA[4]	textarea	VARCHAR or LONGVARCHAR
TIME[1]	time	TIME
URL	url	VARCHAR

getTypeInfo()

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

Table 2: getTypeInfo() Results

TYPE_NAME = AnyType AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = ANYTYPE 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
TYPE_NAME = AutoNumber AUTO_INCREMENT = true CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = AUTONUMBER MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 0 NUM_PREC_RADIX = NULL PRECISION = 30 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[4] For searchable columns, this data type maps to VARCHAR. For non-searchable columns, it maps to LONGVARCHAR.

TYPE_NAME = Binary AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = BINARY MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 5242880 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = CheckBox AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 16 (BOOLEAN) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = CHECKBOX MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 0 NUM_PREC_RADIX = NULL PRECISION = 1 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = ComboBox AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = COMBOBOX 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

TYPE_NAME = Currency AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision*, *scale* DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = CURRENCY MAXIMUM_SCALE = 31	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 31 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = DataCategoryGroupReference AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) 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 = 255 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = Date AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' 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 = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) 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 = Email AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = EMAIL MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 80 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = EncryptedText AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ’ LITERAL_SUFFIX = ’ LOCAL_TYPE_NAME = EncryptedText MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 175 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = HTML AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = HTML MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32000 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = ID AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = ID MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 0 NUM_PREC_RADIX = NULL PRECISION = 18 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = LongTextArea AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = LONGTEXTAREA MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32000 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = MultiSelectPickList AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = MULTISELECTPICKLIST 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
TYPE_NAME = Number AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision*, *scale* DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = NUMBER 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 = Percent AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision*, *scale* DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = PERCENT MAXIMUM_SCALE = 31	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 31 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = Phone AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = PHONE MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 40 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = PickList AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = PICKLIST 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
TYPE_NAME = Reference AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = REFERENCE MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 18 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = Text AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = TEXT MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = TextArea AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = TEXTAREA 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

TYPE_NAME = Time AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 92 (TIME) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = TIME MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 8 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = URL AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = URL 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

SQL escape sequences

The driver supports the following SQL escape sequences.

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

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

Supported scalar functions

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

{fn *scalar-function*}

where:

scalar-function

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

Example:

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

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

Table 3: Supported Scalar Functions

String Functions	Numeric Functions	Timedate Functions	System Functions
ASCII	ABS	CURDATE	CURSESSIONID
BIT_LENGTH	ACOS	CURRENT_DATE	DATABASE
CHAR	ASIN	CURRENT_TIME	IDENTITY
CHAR_LENGTH	ATAN	CURRENT_TIMESTAMP	USER
CHARACTER_LENGTH	ATAN2	CURTIME	
CONCAT	BITAND	DATEDIFF	
DIFFERENCE	BITOR	DATE_ADD	
HEXTORAW	BITXOR	DATE_SUB	
INSERT	CEILING	DAY	
LCASE	COS	DAYNAME	
LEFT	COT	DAYOFMONTH	
LENGTH	DEGREES	DAYOFWEEK	
LOCATE	EXP	DAYOFYEAR	
LOCATE_2	FLOOR	EXTRACT	
LOWER	LOG	HOUR	
LTRIM	LOG10	MINUTE	
OCTET_LENGTH	MOD	MONTH	
RAWTOHEX	PI	MONTHNAME	
REPEAT	POWER	NOW	
REPLACE	RADIANS	QUARTER	
RIGHT	RAND	SECOND	
RTRIM	ROUND	SECONDS_SINCE_MIDNIGHT	
SOUNDEX	ROUNDMAGIC	TIMESTAMPADD	
SPACE	SIGN	TIMESTAMPDIFF	
SUBSTR	SIN	TO_CHAR	

String Functions	Numeric Functions	Timedate Functions	System Functions
SUBSTRING UCASE UPPER	SQRT TAN TRUNCATE	WEEK YEAR	

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.

Additional information

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

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

Contacting Technical Support

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

https://www.progress.com/support

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

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

- Your number or the serial number that corresponds to the product for which you are seeking support, or a case number if you have been provided one for your issue. If you do not have a SupportLink contract, the SupportLink representative assisting you will connect you with our Sales team.
- Your name, phone number, email address, and organization. For a first-time call, you may be asked for full information, including location.
- The Progress DataDirect product and the version that you are using.
- The type and version of the operating system where you have installed your product.
- Any database, database version, third-party software, or other environment information required to understand the problem.
- A brief description of the problem, including, but not limited to, any error messages you have received, what steps you followed prior to the initial occurrence of the problem, any trace logs capturing the issue, and so on. Depending on the complexity of the problem, you may be asked to submit an example or reproducible application so that the issue can be re-created.
- A description of what you have attempted to resolve the issue. If you have researched your issue on Web search engines, our Knowledgebase, or have tested additional configurations, applications, or other vendor products, you will want to carefully note everything you have already attempted.
- A simple assessment of how the severity of the issue is impacting your organization.

2

Getting Started

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

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

For details, see the following topics:

Setting the Classpath

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

`install_dir``/lib/60/sforce.jar`

Windows Example

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

UNIX Example

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

Data Source and Driver Classes

The `Driver` class for the driver is:

`com.ddtek.jdbc.sforce.SForceDriver`

The `DataSource` class for the driver is:

`com.ddtek.jdbcx.sforce.SForceDataSource`

Connecting Using the DriverManager

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

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:sforce://login.salesforce.com;User=abc@defcorp.com;
   Password=secret;SecurityToken=XaBARTsLZReM4Px47qPLOS");
```

Passing the Connection URL

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

Connection URL Syntax

```
jdbc:datadirect:sforce://servername;User=username;Password=password;
  SecurityToken=value[;property=value[;...]]
```

where:

servername

specifies the base Salesforce URL to use for logging in. The default is `login.salesforce.com`.

`User`

specifies the user name that is used to connect to the Salesforce instance.

`Password`

specifies the password to use to connect to your Salesforce instance. A security token may be appended to the password. See "Password" for details.

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

`SecurityToken`

specifies the security token required to make a connection to a Salesforce instance that is configured for a security token. If a security token is required and you do not supply one, the driver returns an error indicating that an invalid user or password was supplied. A security token may be appended to the password. See "SecurityToken" for details.

Important: If setting the security token using a data source, be aware that the SecurityToken property, like all data source properties, is persisted in clear text.

Connection URL Example

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:sforce://login.salesforce.com;User=abc@defcorp.com;
   Password=secret;SecurityToken=XaBARTsLZReM4Px47qPLOS");
```

See also

Password on page 147
SecurityToken on page 155
Using Connection Properties on page 56

Testing the Connection

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

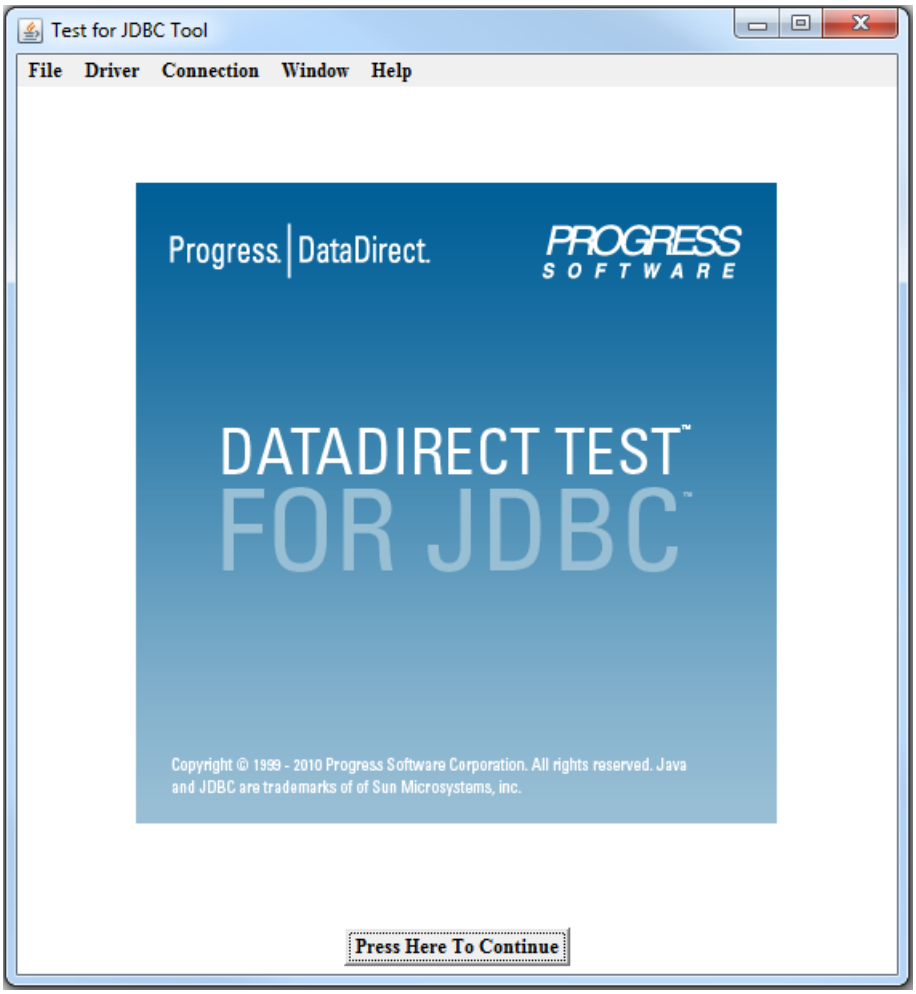
To test the connection from the driver to your data source, follow these steps:

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

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

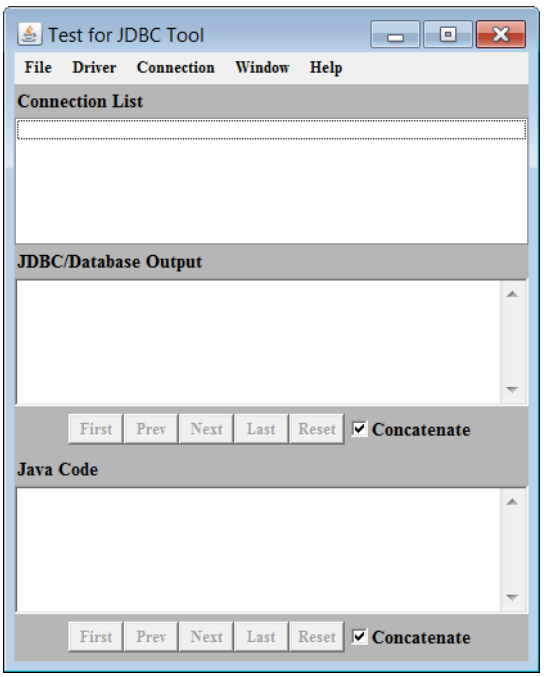
2. From the `testforjdbc` folder, run the platform-specific tool:
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

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



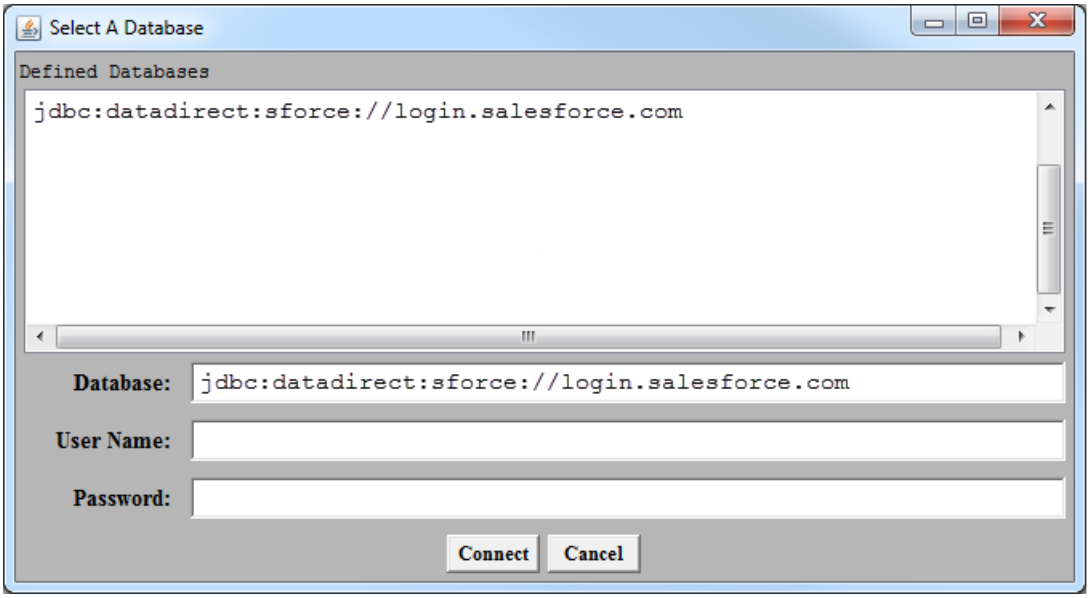
3. Click **Press Here to Continue**.

   The main dialog appears:



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

   The **Select A Database** dialog appears:



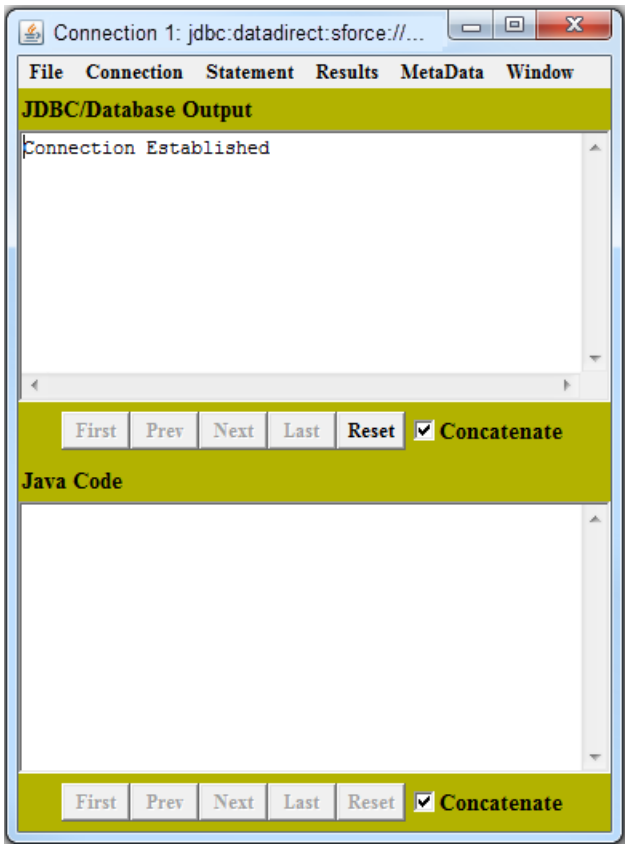
5. Select the appropriate database template from the **Defined Databases** field.
6. In the **Database** field, specify all required connection properties.

   For example:

```
jdbc:datadirect:sforce://login.salesforce.com;User=abc@defcorp.com;
   Password=secret;SecurityToken=XaBARTsLZReM4Px47qPLOS
```

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 using data sources

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

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

How Data Sources Are Implemented

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

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

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

Creating data sources

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

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

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

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

Example Data Source

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

```
import com.ddtek.jdbcx.SForce.SForceDataSource;
```

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

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

Note: In a JDBC data source, string values must be enclosed in double quotation marks, for example, setUser("abc@defcorp.com").

```
SForceDataSource mds = new SForceDataSource();
mds.setDescription("My Salesforce Datasource");
mds.setServerName("login.salesforce.com");
mds.setUser("abc@defcorp.com");
mds.setPassword("secret");
mds.setSecurityToken("XaBARTsLZReM4Px47qPLOS");
```

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

```
if (ds instanceof SForceDataSource)
{
SForceDataSource jmds = (SForceDataSource) ds;
System.out.println("description=" + jmds.getDescription());
System.out.println("serverName=" + jmds.getServerName());
System.out.println("user=" + jmds.getUser());
System.out.println("password=" + jmds.getPassword());
```

```
System.out.println("securityToken=" + jmds.getSecurityToken());
System.out.println();
}
```

Calling a data source in an application

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

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

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

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

Testing a data source connection

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

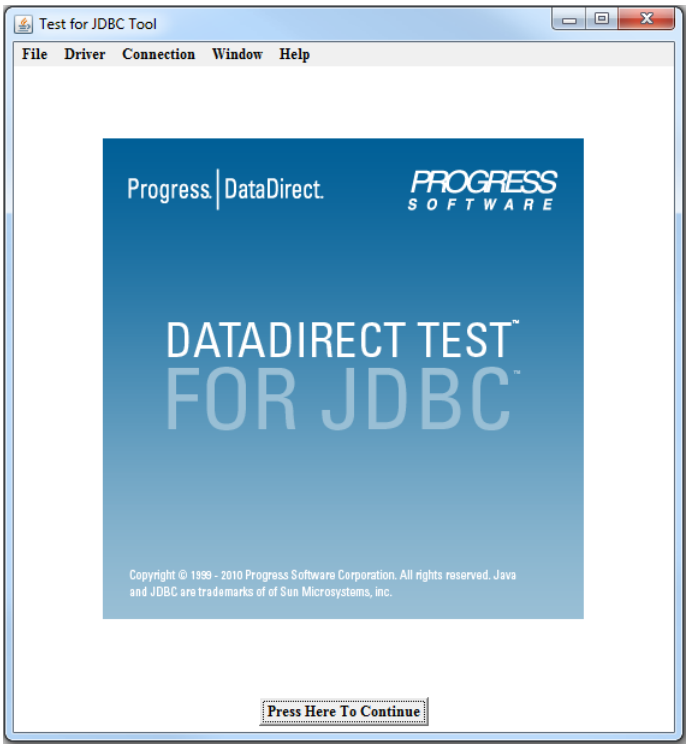
Take the following steps to establish a connection.

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

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

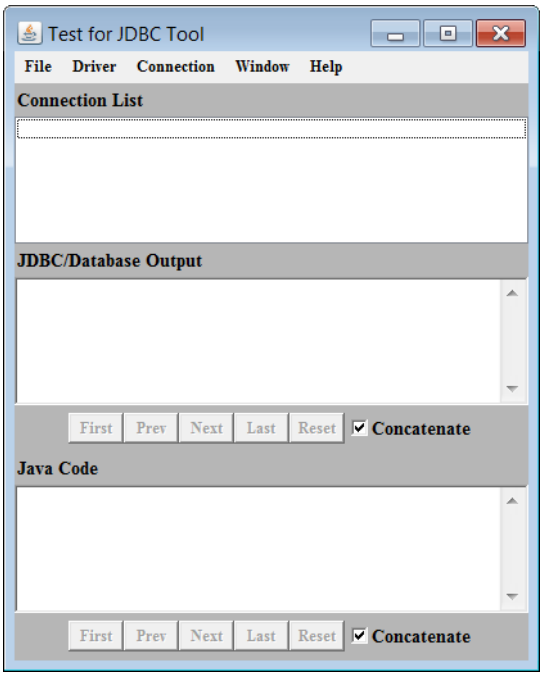
2. From the `testforjdbc` folder, run the platform-specific tool:
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

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



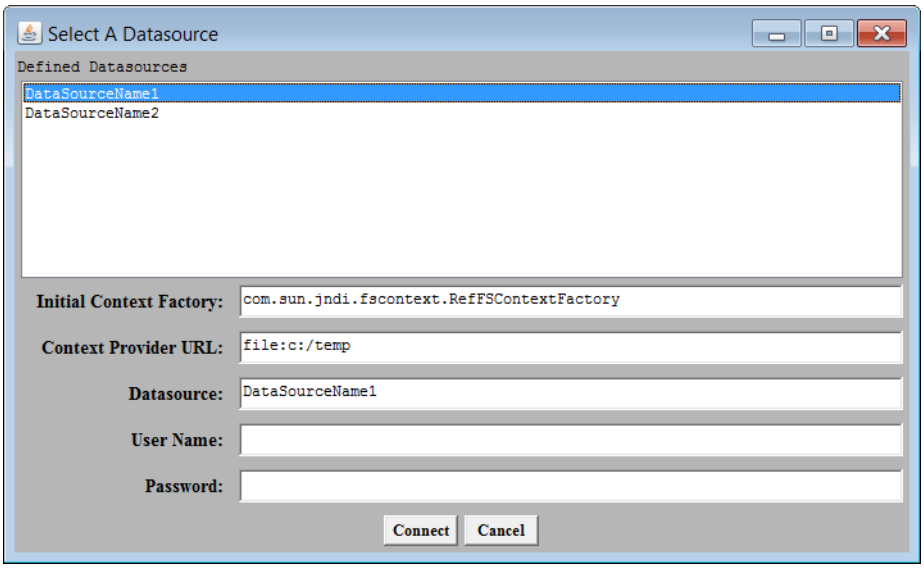
3. Click **Press Here to Continue**.

   The main dialog appears:



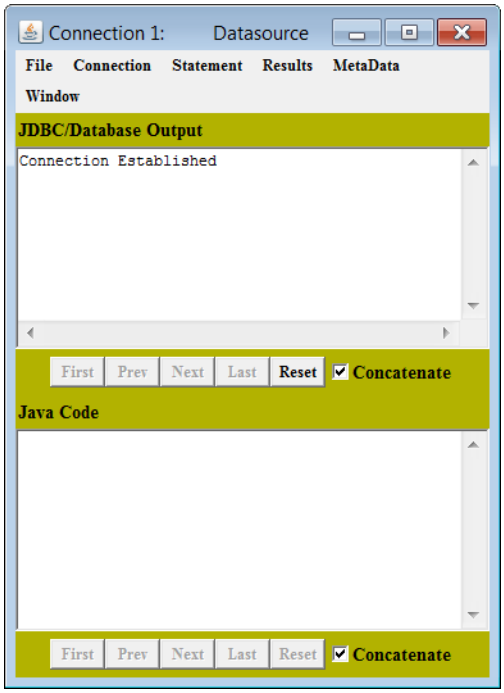
4. From the menu bar, select **Connection > Connect to DB via Data Source**.

   The **Select A Database** dialog appears:



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

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



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
- Using Connection Properties
- Connecting Through a Proxy Server
- Performance Considerations
- Client-Side Caches
- Catalog Tables
- Reports
- Authentication
- Data Encryption
- Using Identifiers
- Failover Support
- IP Addresses
- Statement pooling

- Connection pooling
- Timeouts
- Views and Remote/Local Tables
- Isolation Levels
- Scrollable cursors
- Unicode support
- Error Handling
- Large Object (LOB) Support
- Parameter Metadata Support
- ResultSet MetaData Support
- Rowset Support
- Auto-Generated Keys Support
- DataDirect Bulk Load
- CSV files
- Tracking JDBC calls with DataDirect Spy

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.

