Progress DataDirect for JDBC for Apache Cassandra
User's Guide

6.0.0 Release

Copyright

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

Updated: 2026/05/12

Table of Contents

Welcome to the Progress DataDirect for JDBC for Apache Cassandra Driver....9

What's New in this Release?....10

Requirements....12

Driver and Data Source Classes....12

Version String Information....12

Connection Properties....13

Complex Type Normalization....13

Collection Types....14

Tuple and User-Defined Types....17

Nested Complex Types....19

Data Types....21

getTypeInfo()....22

SQL escape sequences....31

DataDirect tools....32

Troubleshooting....32

Additional information....32

Contacting Technical Support....32

Getting started....35

Driver and Data Source Classes....35

Connecting Using the DriverManager....36

Setting the Classpath....36

Passing the Connection URL....36

Testing the Connection....37

Connecting using data sources....40

How Data Sources Are Implemented....41

Creating data sources....41

Calling a data source in an application....41

Using the Driver....43

Connecting from an application....44

Driver and Data Source Classes....44

Connecting Using the DriverManager....44

Connecting using data sources....49

Interactive SQL for JDBC (JDBCISQL)....50

Using Connection Properties....51

Required Properties....52
Mapping Properties....52
Authentication Properties....53
Data Encryption Properties....53
Statement Pooling Properties....55
Additional Properties....55
Proxy Server Properties....56
Performance Considerations....57
Authentication....58
Configuring User ID/Password Authentication....58
Configuring the Driver for Kerberos Authentication....59
Proxy server support....66
Data Encryption....67
Configuring SSL Encryption....67
Configuring SSL Server Authentication....68
Configuring SSL Client Authentication....68
FIPS (Federal Information Processing Standard)....69
Identifiers....70
IP Addresses....70
Parameter Metadata Support....71
Insert and Update Statements....71
Select Statements....71
Isolation Levels....72
Unicode support....72
Error Handling....72
Large Object (LOB) Support....73
Rowset Support....73
Executing CQL....74

Connection Property Descriptions....75

AuthenticationMethod....79
ClusterNodes....79
ConfigOptions....80
SchemaFormat (configuration option)....81
CreateMap....82
EncryptionMethod....83
FetchSize....84
HostNameInCertificate....85
ImportStatementPool....86
InitializationString....87
InsensitiveResultSetBufferSize....87
KeyPassword....88
KeyspaceName....89
KeyStore....90

KeyStorePassword....................91
LogConfigFile....................91
LoginConfigName....................92
LoginTimeout....................93
MaxPooledStatements....................94
NativeFetchSize....................95
Password....................96
PortNumber....................97
ProxyHost....................97
ProxyPassword....................98
ProxyPort....................98
ProxyUser....................99
ReadConsistency....................99
ReadOnly....................101
RegisterStatementPoolMonitorMBean....................101
ResultMemorySize....................102
SchemaMap....................103
SecureConnectBundle....................105
ServerName....................107
ServicePrincipalName....................108
SpyAttributes....................109
TransactionMode....................111
TrustStore....................112
TrustStorePassword....................113
User....................113
ValidateServerCertificate....................114
WriteConsistency....................115

Supported SQL Functionality....................117

Data Definition Language (DDL)....................117
Native and Refresh Escape Sequences....................118
Delete....................118
Insert....................119
Refresh Map (EXT)....................120
Select....................121
Select Clause....................122
Update....................131
SQL Expressions....................133
Column Names....................134
Literals....................134
Operators....................136
Functions....................140
Conditions....................140
Subqueries....................141

IN Predicate....141
EXISTS Predicate....142
UNIQUE Predicate....142
Correlated Subqueries....142

1

Welcome to the Progress DataDirect for JDBC for Apache Cassandra Driver

The Progress® DataDirect® *for* JDBC™ for Apache Cassandra™ driver supports SQL read-write access to DataStax Enterprise 4.6 or higher and Apache Cassandra 2.0 or higher. To support SQL access to Cassandra, the driver creates a relational map of native Cassandra data and translates SQL statements to CQL. Cassandra complex data types Map, List, Set, Tuple, and user-defined types are supported alongside primitive CQL types. The driver optimizes performance when executing joins by leveraging data relationships among Cassandra objects to minimize the amount of data that needs to be fetched over the network. Relationships among objects can be reported with the metadata methods getTables(), getColumns(), getTypeInfo(), getPrimaryKeys(), getExportedKeys(), and getBestRowIdentifier().

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-apache-cassandra.