Permissions for bulk load from a CSV file

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

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

Connecting from an application

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

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

Setting the Classpath

Before you can connect, 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 `sforce.jar` file as shown, where *`install_dir`* is the path to your product installation directory:

```
install_dir/lib/60/sforce.jar
```

Windows Example

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

UNIX Example

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

Connecting Using the JDBC Driver Manager

One way to connect to a Salesforce instance 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 a data source:

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:sforce://login.salesforce.com;User=abc@defcorp.com;
   Password=secret;SecurityToken=XaBARTsLZReM4Px47qPLOS");
```

Passing the Connection URL

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

Connection URL Syntax

```
jdbc:datadirect:sforce://servername;User=username;Password=password;
  SecurityToken=value[;property=value[;...]]
```

where:

servername

specifies the base Salesforce URL to use for logging in. The default is `login.salesforce.com`.

`User`

specifies the user name that is used to connect to the Salesforce instance.

`Password`

specifies the password to use to connect to your Salesforce instance. A security token may be appended to the password. See "Password" for details.

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

`SecurityToken`

specifies the security token required to make a connection to a Salesforce instance that is configured for a security token. If a security token is required and you do not supply one, the driver returns an error indicating that an invalid user or password was supplied. A security token may be appended to the password. See "SecurityToken" for details.

Important: If setting the security token using a data source, be aware that the SecurityToken property, like all data source properties, is persisted in clear text.

Connection URL Example

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:sforce://login.salesforce.com;User=abc@defcorp.com;
   Password=secret;SecurityToken=XaBARTsLZReM4Px47qPLOS");
```

See also

Password on page 147
SecurityToken on page 155
Using Connection Properties on page 56

Testing the Connection

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

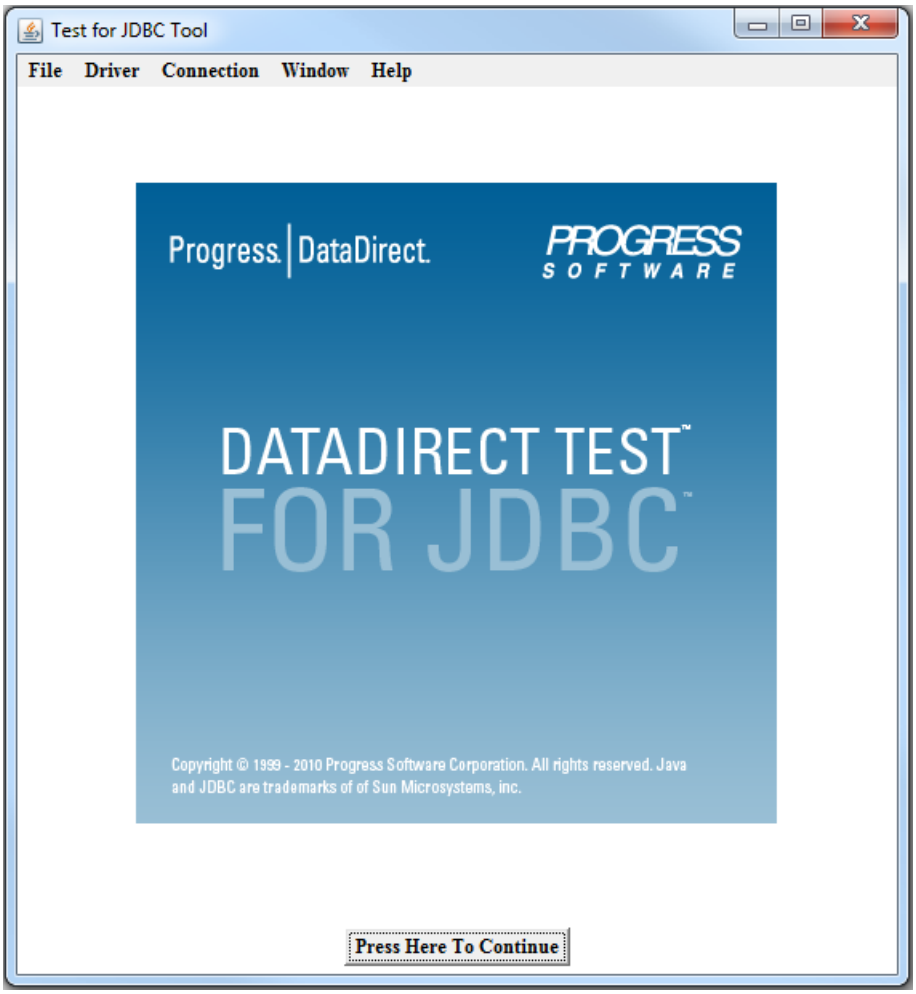
To test the connection from the driver to your data source, follow these steps:

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

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

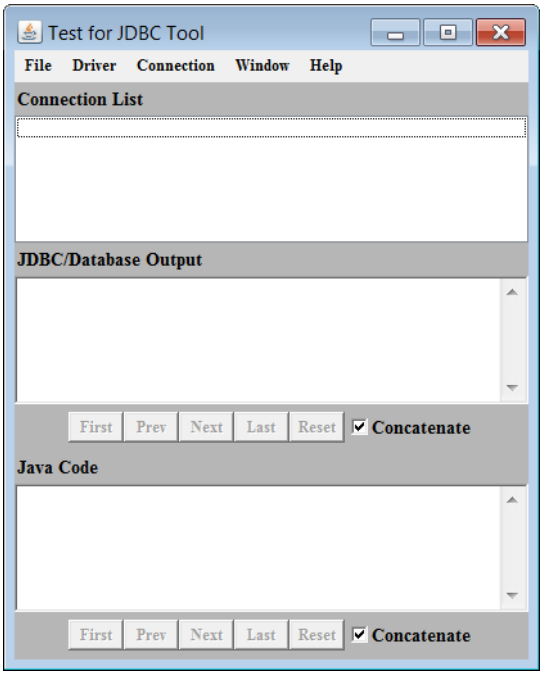
2. From the `testforjdbc` folder, run the platform-specific tool:
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

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



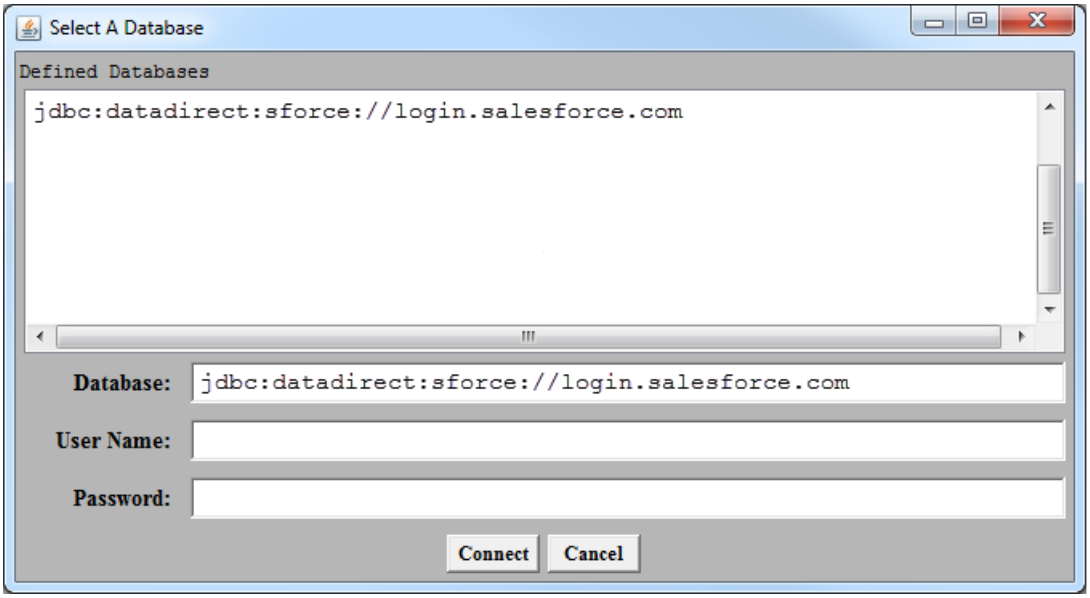
3. Click **Press Here to Continue**.

   The main dialog appears:



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

   The **Select A Database** dialog appears:



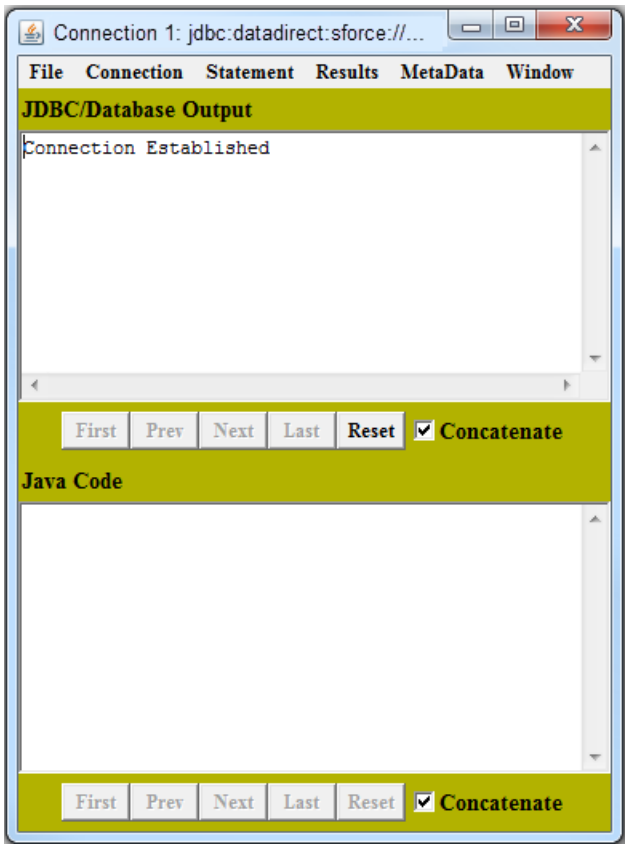
5. Select the appropriate database template from the **Defined Databases** field.
6. In the **Database** field, specify all required connection properties.

   For example:

```
jdbc:datadirect:sforce://login.salesforce.com;User=abc@defcorp.com;
   Password=secret;SecurityToken=XaBARTsLZReM4Px47qPLOS
```

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 using data sources

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

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

How Data Sources Are Implemented

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

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

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

Creating data sources

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

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

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

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

Example Data Source

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

```
import com.ddtek.jdbcx.SForce.SForceDataSource;
```

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

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

Note: In a JDBC data source, string values must be enclosed in double quotation marks, for example, setUser("abc@defcorp.com").

```
SForceDataSource mds = new SForceDataSource();
mds.setDescription("My Salesforce Datasource");
mds.setServerName("login.salesforce.com");
mds.setUser("abc@defcorp.com");
mds.setPassword("secret");
mds.setSecurityToken("XaBARTsLZReM4Px47qPLOS");
```

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

```
if (ds instanceof SForceDataSource)
{
SForceDataSource jmds = (SForceDataSource) ds;
System.out.println("description=" + jmds.getDescription());
System.out.println("serverName=" + jmds.getServerName());
System.out.println("user=" + jmds.getUser());
System.out.println("password=" + jmds.getPassword());
System.out.println("securityToken=" + jmds.getSecurityToken());
System.out.println();
}
```

Calling a data source in an application

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

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

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

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

Testing a data source connection

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

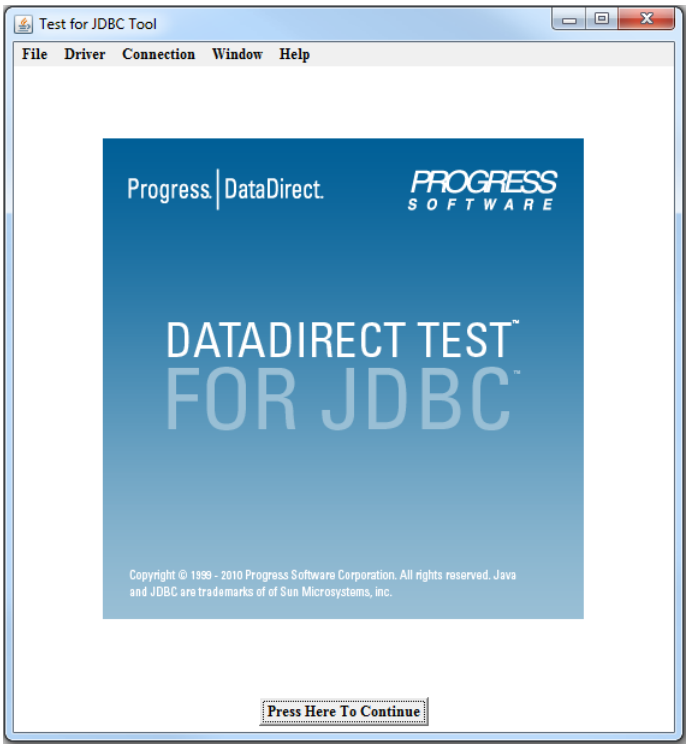
Take the following steps to establish a connection.

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

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

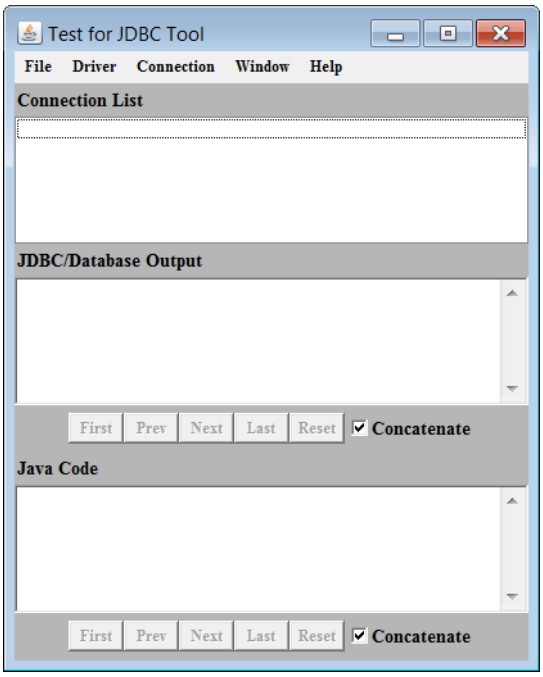
2. From the `testforjdbc` folder, run the platform-specific tool:
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

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



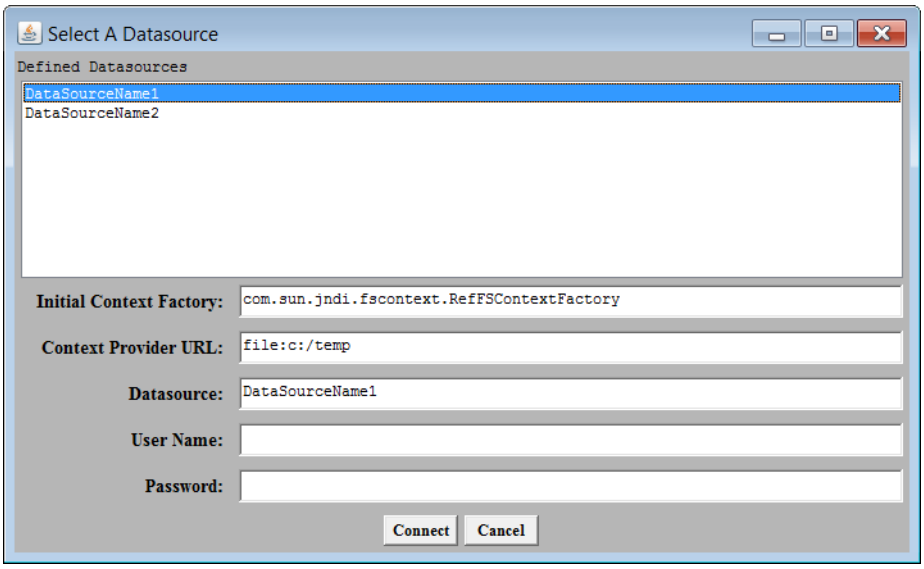
3. Click **Press Here to Continue**.

   The main dialog appears:



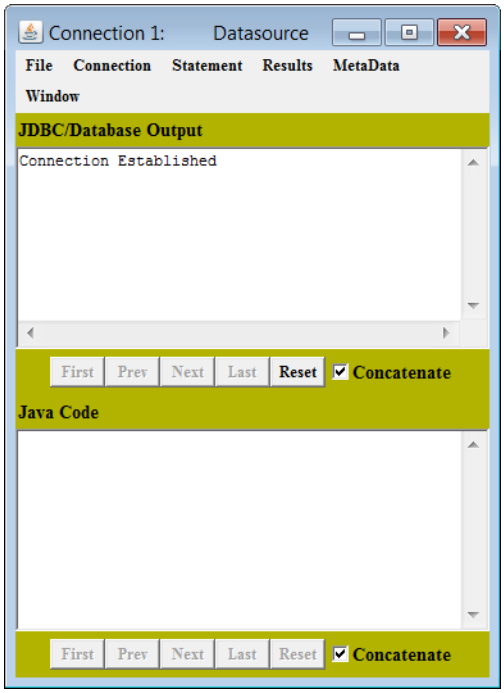
4. From the menu bar, select **Connection > Connect to DB via Data Source**.

   The **Select A Database** dialog appears:



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

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



Using Connection Properties

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

Note: Connection property names are case-insensitive. For example, `Password` is the same as `password`.

Note: In a JDBC data source, string values must be enclosed in double quotation marks, for example, `setUser("abc@defcorp.com")`.

See "Connection Property Descriptions" for an alphabetical list of connection properties and their descriptions.

See also

Connecting Using the JDBC Driver Manager on page 47
Connecting using data sources on page 36
Connection Property Descriptions on page 109

Required Properties

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

Table 4: Required Properties

Property	Characteristic
Password on page 147	Specifies the password to use to connect to your Salesforce 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.

Property	Characteristic
SecurityToken on page 155	Specifies the security token required to make a connection to a Salesforce instance that is configured for a security token. **Note:** A security token is not required when Salesforce has been configured for Trusted IP Ranges and the user is logging in from a trusted IP address. Refer to "Set Trusted IP Ranges for Your Organization" in your Salesforce documentation for details. **Note:** If setting the security token using a data source, be aware that the SecurityToken property, like all data source properties, is persisted in clear text.
ServerName on page 156	Specifies the base Salesforce URL to use for logging in. This value is as follows: • Production environments: `login.salesforce.com` • Sandbox instances: `test.salesforce.com` The default is `login.salesforce.com`.
User on page 161	Specifies the user name that is used to connect to the Salesforce instance.

See also

Connection Property Descriptions on page 109

Mapping Properties

The following table summarizes connection properties involved in mapping the remote Salesforce data model to a local schema map used to support SQL queries against Salesforce.

Table 5: Mapping Properties

Property	Characteristic
ConfigOptions on page 127	Determines how the mapping of the remote Salesforce data model to a local schema map can be configured, customized, and updated. The default is: `(AuditColumns=all;CustomSuffix=include; KeywordConflictSuffix=; MapSystemColumnNames=0; NumberFieldMapping=emulateInteger; UppercaseIdentifiers=true)`

Property	Characteristic
CreateMap on page 135	Specifies whether the driver creates a new map of the Salesforce data model when establishing the connection. If set to `forceNew`, the driver deletes the current schema map specified by the SchemaMap property and creates a new one at the same location. If set to `notExist`, the driver uses the current schema map specified by the SchemaMap property. If one does not exist, the driver creates one. If set to `no`, the driver uses the current schema map specified by the SchemaMap property. If one does not exist, the connection fails. The default is `notExist`.
SchemaMap on page 153	Specifies the fully qualified path of the configuration file where the map of the Salesforce data model is written. The driver looks for this file when connecting to a Salesforce instance. If the file does not exist, the driver creates one. The default varies based on your environment: For Windows: *application_data_folder*`\Local\Progress\DataDirect\SForce_Schema\`*user_name*`.config` For UNIX/Linux: `~/progress/datadirect/SForce_Schema/`*user_name*`.config`

See also

Mapping Objects to Tables on page 16
Connection Property Descriptions on page 109

Authentication Properties

The following table summarizes connection properties which are required for user ID and password authentication.

Table 6: Authentication Properties

Property	Characteristic
AccessToken on page 114	Specifies the access token required to authenticate to a Salesforce instance when OAuth 2.0 is enabled (`AuthenticationMethod=oauth2.0`). **Note:** If a value for the AccessToken property is not specified, the driver uses the value of the RefreshToken property to make a connection. If both values are not specified, the driver cannot make a successful connection. If both are specified, the driver ignores the AccessToken value and uses the RefreshToken value to generate a new AccessToken value.

Property	Characteristic
AuthenticationMethod on page 116	Determines which authentication method the driver uses when establishing a connection. If set to `userIDPassword`, the driver uses user ID/password authentication when establishing a connection. If set to `oauth2.0`, the driver uses OAuth 2.0 authentication when establishing a connection. The default is `userIDPassword.`
ClaimsIssuer on page 123	Specifies the consumer key or the client ID of the Salesforce Connected App.
ClaimsSubject on page 124	Specifies the username of the user.
ClientID on page 124	Specifies the consumer key for your application. The driver uses this value to authenticate to a Salesforce instance when OAuth 2.0 is enabled (`AuthenticationMethod=oauth2.0`).
ClientSecret on page 125	Specifies the consumer secret for your application. This value can optionally be specified when authenticating to a Salesforce instance using OAuth 2.0 (`AuthenticationMethod=oauth2.0`).
JWTCertAlias on page 143	Specifies the alias for the JWT certificate located in the keystore.
JWTCertStore on page 144	Specifies the file path of the certificate store containing the private key used for JWT authentication.
JWTCertPassword on page 143	Specifies the password of the JWT certificate, if the certificate is password protected.
Password on page 147	Specifies the password to use to connect to your Salesforce 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.

Property	Characteristic
RefreshToken on page 152	Specifies the refresh token used to either request a new access token or renew an expired access token. When the refresh token is specified, the access token generated at connection is used to authenticate to a Salesforce instance when OAuth 2.0 is enabled (`AuthenticationMethod=oauth2.0`). **Note:** If a value for the AccessToken property is not specified, the driver uses the value of the RefreshToken property to make a connection. If both values are not specified, the driver cannot make a successful connection. If both are specified, the driver ignores the AccessToken value and uses the RefreshToken value to generate a new AccessToken value.
SecurityToken on page 155	Specifies the security token required to make a connection to a Salesforce instance that is configured for a security token. **Note:** A security token is not required when Salesforce has been configured for Trusted IP Ranges and the user is logging in from a trusted IP address. Refer to "Set Trusted IP Ranges for Your Organization" in your Salesforce documentation for details. **Note:** If setting the security token using a data source, be aware that the SecurityToken property, like all data source properties, is persisted in clear text.
User on page 161	Specifies the user name that is used to connect to the Salesforce instance.

See also

Connection Property Descriptions on page 109

Failover Properties

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

Table 7: Failover Properties

Property	Characteristic
ConnectionRetryCount on page 132	Specifies the number of times the driver retries connection attempts to the primary database server, and if specified, alternate servers before returning a connection failure. If set to `0`, the driver does not try to reconnect after the initial unsuccessful attempt. If set to *x*, the driver retries connection attempts the specified number of times. If a connection is not established during the retry attempts, the driver returns an exception that is generated by the last database server to which it tried to connect. The default is `5`.
ConnectionRetryDelay on page 133	Specifies the number of seconds the driver waits between connection retry attempts when ConnectionRetryCount is set to a positive integer. If set to `0`, the driver does not delay between retries. If set to *x*, the driver waits between connection retry attempts the specified number of seconds. The default is `1`.

See also

- Connection Property Descriptions on page 109

Proxy Server Properties

The following table summarizes proxy server connection properties.

Table 8: Proxy Server Properties

Property	Characteristic
ProxyHost on page 149	Identifies a proxy server to use for the first connection. This can be a named server or an IP address.
ProxyPassword on page 149	Specifies the password needed to connect to a proxy server for the first connection.
ProxyPort on page 150	Specifies the port number where the proxy server is listening for HTTP or HTTPS requests for the first connection. The default is `0`.
ProxyUser on page 150	Specifies the user name needed to connect to a proxy server for the first connection.

See also

- Connection Property Descriptions on page 109
- Connecting Through a Proxy Server on page 72

Web Service Properties

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

Table 9: Web Service Properties

Property	Characteristic
LoginTimeout on page 145	The amount of time, in seconds, that the driver waits for a connection to be established before timing out the connection request. 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. The default is `0` (no timeout).
StmtCallLimit on page 159	Specifies the maximum number of Web service calls the driver can make when executing any single SQL statement or metadata query. If set to `0`, there is no limit. If set to *x*, the driver uses this value to set the maximum number of Web service calls on a single connection that can be made when executing a SQL statement. This limit can be overridden by changing the STMT_CALL_LIMIT session attribute using the ALTER SESSION statement. The default is `100` (Web service calls).
StmtCallLimitBehavior on page 160	Specifies the behavior of the driver when the maximum Web service call limit specified by the StmtCallLimit property is exceeded. 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. The default is `errorAlways`.
WSCompressData on page 162	Specifies whether the driver compresses data it sends to or receives from the Web server. 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. The default is `compress`.

Property	Characteristic
WSFetchSize on page 163	Specifies the number of rows of data the driver attempts to fetch for each JDBC call. If set to `0`, the driver attempts to fetch up to a maximum of 2000 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 2000 can reduce the response time for returning the initial data. Consider using a smaller WSFetch size for interactive applications only.
WSPoolSize on page 164	Specifies the maximum number of Salesforce sessions the driver uses. This allows the driver to have multiple web service requests active when multiple ODBC connections are open, thereby improving throughput and performance. The default is `1`.
WSRetryCount on page 164	The number of times the driver retries a timed-out Select request. Insert, Update, and Delete requests are never retried. 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. The default is `0`.
WSTimeout on page 165	Specifies the time, in seconds, that the driver waits for a response to a Web service request. 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. The default is `120`.

See also

- Connection Property Descriptions on page 109
- Performance Considerations on page 72
- Timeouts on page 91

Bulk API Properties

The following table summarizes connection properties that are used to implement the Salesforce Bulk API for selects, inserts, updates, and deletes.

Table 10: Bulk API Properties

Property	Description
BulkFetchThreshold on page 116	Specifies a number of rows that, if exceeded, signals the driver to use the Salesforce Bulk API for select operations. For this behavior to take effect, the EnableBulkFetch property must be set to `true`. If set to `0`, the driver uses the Salesforce Bulk API for all select operations. If set to *x*, the driver uses the Salesforce Bulk API for select operations when the value of *x* is exceeded. The default is `30000` (rows).
BulkLoadAsync on page 117	Determines whether the driver treats bulk load operations as synchronous or asynchronous. If set to `0`, bulk load operations are synchronous. The driver does not return from the method that invoked an operation until the operation has completed or the operation has timed out. If the operation times out, the driver throws an exception. If set to `1`, bulk load operations are asynchronous. The driver returns from the method that invoked an operation immediately after the operation is submitted to the server. The driver does not verify that the bulk load operation was completed. The default is `0` (synchronous).
BulkLoadBatchSize on page 118	Provides a suggestion to the driver for the number of rows to load to the database at a time when bulk loading data. The default is `10000`.
BulkLoadConcurrencyMode on page 119	Determines whether multiple batches associated with a bulk load operation are processed by Salesforce in parallel or one at a time. If set to `parallel`, multiple batches associated with a bulk load operation are processed in parallel. The order in which the batches are processed can vary. If set to `serial`, multiple batches associated with a bulk load operation are processed one at a time. The default is `parallel`.
BulkLoadJobSize on page 119	Determines the number of rows to load into a single job of a bulk operation when the connection property BulkLoadVersion is set to V2. Performance can be improved by increasing the number of rows the driver loads at a time because fewer network round trips are required. Increasing the number of rows also causes the driver to consume more memory on the client. The default is `150000` (rows).

Property	Description
BulkLoadPollInterval on page 120	Specifies the number of seconds the driver waits to request bulk operation status. This interval is used by the driver the first time it requests status and for all subsequent status requests. The default is `10` (seconds).
BulkLoadThreshold on page 121	Specifies a number of rows that, if exceeded, signals the driver to use the Salesforce Bulk API for inserts, updates, and deletes. For this behavior to take effect, the EnableBulkLoad property must be set to `true`. If set to `0`, the driver uses the Salesforce Bulk API for all inserts, updates, and deletes. If set to *x*, the driver uses the Salesforce Bulk API to execute the insert, update, or delete operation when the value of *x* is exceeded. The default is `4000` (rows).
BulkLoadVersion on page 122	Specifies which version of Salesforce Bulk Load API to use for performing bulk load operations. This is applicable if EnableBulkLoad is set to `true`. If set to `V1`, the driver uses the Salesforce Bulk API V1 to execute the insert, update, and delete operations. If set to `V2`, the driver uses the Salesforce Bulk API V2 to execute the insert, update, or delete operations. The default is `V1`.
EnableBulkFetch on page 136	Specifies whether the driver can use the Salesforce Bulk API for selects based on the value of the BulkFetchThreshold connection property. If the number of rows expected in the result set exceeds the value of BulkFetchThreshold property, the driver uses the Salesforce Bulk API to execute the select operation. Using the Salesforce Bulk API may significantly reduce the number of Web service calls used to execute a statement and, therefore, may improve performance. If set to `true`, the driver can use the Salesforce Bulk API for selects based on the value of the BulkFetchThreshold connection property. If the number of rows expected in the result set exceeds the value of BulkFetchThreshold property, the driver uses the Salesforce Bulk API to execute the select operation. If set to `false`, the driver does not use the Salesforce Bulk API, and the BulkFetchThreshold property is ignored. The default is `true`.

Property	Description
EnableBulkLoad on page 136	Specifies whether the driver can use the Salesforce Bulk API for inserts, updates, and deletes based on the value of the BulkLoadThreshold connection property. If the number of affected rows exceeds the value of BulkLoadThreshold property, the driver uses the Salesforce Bulk API to execute the insert, update, or delete operation. Using the Salesforce Bulk API may significantly reduce the number of Web service calls used to execute a statement and, therefore, may improve performance. If set to `true`, the driver can use the Salesforce Bulk API for inserts, updates, and deletes based on the value of the BulkLoadThreshold connection property. If the number of affected rows exceeds the value of BulkLoadThreshold property, the driver uses the Salesforce Bulk API to execute the insert, update, or delete operation. If set to `false`, the driver does not use the Salesforce Bulk API, and the BulkLoadThreshold property is ignored. The default is `true`.
EnablePKChunking on page 137	Specifies whether the driver uses PK chunking for select operations. PK chunking breaks down bulk fetch operations into smaller, more manageable batches for improved performance. If set to `true`, the driver uses PK chunking for select operations when the expected number of rows in the result set is greater than the values of the BulkFetchThreshold and PKChunkSize properties. For this behavior to take effect, the EnableBulkFetch property must also be set to `true`. If set to `false`, the driver does not use PK chunking when executing select operations, and the PKChunkSize property is ignored. The default is `true`.
PKChunkSize on page 148	Specifies the size, in rows, of a primary key chunk when PK chunking has been enabled via the EnablePKChunking property. The Salesforce Bulk API splits the query into chunks of this size. PKChunkSize may be set to a maximum value of `250000` rows. The default is `100000` (rows).

See also

DataDirect Bulk Load on page 96
Connection Property Descriptions on page 109

Timeout Properties

The following table summarizes timeout connection properties.

Table 11: Timeout Properties

Property	Characteristic
LoginTimeout on page 145	Specifies the amount of time, in seconds, that the driver waits for a connection to be established before timing out the connection request. 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. The default is `0`.
WSRetryCount on page 164	The number of times the driver retries a timed-out Select request. Insert, Update, and Delete requests are never retried. 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. The default is `0`.
WSTimeout on page 165	Specifies the time, in seconds, that the driver waits for a response to a Web service request. 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. The default is `120`.

See also

- Connection Property Descriptions on page 109
- Web Service Properties on page 62

Statement Pooling Properties

The following table summarizes statement pooling connection properties.

Table 12: Statement Pooling Properties

Property	Characteristic
ImportStatementPool on page 139	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.

Property	Characteristic
MaxPooledStatements on page 146	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. If set to `0`, the driver's internal prepared statement pooling is not enabled. If set to *x*, the driver enables the DataDirect Statement Pool and uses the specified value to cache a certain number of 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. The default is `0`.
RegisterStatementPoolMonitorMBean on page 153	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. 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. The default is `false`.

See also

- Refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference* for further details.
- Connection Property Descriptions on page 109

Additional Properties

The following table summarizes additional connection properties.

Table 13: Additional Properties

Property	Characteristic
ApplyToLabel on page 115	Determines whether the driver applies the toLabel() function to the picklist fields when executing queries. If set to `true`, the driver applies the toLabel() function to the picklist fields when executing queries. If set to `false`, the driver does not apply the toLabel() function to the picklist fields when executing queries. The default is `false`.
CatalogOptions on page 122	Determines which type of metadata information is included in result sets when an application calls DatabaseMetaData methods. If set to `2`, result sets do not contain synonyms. If set to `4`, a hint is provided to the driver to emulate getColumns() calls using the ResultSetMetaData object instead of querying database catalogs for column information. Result sets contain synonyms. Using emulation can improve performance because the SQL statement that is formulated by the emulation is less complex than the SQL statement that is formulated using getColumns(). The argument to getColumns() must evaluate to a single table. If it does not, because of a wildcard or null value, for example, the driver reverts to the default behavior for getColumns() calls. The default is `2`.
ClientTimeZone on page 126	Specifies the time zone other than UTC (Universal Time Coordinated) that should be used when translating data store time and timestamp values between the client and the driver. By default, the driver determines the client time zone based on the system-specific time zone settings.
ConvertNull on page 134	Controls how data conversions are handled for null values. 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. The default is `1`.

Property	Characteristic
FetchSize on page 138	Specifies the maximum number of rows that the driver processes before returning data to the application for a single fetch request when executing a Select. This value provides a suggestion to the driver as to the number of rows that should be returned to the application. The driver may fetch fewer rows to conserve memory when processing exceptionally wide rows. If set to `0`, the driver processes all the rows of the result before returning control to the application. When large data sets are being processed, setting FetchSize to `0` can diminish performance and increase the likelihood of out-of-memory errors. If set to *x*, the driver limits the number of rows that may be processed for each fetch request before returning control to the application.
InitializationString on page 140	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.
InsensitiveResultSetBufferSize on page 140	Determines the amount of memory used by the driver to cache insensitive result set data. 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. The default is `2048`.

Property	Characteristic
JavaDoubleToString on page 142	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. 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. The default is `false`.
LogConfigFile on page 145	Specifies the filename of the configuration file used to initialize driver logging. The default is `ddlogging.properties`.
ReadOnly on page 151	Specifies whether the connection supports read-only access to the data source. If set to `true`, the connection has read-only access. The following commands are the only commands that you can use when a connection if read-only: • Call* (if the procedure does not update data) • Explain Plan • Select (except Select Into) • Set Database Collation • Set IgnoreCase • Set Maxrows • Set Schema The driver generates an exception if any other command is executed. If set to `false`, the connection is opened for read/write access, and you can use all commands supported by the product. The default is `false`.

Property	Characteristic
SpyAttributes on page 157	Enables DataDirect Spy to log detailed information about calls issued by the driver on behalf of the application. DataDirect Spy is disabled by default.
TransactionMode on page 161	Specifies how the driver handles manual transactions. If set to `ignore`, the data source does not support transactions and the driver always operates in auto-commit mode. If set to `noTransactions`, the data source and the driver do not support transactions. The default is `noTransactions`.

See also

Connection Property Descriptions on page 109

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 sforce.jar com.acme.myapp.Main
```

Alternatively, you can use the ProxyHost, ProxyPort, ProxyUser, and ProxyPassword connection properties. See "Connection Property Descriptions" for details about these properties.

See also

Connection Property Descriptions on page 109

Performance Considerations

BulkLoadJobSize: The BulkLoadJobSize property can be used to improve performance by increasing the number of rows the driver loads at a time because fewer network round trips are required. However, a higher number of rows also causes the driver to consume more memory on the client.

EnableBulkFetch: The EnableBulkfetch property can be used to improve performance by enabling the driver to use the Salesforce Bulk API for selects. Using the Salesforce Bulk API may significantly reduce the number of Web service calls used to execute a statement and, therefore, may improve performance. When EnableBulkFetch has been set to `true`, the driver uses the Salesforce Bulk API based on the value of the BulkFetchThreshold connection property. If the number of rows expected in the result set exceeds the value of BulkFetchThreshold property, the driver uses the Salesforce Bulk API to execute the select operation.

EnableBulkLoad: The EnableBulkLoad property can be used to improve performance by enabling the driver to use the Salesforce Bulk API for inserts, updates, and deletes. Using the Salesforce Bulk API may significantly reduce the number of Web service calls used to execute a statement and, therefore, may improve performance. When EnableBulkLoad has been set to `true`, the driver uses the Salesforce Bulk API based on the value of the BulkLoadThreshold connection property. If the number of affected rows exceeds the value of BulkLoadThreshold property, the driver uses the Salesforce Bulk API to execute the insert, update, or delete operation.

EnablePKChunking: The EnablePKChunking property can be used to improve performance by enabling the driver to use PK chunking for bulk fetch operations. When EnablePKChunking is set to `true`, the driver uses PK chunking to execute the operation if the expected number of rows in the result set is greater than the values of the BulkFetchThreshold and PKChunkSize properties. For this behavior to take effect, the EnableBulkFetch property must also be set to `true`.

Note: PK chunking is supported for all custom objects and the following standard objects: `Account`, `Campaign`, `CampaignMember`, `Case`, `Contact`, `Lead`, `LoginHistory`, `Opportunity`, `Task`, and `User`. In addition, PK chunking is supported for sharing objects as long as the parent object is supported.

FetchSize/**WSFetchSize**: The connection options 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 `2000` (the maximum), the driver only needs to make 50 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.

WSPoolSize: WSPoolSize determines the maximum number of sessions the driver uses when there are multiple active connections to Salesforce. By increasing this number, you increase the number of sessions the driver uses to distribute calls to Salesforce, thereby improving throughput and performance. For example, if WSPoolSize is set to 1, and you have two open connections, the session must complete a call from one connection before it can begin processing a call from the other connection. However, if WSPoolSize is equal to 2, a second session is opened that allows calls from both connections to be processed simultaneously.

Note: The number specified for WSPoolSize should not exceed the amount of sessions permitted by your Salesforce account.

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. This 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, all subsequent queries that reference Account access the Account cache. Disabling or dropping the cache allows references to the Account table to access 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 create a cache before it can be populated; caches are not created automatically. After you have created a cache on a table, the cache is 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_NAME 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.

The caches maintained by the driver are write-through caches. This means that, for any operation that modifies data in a table that is cached, the driver performs the operation on the remote data first and then updates the cache.

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 "Supported SQL Statements and Extensions" for descriptions of the syntax of these extensions.

See also

Supported SQL Statements and Extensions on page 181

Creating a Cache

You create a cache using the Create Cache statement (see "Create Cache (EXT)"). A cache can be created on a single table or on a set of related tables. When creating a cache on a single table, you specify the name of the table to cache and optionally a filter for the table. The filter determines whether the cache holds all 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
- If the cache is initially enabled
- If the driver checks to see if a refresh is needed at connect time

Creating a cache for a set of related tables is similar to creating a cache on a single table, except that a primary table and one or more referencing tables are specified. This is useful if you want to cache a subset of data for a table and also cache data related to that subset of data. For example, suppose you have three tables (named Account, Contact, and Opportunity), where both a contact and an opportunity belong to a particular account. Using a relational cache, you could specify that accounts that have had activity in the past year be cached, as well as caching the opportunities and contacts for only those cached accounts.

See also

Create Cache (EXT) on page 192

Modifying a Cache Definition

Once a cache has been created, you can modify the definition of the cache or set of related caches with the Alter Cache statement (see Alter Cache (EXT)). Only the attributes of the cache can be modified through the Alter Cache statement; the table or related set of tables cannot be changed and a single table cache cannot be changed to a relational cache.

---

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.

See also

Alter Cache (EXT) on page 182

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.

Assume that a cache was created on the Account table with a filter set to cache accounts that have had activity in the past year. If you want to run a query to get information about an account that has not been active for two years, you can 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.

A better alternative is to 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 periodically refreshes the cache data with data from the remote data source. To minimize the amount of data that needs to be moved when a cache is refreshed, and the time required to refresh it, the driver checks to see which records in the remote table have been added, modified, or deleted since the last time the cache was refreshed. The driver retrieves only data for added or modified records and removes only deleted records from the cache. Your application can explicitly refresh the cache or the driver can refresh the cache automatically.

You can refresh a cache explicitly at any time by using the Refresh Cache statement (see Refresh Cache (EXT)). The Refresh Cache statement can also be used to perform a Clean (complete) refresh in addition to the standard optimized refresh. A Clean refresh discards all the data from the cache and repopulates it with data from the remote data source.

The driver also can refresh a cache automatically. When you create a cache, one of the attributes that you set is the refresh interval for the cache. During each cache query, the driver checks to see whether the time elapsed since the last refresh has exceeded the refresh interval for the cache. If it has, the driver refreshes the cache before satisfying the query.

Update operations to a table that is cached can trigger the driver to refresh the cache automatically. The caches maintained by the driver are write-through caches. For any operation that modifies data in a table that is cached, the driver performs the operation on the remote data first and then updates the cache. The driver may not be able to update the cache with all modifications because some of the modified data may have been generated by the remote data source. For example, if a row is inserted but a value for all columns in the row is not required, any default values generated by the remote data source for columns not specified in the Insert statement would not be set in the cache. Because the driver cannot reflect all the changes made when a cached table is modified, it sets the cache state to dirty. When a cache state is dirty, the next query that attempts to fetch data from that cache causes the driver to refresh the cache before the fetch operation is performed. This allows the fetch to access the values populated by the remote data source.

The state of a cache can be viewed by selecting the STATUS column of the INFORMATION_SCHEMA.SYSTEM_CACHES table. See "SYSTEM_CACHES Catalog Table" for more information.

See also

Refresh Cache (EXT) on page 218
SYSTEM_CACHES Catalog Table on page 77

Dropping a Cache

You can drop an existing cache using the Drop Cache statement (see Drop Cache (EXT)). If a cache is a relational cache, the Drop Cache statement drops the cache for the primary table as well as the caches for the related tables.

Note: When a cache is dropped, all the data in that cache is discarded.

See also

Drop Cache (EXT) on page 213

Caching MetaData

The driver maintains information about the caches that have been created. The driver provides two system tables to expose the cache information:

- SYSTEM_CACHES
- SYSTEM_CACHE_REFERENCES

Both system tables are in the INFORMATION_SCHEMA schema. See "Catalog Tables" for a complete description of the contents of these system tables.

See also

Catalog Tables on page 77

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").

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 14: 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 the driver.
TABLE_SCHEM	VARCHAR(128), NULLABLE	The schema that contains the remote table on which the cache is defined.
TABLE_NAME	VARCHAR(128), NOT NULL	The name of the remote table on which the cache is defined.

Column Name	Data Type	Description
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.

See also

SYSTEM_CACHE_REFERENCES Catalog Table on page 78

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 15: SYSTEM_CACHE_REFERENCES Catalog Table

Column	Data Type	Description
PRIMARY_TABLE_CAT	VARCHAR (128), NULLABLE	The catalog that contains the primary table of the relational cache. It is NULL for the 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 16: 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 Salesforce.
INSTANCE	VARCHAR(128)	The remote session instance name or null if the remote data source does not have multiple instances.The Salesforce value for INSTANCE has the following form:Organization_Name [Sandbox]where Organization_Name is the organization name of the Salesforce instance to which the connection is established. If the connection is established to a sandbox of the organization, then the word Sandbox is added to the end of the name.

Column Name	Data Type	Description
VERSION	VARCHAR(30), NOT NULL	The version of the remote data source to which the session is connected. For Salesforce, this is the version of the Web Service API the driver is using to connect to Salesforce.
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 17: SYSTEM_SESSIONS Catalog Table

Column	Data Type	Description
SESSION_ID	INTEGER, NOT NULL	A unique ID that identifies this session. The system function CURSESSIONID( ) returns the session ID associated with the connection. See "Functions" for more information about the CURSESSIONID() system function.
CONNECTED	DATETIME, NOT NULL	The date and time the session was established.
USERNAME	VARCHAR (128), NOT NULL	The name of the schema map that the session is using.

Column	Data Type	Description
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.

See also

Functions on page 237

Reports

The driver exposes reports defined on a Salesforce instance as stored procedures. An application can obtain a list of the reports defined on a Salesforce 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.

Salesforce organizes reports into folders. The Salesforce driver incorporates the folder name and report name into the procedure name reported by getProcedures(). The driver creates the reported procedure name by prepending the folder name to the report name using an underscore (_) to join them. Additionally, any spaces in the report or folder names are replaced with an underscore character. Like all identifier name metadata returned by the driver, the procedure name is uppercase. For example, if a report named Opportunity Pipeline is in the folder Opportunity Reports, it would be:

OPPORTUNITY_REPORTS_OPPORTUNITY_PIPELINE

An application invokes a report using the standard Call escape syntax, `{call report name}`, and JDBC mechanisms for calling a stored procedure that returns a result set. The following example shows one way to invoke the Opportunity Pipeline report:

```
String sql = "{call OPPORTUNITY_REPORTS_OPPORTUNITY_PIPELINE()}";
CallableStatement callStmt = con.prepareCall(sql);
boolean isResultSet = callStmt.execute();
if (isResultSet) {
  resultSet = callStmt.getResultSet();
  // process the resultset
}
```

Note: The API used by the driver to obtain the list of reports and execute the reports is not an API that is documented by Salesforce. This API may change or may not be supported in the future.

Note: When passing parameters to stored procedures, reports are not supported.

Authentication

Authentication ensures that only the authorized users are allowed to connect to a Salesforce instance.

The Salesforce driver supports the following methods of authentication:

- *User ID/password authentication* authenticates the user to a Salesforce instance using the user ID and password specified by the application.
- *OAuth 2.0 authentication* allows the user to authenticate to a Salesforce instance without having to specify user ID and password.

See also

Using Connection Properties on page 56
Authentication Properties on page 58

Configuring user ID/password authentication

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

1. Set the AuthenticationMethod property to `userIDPassword`.
2. Set the User property to provide the user ID.
3. Set the Password property to provide the password.

See also

AuthenticationMethod on page 116

Configuring OAuth 2.0 authentication

The driver supports OAuth 2.0 to access Salesforce resources. Before you can configure the driver for OAuth, you must register your client application with Salesforce and obtain information such as the client ID, the client secret, and the refresh token. The following workflow describes the process for setting up OAuth 2.0 access.

1. Register your application with Salesforce. See this topic for step-by-step instructions for registering your application.
2. Obtain client ID, client secret, and scopes. If the application has already been registered, see this topic for steps to obtain information required to configure the driver.
3. Obtain access and refresh tokens with Postman. To configure the driver, you will need to specify either an access token or a refresh token. This topic provides instructions to obtain these tokens using Postman.
4. Configure the driver to use OAuth 2.0. You can configure the driver to access Salesforce resources by specifying the connection properties described in this topic.
5. Configure the driver to use OAuth 2.0 with JWT grant. You can configure the driver to use JWT grant in OAuth 2.0 by specifying the connection properties described in this topic.

Note: For more information, refer to the Authorize Apps with OAuth section of the *Salesforce Help*.

Registering your application with Salesforce

Prerequisites:

- The callback URL (or redirect URI) for the client application

Take the following steps to register your application as a Salesforce Connected App.

1. Sign in to Salesforce.

   https://login.salesforce.com

2. Navigate to the **Setup** page.
3. Navigate to **Apps > App Manager**.
4. Click **New Connected App** to open the **New Connected App** form.
5. Complete and save the **New Connected App** form. The following list of parameters are required for OAuth.
   - **Basic Information**
     - **Connected App Name**: An application name must be specified.
     - **API Name**: The API Name is based on the application name and generated automatically, but you may modify it.
     - **Contact Email**: A contact email must be specified.
   - **API (Enable OAuth Settings)**
     - **Enable OAuth Settings**: Check this box to enable OAuth.
     - **Callback URL (OAuth Redirect URI)**: Specify one or more callback URLs. This is the URL that Salesforce calls back to your application during the authorization code flow.

       Notes:

- If you are using **Postman** to implement or test OAuth connectivity, the callback URL may be either of the following:

  `https://oauth.pstmn.io/v1/callback`

  `https://www.getpostman.com/oauth2/callback`

  For details, see Obtaining access and refresh tokens with Postman.

- If you are using **Hybrid Data Pipeline**, the callback URL is:

  `https://MyHDPDomain:8443/hdpui/oauth2callback`

  where *MyHDPDomain* is the name of the Hybrid Data Pipeline host or load balancer.

- **Use digital signatures**. Check this box if you are using the JWT OAuth flow, and upload a certificate if required.
- **Selected OAuth Scopes**: Choose OAuth scopes to define permissions for the application. For example, to grant an application full access to Salesforce resources at any time, you would specify the following scopes:
  - `Full access (full)`: Allows access to all data accessible by the logged-in user, and encompasses all other scopes.
  - `Perform requests at any time (refresh_token, offline_access)`: Allows a refresh token to be returned when the requesting client is eligible to receive one.

Note: The values you select should be saved to a secure location. You will need to specify these values if you use are using the authorization code flow to obtain access and refresh tokens, as described in Obtaining access and refresh tokens with Postman.

6. After saving the **New Connected App** form, the page for the Connected App will be displayed.
7. Click **Manage Consumer Details** to obtain the client ID and client secret (Consumer Key and Consumer Secret).

Note: The values for the client ID and secret should be saved to a secure location. You will need to specify these values for the ClientID and ClientSecret connection properties.

Results:

You have successfully registered your application as a Salesforce Connected App, as well as obtained the OAuth client ID and client secret.

Obtaining the client ID, client secret, and scopes

Take the following steps to obtain the client ID, the client secret, and the scope.

1. Sign in to Salesforce.

   https://login.salesforce.com

2. Navigate to the **Setup** page.
3. Navigate to **Apps > App Manager**.

4. Select the application for which you are obtaining the client ID and secret from the list of Connected Apps.
5. For the scopes, navigate to **API (Enable OAuth Settings) > Selected OAuth Scopes**.

   In most cases, the client application will be granted full access to Salesforce resources at any time. In this scenario, the following scopes would be specified:

   - `Full access (full)`: Allows access to all data accessible by the logged-in user, and encompasses all other scopes.
   - `Perform requests at any time (refresh_token, offline_access)`: Allows a refresh token to be returned when the requesting client is eligible to receive one.

   Note: You will need to specify scopes if you use are using the authorization code flow to obtain access and refresh tokens, as described in Obtaining access and refresh tokens with Postman.

6. For the client ID and client secret (Consumer Key and Consumer Secret), click **Manage Consumer Details**.

   Note: The values for the client ID and secret should be saved to a secure location. You will need to specify these values for the ClientID and ClientSecret connection properties.

Results:

You have obtained the OAuth client ID and client secret.

Obtaining access and refresh tokens with Postman

Prerequisites:

- The client ID and client secret (Consumer Key and Consumer Secret) for the Salesforce Connected App
- The Postman callback URL (or redirect URI) for the client application.

The following steps show how to use Postman to obtain access and refresh tokens from Salesforce.

1. Open Postman.
2. Select the **Authorization** tab.
3. Select **OAuth 2.0**. Then, enter required values. For example:

   Token Name: `MySalesforceToken`

   Grant Type: `Authorization Code`

   Callback URL: *Depending on the version of Postman you are using, this may be either of the following endpoints:*

   - `https://oauth.pstmn.io/v1/callback`
   - `https://www.getpostman.com/oauth2/callback`

   Auth URL: `https://login.salesforce.com/services/oauth2/authorize`

   Token URL: `https://login.salesforce.com/services/oauth2/token`

   Client ID: *`client-id`*

   Client Secret: *`client-secret`*

Scope: *scope* where *scope* is the list of scopes that define application permissions. In most cases, the client application will be granted full access to Salesforce resources at any time. In this scenario, the following scopes would be specified:

- `Full access (full)`: Allows access to all data accessible by the logged-in user, and encompasses all other scopes.
- `Perform requests at any time (refresh_token, offline_access)`: Allows a refresh token to be returned when the requesting client is eligible to receive one.

4. Press **Get New Access Token**.

   Salesforce returns the access and refresh tokens to Postman.

Results:

You have obtained access and refresh tokens for OAuth access to Salesforce.

Configuring the driver to use OAuth 2.0

Prerequisites:

- Client application registered as a Salesforce Connected App
- An access token or refresh token

After you have registered your client application as a Salesforce Connected App and obtained the required OAuth information, you may configure the driver to access Salesforce resources using OAuth 2.0.

To configure the driver, set the following connection properties:

- Set the AuthenticationMethod property to `oauth2.0`.
- Set the SchemaMap property to specify either the name or the absolute path and name of the configuration file where the map of the Salesforce data model is written. Note that a value for the SchemaMap property must be specified every time you authenticate to a Salesforce instance using OAuth 2.0.
- Set either of the following properties:
  - Set the AccessToken property to specify the access token you have obtained from Salesforce. Generally, an access token is available for a short period of time and may only be used for a single session.
  - Set the RefreshToken property to specify the refresh token you have obtained from Salesforce. With a valid refresh token, the driver can obtain a new access token. In this way, a refresh token enables application access to Salesforce for multiple sessions over an extended period of time, as dictated by your organization's policies.

  If a value for the AccessToken property is not specified, the driver uses the value of the RefreshToken property to make a connection. If both values are not specified, the driver cannot make a successful connection. If both are specified, the driver ignores the AccessToken value and uses the RefreshToken value to generate a new AccessToken value.

  See Obtaining access and refresh tokens with Postman for details on how to obtain access or refresh tokens using Postman.

Access token example URL

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:sforce://login.salesforce.com;AuthenticationMethod=oauth2.0;
SchemaMap=schema-map-name;AccessToken=access-token;");
```

Refresh token example URL

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:sforce://login.salesforce.com;AuthenticationMethod=oauth2.0;
SchemaMap=schema-map-name;RefreshToken=refresh-token;");
```

Configuring the driver to use OAuth 2.0 with JWT grant

Prerequisites:

- A client application registered as a Salesforce Connected App.
- A JWT certificate containing the private key for the registered application.

After you have registered your client application as a Salesforce Connected App and obtained the required OAuth information, you may configure the driver to access Salesforce resources using OAuth 2.0. with JWT grant.

To configure the driver, set the following connection properties:

- Set the AuthenticationMethod property to `oauth2.0`.
- Set the JWTCertStore property to specify the file path to the certificate store containing the private key for your application.
- Set the ClaimsIssuer property to specify the client ID or consumer key of the Salesforce Connected App.
- Set the ClaimsSubject property to specify the username of the user.
- Set the SchemaMap property to specify either the name or the absolute path and name of the configuration file where the map of the Salesforce data model is written. Note that a value for the SchemaMap property must be specified every time you authenticate to a Salesforce instance using OAuth 2.0.
- Set the JWTCertPassword property to specify the password for the JWT certficate, if any.
- Set the JWTCertAlias property to specify an alias for the JWT certficate, if any.

JWT grant example URL

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:sforce://login.salesforce.com;AuthenticationMethod=oauth2.0;
SchemaMap=schema-map-name;JWTCertStore=file-path;
ClaimsIssuer=client-ID;ClaimsSubject=user-name;
JWTCertPassword=certificate-password;JWTCertAlias=certificate-alias");
```

See also

ClaimsIssuer on page 123
ClaimsSubject on page 124
JWTCertAlias on page 143
JWTCertPassword on page 143
JWTCertStore on page 144

Data Encryption

All communication between the driver and Salesforce, including user ID/password authentication, is encrypted using TLS/SSL. See "Using Connection Properties" for information on specifying a user ID and Password.

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 also

Using Connection Properties on page 56

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.

Using 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 Salesforce 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.

Failover Support

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

See also

Failover Properties on page 60
ConnectionRetryCount on page 132
ConnectionRetryDelay on page 133

IP Addresses

The driver supports Internet Protocol (IP) addresses in IPv4 and IPv6 format.

The server name specified in the connection URL, or data source, can resolve to an IPv4 or IPv6 address. For example, the following URL can resolve to either type of address:

```
jdbc:datadirect:sforce://login.salesforce.com;User=abc@defcorp.com;
   Password=secret;SecurityToken=XaBARTsLZReM4Px47qPLOS
```

Note: The server name specifies the base Salesforce URL to use for logging in, for example, `login.salesforce.com`.

Alternately, you can specify addresses using IPv4 or IPv6 format in the server portion of the connection URL. For example, the following connection URL specifies the server using an IPv4 address:

```
jdbc:datadirect:sforce://123.456.78.90;User=abc@defcorp.com;
   Password=secret;SecurityToken=XaBARTsLZReM4Px47qPLOS
```

You also can specify addresses in either format using the ServerName data source property. The following example shows a data source definition that specifies the server name using IPv6 format:

```
SalesforceDataSource mds = new SalesforceDataSource();
   mds.setDescription("My SalesforceDataSource");
   mds.setServerName("[ABCD:EF01:2345:6789:ABCD:EF01:2345:6789]");
   ...
```

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

In addition to the normal IPv6 format, the drivers support IPv6 alternative formats for compressed and IPv4/IPv6 combination addresses. For example, the following connection URL specifies the server using IPv6 format, but uses the compressed syntax for strings of zero bits:

```
jdbc:datadirect:sforce://[2001:DB8:0:0:8:800:200C:417A];User=abc@defcorp.com;
   Password=secret;SecurityToken=XaBARTsLZReM4Px47qPLOS
```

Similarly, the following connection URL specifies the server using a combination of IPv4 and IPv6:

```
jdbc:datadirect:sforce://[0000:0000:0000:0000:0000:FFFF:123.456.78.90];User=abc@defcorp.com;
   Password=secret;SecurityToken=XaBARTsLZReM4Px47qPLOS
```

For complete information about IPv6, go to the following URL:

http://tools.ietf.org/html/rfc4291#section-2.2

See also

ServerName on page 156

Statement pooling

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

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

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

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

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

Connection pooling

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

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

Timeouts

The driver supports the LoginTimeout, WSTimeout, and WSRetryCount connection properties. The LoginTimeout property specifies the amount of time, in seconds, that the driver waits for a connection to be established before timing out the connection request. In contrast, the WSTimeout property specifies the time, in seconds, that the driver waits for a response to a Web service request. The WSTimeout property can be used in conjunction with the WSRetryCount property. The WSRetryCount connection property can be used to retry select queries that have timed out.

Session 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. This is particularly true when connection pooling is used. The driver automatically attempts to re-establish a new session if the driver receives a session timeout error from a data source. The driver uses the initial server name, remote user ID, and remote password 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, you can configure the driver to retry the request a specified number of times by setting the WSRetryCount connection property. If subsequent attempts to retry a request fail, the driver returns an error indicating that the service request timed out and the subsequent requests failed.

See also

- Timeout Properties on page 66
- Refer to "Connection Pool Manager" in the *Progress DataDirect for JDBC Drivers Reference* for more information.

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 a Salesforce object and is exposed in the `SF0RCE` 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" for details on the Create View and Create Table statements and other SQL statements supported by the driver.

See also

Supported SQL Statements and Extensions on page 181

Isolation Levels

The driver supports the TRANSACTION_NONE isolation level because Salesforce does not support transactions. However, manual transactions can be handled to some degree with the TransactionMode connection property.

See also

TransactionMode on page 161

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.

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

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

Large Object (LOB) Support

The Salesforce 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.

Parameter Metadata Support

The driver supports returning parameter metadata as described in "Insert and Update Statements" and "Select Statements."

Insert and Update Statements

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

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

where:

operator

is any of the following SQL operators:

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

Select Statements

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

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

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

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

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

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

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

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

```
SELECT col1, col2 FROM foo WHERE col1 = ? AND col2 > ?
SELECT ... WHERE colname = (SELECT col2 FROM t2 WHERE col3 = ?)
SELECT ... WHERE colname LIKE ?
SELECT ... WHERE colname BETWEEN ? and ?
SELECT ... WHERE colname IN (?, ?, ?)
SELECT ... WHERE EXISTS(SELECT ... FROM T2 WHERE col1 < ?)
```

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

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

Joins are supported. For example:

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

Fully qualified names and aliases are supported. For example:

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

ResultSet MetaData Support

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

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

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

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

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

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

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

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

Rowset Support

The Salesforce 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.

Auto-Generated Keys Support

The Salesforce driver supports retrieving the values of auto-generated keys. An auto-generated key returned by the Salesforce 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.

DataDirect Bulk Load

In addition to supporting bulk operations via the Salesforce Bulk API, the driver supports DataDirect Bulk Load. DataDirect Bulk Load allows you to perform bulk load operations by creating a `DDBulkLoad` object and using the methods provided by the `DDBulkLoad` interface in the `com.ddtek.jdbc.extensions` package for bulk load. You may want to use this method if you are developing a new application that needs to perform bulk load operations.

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

See also

Bulk API Properties on page 63

Using a DDBulkLoad Object

The first step in performing a bulk load operation using a `DDBulkLoad` object is to create a `DDBulkLoad` object. A `DDBulkLoad` must be created with the `getInstance` method using the `DDBulkLoadFactory` class as shown in the following example.

```
import com.ddtek.jdbc.extensions.*
// Get Salesforce Connection
Connection con = DriverManager.getConnection(
   "jdbc:datadirect:sforce://login.salesforce.com;User=abc@defcorp.com;
    Password=secret;SecurityToken=XaBARTsLZReM4Px47qPLOS");

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

Once the DDBulkLoad object has been created, DDBulkLoad methods can be used to instruct the driver to obtain data from one location and load it to another location. The driver can obtain data either from a ResultSet object or from a CSV file.

Migrating Data Using a ResultSet Object

The following steps would need to be taken to migrate data from Oracle to Salesforce using a ResultSet object.

Important: This scenario assumes that you are using the DataDirect Oracle driver to query the Oracle database and the DataDirect Salesforce driver to insert data on the Salesforce data store.

---

1. The application creates a DDBulkLoad object.
2. The application executes a standard query on the Oracle database, and the Oracle driver retrieves the results as a ResultSet object.
3. The application instructs the Salesforce driver to load the data from the ResultSet object into Salesforce. (See "Loading Data From a ResultSet Object".)

Migrating Data Using a CSV File

The following steps would need to be taken to migrate data from Oracle to Salesforce using a CSV file.

1. The application creates a DDBulkLoad object.
2. The application instructs the Oracle driver to export the data from the Oracle database into a CSV file. (See "Exporting Data to a CSV File.")
3. The application instructs the Salesforce driver to load data from the CSV file into Salesforce. (See "Loading Data From a CSV File.")

See also

- Refer to "JDBC extensions" in the *Progress DataDirect for JDBC Drivers Reference* for more information about bulk load methods.
- CSV files on page 100
- Permissions for bulk load from a CSV file on page 46

Exporting data to a CSV file

Using a BulkLoad object, data can be exported either as a table or ResultSet object.

Exporting Data as a Table

To export data as a table, the application must specify the table name with the setTableName method and then export the data to a CSV file using the export method. For example, the following code snippet specifies the GBMAXTABLE table and exports it to a CSV file called tmp.csv.

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

Alternatively, you can create a file reference to the CSV file and use the export method to specify the file reference. For example:

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

Exporting Data as a ResultSet object

To export data as a `ResultSet` object, the application must first create a file reference to a CSV file, and then, using the `export` method, specify the `ResultSet` object and the file reference. For example, the following code snippet creates the `tmp.csv` file reference and then specifies `MyResults` (the `ResultSet` object be exported).

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

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

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

See also

CSV files on page 100

Loading Data from a ResultSet Object

The `setTableName` method should be used to specify the table into which you want to load the data. Then, the `load` method is used to specify the `ResultSet` object that contains the data you are loading. For example, the following code snippet loads data from a `ResultSet` object named `MyResults` into a table named `GBMAXTABLE`

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

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

You can use the BulkLoadBatchSize property to specify the number of rows the driver loads at a time when bulk loading data. Performance can be improved by increasing the number of rows the driver loads at a time because fewer network round trips are required. Be aware that increasing the number of rows that are loaded also causes the driver to consume more memory on the client.

Loading Data from a CSV File

The `setTableName` method should be used to specify the table into which you want to load the data. Then, the `load` method is used to specify the CSV file that contains the data you are loading. For example:

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

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

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

For the Salesforce driver, you also can specify the columns to which you want to load the data. This example loads the first column in the CSV file to the `ColName1` column, the second column in the CSV file to the `ColName2` column, and the third column in the CSV file to the `ColName3` column:

```
bulkLoad.setTableName("GBMAXTABLE(ColName1, ColName2, ColName3)");
bulkLoad.load("tmp.csv");
```

Use the BulkLoadBatchSize property to specify the number of rows the driver loads at a time when bulk loading data. Performance can be improved by increasing the number of rows the driver loads at a time because fewer network round trips are required. Be aware that increasing the number of rows that are loaded also causes the driver to consume more memory on the client. See "JDBC Extensions" for more information about bulk load methods.

Specifying the bulk load operation

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

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

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

Logging

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

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

- Total number of rows that were read
- Total number of rows that successfully loaded

  Note: The total number of rows that successfully loaded is not provided for Microsoft Azure Synapse Analytics or Microsoft Analytics Platform System.

- Total number of rows that failed to load

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

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

Enabling Logging on Windows

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

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

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

Enabling Logging on UNIX/Linux

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

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

CSV files

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

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

See also

Exporting data to a CSV file on page 97

Bulk load configuration file

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

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

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

```
<?xml version="1.0" encoding="utf-8"?>
<table codepage="UTF-8" xsi:noNamespaceSchemaLocation=
"http://media.datadirect.com/download/docs/ns/bulk/BulkData.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<row>
   <column datatype="DECIMAL" precision="38" scale="0" nullable="false">
      INTEGERCOL</column>
   <column datatype="VARBINARY" length="10"
nullable="true">VARBINCOL</column>
   <column datatype="VARCHAR" length="10" sourcecodepage="Windows-1252"
      externalfilecodepage="Windows-1252"
nullable="true">VCHARCOL</column>
   <column datatype="VARCHAR" length="10" sourcecodepage="Windows-1252"
      externalfilecodepage="Windows-1252"
nullable="true">UNIVCHARCOL</column>
```

```
</row>
</table>
```

Bulk load configuration file schema

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

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

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

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

Character set conversions

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

To avoid character set conversions, your application can specify which code page to use for the CSV file when exporting data. For example, if the source database table uses a SHIFT-JIS code page and the target table uses a EUC-JP code page, specify `setCodePage("EUC_JP")` to ensure that the CSV file will use the same code page as the target table.

You can specify any of the following code pages.

Note: If the code page you need to use is not listed, contact Customer Support to request support for that code page.

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

External overflow files

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

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

The format for overflow file names is:

`CSV_file_name``.xxxxxx.lob`

where:

`CSV_file_name` is the name of the CSV file.

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

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

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

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

Discard file

If the driver was unable to load rows into the database for a bulk load operation from a CSV file, it can record all the rows that were not loaded in a discard file. The contents of the discard file is in the same format as that of the CSV file. After fixing reported issues in the discard file, the bulk load can be reissued, using the discard file as the CSV file.

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

Creating a discard file on Windows

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

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

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

Creating a discard file on UNIX/Linux

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

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

Tracking JDBC calls with DataDirect Spy

DataDirect Spy is functionality that is built into the drivers. It is used to log detailed information about calls your driver makes and provide information you can use for troubleshooting. DataDirect Spy provides the following advantages:

- Logging is JDBC 4.0-compliant.
- All parameters and function results for JDBC calls can be logged.
- Logging can be enabled without changing the application.

When you enable DataDirect Spy for a connection, you can customize logging by setting one or multiple options for DataDirect Spy. For example, you may want to direct logging to a local file on your machine.

Once logging is enabled for a connection, you can turn it on and off at runtime using the `setEnableLogging` method in the `com.ddtek.jdbc.extensions.ExtLogControl` interface.

Refer to Troubleshooting your application in the *Progress DataDirect for JDBC Drivers Reference* for information about using a DataDirect Spy log for troubleshooting.

Enabling DataDirect Spy

You can enable and customize DataDirect Spy logging in either of the following ways.

- Specifying the SpyAttributes connection property for connections using the JDBC `DriverManager`.
- Specifying DataDirect Spy attributes using a JDBC data source.

Using the JDBC DriverManager

The SpyAttributes connection property allows you to specify a semi-colon separated list of DataDirect Spy attributes. The format for the value of the SpyAttributes property is:

```
(
spy_attribute
[;
spy_attribute
]...)
```

where *`spy_attribute`* is any valid DataDirect Spy attribute. See "DataDirect Spy Attributes" for a list of supported attributes.

Windows Example

```
Class.forName("com.ddtek.jdbc.sforce.SForceDriver");
Connection conn = DriverManager.getConnection
   ("jdbc:datadirect:sforce://login.salesforce.com;User=abc@defcorp.com;
   Password=secret;SecurityToken=XaBARTsLZReM4Px47qPLOS;
   SpyAttributes=(log=(filePrefix)C:\\temp\\spy_;linelimit=80;logTName=yes;
   timestamp=yes)");
```

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

DataDirect Spy loads the Salesforce driver and logs all JDBC activity to the `spy_x.log` file located in the `C:\temp` directory (`log=(filePrefix)C:\\temp\\spy_`), where *x* is an integer that increments by 1 for each connection on which the prefix is specified. The `spy_x.log` file logs a maximum of 80 characters on each line (`linelimit=80`) and includes the name of the current thread (`logTName=yes`) with a timestamp on each line in the log (`timestamp=yes`).

UNIX Example

```
Class.forName("com.ddtek.jdbc.sforce.SForceDriver");
Connection conn = DriverManager.getConnection
   ("jdbc:datadirect:sforce://login.salesforce.com;User=abc@defcorp.com;
   Password=secret;SecurityToken=XaBARTsLZReM4Px47qPLOS;
   SpyAttributes=(log=(filePrefix)/tmp/spy_;logTName=yes;timestamp=yes)");
```

DataDirect Spy loads the Salesforce driver and logs all JDBC activity to the `spy_x.log` file located in the `/tmp` directory (`log=(filePrefix)/tmp/spy_`), where *x* is an integer that increments by 1 for each connection on which the prefix is specified. The `spy_x.log` file includes the name of the current thread (`logTName=yes`) with a timestamp on each line in the log (`timestamp=yes`).

See also

DataDirect Spy attributes on page 106

Using JDBC Data Sources

You can use DataDirect Spy to track JDBC calls made by a running application with either of these features:

- JNDI for Naming Databases
- Connection Pooling

The `com.ddtek.jdbcx.SForce.SForceDataSource` class supports setting a semi-colon separated list of DataDirect Spy attributes.

Windows Example

```
SForceDataSource sds = new SForceDataSource();
sds.setDescription("My Salesforce Datasource");
sds.setServerName("login.salesforce.com");
sds.setUser("abc@defcorp.com");
sds.setPassword("secret");
sds.setSecurityToken("XaBARTsLZReM4Px47qPLOS");
sds.setSpyAttributes("log=(file)C:\\temp\\spy.log;logIS=yes;logTName=yes");
Connection conn=sds.getConnection;
...
```

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

DataDirect Spy loads the Salesforce driver and logs all JDBC activity to the `spy.log` file located in the `C:\temp` directory (`log=(file)C:\\temp\\spy.log`). In addition to regular JDBC activity, the `spy.log` file also logs activity on `InputStream` and `Reader` objects (`logIS=yes`). It also includes the name of the current thread (`logTName=yes`).

UNIX Example