For details, see the following topics:

- What's New in this Release?
- Requirements
- Driver and Data Source Classes
- Version String Information
- Connection Properties
- Complex Type Normalization

- Data Types
- SQL escape sequences
- DataDirect tools
- Troubleshooting
- Additional information
- Contacting Technical Support

What's New in this Release?

Support and certification

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

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

Changes since 6.0.0 Release

- **Driver Enhancements**
  - The driver has been enhanced to support connection failover and client load balancing when connecting to a cluster. You can enable this new functionality by specifying a comma-separated list of member nodes using the new ClusterNodes connection property. See ClusterNodes on page 79 for details.
  - 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 69 for details.
  - The driver has been enhanced to connect using secure connect bundles, which contain security certificates and credentials for your database. You can configure your driver to use secure connect bundles by specifying the location of the `.zip` file using the new SecureConnectBundle property. See SecureConnectBundle on page 105 for details.

    Note that secure connect bundles are only supported by certain databases, such as DataStax Enterprise running in IBM Cloud. Refer to your database documentation for support information.
  - The driver has been enhanced to allow you to configure how collections are mapped in the relational view of your data. By configuring the new SchemaFormat config option, you can determine whether the driver normalizes collections and collections labeled FROZEN. See SchemaFormat (configuration option) on page 81 for details.
  - The driver has been enhanced to include a timestamp in the Spy and JDBC packet logs by default. See SpyAttributes on page 109 for details.
  - The driver has been enhanced to support the Duration data type, which maps to the Varchar JDBC type. See Data Types on page 21 for details.
  - Interactive SQL for JDBC (JDBCISQL) is now installed with the product. JDBCISQL is a command-line interface that supports connecting your driver to a data source, executing SQL statements and retrieving

results in a terminal. This tool provides a method to quickly test your drivers in an environment that does not support GUIs. See Interactive SQL for JDBC (JDBCISQL) on page 50 for details.

- The driver has been enhanced to support all the data consistency levels for read and write operations that are supported by Apache Cassandra data stores. Data consistency levels are configured using the ReadConsistency and WriteConsistency connection properties. For additional information, see ReadConsistency on page 99 and WriteConsistency on page 115.
- The driver has been enhanced to support SSL, incorporating the addition of eight new connection properties. See Data Encryption on page 67 and Data Encryption Properties on page 53 for details.
- The driver has been enhanced to support Kerberos authentication. See Authentication on page 58 and Authentication Properties on page 53 for details.
- The driver has been enhanced to improve the handling of large result sets and reduce the likelihood of out-of-memory errors through the introduction of the FetchSize, NativeFetchSize, and ResultMemorySize connection properties. See FetchSize on page 84, NativeFetchSize on page 95, and WriteConsistency on page 115 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 109 for details.
  - Java SE 7 has reached the end of its product life cycle and will no longer receive generally available security updates. As a result, the drivers will no longer support JVMs that are version Java SE 7 or earlier. Support for distributed versions of Java SE 7 and earlier will also end, including IBM SDK (Java Edition).
  - The SchemaDefinition connection property has been replaced with the SchemaMap connection property. See SchemaMap on page 103 for details.

Highlights of the 6.0.0 Release

- Supports SQL read-write access to DataStax Enterprise 4.6 or higher and Apache Cassandra 2.0 or higher. See Supported SQL Functionality on page 117 for details.
- The driver supports JDBC core functions. For details, refer to "JDBC support" in the *Progress DataDirect for JDBC Drivers Reference*.
- Supports user id/password authentication. See Authentication on page 58 for details.
- Supports Cassandra data types, including the complex types Tuple, user-defined types, Map, List and Set. See Data Types on page 21 for details.
- Generates a relational view of Cassandra data. Tuple and user-defined types are flattened into a relational parent table, while collection types are mapped as relational child tables. See Complex Type Normalization on page 13 for details.
- Supports Native and Refresh escape sequences to embed CQL commands in SQL-92 statements. See Native and Refresh Escape Sequences on page 118 for details.
- Supports Cassandra's tunable consistency functionality with ReadConsistency on page 99 and WriteConsistency on page 115 connection properties.
- Supports the handling of large result sets with FetchSize on page 84, NativeFetchSize on page 95, and ResultMemorySize on page 102 connection properties.
- Includes the TransactionMode on page 111 connection property which allows you to configure the driver to report that it supports transactions, even though Cassandra does not support transactions. This provides a workaround for applications that do not operate with a driver that reports transactions are not supported.