```
SForceDataSource mds = new SForceDataSource();
mds.setDescription("My Salesforce Datasource");
mds.setServerName("login.salesforce.com");
mds.setUser("abc@defcorp.com");
mds.setPassword("secret");
mds.setSecurityToken("XaBARTsLZReM4Px47qPLOS");
mds.setSpyAttributes("log=(file)/tmp/spy.log;logIS=yes;logTName=yes");
Connection conn=mds.getConnection;
...
```

DataDirect Spy loads the Salesforce driver and logs all JDBC activity to the `spy.log` file located in the `/tmp` directory (`log=(file)/tmp/spy.log`). In addition to regular JDBC activity, the `spy.log` file also logs activity on `InputStream` and `Reader` objects (`logIS=yes`). It also includes the name of the current thread (`logTName=yes`).

See also

DataDirect Spy attributes on page 106

DataDirect Spy attributes

DataDirect Spy supports the attributes described in the following table.

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

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

4

Connection Property Descriptions

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

Note:

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

The following table provides a summary of the connection properties supported by the driver, their corresponding data source methods, and their default values.

Table 18: Salesforce Driver Properties

Property	Data Source Method	Default
AccessToken on page 114	`setAccessToken`	None

Property	Data Source Method	Default
ApplyToLabel on page 115	setApplyToLabel	false
AuthenticationMethod on page 116	setAuthenticationMethod	userIDPassword
BulkFetchThreshold on page 116	setBulkFetchThreshold	30000 (rows)
BulkLoadAsync on page 117	setBulkLoadAsync	0 (bulk load operations are synchronous)
BulkLoadBatchSize on page 118	setBulkLoadBatchSize	10000 (rows)
BulkLoadConcurrencyMode on page 119	setBulkLoadConcurrencyMode	parallel
BulkLoadJobSize on page 119	setBulkLoadJobSize	150000
BulkLoadPollInterval on page 120	setBulkLoadPollInterval	10 (seconds)
BulkLoadThreshold on page 121	setBulkLoadThreshold	4000 (rows)
BulkLoadVersion on page 122	setBulkLoadVersion	V1
CatalogOptions on page 122	setCatalogOptions	2
ClaimsIssuer on page 123	setClaimsIssuer	None
ClaimsSubject on page 124	setClaimsSubject	None
ClientID on page 124	setClientID	None
ClientSecret on page 125	setClientSecret	None
ClientTimeZone on page 126	setClientTimeZone	None. The driver determines the client time zone based on the system-specific time zone settings.
ConfigOptions on page 127 **Important:** The configuration options have been deprecated. However, the driver will continue to support them until the next major release of the driver.	setConfigOptions	AuditColumns=all; CustomSuffix=include; KeywordConflictSuffix=; MapSystemColumnNames=0; NumberFieldMapping=emulateInteger; UppercaseIdentifiers=true
ConnectionRetryCount on page 132	setConnectionRetryCount	5
ConnectionRetryDelay on page 133	setConnectionRetryDelay	1 (second)

Property	Data Source Method	Default
ConvertNull on page 134	setConvertNull	1 (data type check is performed if column value is null)
CreateMap on page 135	setCreateMap	notExist
EnableBulkFetch on page 136	setEnableBulkFetch	true
EnableBulkLoad on page 136	setEnableBulkLoad	true
EnablePKChunking on page 137	setEnablePKChunking	true
FetchSize on page 138	setFetchSize	100 (rows)
ImportStatementPool on page 139	setImportStatementPool	empty string
InitializationString on page 140	setInitializationString	None
InsensitiveResultSetBufferSize on page 140	setInsensitiveResultSetBufferSize	2048 (KB of memory)
InValuesLimit on page 141	setInValuesLimit	200
JavaDoubleToString on page 142	setJavaDoubleToString	false
JWTCertAlias on page 143	setJwtCertAlias	None
JWTCertPassword on page 143	setJwtCertPassword	None
JWTCertStore on page 144	setJwtCertStore	None
LogConfigFile on page 145	setLogConfigFile	ddlogging.properties
LoginTimeout on page 145	setLoginTimeout	0 (no timeout)
MaxPooledStatements on page 146	setMaxPooledStatements	0 (driver's internal prepared statement pooling is not enabled)
Password on page 147	setPassword	None
PKChunkSize on page 148	setPKChunkSize	100000 (rows)
ProxyHost on page 149	setProxyHost	empty string
ProxyPassword on page 149	setProxyPassword	empty string
ProxyPort on page 150	setProxyPort	0
ProxyUser on page 150	setProxyUser	empty string
ReadOnly on page 151	setReadyOnly	false

Property	Data Source Method	Default
RefreshToken on page 152	setRefreshToken	None
RegisterStatementPoolMonitorMBean on page 153	setRegisterStatementPoolMonitorMBean	false
SchemaMap on page 153	setSchemaMap	Default value depends on environment
SecurityToken on page 155	setSecurityToken	None
ServerName on page 156	setServerName	login.salesforce.com
SpyAttributes on page 157	setSpyAttributes	None
StmtCallLimit on page 159	setStmtCallLimit	100 (Web service calls)
StmtCallLimitBehavior on page 160	setStmtCallLimitBehavior	errorAlways
TransactionMode on page 161	setTransactionMode	noTransactions
User on page 161	setUser	None
WSCompressData on page 162	setWSCompressData	compress
WSFetchSize on page 163	setWSFetchSize	0 (maximum of 2000 rows)
WSPoolSize on page 164	setWSPoolSize	1
WSRetryCount on page 164	setWSRetryCount	0 (no retries for timed-out requests)
WSTimeout on page 165	setWSTimeout	120 (seconds)

For details, see the following topics:

- AccessToken
- ApplyToLabel
- AuthenticationMethod
- BulkFetchThreshold
- BulkLoadAsync
- BulkLoadBatchSize
- BulkLoadConcurrencyMode
- BulkLoadJobSize
- BulkLoadPollInterval
- BulkLoadThreshold

- BulkLoadVersion
- CatalogOptions
- ClaimsIssuer
- ClaimsSubject
- ClientID
- ClientSecret
- ClientTimeZone
- ConfigOptions
- ConnectionRetryCount
- ConnectionRetryDelay
- ConvertNull
- CreateMap
- EnableBulkFetch
- EnableBulkLoad
- EnablePKChunking
- FetchSize
- ImportStatementPool
- InitializationString
- InsensitiveResultSetBufferSize
- InValuesLimit
- JavaDoubleToString
- JWTCertAlias
- JWTCertPassword
- JWTCertStore
- LogConfigFile
- LoginTimeout
- MaxPooledStatements
- Password
- PKChunkSize
- ProxyHost
- ProxyPassword
- ProxyPort
- ProxyUser

AccessToken

Purpose

Specifies the access token required to authenticate to a Salesforce instance when OAuth 2.0 is enabled (`AuthenticationMethod=oauth2.0`).

The access token acts as a session ID for the connection. Refer to the Salesforce documentation to know how to obtain an access token.

Valid Values

`string`

where:

`string`

is the access token you have obtained from Salesforce.

Notes

- If a value for the AccessToken property is not specified, the driver uses the value of the RefreshToken property to make a connection.
- If both AccessToken and RefreshToken values are not specified, the driver cannot make a successful connection.
- If both AccessToken and RefreshToken values are specified, the driver ignores the AccessToken value and uses the RefreshToken value to generate a new AccessToken value.

Data Source Method

`setAccessToken`

Default

None

Data Type

String

See also

Configuring OAuth 2.0 authentication on page 83

AuthenticationMethod on page 116

RefreshToken on page 152

ApplyToLabel

Purpose

Determines whether the driver applies the toLabel() function to the picklist fields when executing queries. The toLabel() function translates the result set of a query into the language of the user.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver applies the toLabel() function to the picklist fields when executing queries.

If set to `false`, the driver does not apply the toLabel() function to the picklist fields when executing queries.

Notes

- The driver does not apply the toLabel() function to the picklist fields specified using the Order By clause.

Data Source Methods

`setApplyTolabel`

Default Value

`false`

Data Type

Boolean

AuthenticationMethod

Purpose

Determines which authentication method the driver uses when establishing a connection.

Valid values

`userIDPassword` | `oauth2.0`

Behavior

If set to `userIDPassword`, the driver uses user ID/password authentication when establishing a connection.

If set to `oauth2.0`, the driver uses OAuth 2.0 authentication when establishing a connection.

Data Source Method

`setAuthenticationMethod`

Default

`userIDPassword`

Data Type

String

See Also

Configuring user ID/password authentication on page 82

Configuring OAuth 2.0 authentication on page 83

BulkFetchThreshold

Purpose

Specifies a number of rows that, if exceeded, signals the driver to use the Salesforce Bulk API for select operations. For this behavior to take effect, the EnableBulkFetch property must be set to `true`.

Valid Values

`0` | *x*

where:

x

is a positive integer that represents a number of rows.

Behavior

If set to `0`, the driver uses the Salesforce Bulk API for all select operations.

If set to *x*, the driver uses the Salesforce Bulk API for select operations when the value of *x* is exceeded.

Notes

- If the EnableBulkFetch property is set to `false`, the BulkFetchThreshold property is ignored.
- The EnablePKChunking connection property can be used to break bulk fetch operations into smaller, more manageable batches for improved performance.
- Do not set the BulkFetchThreshold property to a value greater than the Web service call limit set by the StmtCallLimit property. If the value set for BulkFetchThreshold is greater than the value of StmtCallLimit, the driver would not use the Salesforce Bulk API for fetch operations because the Web service call limit would be reached before the driver reached the threshold to use the Salesforce Bulk API.

Data Source Method

`setBulkFetchThreshold`

Default

`30000` (rows)

Data Type

int

See also

- EnableBulkFetch on page 136
- EnablePKChunking on page 137
- StmtCallLimit on page 159

BulkLoadAsync

Purpose

Determines whether the driver treats bulk load operations as synchronous or asynchronous..

Valid Values

`0 | 1`

Behavior

If set to `0`, bulk load operations are synchronous. The driver does not return from the method that invoked an operation until the operation has completed or the operation has timed out. If the operation times out, the driver throws an exception.

If set to 1, bulk load operations are asynchronous. The driver returns from the method that invoked an operation immediately after the operation is submitted to the server. The driver does not verify that the bulk load operation was completed.

Default

0

Data Source Method

setBulkLoadAsync

Data Type

int

See also

DataDirect Bulk Load on page 96

BulkLoadBatchSize

Purpose

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

Valid Values

x

where:

x

is a positive integer that represents a number of rows.

Behavior

The driver loads the specified number of rows at a time. If the specified value exceeds the maximum batch size allowed by the data source, the driver uses the maximum batch size allowed by the data source.

Notes

The batchSize() method of the DDBulkLoad interface can be used to override this value for a particular bulk load operation.

Data Source Method

setBulkLoadBatchSize

Default

10000 (rows)

Data Type

long

See also

DataDirect Bulk Load on page 96

BulkLoadConcurrencyMode

Purpose

Determines whether multiple batches associated with a bulk load operation are processed by Salesforce in parallel or one at a time.

Valid Values

`parallel|serial`

Behavior

If set to `parallel`, multiple batches associated with a bulk load operation are processed in parallel. The order in which the batches are processed can vary.

If set to `serial`, multiple batches associated with a bulk load operation are processed one at a time.

Data Source Method

`setBulkLoadConcurrencyMode`

Default

`parallel`

Data Type

String

See also

DataDirect Bulk Load on page 96

BulkLoadJobSize

Purpose

Determines the number of rows to load into a single job of a bulk operation when the connection property BulkLoadVersion is set to V2. Performance can be improved by increasing the number of rows the driver loads at a time because fewer network round trips are required. Increasing the number of rows also causes the driver to consume more memory on the client.

Valid Values

x

where:

x

is a positive integer less than or equal to 1000000 that represents a number of rows.

Behavior

The driver loads the specified number of rows into a single job of a bulk operation.

Data Source Methods

```
public Integer getBulkLoadJobSize()
```

```
public void setBulkLoadJobSize(Integer)
```

Default Value

`150000`

Data Type

Integer

BulkLoadPollInterval

Purpose

Specifies the number of seconds the driver waits to request bulk operation status. This interval is used by the driver the first time it requests status and for all subsequent status requests.

Valid Values

x

where:

x

is a positive integer that represents a number of seconds the driver waits before requesting bulk operation status.

Data Source Method

`setBulkLoadPollInterval`

Default

`10` (seconds)

Data Type

int

See also

DataDirect Bulk Load on page 96

BulkLoadThreshold

Purpose

Specifies a number of rows that, if exceeded, signals the driver to use the Salesforce Bulk API for inserts, updates, and deletes. For this behavior to take effect, the EnableBulkLoad property must be set to `true`.

Valid Values

`0` | *x*

where:

x

is a positive integer that represents a number of rows.

Behavior

If set to `0`, the driver uses the Salesforce Bulk API for all inserts, updates, and deletes.

If set to *x*, the driver uses the Salesforce Bulk API to execute the insert, update, or delete operation when the value of *x* is exceeded.

Notes

- If the EnableBulkLoad property is set to `false`, the BulkLoadThreshold property is ignored.
- Do not set the BulkLoadThreshold property to a value greater than the Web service call limit set by the StmtCallLimit property. If the value set for BulkLoadThreshold is greater than the value of StmtCallLimit, the driver would not use the Salesforce Bulk API because the Web service call limit would be reached before the driver reached the threshold to use the Salesforce Bulk API.

Data Source Method

`setBulkLoadThreshold`

Default

`4000` (rows)

Data Type

int

See also

- EnableBulkLoad on page 136
- DataDirect Bulk Load on page 96

BulkLoadVersion

Purpose

Specifies which version of Salesforce Bulk Load API to use for performing bulk load operations.

Valid Values

V1 | V2

Behavior

If set to V1, the driver uses the Salesforce Bulk API V1 for all insert, update, and delete operations.

If set to V2, the driver uses the Salesforce Bulk API V2 for all insert, update, and delete operations.

Notes

- This is applicable if EnableBulkLoad is set to `true`.

Data Source Methods

```
public String getBulkLoadVersion()
```

```
public void setBulkLoadVersion(String)
```

Default Value

V1

Data Type

String

CatalogOptions

Purpose

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

Valid Values

2 | 4

Behavior

If set to 2, result sets do not contain synonyms.

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

Data Source Method

`setCatalogOptions`

Default

2

Data Type

int

ClaimsIssuer

Purpose

Specifies the consumer key or the client ID of the Salesforce Connected App. This property must be specified to use the JWT grant in OAuth 2.0 authentication.

Valid Values

`string`

where:

`string`

is the client ID.

Data Source Method

`setClaimsIssuer`

Default

None

Data Type

String

See also

Configuring the driver to use OAuth 2.0 with JWT grant on page 87

ClaimsSubject on page 124

JWTCertStore on page 144

JWTCertPassword on page 143

JWTCertAlias on page 143

ClaimsSubject

Purpose

Specifies the username of the user. This property must be specified to use the JWT grant in OAuth 2.0 authentication.

Valid Values

string

where:

string

is the username.

Data Source Method

`setClaimsSubject`

Default

None

Data Type

String

See also

Configuring the driver to use OAuth 2.0 with JWT grant on page 87

JWTCertStore on page 144

JWTCertPassword on page 143

JWTCertAlias on page 143

ClaimsIssuer on page 123

ClientID

Purpose

Specifies the consumer key for your application. The driver uses this value when authenticating to a Salesforce instance using OAuth 2.0 (`AuthenticationMethod=oauth2.0`).

Refer to the Salesforce documentation to know how to obtain the consumer key for your application.

Valid Values

string

where:

string

is the consumer key for your application.

Data Source Method

`setClientID`

Default

None

Data Type

String

See also

Configuring OAuth 2.0 authentication on page 83

AuthenticationMethod on page 116

ClientSecret

Purpose

Specifies the consumer secret for your application. This value can optionally be specified when authenticating to a Salesforce instance using OAuth 2.0 (`AuthenticationMethod=oauth2.0`).

Refer to the Salesforce documentation to know how to obtain the consumer secret for your application.

Valid Values

string

where:

string

is the consumer secret for your application.

Data Source Method

`setClientSecret`

Default

None

Data Type

String

See also

Configuring OAuth 2.0 authentication on page 83

AuthenticationMethod on page 116

ClientTimeZone

Purpose

Specifies the time zone other than UTC (Universal Time Coordinated) that should be used when translating data store time and timestamp values between the client and the driver. Most data stores use UTC, while applications use and supply time and timestamp values in local time for the client. If the ClientTimeZone value is not specified (the initial default value), the driver uses system-specific client time zone settings to convert between the UTC time used by the cloud service and the local time used by the client application. The driver returns an error at connect time if it cannot obtain the client time zone. Client time zone settings vary among operating systems:

- On Windows systems, the driver translates the system time zone to an equivalent client time zone.
- On UNIX and Linux systems, the client time zone is set using the `TZ` variable, with the following exceptions:
  - On Linux systems, if `TZ` is NULL or empty, the client time zone comes from the `ZONE` value in the following file: `/etc/sysconfig/clock`
  - On Solaris systems, `TZ` may contain `localtime`. In this case, the driver uses the `/etc/localtime` link to determine the client time zone.

  If you want the connectivity service to verify that this time zone has certain characteristics, you can append some verification information onto the time zone, such as an offset and daylight saving time indicator. The offset specification is the number of hours (and optional minutes) behind (indicated by a leading minus sign) or ahead (indicated by no sign or a plus sign) of UTC. If the time zone is expected to support daylight saving time, append `D` to the offset.

Valid Values

timezone[,[+ | -]*HH*[:*MM*] [D]

where:

timezone

is a valid Java TimeZone ID. See your Java documentation or use the `TimeZone.getAvailableIDs()` method to return a list of valid IDs.

`+ | -`

optionally specifies whether the offset is before or after Greenwich Mean Time.

HH:*MM*

optionally specifies the number of hours and minutes to offset the time from Greenwich Mean Time.

`D`

signifies whether the time zone adjusts for daylight savings time.

Examples

- `America/New_York`
- `America/New_York,-5D`
- `America/New_York,-05:00D`
- `Asia/Calcutta,5:30`
- `Asia/Calcutta,+5:30`

Data Source Method

`setClientTimeZone`

Default

None. The driver determines the client time zone based on the system-specific time zone settings.

Data Type

String

ConfigOptions

Important: The configuration options have been deprecated. However, the driver will continue to support them until the next major release of the driver.

Purpose

Determines how the mapping of the remote Salesforce data model to a local schema map can be configured, customized, and updated.

Notes

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

Valid Values

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

where:

key

is one of the following configuration options:

- `AuditColumns`
- `CustomSuffix`
- `KeywordConflictSuffix`
- `MapSystemColumnNames`

- NumberFieldMapping
- UppercaseIdentifiers

value

specifies a setting for the configuration option.

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

```
ConfigOptions=(AuditColumns=none;CustomSuffix=strip;KeywordConflictSuffix=;
MapSystemColumnNames=1;NumberFieldMapping=emulateInteger;
UppercaseIdentifiers=true)
```

Data Source Method

`setConfigOptions`

Default

```
AuditColumns=all;
CustomSuffix=include;
KeywordConflictSuffix=;
MapSystemColumnNames=0;
NumberFieldMapping=emulateInteger;
UppercaseIdentifiers=true
```

Data Type

String

AuditColumns (Configuration Option)

Purpose

Determines whether the driver includes audit fields, which Salesforce adds to all objects defined in a Salesforce instance, as table columns when mapping the Salesforce data model.

The audit columns added by Salesforce are:

- IsDeleted
- CreatedById
- CreatedDate
- LastModifiedById
- LastModifiedDate
- SystemModstamp

Valid Values

`all` | `auditOnly` | `masterOnly` | `none`

Behavior

If set to `all`, the driver includes the all of the audit columns and the master record id column in its table definitions.

If set to `auditOnly`, the driver adds only the audit columns in its table definitions.

If set to `masterOnly`, the driver adds only the MasterRecordId column in its table definitions.

If set to `none`, the driver does not add the audit columns or the MasterRecordId column in its table definitions.

Notes

In a typical Salesforce instance, not all users are granted access to the Audit or MasterRecordId columns. If AuditColumns is set to a value other than `none` and the driver cannot include the columns requested, the connection fails and the driver generates a SQLException with a SQLState of 08001.

Default

`all`

See also

ConfigOptions on page 127

CustomSuffix (Configuration Option)

Purpose

Determines whether the driver includes or strips the __c suffix from the table and column names when mapping the Salesforce data model. Salesforce adds the suffix to all custom objects and fields.

Valid Values

`include` | `strip`

Behavior

If set to `include`, the driver includes the __c suffix.

If set to `strip`, the driver strips the __c suffix.

Default

`include`

See also

ConfigOptions on page 127

KeywordConflictSuffix (Configuration Option)

Purpose

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

Valid Values

string

where:

string

is a string of up to five alphanumeric characters.

Example

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

Notes

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

Default

Empty string

See also

ConfigOptions on page 127

MapSystemColumnNames (Configuration Option)

Purpose

Determines how the driver maps Salesforce system columns.

Valid Values

0 | 1

Behavior

If set to 0, the driver does not change the names of the Salesforce system columns.

If set to 1, the driver changes the names of the Salesforce system columns as described in the following table:

Table 19: Mapped Names for Salesforce Field Names When Using MapSystemColumnNames

Field Name	Mapped Name
Id	ROWID
Name	SYS_NAME
IsDeleted	SYS_ISDELETED
CreatedDate	SYS_CREATEDDATE
CreatedById	SYS_CREATEDBYID
LastModifiedDate	SYS_LASTMODIFIEDDATE
LastModifiedId	SYS_LASTMODIFIEDID
SystemModstamp	SYS_SYSTEMMODSTAMP
LastActivityDate	SYS_LASTACTIVITYDATE
OwnerId	SYS_OWNERID

Default
0

See also
ConfigOptions on page 127

NumberFieldMapping (Configuration Option)

Purpose
Determines how the driver maps fields defined as NUMBER in Salesforce. The Salesforce API uses DOUBLE values to transfer data to and from NUMBER fields, which can cause problems when the precision of the NUMBER field is greater than the precision of a DOUBLE value. Rounding can occur when converting large values to and from DOUBLE. The NumberFieldMapping option allows you to map the NUMBER fields to the required SQL types based on their precision.

Valid Values
emulateInteger | alwaysDouble | alwaysDecimal

Behavior
If set to emulateInteger, the driver maps NUMBER fields with a precision of 9 or less and a scale of 0 to the INTEGER SQL type and maps all other NUMBER fields to the DOUBLE SQL type.

If set to alwaysDouble, the driver maps all NUMBER fields to the DOUBLE SQL type regardless of their precision.

If set to `alwaysDecimal`, the driver maps all NUMBER fields to the DECIMAL SQL type. It can be used when the precision of the NUMBER field is greater than the precision of a DOUBLE value.

Default
`emulateInteger`

See also
ConfigOptions on page 127

UppercaseIdentifiers (Configuration Option)

Purpose

Specifies whether the driver maps all identifier names to uppercase.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver maps identifiers to uppercase.

If set to `false` , the driver maps identifiers to the mixed case name of the object being mapped. If mixed case identifiers are used, SQL statements must enclose those identifiers in double quotes and the case of the identifier must exactly match the case of the identifier name.

Example

For example, if `UppercaseIdentifiers=false`, to query the Account table you specify:

```
SELECT "Phone", "Website" FROM "Account"
```

Notes

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

Default
`true`

See also
ConfigOptions on page 127

ConnectionRetryCount

Purpose

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

Valid Values

`0` | *x*

where *x* is a positive integer that represents the number of retries.

Behavior

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

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

Example

If this property is set to 2, the driver retries the server twice after the initial retry attempt.

Notes

- If an application sets a login timeout value (for example, using DataSource.loginTimeout or DriverManager.loginTimeout), and the login timeout expires, the driver ceases connection attempts.
- The ConnectionRetryDelay property specifies the wait interval, in seconds, to occur between retry attempts.

Data Source Method

`setConnectionRetryCount`

Default

5

Data Type

int

ConnectionRetryDelay

Purpose

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

Valid Values

`0` | *x*

where:

x

is a number of seconds.

Behavior

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

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

Example

If ConnectionRetryCount is set to 2 and this property is set to 3, the driver retries the server twice after the initial retry attempt. The driver waits 3 seconds between retry attempts.

Data Source Method

`ConnectionRetryDelay`

Default

1 (second)

Data Type

int

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.

Data Source Method

`setConvertNull`

Default

`1`

Data Type

int

CreateMap

Purpose

Specifies whether the driver creates a new map of the Salesforce data model when establishing the connection.

Valid Values

`forceNew` | `notExist` | `no`

Behavior

If set to `forceNew`, the driver deletes the current schema map specified by the SchemaMap property and creates a new one at the same location.

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

If set to `notExist`, the driver uses the current schema map specified by the SchemaMap property. If one does not exist, the driver creates one.

If set to `no`, the driver uses the current schema map specified by the SchemaMap property. If one does not exist, the connection fails.

Notes

Alternatively, you can refresh a schema manually at any time by using the Refresh Map statement. See "Refresh Map (EXT)" for details.

Data Source Method

`setCreateMap`

Default

`notExist`

Data Type

String

See Also

- SchemaMap on page 153
- Refresh Map (EXT) on page 218

EnableBulkFetch

Purpose

Specifies whether the driver can use the Salesforce Bulk API for selects based on the value of the BulkFetchThreshold connection property. If the number of rows expected in the result set exceeds the value of BulkFetchThreshold property, the driver uses the Salesforce Bulk API to execute the select operation. Using the Salesforce Bulk API may significantly reduce the number of Web service calls used to execute a statement and, therefore, may improve performance.

Valid Values

`1 | 0`

Behavior

If set to `1`, the driver can use the Salesforce Bulk API for selects based on the value of the BulkFetchThreshold connection property. If the number of rows expected in the result set exceeds the value of BulkFetchThreshold property, the driver uses the Salesforce Bulk API to execute the select operation.

If set to `0`, the driver does not use the Salesforce Bulk API, and the BulkFetchThreshold property is ignored.

Data Source Method

`setEnableBulkFetch`

Default

`1`

Data Type

boolean

See Also

- BulkFetchThreshold on page 116
- DataDirect Bulk Load on page 96

EnableBulkLoad

Purpose

Specifies whether the driver can use the Salesforce Bulk API for inserts, updates, and deletes based on the value of BulkLoadThreshold connection property. If the number of affected rows exceeds the value of BulkLoadThreshold property, the driver uses the Salesforce Bulk API to execute the insert, update, or delete operation. Using the Salesforce Bulk API may significantly reduce the number of Web service calls used to execute a statement and, therefore, may improve performance.

Valid Values

`true | false`

Behavior

If set to `1`, the driver can use the Salesforce Bulk API for inserts, updates, and deletes based on the value of BulkLoadThreshold connection property. If the number of affected rows exceeds the value of BulkLoadThreshold property, the driver uses the Salesforce Bulk API to execute the insert, update, or delete operation.

If set to `0`, the driver does not use the Salesforce Bulk API, and the BulkLoadThreshold property is ignored.

Data Source Method

`setEnableBulkLoad`

Default

`true`

Data Type

boolean

See Also

- BulkLoadThreshold on page 121
- DataDirect Bulk Load on page 96

EnablePKChunking

Purpose

Specifies whether the driver uses PK chunking for select operations. PK chunking breaks down bulk fetch operations into smaller, more manageable batches for improved performance.

Valid Values

`1|0`

Behavior

If set to `1`, the driver uses PK chunking for select operations when the expected number of rows in the result set is greater than the values of the BulkFetchThreshold and PKChunkSize properties. For this behavior to take effect, the EnableBulkFetch property must also be set to `1`.

If set to `0`, the driver does not use PK chunking when executing select operations, and the PKChunkSize property is ignored.

Notes

- PK chunking is supported for all custom objects and the following standard objects: `Account`, `Campaign`, `CampaignMember`, `Case`, `Contact`, `Lead`, `LoginHistory`, `Opportunity`, `Task`, and `User`. In addition, PK chunking is supported for sharing objects as long as the parent object is supported.

Data Source Method

`setEnablePKChunking`

Default

1

Data Type

boolean

See Also

- BulkFetchThreshold on page 116
- PKChunkSize on page 148
- EnableBulkFetch on page 136

FetchSize

Purpose

Specifies the maximum number of rows that the driver processes before returning data to the application when executing a Select. This value provides a suggestion to the driver as to the number of rows it should internally process before returning control to the application. The driver may fetch fewer rows to conserve memory when processing exceptionally wide rows.

Valid Values

0 | *x*

where:

x

is a positive integer indicating the number of rows that should be processed.

Behavior

If set to 0, the driver processes all the rows of the result before returning control to the application. When large data sets are being processed, setting FetchSize to 0 can diminish performance and increase the likelihood of out-of-memory errors.

If set to *x*, the driver limits the number of rows that may be processed for each fetch request before returning control to the application.

Notes

- To optimize throughput and conserve memory, the driver uses an internal algorithm to determine how many rows should be processed based on the width of rows in the result set. Therefore, the driver may process fewer rows than specified by FetchSize when the result set contains exceptionally wide rows. Alternatively, the driver processes the number of rows specified by FetchSize when the result set contains rows of unexceptional width.
- FetchSize and WSFetchSize can be used to adjust the trade-off between throughput and response time. Smaller fetch sizes can improve the initial response time of the query. Larger fetch sizes can improve overall response times at the cost of additional memory.

- You can use FetchSize to reduce demands on memory and decrease the likelihood of out-of-memory errors. Simply, decrease FetchSize to reduce the number of rows the driver is required to process before returning data to the application.

Data Source Method

setFetchSize

Default

`100` (rows)

Data Type

Int

See also

- Additional Properties on page 68
- Performance Considerations on page 72

ImportStatementPool

Purpose

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

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

Valid Values

`string`

where:

`string`

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

Data Source Method

`setImportStatementPool`

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

Because fetching metadata and generating mapping files can significantly increase the time it takes to connect to Salesforce, the driver caches this information on the client the first time the driver connects on behalf of each user. The cached metadata is used in subsequent connections made by the user instead of re-fetching the metadata from Salesforce. To force the driver to re-fetch the metadata information for a connection, use the InitializationString property to pass the REFRESH SCHEMA SFORCE command in the connection URL. For example:

```
jdbc:datadirect:sforce://login.salesforce.com;User=test@abccorp.com;
Password=secret;InitializationString=(REFRESH SCHEMA SFORCE)
```

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.

Data Source Method

`setInsensitiveResultSetBufferSize`

Default

`2048`

Data Type

int

InValuesLimit

Purpose

Determines the number of values in an `IN` clause subquery to be materialized. If this value is exceeded, the driver processes the subquery as a JOIN clause. You can use this property if your query exceeds the maximum SOQL length set by Salesforce.

Valid Values

x

where:

x

is a positive integer that represents the number of values in an `IN` clause.

Behavior

If set to *x*, the driver limits the number of values in an `IN` clause subquery to the specified value. If the number of values exceeds this limit then the subquery is processed as a `JOIN` clause.

Data Source Method

`setInValuesLimit`

Default

`200`

Data Type

int

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

`1|0`

Behavior

If set to `1`, 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 `1` to use the JVM conversion algorithm.

If set to `0`, the driver uses its own internal algorithm when converting a double or float value to a string value. This value improves performance, but slight rounding differences within the allowable error of the double and float data types can occur when compared to the same conversion using the JVM algorithm.

Data Source Method

`setJavaDoubleToString`

Default

`0`

Data Type

boolean

JWTCertAlias

Purpose

Specifies the alias for the JWT certificate stored in the keystore. This property is optional.

Valid Values

string

where:

string

is the alias for the JWT certificate.

Data Source Method

`setJwtCertAlias`

Default

None

Data Type

String

See also

Configuring the driver to use OAuth 2.0 with JWT grant on page 87

JWTCertStore on page 144

JWTCertPassword on page 143

ClaimsIssuer on page 123

ClaimsSubject on page 124

JWTCertPassword

Purpose

Specifies the password of the JWT certificate, if the certificate is password protected. This connection property is optional.

Valid Values

string

where:

string

is the password for the certificate.

Data Source Method

`setJwtCertPassword`

Default

None

Data Type

String

See also

Configuring the driver to use OAuth 2.0 with JWT grant on page 87

ClaimsIssuer on page 123

ClaimsSubject on page 124

JWTCertStore on page 144

JWTCertAlias on page 143

JWTCertStore

Purpose

Specifies the file path of the certificate store containing the private key used for JWT authentication. This property must be specified to use the JWT grant in OAuth 2.0 authentication.

Valid Values

string

where:

string

is the file path for the JWT certificate store.

Data Source Method

`setJwtCertStore`

Default

None

Data Type

String

See also

Configuring the driver to use OAuth 2.0 with JWT grant on page 87

JWTCertAlias on page 143

JWTCertPassword on page 143

ClaimsIssuer on page 123

ClaimsSubject on page 124

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 as described in "Using Java Logging."

Data Source Method

`setLogConfigFile`

Default

`ddlogging.properties`

Data Type

String

See also

Using Java Logging on page 175

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.

Data Source Method

`setLoginTimeout`

Default

`0`

Data Type

int

MaxPooledStatements

Purpose

The maximum number of pooled prepared statements for this connection. Setting MaxStatements to an integer greater than zero (0) enables the driver's internal prepared statement pooling, which is useful when the driver is not running from within an application server or another application that provides its own prepared statement pooling.

Valid Values

`0` | *x*

where

x

is a positive integer that represents a number of pooled prepared statements.

Behavior

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

If set to *x*, the driver enables the DataDirect Statement Pool and uses the specified value to cache a certain number of prepared statements created by an application. If the value set for this property is greater than the number of prepared statements that are used by the application, all prepared statements that are created by the application are cached. Because CallableStatement is a sub-class of PreparedStatement, CallableStatements also are cached.

Example

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

Data Source Method

`setMaxPooledStatements`

Default

`0`

Data Type

int

Password

Description

Specifies the password to use to connect to your Salesforce 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

`password` | *`password+securitytoken`*

where:

`password`

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

`password+securitytoken`

is a valid password appended by the security token required to connect to the Salesforce instance, for example, `secretXaBARTsLZReM4Px47qPL0S`, where `secret` is the password and the remainder of the value is the security token. Both the password and security token are case-sensitive.

Note: Optionally, you can specify the security token in the SecurityToken property. Do not specify the security token in both properties.

Data Source Method

`setPassword`

Default

None

Data Type

String

PKChunkSize

Purpose

Specifies the size, in rows, of a primary key chunk when PK chunking has been enabled via the EnablePKChunking property. The Salesforce Bulk API splits the query into chunks of this size.

Valid Values

x

where:

x

is a positive integer less than or equal to 250,000 rows.

Behavior

The driver requests, via the Salesforce Bulk API, that Salesforce divide the query into chunks of this size.

Notes

- Fewer rows may be returned if a `WHERE` clause has been applied to the fetch operation, or if soft-deleted records are included within the boundaries of a given chunk.
- Each chunk is processed as a separate batch that counts toward your daily batch limit. Larger chunks will result in fewer batches, but may reduce performance. PKChunkSize may be set to a maximum value of 250,000 rows.
- For PK chunking to be used in select operations, the expected number of rows in the result set must be greater than the values of the BulkFetchThreshold and PKChunkSize properties.

Data Source Method

`setPKChunkSize`

Default

`100000` (rows)

Data Type

long

See also

- EnablePKChunking on page 137
- BulkFetchThreshold on page 116

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 "Using IP Addresses" for details about using these formats.

Data Source Method

`setProxyHost`

Default

empty string

See also

- IP Addresses on page 89
- Connecting Through a Proxy Server on page 72

ProxyPassword

Purpose

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

Valid Values

password

where:

password

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

Data Source Method

`setProxyPassword`

Default

empty string

See also

Connecting Through a Proxy Server on page 72

ProxyPort

Purpose

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

Valid Values

port

where:

port

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

Data Source Method

`setProxyPort`

Default

`0`

See also

Connecting Through a Proxy Server on page 72

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.

Data Source Method

`setProxyUser`

Default

empty string

See also

Connecting Through a Proxy Server on page 72

ReadOnly

Purpose

Specifies whether the connection has read-only access to the data source.

Valid Values

`1 | 0`

Behavior

If set to `1`, the connection has read-only access. The following commands are the only commands that you can use when a connection if read-only:

- Call* (if the procedure does not update data)
- Explain Plan
- Select (except Select Into)
- Set Database Collation
- Set IgnoreCase
- Set Maxrows
- Set Schema

The driver generates an exception if any other command is executed.

If set to `0`, the connection is opened for read/write access, and you can use all commands supported by the product.

Notes

- You also can use the JDBC connection method setReadOnly to set a read-only state for a connection.

Data Source Method

`setReadOnly`

Default

`0`

Data Type

boolean

RefreshToken

Purpose

Specifies the refresh token used to either request a new access token or renew an expired access token. When the refresh token is specified, the access token generated at connection is used to authenticate to a Salesforce instance when OAuth 2.0 is enabled (`AuthenticationMethod=oauth2.0`).

Refer to the Salesforce documentation to know how to obtain a refresh token.

Valid Values

`string`

where:

`string`

is the refresh token you have obtained from Salesforce.

Notes

- If a value for the AccessToken property is not specified, the driver uses the value of the RefreshToken property to make a connection.
- If both AccessToken and RefreshToken values are not specified, the driver cannot make a successful connection.
- If both AccessToken and RefreshToken values are specified, the driver ignores the AccessToken value and uses the RefreshToken value to generate a new AccessToken value.

Data Source Method

`setRefreshToken`

Default

None

Data Type

String

See also

Configuring OAuth 2.0 authentication on page 83

AuthenticationMethod on page 116

AccessToken on page 114

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

-
- In addition to `true` and `false`, the driver accepts `1`, `0`, `on`, and `off` as valid values. `1` and `On` are equivalent to `true`, and `0` and `Off` are equivalent to `false`.

Data Source Method

setRegisterStatementPoolMonitorMBean

Default

`false`

Data Type

Boolean

See also

- Refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference* for further details.
- Statement Pooling Properties on page 67
- MaxPooledStatements on page 146

SchemaMap

Purpose

Specifies either the name or the absolute path and name of the configuration file where the map of the Salesforce data model is written. The driver looks for this file when connecting to a Salesforce instance. If the file does not exist, the driver creates one.

Valid Values

string

where:

string

is either the name or the absolute path and name (including the `.config` extension) of the configuration file. For example, if SchemaMap is set to a value of:

- `ABC`, the driver either creates or looks for the configuration file `ABC` in the working directory of your application.
- `C:\\Users\\Default\\AppData\\Local\\Progress\\DataDirect\\SForce_Schema\\abc@defcorp.com.config`, the driver either creates or looks for the configuration file `abc@defcorp.com.config` in the directory `C:\Users\Default\AppData\Local\Progress\DataDirect\SForce_Schema`.

Notes

- If using OAuth 2.0 authentication, a value for the SchemaMap property must be specified for every connection.
- When connecting to a Salesforce instance, the driver looks for the schema map configuration file. If the configuration file does not exist, the driver creates the schema map configuration file using the name and location you have provided. If you do not provide a name and location for the configuration file, the driver creates it using default values.
- The driver uses the path specified in this connection property to store additional internal files.
- You can refresh the internal files related to an existing view of your data by using the SQL extension Refresh Map. Refresh Map runs a discovery against your native data and updates your internal files accordingly.

Example

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

Driver Manager Example

```
jdbc:datadirect:sforce://login.salesforce.com;User=abc@defcorp;Password=secret;
SecurityToken=XaBARTsLZReM4Px47qPLOS;
SchemaMap=C:\Users\Default\AppData\Local\Progress\DataDirect\
SForce_Schema\abc@defcorp.com.config
```

Data Source Example

```
SForceDataSource ds = new SForceDataSource();
ds.setDescription("My Salesforce DataSource");
ds.setServerName("login.salesforce.com");
ds.setUser("abc@defcorp.com");
ds.setPassword("secret");
ds.setSecurityToken("XaBARTsLZReM4Px47qPLOS");
ds.setSchemaMap("C:\\Users\\Default\\AppData\\Local\\Progress
             \\DataDirect\\SForce_Schema\\abc@defcorp.com.config")
```

Data Source Method

`setSchemaMap`

Default

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

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

  For Windows, the file path takes the following format:

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

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

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

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

Data Type

String

See also

Refresh Map (EXT) on page 218

Configuring OAuth 2.0 authentication on page 83

SecurityToken

Purpose

Specifies the security token required to make a connection to a Salesforce instance that is configured for a security token. If a security token is required and you do not supply one, the driver returns an error indicating that an invalid user or password was supplied. Contact your Salesforce administrator to find out if a security token is required.

Important: If setting the security token using a data source, be aware that the SecurityToken property, like all data source properties, is persisted in clear text.

Valid Values

string

where:

string

is the value of the security token assigned to the user.

Notes

- Optionally, you can specify the security token in the Password property by appending the security token to the password, for example, `secretXaBARTsLZReM4Px47qPL0S`, where `secret` is the password and the remainder of the value is the security token. Do not specify the security token in both properties.
- A security token is not required when Salesforce has been configured for Trusted IP Ranges and the user is logging in from a trusted IP address. Refer to "Set Trusted IP Ranges for Your Organization" in your Salesforce documentation for details.
- If setting the security token using a data source, be aware that the SecurityToken property, like all data source properties, is persisted in clear text.

Data Source Method

`setSecurityToken`

Default

None

Data Type

String

ServerName

Purpose

Specifies the base Salesforce URL to use for logging in.

Valid Values

url

where:

url

is the root of the Salesforce URL to which you want to connect.

Example

Suppose you have a Salesforce instance that is configured with a production instance and a sandbox instance. You can specify `login.salesforce.com` as the value for the ServerName property to connect to the production instance or `test.salesforce.com` to connect to the sandbox instance:

Salesforce Instance	URL
Production	`login.salesforce.com`
Sandbox	`test.salesforce.com`

Data Source Method

`setServerName`

Default

`login.salesforce.com`

Data Type

String

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. See "DataDirect Spy Attributes" for a list of supported attributes.

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).

Attribute	Description
log=(file)*filename*	Directs logging to the file specified by *filename*. For Windows, if coding a path to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: log=(file)C:\\temp\\spy.log;logIS=yes;logTName=yes.
log=(filePrefix)*file_prefix*	Directs logging to a file prefixed by *file_prefix*. The log file is named *file_prefixX*.log where: *X* is an integer that increments by 1 for each connection on which the prefix is specified. For example, if the attribute log=(filePrefix)C:\\temp\\spy_ is specified on multiple connections, the following logs are created: C:\temp\spy_1.log C:\temp\spy_2.log C:\temp\spy_3.log ... If coding a path to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: log=(filePrefix)C:\\temp\\spy_;logIS=yes;logTName=yes.
log=System.out	Directs logging to the Java output standard, System.out.
logIS={ yes | no | nosingleread }	Specifies whether DataDirect Spy logs activity on InputStream and Reader objects. When logIS=nosingleread, logging on InputStream and Reader objects is active; however, logging of the single-byte read InputStream.read or single-character Reader.read is suppressed to prevent generating large log files that contain single-byte or single character read messages. The default is no.
logLobs={ yes | no }	Specifies whether DataDirect Spy logs activity on BLOB and CLOB objects.

Attribute	Description
logTName={yes | no}	Specifies whether DataDirect Spy logs the name of the current thread. The default is no.
timestamp={yes | no}	Specifies whether a timestamp is included on each line of the DataDirect Spy log. The default is no.

Notes

- If coding a path on Windows to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: `log=(file)C:\\temp\\spy.log`.
- If a log file name does not include the `.log` extension, the driver automatically appends it. For example, a file named `spy.jsp` is renamed to `spy.jsp.log` by the driver.

Example

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

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

Data Source Method

setSpyAttributes

Default

Empty string

Data Type

String

See also

- Tracking JDBC calls with DataDirect Spy on page 103
- DataDirect Spy attributes on page 106

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

`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 `0`, there is no limit.

If set to x, the driver uses this value to set the maximum number of Web service calls on a single connection that can be made when executing a SQL statement. This limit 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.

Data Source Method

`setStmtCallLimit`

Default

`100` (Web service calls)

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.

Data Source Method

`setStmtCallLimitBehavior`

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.

Data Source Method

`setTransactionMode`

Default

`noTransactions`

Data Type

String

User

Purpose

Specifies the user name that is used to connect to the Salesforce instance. A user name is required.

Valid Values

string

where:

`string`

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

Data Source Method

`setUser`

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.

Data Source Method

setWSCompressData

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 from 1 to 2000 that defines a number of rows.

Behavior

If set to `0`, the driver attempts to fetch up to a maximum of 2000 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 2000 can reduce the response time for returning the initial data. Consider using a smaller WSFetch size for interactive applications only.

Notes

WSFetchSize and FetchSize can be used to adjust the trade-off between throughput and response time. Smaller fetch sizes can improve the initial response time of the query. Larger fetch sizes can improve overall response times at the cost of additional memory.

Data Source Method

setWSFetchSize

Default

`0` (up to a maximum of 2000 rows)

Data Type

Int

See also

- Web Service Properties on page 62
- Performance Considerations on page 72

WSPoolSize

Purpose

Specifies the maximum number of Salesforce sessions the driver uses. This allows the driver to have multiple web service requests active when multiple JDBC connections are open, thereby improving throughput and performance.

Valid Values

x

where:

x

is the number of Salesforce sessions the driver uses to distribute calls. This value should not exceed the number of sessions permitted by your Salesforce account.

Notes

- You can improve performance by increasing the number of sessions specified by this property. By increasing the number of sessions the driver uses, you can improve throughput by distributing calls across multiple sessions when multiple connections are active.
- The maximum number of sessions is determined by the setting of WSPoolSize for the connection that initiates the session. For subsequent connections to an active session, the setting is ignored and a warning is returned. To change the maximum number of sessions, close all connections using the Salesforce driver; then, open a new Salesforce connection with desired limit specified for this property.

Data Source Method

`setWSPoolSize`

Default

`1`

Data Type

Int

See also

Performance Considerations on page 72

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.

Data Source Method

setWSRetryCount

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.

Data Source Method

setWSTimeout

Default

`120` (seconds)

Data Type

Int

5

Troubleshooting

This section provides information that can help you troubleshoot problems when they occur.

For details, see the following topics:

- Troubleshooting your application
- Troubleshooting connection pooling
- Troubleshooting statement pooling
- Using Java Logging

Troubleshooting your application

To help you troubleshoot any problems that occur with your application, you can use DataDirect Spy to log detailed information about calls issued by the drivers on behalf of your application. When you enable DataDirect Spy for a connection, you can customize DataDirect Spy logging by setting one or multiple options. See "Tracking JDBC calls with DataDirect Spy" for information about using DataDirect Spy and instructions on enabling and customizing logging.

See also

Tracking JDBC calls with DataDirect Spy on page 103

Turning On and Off DataDirect Spy Logging

Once DataDirect Spy logging is enabled for a connection, you can turn on and off the logging at runtime using the `setEnableLogging` method in the `com.ddtek.jdbc.extensions.ExtLogControl` interface. When DataDirect Spy logging is enabled, all Connection objects returned to an application provide an implementation of the `ExtLogControl` interface.

The following code snippet shows how to turn off logging using `setEnableLogging(false)`.

```
import com.ddtek.jdbc.extensions.*

// Get Database Connection
Connection con = DriverManager.getConnection
   ("jdbc:datadirect:sforce://login.salesforce.com;User=abc@defcorp.com;
   Password=secret;SecurityToken=XaBARTsLZReM4Px47qPLOS;
   SpyAttributes=(log=(filePrefix)/tmp/spy_;logTName=yes;timestamp=yes)");

((ExtLogControl) con).setEnableLogging(false);
...
```

The `setEnableLogging` method only turns on and off logging if DataDirect Spy logging has already been enabled for a connection; it does not set or change DataDirect Spy attributes. See "Enabling DataDirect Spy" for information about enabling and customizing DataDirect Spy logging.

If Java logging is also enabled and Java API logger levels in the properties file are set to `FINER` or `FINEST`, the driver ignores the file settings of Spy logging and logs all the logging information (Spy logs and Java logs) into the log file defined by the Java logging config settings.

For example:

If both Spy logging and Java logging are specified in the URL:

```
LogConfigFile=C:\\props\\spy\\ddlogging_DD.properties;
SpyAttributes=(log=(file)spy.log;timestamp=yes)
```

And the Java logger levels are set to `FINER` or `FINEST`:

```
# logger for web service adapter: use CONFIG, FINE, FINER, FINEST
datadirect.cloud.adapter.level = FINEST

# logger for sql engine: use CONFIG, FINE, FINER, FINEST
datadirect.cloud.sql.level = FINEST

# logger for jdbc spy: use FINER or FINEST
datadirect.jdbc.cloud.level = FINEST
```

And Java logging argument is:

```
java -Djava.util.logging.config.file=myfile
```

Then both the Spy logs and Java logs will be logged into the log file defined by the Java logging argument, which is `myfile` in this example.

For more information about Java logging, see "Using Java Logging."

See also

Enabling DataDirect Spy on page 104
Using Java Logging on page 175

DataDirect Spy Log Example

This section provides information to help you understand the content of your own DataDirect Spy logs.

For example, suppose your application executes the following code and performs some operations:

```
Class.forName("com.ddtek.jdbc.sforce.SForceDriver");
DriverManager.getConnection("jdbc:datadirect:sforce://login.salesforce.com;
User=abc@defcorp.com;Password=secret;SecurityToken=XaBARTsLZReM4Px47qPLOS;
spyAttributes=(log=(file)c:\\temp\\spy.log)");
```

The log file generated by DataDirect Spy would look similar to the following example. Notes provide explanations for the referenced text.

```
spy>> Connection[1].getMetaData()
spy>> OK (DatabaseMetaData[1])

spy>> DatabaseMetaData[1].getURL()
spy>> OK
(jdbc:datadirect:sforce:;MAXSOQLLENGTH=20000;JDBCBEHAVIOR=1;LOGINHOST=;APPLICATIONNAME=;
WSFETCHSIZE=2000;PROXYHOST=;CATALOGOPTIONS=2;BULKLOADCONCURRENCYMODE=PARALLEL;LOGSOAP=false;
WSPOOLSIZE=1;BULKLOADBATCHSIZE=10000;ENABLEBULKLOAD=false;STMTCALLLIMIT=0;
CONNECTIONRETRYDELAY=1;READONLY=false;CLIENTUSER=;BULKLOADTHRESHOLD=4000;WORKAROUNDS=0;
CONVERTNULL=1;RECORDRESTEVENTS=false;CONNECTIONRETRYCOUNT=5;READAHEAD=0;CUSTOMSUFFIX=strip;
TOKEN=;STMTCALLLIMITBEHAVIOR=ErrorAlways;LOADLIBRARYPATH=;ENABLEHTTPCHUNKING=true;
MAXPOOLEDSTATEMENTS=0;CRYPTOPROTOCOLVERSION=;QUERYTIMEOUT=0;PROXYPASSWORD=;
SERIALIZECREATECOLUMNS=false;TRANSACTIONMODE=NoTransactions;WSRETRYCOUNT=0;PROGRAMID=;
PROXYPORT=0;WSCOMPRESSDATA=Compress;SECURERANDOMALGORITHM=;DEBUGPLAYBACK=;
REGISTERSTATEMENTPOOLMONITORMBEAN=false;BULKLOADPOLLINTERVAL=10;IMPORTSTATEMENTPOOL=;SERVERPATH=;
LOADBALANCING=false;LOGINTIMEOUT=0;WSTIMEOUT=120;RANDOMGENERATOR=SECURERANDOM;PROXYUSER=;
FAILOVERGRANULARITY=nonAtomic;ACCOUNTINGINFO=;LOGCONFIGFILE=ddlogging.properties;
ENCRYPTIONMETHOD=noEncryption;FAILOVERMODE=connect;INITIALIZATIONSTRING=;
BATCHPERFORMANCEWORKAROUND=false;JAVADOUBLETOSTRING=false;ENABLETESTPROCS=false;DEBUGRECORD=;
AUTHENTICATIONMETHOD=none;CLIENTHOSTNAME=;CONFIGOPTIONS=;CREATEDB=NotExist;REFRESHSCHEMA=false;
RESULTSETMETADATAOPTIONS=0;FAILOVERPRECONNECT=false;D2CDATASTOREID=0;
SPYATTRIBUTES=(log=(file)c:\temp\spy.log);MAXDESCRIBEOBJECTS=100;INSENSITIVERESULTSETBUFFERSIZE=2048;
FETCHSIZE=100;SECURITYTOKEN=89mqiEVzSigEXmr7vQqCoOuN2;SCHEMAMAP=;ENABLEREPORTPARAMETERS=false;
ALTERNATESERVERS=;BULKLOADASYNC=false)
[5]
spy>> DatabaseMetaData[1].getDriverName()
spy>> OK (SForce)

spy>> DatabaseMetaData[1].getDriverVersion()
spy>> OK (6.0.0.0000 (C0000.F000000.U000000))

spy>> DatabaseMetaData[1].getDatabaseProductName()
spy>> OK (Salesforce)

spy>> DatabaseMetaData[1].getDatabaseProductVersion()
spy>> OK (41.0)

spy>> Connection Options :[6]
spy>>   MAXSOQLLENGTH=20000
spy>>   JDBCBEHAVIOR=1
spy>>   LOGINHOST=
spy>>   APPLICATIONNAME=
spy>>   WSFETCHSIZE=2000
spy>>   ...
spy>>   LOGINTIMEOUT=0
spy>>   WSTIMEOUT=120
spy>>   RANDOMGENERATOR=SECURERANDOM
spy>>   PROXYUSER=
spy>>   FAILOVERGRANULARITY=nonAtomic
spy>>   ...
spy>>   SCHEMAMAP=
spy>>   ENABLEREPORTPARAMETERS=false
```

[5] The combination of the URL specified by the application and the default values of all connection properties not specified.

[6] The combination of the connection properties specified by the application and the default values of all connection properties not specified.

```
spy>>   ALTERNATESERVERS=
spy>>   BULKLOADASYNC=false
spy>> Driver Name = SForce[7]
spy>> Driver Version = 6.0.0.0000 (C0000.F000000.U000000))[8]
spy>> Database Name = Salesforce[9]
spy>> Database Version = 41.0[10]
spy>> Connection[1].getWarnings()
spy>> OK[11]
spy>> Connection[1].createStatement
spy>> OK (Statement[1])
spy>> Statement[1].executeQuery(String sql)
spy>> sql = select empno,ename,job from emp where empno=7369
spy>> OK (ResultSet[1])[12]
spy>> ResultSet[1].getMetaData()
spy>> OK (ResultSetMetaData[1])[13]
spy>> ResultSetMetaData[1].getColumnCount()
spy>> OK (3)[14]
spy>> ResultSetMetaData[1].getColumnLabel(int column)
spy>> column = 1
spy>> OK (EMPNO)[15]
spy>> ResultSetMetaData[1].getColumnLabel(int column)
spy>> column = 2
spy>> OK (ENAME)[16]
spy>> ResultSetMetaData[1].getColumnLabel(int column)
spy>> column = 3
spy>> OK (JOB)[17]
spy>> ResultSet[1].next()
spy>> OK (true)[18]
spy>> ResultSet[1].getString(int columnIndex)
spy>> columnIndex = 1
spy>> OK (7369)[19]
spy>> ResultSet[1].getString(int columnIndex)
spy>> columnIndex = 2
spy>> OK (SMITH)[20]
spy>> ResultSet[1].getString(int columnIndex)
spy>> columnIndex = 3
spy>> OK (CLERK)[21]
spy>> ResultSet[1].next()
spy>> OK (false)[22]
spy>> ResultSet[1].close()
spy>> OK[23]
spy>> Connection[1].close()
spy>> OK[24]
```

[7] The name of the driver.
[8] The version of the driver.
[9] The name of the database server to which the driver connects.
[10] The version of the database to which the driver connects.
[11] The application checks to see if there are any warnings. In this example, no warnings are present.
[12] The SELECT statement is executed.
[13] Some metadata is requested.
[14] Some metadata is requested.
[15] Some metadata is requested.
[16] Some metadata is requested.
[17] Some metadata is requested.
[18] The first row is retrieved and the application retrieves the result values.
[19] The first row is retrieved and the application retrieves the result values.
[20] The first row is retrieved and the application retrieves the result values.
[21] The first row is retrieved and the application retrieves the result values.
[22] The application attempts to retrieve the next row, but only one row was returned for this query.
[23] After the application has completed retrieving result values, the result set is closed.
[24] The application finishes and disconnects.

Troubleshooting connection pooling

Connection pooling allows connections to be reused rather than created each time a connection is requested. If your application is using connection pooling through the DataDirect Connection Pool Manager, you can generate a trace file that shows all the actions taken by the Pool Manager. See "Connection Pool Manager" for information about using the Pool Manager.

Enabling tracing with the setTracing method

You can enable Pool Manager logging by calling `setTracing(true)` on the `PooledConnectionDataSource` connection. To disable tracing, call `setTracing(false)` on the connection.

By default, the DataDirect Connection Pool Manager logs its pool activities to the standard output `System.out`. You can change where the Pool Manager trace information is written by calling the `setLogWriter()` method on the `PooledConnectionDataSource` connection.

Pool Manager Trace File Example

The following example shows a DataDirect Connection Pool Manager trace file. Notes provide explanations for the referenced text to help you understand the content of your own Pool Manager trace files.

```
jdbc/SalesforceNCMarkBPool: *** ConnectionPool Created[25]
   (jdbc/SalesforceNCMarkBPool[26],
   com.ddtek.jdbcx.sforce.SForceDataSource@1835282, 5, 5, 10, abc@defcorp.com)[27]
jdbc/SalesforceNCMarkBPool: Number pooled connections = 0.
jdbc/SalesforceNCMarkBPool: Number free connections = 0.

jdbc/SalesforceNCMarkBPool: Enforced minimum![28]
NrFreeConnections was: 0
jdbc/SalesforceNCMarkBPool: Number pooled connections = 5.
jdbc/SalesforceNCMarkBPool: Number free connections = 5.

jdbc/SalesforceNCMarkBPool: Reused free connection.[29]
jdbc/SalesforceNCMarkBPool: Number pooled connections = 5.
jdbc/SalesforceNCMarkBPool: Number free connections = 4.

jdbc/SalesforceNCMarkBPool: Reused free connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 5.
jdbc/SalesforceNCMarkBPool: Number free connections = 3.

jdbc/SalesforceNCMarkBPool: Reused free connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 5.
jdbc/SalesforceNCMarkBPool: Number free connections = 2.

jdbc/SalesforceNCMarkBPool: Reused free connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 5.
jdbc/SalesforceNCMarkBPool: Number free connections = 1.

jdbc/SalesforceNCMarkBPool: Reused free connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 5.
```

[25] The Pool Manager creates a connection pool.

[26] The JNDI name used to look up the connection pool

[27] The DataSource class associated with the connection pool followed by the initial pool size, the minimum pool size, the maximum pool size, and the user name.

[28] The Pool Manager checks the pool size. Because the minimum pool size is five connections, the Pool Manager creates new connections to satisfy the minimum pool size.

[29] The driver requests a connection from the connection pool. The driver retrieves an available connection.

```
jdbc/SalesforceNCMarkBPool: Number free connections = 0.

jdbc/SalesforceNCMarkBPool: Created new connection.[30]
jdbc/SalesforceNCMarkBPool: Number pooled connections = 6.
jdbc/SalesforceNCMarkBPool: Number free connections = 0.

jdbc/SalesforceNCMarkBPool: Created new connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 7.
jdbc/SalesforceNCMarkBPool: Number free connections = 0.

jdbc/SalesforceNCMarkBPool: Created new connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 8.
jdbc/SalesforceNCMarkBPool: Number free connections = 0.

jdbc/SalesforceNCMarkBPool: Created new connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 9.
jdbc/SalesforceNCMarkBPool: Number free connections = 0.

jdbc/SalesforceNCMarkBPool: Created new connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 10.
jdbc/SalesforceNCMarkBPool: Number free connections = 0.

jdbc/SalesforceNCMarkBPool: Created new connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 11.
jdbc/SalesforceNCMarkBPool: Number free connections = 0.

jdbc/SalesforceNCMarkBPool: Connection was closed and added to the cache.[31]
jdbc/SalesforceNCMarkBPool: Number pooled connections = 11.
jdbc/SalesforceNCMarkBPool: Number free connections = 1.

jdbc/SalesforceNCMarkBPool: Connection was closed and added to the cache.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 11.
jdbc/SalesforceNCMarkBPool: Number free connections = 2.

jdbc/SalesforceNCMarkBPool: Connection was closed and added to the cache.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 11.
jdbc/SalesforceNCMarkBPool: Number free connections = 3.

jdbc/SalesforceNCMarkBPool: Connection was closed and added to the cache.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 11.
jdbc/SalesforceNCMarkBPool: Number free connections = 4.

jdbc/SalesforceNCMarkBPool: Connection was closed and added to the cache.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 11.
jdbc/SalesforceNCMarkBPool: Number free connections = 5.

jdbc/SalesforceNCMarkBPool: Connection was closed and added to the cache.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 11.
jdbc/SalesforceNCMarkBPool: Number free connections = 6.

jdbc/SalesforceNCMarkBPool: Connection was closed and added to the cache.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 11.
jdbc/SalesforceNCMarkBPool: Number free connections = 7.

jdbc/SalesforceNCMarkBPool: Connection was closed and added to the cache.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 11.
jdbc/SalesforceNCMarkBPool: Number free connections = 8.

jdbc/SalesforceNCMarkBPool: Connection was closed and added to the cache.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 11.
jdbc/SalesforceNCMarkBPool: Number free connections = 9.

jdbc/SalesforceNCMarkBPool: Connection was closed and added to the cache.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 11.
jdbc/SalesforceNCMarkBPool: Number free connections = 10.
```

[30] The driver requests a connection from the connection pool. Because a connection is unavailable, the Pool Manager creates a new connection for the request.

[31] A connection is closed by the application and returned to the connection pool.

```
jdbc/SalesforceNCMarkBPool: Connection was closed and added to the cache.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 11.
jdbc/SalesforceNCMarkBPool: Number free connections = 11.

jdbc/SalesforceNCMarkBPool: Enforced minimum![32]
NrFreeConnections was: 11
jdbc/SalesforceNCMarkBPool: Number pooled connections = 11.
jdbc/SalesforceNCMarkBPool: Number free connections = 11.

jdbc/SalesforceNCMarkBPool: Enforced maximum![33]
NrFreeConnections was: 11
jdbc/SalesforceNCMarkBPool: Number pooled connections = 10.
jdbc/SalesforceNCMarkBPool: Number free connections = 10.

jdbc/SalesforceNCMarkBPool: Enforced minimum!
NrFreeConnections was: 10
jdbc/SalesforceNCMarkBPool: Number pooled connections = 10.
jdbc/SalesforceNCMarkBPool: Number free connections = 10.

jdbc/SalesforceNCMarkBPool: Enforced maximum!
NrFreeConnections was: 10
jdbc/SalesforceNCMarkBPool: Number pooled connections = 10.
jdbc/SalesforceNCMarkBPool: Number free connections = 10.

jdbc/SalesforceNCMarkBPool: Enforced minimum!
NrFreeConnections was: 10
jdbc/SalesforceNCMarkBPool: Number pooled connections = 10.
jdbc/SalesforceNCMarkBPool: Number free connections = 10.

jdbc/SalesforceNCMarkBPool: Enforced maximum!
NrFreeConnections was: 10
jdbc/SalesforceNCMarkBPool: Number pooled connections = 10.
jdbc/SalesforceNCMarkBPool: Number free connections = 10.

jdbc/SalesforceNCMarkBPool: Dumped free connection.[34]
jdbc/SalesforceNCMarkBPool: Number pooled connections = 9.
jdbc/SalesforceNCMarkBPool: Number free connections = 9.

jdbc/SalesforceNCMarkBPool: Dumped free connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 8.
jdbc/SalesforceNCMarkBPool: Number free connections = 8.

jdbc/SalesforceNCMarkBPool: Dumped free connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 7.
jdbc/SalesforceNCMarkBPool: Number free connections = 7.

jdbc/SalesforceNCMarkBPool: Dumped free connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 6.
jdbc/SalesforceNCMarkBPool: Number free connections = 6.

jdbc/SalesforceNCMarkBPool: Dumped free connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 5.
jdbc/SalesforceNCMarkBPool: Number free connections = 5.

jdbc/SalesforceNCMarkBPool: Dumped free connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 4.
jdbc/SalesforceNCMarkBPool: Number free connections = 4.

jdbc/SalesforceNCMarkBPool: Dumped free connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 3.
jdbc/SalesforceNCMarkBPool: Number free connections = 3.
```

[32] The Pool Manager checks the pool size. Because the number of connections in the connection pool is greater than the minimum pool size, five connections, no action is taken by the Pool Manager.

[33] The Pool Manager checks the pool size. Because the number of connections in the connection pool is greater than the maximum pool size, 10 connections, a connection is closed and discarded from the pool.

[34] The Pool Manager detects that a connection was idle in the connection pool longer than the maximum idle timeout. The idle connection is closed and discarded from the pool.

```
jdbc/SalesforceNCMarkBPool: Dumped free connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 2.
jdbc/SalesforceNCMarkBPool: Number free connections = 2.

jdbc/SalesforceNCMarkBPool: Dumped free connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 1.
jdbc/SalesforceNCMarkBPool: Number free connections = 1.

jdbc/SalesforceNCMarkBPool: Dumped free connection.
jdbc/SalesforceNCMarkBPool: Number pooled connections = 0.
jdbc/SalesforceNCMarkBPool: Number free connections = 0.

jdbc/SalesforceNCMarkBPool: Enforced minimum![35]
NrFreeConnections was: 0
jdbc/SalesforceNCMarkBPool: Number pooled connections = 5.
jdbc/SalesforceNCMarkBPool: Number free connections = 5.

jdbc/SalesforceNCMarkBPool: Enforced maximum!
NrFreeConnections was: 5
jdbc/SalesforceNCMarkBPool: Number pooled connections = 5.
jdbc/SalesforceNCMarkBPool: Number free connections = 5.

jdbc/SalesforceNCMarkBPool: Closing a pool of the group
      jdbc/SalesforceNCMarkBPool[36]
jdbc/SalesforceNCMarkBPool: Number pooled connections = 5.
jdbc/SalesforceNCMarkBPool: Number free connections = 5.

jdbc/SalesforceNCMarkBPool: Pool closed[37]
jdbc/SalesforceNCMarkBPool: Number pooled connections = 0.
jdbc/SalesforceNCMarkBPool: Number free connections = 0.
```

Troubleshooting statement pooling

Similar to connection pooling, statement pooling provides performance gains for applications that execute the same SQL statements multiple times in the life of the application. The DataDirect Statement Pool Monitor provides the following functionality to help you troubleshoot problems that may occur with statement pooling:

- You can generate a statement pool export file that shows you all statements in the statement pool. Each statement pool entry in the file includes information about statement characteristics such as the SQL text used to generate the statement, statement type, result set type, and result set concurrency type.
- You can use the following methods of the `ExtStatementPoolMonitorMBean` interface to return useful information to determine if your workload is using the statement pool effectively:
  - The `getHitCount` method returns the hit count for the statement pool. The hit count should be high for good performance.
  - The `getMissCount` method returns the miss count for the statement pool. The miss count should be low for good performance.

See also

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

[35] The Pool Manager detects that the number of connections dropped below the limit set by the minimum pool size, five connections. The Pool Manager creates new connections to satisfy the minimum pool size.

[36] The Pool Manager closes one of the connection pools in the pool group. A pool group is a collection of pools created from the same PooledConnectionDataSource call. Different pools are created when different user IDs are used to retrieve connections from the pool. A pool group is created for each user ID that requests a connection. In our example, because only one user ID was used, only one pool group is closed.

[37] The Pool Manager closed all the pools in the pool group. The connection pool is closed.

Generating an export file with the exportStatement method

You can generate an export file by calling the `exportStatements` method of the `ExtStatementPoolMonitorMBean` interface. For example, the following code exports the contents of the statement pool associated with the connection to a file named `stmt_export`.

```
ExtStatementPoolMonitor monitor =
   ((ExtConnection) con).getStatementPoolMonitor();
exportStatements(stmt_export.txt)
```

Statement pool export file example

The following example shows a sample export file. The footnotes provide explanations for the referenced text to help you understand the content of your own statement pool export files.

```
[DDTEK_STMT_POOL][38]
VERSION=1[39]

[STMT_ENTRY][40]
SQL_TEXT=[
INSERT INTO emp(id, name) VALUES(?,?)
]
STATEMENT_TYPE=Prepared Statement
RESULTSET_TYPE=Forward Only
RESULTSET_CONCURRENCY=Read Only
AUTOGENERATEDKEYSREQUESTED=false
REQUESTEDKEYCOLUMNS=