- Supports Binary Large Objects (BLOBs). See Large Object (LOB) Support on page 73 for details.
- Supports connection pooling. For details, refer to "Connection Pool Manager" in the *Progress DataDirect for JDBC Drivers Reference*.
- Supports statement pooling. For details, refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference*.
- Includes the LoginTimeout on page 93 connection property which allows you to specify how long the driver waits for a connection to be established before timing out the connection request.

Requirements

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

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

Driver and Data Source Classes

The driver class is:

`com.ddtek.jdbc.cassandra.CassandraDriver`

Two data source classes are provided with the driver. Which data source class you use depends on the JDBC functionality your application requires. The following table shows the recommended data source class to use with different JDBC specifications.

Table 1: Choosing a Data Source Class

If your application requires...	Data Source Class
JDBC 4.0 functionality and higher	`com.ddtek.jdbcx.cassandra.CassandraDataSource40`
JDBC 3.x functionality and earlier specifications	`com.ddtek.jdbcx.cassandra.CassandraDataSource`

See Connecting using data sources on page 40 for information about Progress DataDirect data sources.

Version String Information

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

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

where:

M is the major version number.

m is the minor version number.

s is the service pack number.

bbbbbb is the driver build number.

XXXX is the cloud adapter build number.

YYYYYY is the framework build number.

ZZZZZZ is the utl build number.

For example:

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

Connection Properties

The driver includes over 20 connection properties. You can use these connection properties to customize the driver for your environment. Connection properties can be used to accomplish different tasks, such as implementing driver functionality and optimizing performance. You can specify connection properties in a connection URL or within a JDBC data source object.

See Using Connection Properties on page 51 and Connection Property Descriptions on page 75 for more information.

Complex Type Normalization

To support SQL access to Apache Cassandra, the driver maps the Cassandra data model to a relational schema. This process involves the normalization of complex types. You may need to be familiar with the normalization of complex types to formulate SQL queries correctly. The driver handles the normalization of complex types in the following manner:

- If collection types (Map, List, and Set) are discovered, the driver normalizes the Cassandra table into a set of parent-child tables. Primitive types are mapped to a parent table, while each collection type is mapped to a child table that has a foreign key relationship to the parent table.
- Non-nested Tuple and user-defined types (also referred to as *Usertype*) are flattened into a parent table alongside primitive types.
- Any nested complex types (Tuple, user-defined types, Map, List, and Set) are exposed as JSON-style strings in the parent table.

The normalization of complex types is described in greater detail in the following topics.

Note: This section describes the default mapping behavior of the driver. You can configure this behavior using the SchemaFormat configuration option. See SchemaFormat (configuration option) on page 81 for details.

Collection Types

Cassandra collection types include the Map, List, and Set types. If collection types are discovered, the driver normalizes the native data into a set of parent-child tables. Primitive types are normalized in a parent table, while each collection type is normalized in a child table that has a foreign key relationship to the parent table. Take for example the following Cassandra table:

```
CREATE TABLE employee (
   empid int PRIMARY KEY,
   phone map<varchar, varint>,
   client list<varchar>,
   review set<date>);
```

The following `employee` table is a tabular representation of the native Cassandra table with data included. In this example, four distinct relational tables are created. A parent table is created based on the `empid` column, and a child table is created for each of the three collection types (Map, List, and Set).

Table 2: employee (native)

empid (primary key)	phone	client	review
int	**map<varchar, varint>**	**list<varchar>**	**set<date>**
103	home: 2855551122 mobile: 2855552347 office: 2855555566 spouse: 2855556782	Li Kumar Jones	2013-12-07 2015-01-22 2016-01-10
105	home: 2855555678 mobile: 2855553335 office: 2855555462	Yanev Bishop Bogdanov	2015-01-22 2016-01-12

The Parent Table

The parent table is comprised of the primitive integer type column `empid` and takes its name from the native table. A SQL statement would identify the column as `employee.empid`.

Table 3: employee (relational parent)

empid (primary key)
int
103
105

A SQL insert on the employee parent table would take the form:

```
INSERT INTO employee (empid) VALUES (107)
```

The Map Child Table

The Map collection is normalized into a three column child table called employee_phone. The name of the table is formulated by concatenating the name of the native table and the name of the Map column. A foreign key relationship to the parent table is maintained via the employee_empid column, and the Map's key value pairs are resolved into separate keycol and valuecol columns. In a SQL statement, these columns would be identified as the employee_phone.employee_empid, employee_phone.keycol, and employee_phone.valuecol, respectively.