[STMT_ENTRY][41]
SQL_TEXT=[
INSERT INTO emp(id, name) VALUES(99,?)
]
STATEMENT_TYPE=Prepared Statement
RESULTSET_TYPE=Forward Only
RESULTSET_CONCURRENCY=Read Only
AUTOGENERATEDKEYSREQUESTED=false
REQUESTEDKEYCOLUMNS=id,name
```

Using Java Logging

The Salesforce driver provides a flexible and comprehensive logging mechanism that allows logging to be incorporated seamlessly with the logging of your own application or allows logging to be enabled and configured independently from the application. The logging mechanism can be instrumental in investigating and diagnosing issues. It also provides valuable insight into the type and number of operations requested by the application from the driver and requested by the driver from the remote data source. This information can help you tune and optimize your application.

[38] A string that identifies the file as a statement pool export file.
[39] The version of the export file.
[40] The first statement pool entry. Each statement pool entry lists the SQL text, statement type, result set type, result set concurrency type, and generated keys information.
[41] The next statement pool entry.

Logging Components

The Salesforce driver uses the Java Logging API to configure the loggers (individual logging components) used by the driver. The Java Logging API is built into the JVM.

The Java Logging API allows applications or components to define one or more named loggers. Messages written to the loggers can be given different levels of importance. For example, errors that occur in the driver can be written to a logger at the `CONFIG` level, while progress or flow information may be written to a logger at the `FINE` or `FINER` level. Each logger used by the driver can be configured independently. The configuration for a logger includes what level of log messages are written, the location to which they are written, and the format of the log message.

The Java Logging API defines the following levels:

- `SEVERE`
- `WARNING`
- `INFO`
- `CONFIG`
- `FINE`
- `FINER`
- `FINEST`

Note: Log messages logged by the driver only use the `CONFIG`, `FINE`, `FINER`, and `FINEST` logging levels.

Setting the log threshold of a logger to a particular level causes the logger to write log messages of that level and higher to the log. For example, if the threshold is set to `FINE`, the logger writes messages of levels `FINE`, `CONFIG`, `INFO`, `WARNING`, and `SEVERE` to its log. Messages of level `FINER` or `FINEST` are not written to the log.

The driver exposes loggers for the following functional areas:

- JDBC API
- SQL Engine
- Web service adapter

JDBC API Logger

Name

`com.ddtek.jdbc.cloud.level`

Purpose

Logs the JDBC calls made by the application to the driver and the responses from the driver back to the application. DataDirect Spy is used to log the JDBC calls.

Message Levels

FINER - Calls to the JDBC methods are logged at the FINER level. The value of all input parameters passed to these methods and the return values passed from them are also logged, except that input parameter or result data contained in InputStream, Reader, Blob, or Clob objects are not written at this level.

FINEST - In addition to the same information logged by the FINER level, input parameter values and return values contained in InputStream, Reader, Blob and Clob objects are written at this level.

OFF - Calls to the JDBC methods are not logged.

SQL Engine Logger

Name

com.ddtek.cloud.sql.level

Purpose

Logs the operations that the SQL engine performs while executing a query. Operations include preparing a statement to be executed, executing the statement, and fetching the data, if needed. These are internal operations that do not necessarily directly correlate with Web service calls made to the remote data source.

Message Levels

CONFIG - Any errors or warnings detected by the SQL engine are written at this level.

FINE - In addition to the same information logged by the CONFIG level, SQL engine operations are logged at this level. In particular, the SQL statement that is being executed is written at this level.

FINER - In addition to the same information logged by the CONFIG and FINE levels, data sent or received in the process of performing an operation is written at this level.

Web Service Adapter Logger

Name

com.ddtek.cloud.adapter.level

Purpose

Logs the Web service calls the driver makes to the remote data source and the responses it receives from the remote data source.

Message Levels

CONFIG - Any errors or warnings detected by the Web service adapter are written at this level.

FINE - In addition to the same information logged by the CONFIG level, information about Web service calls made by the Web service adapter and responses received by the Web service adapter are written at this level. In particular, the Web service calls made to execute the query and the calls to fetch or send the data are logged. The log entries for the calls to execute the query include the Salesforce-specific query being executed. The actual data sent or fetched is not written at this level.

FINER - In addition to the same information logged by the CONFIG and FINE levels, this level provides additional information.

FINEST - In addition to the same information logged by the CONFIG, FINE, and FINER levels, data associated with the Web service calls made by the Web service adapter is written.

Configuring Logging

You can configure logging using a standard Java properties file in either of the following ways:

- Using the properties file that is shipped with your JVM. See "Using the JVM" for details.
- Using the driver. See "Using the Driver" for details.

Using the JVM for Logging

If you want to configure logging using the properties file that is shipped with your JVM, use a text editor to modify the properties file in your JVM. Typically, this file is named `logging.properties` and is located in the `JRE/lib` subdirectory of your JVM. The JRE looks for this file when it is loading.

You can also specify which properties file to use by setting the java.util.logging.config.file system property. At a command prompt, enter:

`java -Djava.util.logging.config.file=`*`properties_file`*

where:

`properties_file`

is the name of the properties file you want to load.

Using the Driver for Logging

If you want to configure logging using the driver, you can use either of the following approaches:

- Use a single properties file for all Salesforce connections.
- Use a different properties file for each schema map. For example, if you have two maps (for example, `C:\data\schemamaps\test1map.config` and `C:\data\schemamaps\test2map.config`), you can load one properties file for the `test1map.config` schema map and load another properties file for the `test2map.config` schema map.

Note: See "SchemaMap" for information on SchemaMap default values and how to specify valid values for SchemaMap.

By default, the driver looks for the file named `ddlogging.properties` in the current working directory to load for all Salesforce connections.

If a properties file is specified for the LogConfigFile connection property, the driver uses the following process to determine which file to load.

1. The driver looks for the file specified by the LogConfigFile property.
2. If the driver cannot find the file in Step 1 on page 178, it looks for a properties file named *`user_name`*`.logging.properties` in the directory containing the schema map for the connection, where *`user_name`* is your user ID used to connect to the Salesforce instance.
3. If the driver cannot find the file in Step 2 on page 178, it looks for a properties file named `ddlogging.properties` in the current working directory.
4. If the driver cannot find the file in Step 3 on page 178 , it abandons its attempt to load a properties file.

If any of these files exist, but the logging initialization fails for some reason while using that file, the driver writes a warning to the standard output (`System.out`), specifying the name of the properties file being used.

A sample properties file is installed in the `install_dir/testforjdbc` and `install_dir/Examples/SforceSamples` directories. The file is named `ddlogging.properties`. You can copy this file to the current working directory of your application or schema map configuration file directory, and modify it using a text editor for your needs.

See also

SchemaMap on page 153
LogConfigFile on page 145

6

Supported SQL Statements and Extensions

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

For details, see the following topics:

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

Alter Cache (EXT)

Purpose

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}
  [REFERENCING (remote_table_ref[,remote_table_ref]...)]
  [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. When altering a relational cache, *remote_table* must specify the primary table of the relational cache.

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.

`REFERENCING`

is an optional clause that specifies the name of the remote table(s) for which a relationship cache is to be created. See "Relational Caches" and "Referencing Clause" for a complete explanation.

`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 grammar description.

See also

Relational Caches on page 193
Referencing Clause on page 194
Refresh Interval Clause on page 194
Initial Check Clause on page 195
Persist Clause on page 195
Enabled Clause on page 196
Call Limit Clause on page 197
Filter Clause on page 197

Relational Caches

If the Referencing clause is specified, the Alter Cache statement drops the existing cache and any referenced caches and creates a new set of related caches, one for each of the tables specified in the statement. The cache attributes for the existing cache are the default cache attributes for the new relational cache. Any attributes specified in the Alter Cache statement override the default attributes. If the Referencing clause is not specified, the existing cache references, if any, are used.

If the cache being altered is a relational cache, the attributes specified in the Alter Cache statement apply to all of the caches that comprise the relational cache.

Alter Index

Purpose

Changes the name of an existing index.

Syntax

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

where:

index_name

specifies an existing index name.

new_name

specifies the new index name.

Notes

- Index names must not conflict with other user-defined or system-defined names.
- Indexes on remote tables cannot be created, altered or dropped. Indexes can only be defined on local tables.

Alter Sequence

Purpose

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").

See also

Next Value For Clause on page 199

Alter Session (EXT)

Purpose

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

Syntax

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

where:

attribute_name

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

value

specifies the value for that attribute.

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

Table 20: Alter Session Attributes

Attribute Name	Session Type	Description
Current_Schema	Local	Sets the current schema for the local session. The current schema is the schema used when an identifier in a SQL statement is unqualified. The string value must be the name of a schema visible in the local session. For example: `ALTER SESSION SET CURRENT_SCHEMA=sforce`

Attribute Name	Session Type	Description
Stmt_Call_Limit	Local	Sets the maximum number of Web service calls the driver can make in executing a statement. Setting the Stmt_Call_Limit attribute has the same effect as setting the StmtCallLimit connection property. 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 must be a positive integer or `0`. The value `0` means that no call limit exists. For example: `ALTER SESSION SET STMT_CALL_LIMIT=150`

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

Alter Table

The Alter Table statement adds or removes a column. The table being altered can be either a remote or local table. A remote table is a Salesforce object and is exposed in the SFORCE 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.

- For information on altering a remote table, see "Altering a Remote Table."
- For information on altering a local table, see "Altering a Local Table."

Altering a Remote Table

Syntax

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

where:

table_name

specifies an existing remote table.

add_clause

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

drop_clause

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

Notes

- You cannot drop a constraint from a remote table.

Add Clause: Columns

Purpose

Adds a column to an existing table. It is optional.

Syntax

```
ADD [COLUMN] column_name Datatype ...
[DEFAULT default_value] [[NOT]NULL] [EXT_ID] [PRIMARY KEY]
[START WITH starting_value]
```

default_value

is the default value to be assigned to the column. See "Column Definition for Remote Tables" for details.

starting_value

is the starting value for the Identity column. The default start value is `0`.

Notes

- 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.
- You cannot specify ANYTYPE, BINARY, COMBOBOX, or TIME data types in the column definition of Alter Table statements.
- 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 SFORCE, this example adds the `status` column with a default value of `ACTIVE` to the `test` table.

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

Example B

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

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

See also

Creating a Remote Table on page 200

Add Clause: Constraints

Purpose

Adds a constraint to an existing table. It is optional.

Syntax

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

Notes

- The only type of constraint you can add is a foreign key constraint.
- When adding a foreign key constraint, the table that contains the foreign key must be empty.

Example

Assuming the current schema is SFORCE, a foreign key constraint is added to the `deptId` column of the `test` table, referencing the `rowId` of the `dept` table. For the operation to succeed, the `dept` table must be empty.

```
ALTER TABLE test ADD FOREIGN KEY (deptId) REFERENCES dept(rowId)
```

Drop Clause: Columns

Purpose

Drops 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

- The column being dropped cannot have a constraint defined on it.
- Drop fails if a SQL view includes the column.

Example

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
```

Altering a Local Table

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.

See also

Add Clause: Columns on page 189
Add Clause: Constraints on page 190
Drop Clause: Columns on page 191
Drop Clause: Constraints on page 191
Rename Clause on page 191

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 "Creating a Local Table"). 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

- You cannot specify ANYTYPE, BINARY, COMBOBOX, or TIME data types in the column definition of Alter Table statements.
- 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

Creating a Local Table on page 204

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").

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)
```

See also

Constraint Definition for Local Tables on page 207

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
```

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}
  [REFERENCING (remote_table_ref[,remote_table_ref]...)]
  [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.

`REFERENCING`

is an optional clause that specifies the name of the remote table(s) for which a relationship cache is to be created. See "Relational Caches" and "Referencing Clause" for a complete explanation.

`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 grammar description.

See also

Relational Caches on page 193
Referencing Clause on page 194
Refresh Interval Clause on page 194
Initial Check Clause on page 195
Persist Clause on page 195
Enabled Clause on page 196
Call Limit Clause on page 197
Filter Clause on page 197

Relational Caches

If the Referencing clause is specified, the Create Cache statement creates a set of related caches, one for each of the tables specified in the statement. This set of caches is referred to as a related or relational cache. The set of caches in a relational cache is treated as a single entity. They are refreshed, altered, and dropped as a unit. Any attributes specified in the Create Cache statement apply to the cache created for the primary table and to the caches created for all of the referenced tables specified.

A local session can have both standalone and relational caches defined, but only one cache can be defined on a table. If a table is referenced in a relational cache definition, a standalone cache cannot be created on that table.

Referencing Clause

Purpose

The Referencing clause specifies the name of the remote table(s) for which a relationship cache is to be created; it is optional. The specified remote table must be related to either the primary table being cached or one of the other specified related tables. The remote table name cannot include a schema name. The referenced tables must exist in the same schema as the primary table.

Syntax

```
REFERENCING (remote_table_ref[,remote_table_ref]...)]
```

where:

remote_table_ref

represents *remote_table*[.*foreign_key_name*]

remote_table

specifies one or more tables related to the primary table that are to be cached in conjunction with the primary table.

foreign_key_name

specifies the name of the foreign key relationship between the remote table and the primary table (or, optionally, another related table). If a foreign key name is not specified, the driver attempts to find a relationship between the remote table and one of the other tables specified in the relational cache. The driver first looks for a relationship to the primary table. If a relationship to the primary table does not exist, the driver then looks for a relationship to other referenced tables.

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

- 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.

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 local 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 185

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

The following example filters by last activity date.

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

Example A

The Referencing clause allows multiple related tables to be cached as a single entity. The following example creates a cache on the remote table account. The cache is populated with all accounts that had activity in 2010. Additionally, caches are created for the following remote tables: `opportunity`, `contact`, and `opportunitylineitem`. These caches are populated with the opportunities and contacts that are associated with the accounts stored in the accounts cache and the opportunity line items associated with the opportunities stored in the opportunity cache.

```
CREATE CACHE ON account REFERENCING (opportunity, contact, opportunitylineitem)
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 224

Create Index

Purpose

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

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

Specifies 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

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

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

Create Table

Use the Create Table statement to create a new table. You can create either a remote or local table. A remote table is a Salesforce object and is exposed in the SFORCE schema. Creating a table in the SFORCE schema creates a remote 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. Creating a table in the PUBLIC schema creates a local table.

- For information on creating remote tables, see "Creating a Remote Table."
- For information on creating local tables, see "Creating a Local Table."

Creating a Remote Table

Purpose

Defines the syntax to define a column for remote tables.

Syntax

```
CREATE TABLE table_name (column_definition [, ...] [, constraint_definition...])
```

where:

table_name

specifies the name of the new remote table. The table name can be qualified by a schema name using the format *schema*.*table*. If the schema is not specified, the table is created in the current schema. See "Alter Session (EXT)" for information about changing the current schema.

column_definition

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

constraint_definition

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

Notes

- Creating tables in Salesforce is not a quick operation. It can take several minutes for Salesforce to create the table and its relationships.

See also

Alter Session (EXT) on page 185

Column Definition for Remote Tables

Purpose

Defines the syntax to define a column for remote tables.

Syntax

```
column_name Datatype [(precision[,scale])...]
[DEFAULT default_value][[NOT]NULL][EXT_ID][PRIMARY KEY]
[START WITH starting_value]
```

where:

column_name

is the name to be assigned to the column.

Datatype

is the data type of the column to be created. See Data Types in the *DataDirect Connect Series for JDBC User's Guide* for a list of supported Salesforce data types. You cannot specify ANYTYPE, BINARY, COMBOBOX, ENCRYPTEDTEXT, or TIME data types in the column definition of Create Table statements.

precision

is the total number of digits for NUMBER, CURRENCY, and PERCENT columns, and the length of HTML, LONGTEXTAREA, and TEXT columns.

scale

is the number of digits to the right of the decimal point for NUMBER, CURRENCY, and PERCENT columns.

default_value

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

- For character columns, a single-quoted string or NULL.
- 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_ TIMESTAMP, TODAY, or NOW.
- For boolean columns, the literals FALSE, TRUE, NULL.
- For numeric columns, any valid number or NULL.

starting_value

is the starting value for the Identity column. The default start value is 0.

[NOT]NULL

is used to specify whether NULL values are allowed or not allowed in a column. If NOT NULL is specified, all rows in the table must have a column value. If NULL is specified or if neither NULL or NOT NULL is specified, NULL values are allowed in the column.

EXT_ID

is used to specify that the column is an external ID column.

PRIMARY KEY

can only be specified when the data type of the column is ID. ID columns are always the primary key column for Salesforce.

START WITH

specifies the sequence of numbers generated for the Identity column. It can only be used when the data type of the column definition is AUTONUMBER.

Example A

Assuming the current schema is SFORCE, the remote table Test is created in the SFORCE schema. The id column has a starting value of 1000.

```
CREATE TABLE Test (id AUTONUMBER START WITH 1000, Name TEXT(30))
```

Example B

The table name is qualified with a schema name that is not the current schema, creating the Test table in the SFORCE schema. The table is created with the following columns: id, Name, and Status. The Status column contains a default value of ACTIVE.

```
CREATE TABLE SFORCE.Test (id NUMBER(9, 0), Name TEXT(30), Status TEXT(10) DEFAULT
'ACTIVE')
```

Example C

Assuming the current schema is SFORCE, the remote table dept is created with the name and deptId columns. The deptId column can be used as an external ID column.

```
CREATE TABLE dept (name TEXT(30), deptId NUMBER(9, 0) EXT_ID)
```

Constraint Definition for Remote Tables

Purpose

Defines a constraint for a remote table.

Syntax

[CONSTRAINT [*constraint_name*] {foreign_key_constraint}]

where:

constraint_name

is ignored. The driver uses the Salesforce relationship naming convention to generate the constraint name.

foreign_key_constraint

defines a link between related tables. See "Foreign Key Clause" for the 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)
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

See also

Foreign Key Clause on page 204

Foreign Key Clause

Purpose

Specifies a foreign key for a constraint.

Syntax

```
FOREIGN KEY (fcolumn_name)
   REFERENCES ref_table (pcolumn_name)
```

where:

fcolumn_name

specifies the foreign key column to which the constraint is applied. The data type of this column must be the same as the data type of the column it references.

ref_table

specifies the table to which the foreign key refers.

pcolumn_name

specifies the primary key column in the referenced table. For Salesforce, the primary key column is always the rowId column.

Example

Assuming the current schema is SFORCE, the remote table `emp` is created with the `name`, `empId`, and `deptId` columns. The table contains a foreign key constraint on the `deptId` column, referencing the `rowId` in the `dept` table created in Example C. For the operation to succeed, the data type of the `deptId` column must be the same as that of the `rowId` column.

```
CREATE TABLE emp (name TEXT(30), empId NUMBER(9, 0) EXT_ID, deptId TEXT(18),
FOREIGN KEY(deptId) REFERENCES dept(rowId))
```

Creating a Local Table

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 local session. When the session is closed, the data for the memory table is persisted to disk.

DISK

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

TEMPORARY | TEMP

creates 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.

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.

Notes

- If MEMORY, DISK, or TEMPORARY|TEMP is not specified, the new table is created in memory.

Column Definition for Local Tables

Purpose

Defines 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 "Data Types" for a list of supported Salesforce data types. You cannot specify ANYTYPE, BINARY, COMBOBOX, or TIME data types in the column definition of Create Table statements.

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 fractional second digits 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`

define an auto-increment column. You can only specify these clauses on INTEGER and 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))
```

See also

Data Types on page 17

Constraint Definition for Local Tables

Purpose

Defines a constraint 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 the 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 the 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 the 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)
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}]
```

where:

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

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:

- 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

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

- 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)
```

See also

Data Types on page 17

Create View

Purpose

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. The view is defined by a Select statement that is stored locally, though the data itself is not stored locally. 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 the following 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 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 this 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 219

Delete

Purpose

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

Syntax

```
DELETE FROM table_name [WHERE search_condition]
```

where:

table_name

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

search_condition

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

Notes

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

Example A

This example shows a Delete statement on the emp table.

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

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

Example B

This example shows using a subquery in a Delete clause.

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

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

Notes

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

See also

Where Clause on page 224

Drop Cache (EXT)

Purpose

Drops the cache defined on a remote table. To drop a relational cache, the specified table must be the primary table of the relational cache. If a relational cache is specified, the cache for the primary table and all referenced caches are dropped.

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

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

Drops a sequence for a local table.

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 local objects.

Drop Table

Purpose

Drops (removes) a remote or local table, its data, and its indexes. A remote table is a Salesforce object and is exposed in the SFORCE schema. Dropping a table in the SFORCE schema drops a remote 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. 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. If the specified table is a remote table, any tables that reference the specified table also are dropped.

Drop View

Purpose

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

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

Syntax

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

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

Insert

Purpose

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

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

Syntax

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

table_name

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

column_name

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

expression

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

select_statement

is a query that returns values for each *column_name* value specified in the column list. Using a Select statement instead of a list of value expressions lets you select a set of rows from one table and insert it into another table using a single Insert statement. The Select statement is evaluated before any values are inserted. This query cannot be made on the table into which values are inserted. See "Select" for information about Select statements.

Notes

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

See also

Literals on page 231
Select on page 219

Specifying an External ID Column

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

Syntax

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

where:

EXT_ID

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

schema_name

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

table_name

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

ext_id_column

is the external ID column.

Example A

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

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

Example B

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

```
INSERT INTO emp1 (first_name,
                  last_name,
                  emp_id,
                  dept,
                  salary)
SELECT first_name, last_name, emp_id, dept, salary FROM emp
WHERE dept = 'D050'
```

Example C

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

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

Refresh Cache (EXT)

Purpose

Forces the data in the cache for the specified remote table to be refreshed.

Syntax

`REFRESH CACHE ON {`*`remote_table`* `| ALL} [CLEAN]`

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.

`CLEAN`

is optional and discards the data in the cache for the specified table or view, or all cache data if `ALL` is specified, and repopulates the cache with the data in the remote table or view.

Notes

- Caches on views are not supported.

Refresh Map (EXT)

Purpose

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

Syntax

```
REFRESH MAP
```

Notes

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

Select

Purpose

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

Syntax

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

where:

select_clause

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

from_clause

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

where_clause

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

groupby_clause

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

having_clause

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

UNION

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

`INTERSECT`

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

`EXCEPT | MINUS`

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

`orderby_clause`

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

`limit_clause`

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

Select Clause

Purpose

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

Syntax

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

where:

`LIMIT` *`offset number`*

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

`TOP` *`number`*

is equivalent to `LIMIT 0`*`number`*.

`column_expression`

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

column_alias

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

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

DISTINCT

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

```
SELECT DISTINCT dep FROM emp
```

Notes

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

See also

SQL Expressions on page 230

Aggregate Functions

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

The following table lists supported aggregate functions.

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

Table 21: Aggregate Functions

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

MAX	The maximum value in any column expression. For example, `MAX(salary)` returns the maximum salary column value.
MIN	The minimum value in any column expression. For example, `MIN(salary)` returns the minimum salary column value.
SUM	The total of the values in a numeric column expression. For example, `SUM(salary)` returns the sum of all salary column values.

Example

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

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

From Clause

Purpose

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

Syntax

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

where:

`table_name`

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

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

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

`table_alias`

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

Example

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

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

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

```
FROM emp E
```

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

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

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

Join in a From Clause

Purpose

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

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

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

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

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

Syntax

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

Example

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

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

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

Subquery in a From Clause

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

Example

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

See also

Subqueries on page 238

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 238
SQL Expressions on page 230

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 238
SQL Expressions on page 230

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

Subqueries on page 238
SQL Expressions on page 230

Union Operator

Purpose

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

Syntax

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

Notes

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

Example A

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

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

Example B

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

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

Intersect Operator

Purpose

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

Syntax

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

where:

`DISTINCT`

eliminates duplicate rows from the results.

Notes

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

Example A

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

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

Example B

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

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

Except and Minus Operators

Purpose

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

Syntax

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

where:

`DISTINCT`

eliminates duplicate rows from the results.

Notes

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

Example A

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

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

Example B

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

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

Order By Clause

Purpose

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

Syntax

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

where:

`sort_expression`

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

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

Example

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

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

or

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

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

See also

SQL Expressions on page 230

Limit Clause

Purpose

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

Syntax

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

where:

number_of_rows

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

OFFSET

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

Notes

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

Example

The following example returns a maximum of 20 rows.

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

Update

Purpose

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

Syntax

```
UPDATE table_name SET column_name = expression
[, column_name = expression] [WHERE conditions]
```

table_name

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

column_name

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

expression

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

Example A

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

```
UPDATE emp SET salary=32000, exempt=1
```

```
WHERE emp_id = 'E10001'
```

Example B

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

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

Notes

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

See also

Subqueries on page 238
Where Clause on page 224

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

Quoted identifiers must be enclosed in double quotation marks (""). A quoted identifier can contain any Unicode character including the space character. The Salesforce 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 22: Literal Syntax Examples

SQL Type	Literal Syntax	Example
BIGINT	*n* where *n* is any valid integer value in the range of the INTEGER data type	`12 or -34 or 0`
BOOLEAN	Min Value: 0 Max Value: 1	`0` `1`
DATE	DATE'*date*'	`'2010-05-21'`
DATETIME	TIMESTAMP'*ts*'	`'2010-05-21 18:33:05.025'`

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

DOUBLE	*n.f*E*x* where: *n* is the integral part *f* is the fractional part *x* is the exponent	`1.2E0` or `2.5E40` or `-3.45E2` or `5.67E-4`

SQL Type	Literal Syntax	Example
INTEGER	*n* where *n* is a valid integer value in the range of the INTEGER data type	`12` or `-34` or `0`
LONGVARBINARY	'*hex_value*'	`'000482ff'`
LONGVARCHAR	'*value*'	`'This is a string literal'`
TIME	TIME'*time*'	`'2010-05-21 18:33:05.025'`
VARCHAR	'*value*'	`'This is a string literal'`

Character String Literals

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

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

Example

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

Numeric Literals

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

Example

```
+1894.1204
```

Binary Literals

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

Example

```
'00af123d'
```

Date/Time Literals

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

- The format for a Date literal is DATE'*date*'.

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

Integer Literals

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

Notes

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

Example

```
1994 or -2
```

Operators

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

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

Unary Operator

A unary operator operates on only one operand.

operator operand

Binary Operator

A binary operator operates on two operands.

operand1 operator operand2

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

Arithmetic Operators

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

Table 23: Arithmetic Operators

Operator	Purpose	Example
+ -	Denotes a positive or negative expression. These are unary operators.	`SELECT * FROM emp WHERE comm = -1`

Operator	Purpose	Example
* /	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 24: Concatenation Operator

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

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

Comparison Operators

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

The following table lists the supported comparison operators.

Table 25: 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`

Operator	Purpose	Example
LIKE	% and _ wildcards can be used to search for a pattern in a column. The percent sign denotes zero, one, or multiple characters, while the underscore denotes a single character. The right-hand side of a LIKE expression must evaluate to a string or binary.	`SELECT * FROM emp WHERE ENAME LIKE 'J%'`

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

[NOT] IN	"Equal to any member of" test.	`SELECT * FROM emp WHERE job IN ('CLERK','ANALYST') SELECT * FROM emp WHERE sal IN (SELECT sal FROM emp WHERE deptno = 30)`
[NOT] BETWEEN x AND y	"Greater than or equal to x" and "less than or equal to y."	`SELECT * FROM emp WHERE sal BETWEEN 2000 AND 3000`
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 26: 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 27: Operator Precedence

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

Example A

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

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

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

Example B

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

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

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

Functions

The driver supports a number of functions that you can use in expressions, as listed and described in Supported scalar functions on page 27.

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 28: 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)
```