Table 4: employee_phone (relational child of the map column)

employee_empid (foreign key)	keycol	valuecol
int	**varchar**	**varint**
103	home	2855551122
103	mobile	2855552347
103	office	2855555566
103	spouse	2855556782
105	home	2855555678
105	mobile	2855553335
105	office	2855555462

A SQL insert on the employee_phone child table would take the form[1] :

```
INSERT INTO employee_phone (employee_empid, keycol, valuecol) VALUES (107, 'mobile',
2855552391)
```

The List Child Table

The List collection is normalized into a three column child table called employee_client. The name of the table is formulated by concatenating the name of the native table and the name of the List column. A foreign key relationship to the parent table is maintained via the employee_empid column; the order of the elements in the List is maintained via the current_list_index column; and the elements themselves are contained in the client column. SQL statements would identify these columns as employee_client.employee_empid, employee_client.current_list_index, and employee_client.client, respectively.

[1] The driver supports an insert on a child table prior to an insert on a parent table, circumventing referential integrity constraints associated with traditional RDBMS. To maintain integrity between parent and child tables, it is recommended that an insert be performed on the parent table for each foreign key value added to the child. If such an insert is not first performed, the driver automatically inserts a row into the parent tables that contains only the primary key values and NULL values for all non-primary key columns.

Table 5: employee_client (relational child of the list column)

employee_empid (foreign key)	current_list_index	client
int	int	varchar
103	0	Li
103	1	Kumar
103	2	Jones
105	0	Yanev
105	1	Bishop
105	2	Bogdanov

A SQL insert on the `employee_client` child table would take the form[1] :

```
INSERT INTO employee_client (employee_empid, client) VALUES (107, 'Nelson')
```

The Set Child Table

The Set collection is normalized into a two column child table called `employee_review`. The name of the table is formulated by concatenating the name of the native table and the name of the Set column. A foreign key relationship to the parent table is maintained via the `empid` column, while the elements of the Set are given in natural order in the `review` column. In this child table, SQL statements would identify these columns as `employee_review.employee_empid` and `employee_review.review`

Table 6: employee_review (relational child of the set column)

employee_empid (foreign key)	review
int	date
103	2013-12-07
103	2015-01-22
103	2016-01-10
105	2015-01-22
105	2016-01-12

A SQL insert on the `employee_client` child table would take the form[1] :

```
INSERT INTO employee_review (employee_empid, review) VALUES (107, '2015-01-20')
```

Update Support

Update is supported for primitive types, non-nested Tuple types, and non-nested user-defined types. Update is also supported for value columns (`valuecol`) in non-nested Map types. The driver does not support updates on List types, Set types, or key columns (`keycol`) in Map types because the values in each are part of the primary key of their respective child tables and primary key columns cannot be updated. If an Update is attempted when not allowed, the driver issues the following error message:

[DataDirect][Cassandra JDBC Driver][Cassandra]syntax error or access rule violation: UPDATE not permitted for column: *column_name*

Tuple and User-Defined Types

The driver supports Tuple and user-defined complex types which were introduced with Apache Cassandra 2.1. As long as there are no complex types nested in either the Tuple or user-defined types, the driver normalizes Tuple and user-defined types by flattening them into a relational version of the native Cassandra table. Take for example the following Cassandra table:

```
CREATE TABLE agents1 (
   agentid int PRIMARY KEY,
   email varchar,
   contact tuple<varchar,varchar,varchar>);
```

The following `agents1` table is a tabular representation of the native Cassandra table with data included.

Table 7: agents1 (native)

agentid (primary key)	email	contact
int	**varchar**	**tuple<varchar, varchar, varchar>**
272	barronr@email.com	tv newspaper blog
564	rothm@email.com	radio tv magazine

For the relational version of `agents1`, all fields are retained as separate columns, and columns with primitive types (`agentid` and `email`) correspond directly to columns in the native table. In turn, tuple fields are flattened into columns using a `<tuplename>_<ordinal>` naming pattern. The driver normalizes `agents1` in the following manner.

Table 8: agents1 (relational)

agentid (primary key)	email	contact_1	contact_2	contact_3
int	**varchar**	**varchar**	**varchar**	**varchar**
272	barronr@email.com	tv	newspaper	blog
564	rothm@email.com	radio	tv	magazine

A SQL command would take the following form:

```
INSERT INTO agents1 (agentid,email,contact_1,contact_2,contact_3) VALUES
(839,'gonzalesn@email.com','radio','tv','magazine')
```

The driver also flattens user-defined types when normalizing native Cassandra tables. In the following example, the native Cassandra `agents2` table incorporates the user-defined *address* type.

```
CREATE TYPE address (
   street varchar,
   city varchar,
   state varchar,
   zip int);
CREATE TABLE agents2 (
   agentid int PRIMARY KEY,
   email varchar,
   location frozen<address>);
```

The following `agents2` table is a tabular representation of the native Cassandra table with data included.

Table 9: agents2 (native)

agentid (primary key)	email	location
int	**varchar**	**address<street: varchar, city: varchar, state: varchar, zip: int>**
095	barronr@email.com	street: 1551 Main Street city: Pittsburgh state: PA zip: 15237
237	rothm@email.com	street: 422 First Street city: Richmond state: VA zip: 23235

As with the previous example, all fields are retained as separate columns in the relational version of the table, and columns with primitive types (agentid and email) correspond directly to columns in the native table. Here a <columnname>_<fieldname> naming pattern is used to flatten the fields of the user-defined address type into columns. The driver normalizes agents2 in the following manner.

Table 10: agents2 (native)

agentid (primary key)	email	location_street	location_city	location_state	location_zip
int	**varchar**	**varchar**	**varchar**	**varchar**	**int**
095	barronr@email.com	1551 Main Street	Pittsburgh	PA	15237
237	rothm@email.com	422 First Street	Richmond	VA	23235

A SQL command would take the following form:

```
INSERT INTO agents2
(agentid,email,location_street,location_city,location_state,location_zip) VALUES
(839,'gonzalesn@email.com','9 Fifth Street', 'Morrisville', 'NC', 27566)
```

Nested Complex Types

The nesting of complex types within Tuple and user-defined types is permitted in CQL. The driver does not normalize such nested types, but rather the data is passed as a JSON-style string. For example, consider the table contacts which contains the columns id and contact. While id is a primitive int column, contact is a user-defined *info* column which contains name, email, and location fields. The location field itself is a nested user-defined *address* column which contains street, city, state, and zip fields. In CQL, the structure of this table would take the following form:

```
CREATE TYPE address (
   street varchar,
   city varchar,
   state varchar,
   zip int);
CREATE TYPE info (
   name varchar,
   email varchar,
   location frozen<address>);
CREATE TABLE contacts (
   id int PRIMARY KEY,
   contact frozen<info>);
```

The following tabular representation of the contacts table shows how the driver returns data when complex types are nested in other complex types. Because the complex user-defined type address is embedded in the complex user-defined type info, the entire contact column is returned by the driver as a JSON string.

Note: You can retrieve this string data by calling the DatabaseMetaData.getColumns() method.

Table 11: contacts (relational)

id (primary key)	contact
int	**info<name: varchar, email: varchar, location: address<street: varchar, city: varchar, state: varchar, zip: int>>**
`034`	`{name: 'Jude', email: 'jnichols@email.com', location: {street: '101 Main Street', city: 'Albany', state:'NY', zip: 12210}}`
`056`	`{name: 'Karen', email: 'kbrown@email.com', location: {street: '150 First Street', city: 'Portland', state: 'OR', zip: 97214}}`

When executing SQL commands involving nested complex types, the data must be passed as a JSON string. Furthermore, the syntax you use to connote the JSON string depends on whether you are passing the string directly in a SQL command or setting the JSON string as a parameter on a prepared statement.

Note: Hints for parsing JSON-style strings are provided in the `Remark` column of the getColumns() result.

Connoting the JSON-Style String in a SQL Statement

When passing the string directly in a SQL command, you must use the correct SQL syntax and escapes to maintain the structure of the data. To begin, the entire JSON string must be passed in single quotation marks ('). Furthermore, if the JSON string contains nested strings, two single quotation marks are used to indicate string values. The first quotation mark is an escape connoting the second embedded quotation mark. The following command inserts a new row into the `contacts` table.

Note: In accordance with Java syntax, the Insert statement is placed in double quotation marks. However, in the JSON string, two single quotation marks are used to indicate string values. The first quotation mark is an escape connoting the second embedded quotation mark.

```
Stmt.executeUpdate(
   "INSERT INTO contacts (id, contact) VALUES (075, '{name: ''Albert'', " +
     "email: ''aocampo@email.com'', location: {street: ''12 North Street'', " +
     "city: ''Durham'', state:''NC'', zip: 27704}}')");
```

After the insert has been executed, the Select command `SELECT contact FROM contacts WHERE id = 75` returns:

```
{name: 'Albert',
email: 'aocampo@email.com',
location: {street: '12 North Street',
           city: 'Durham',
           state:'NC',
           zip: 27704
           }
}
```

Connoting the JSON-Style String as a Parameter Value on a Prepared Statement

When setting the JSON string as a parameter value, you must follow Java syntax by placing the JSON string in double quotation marks. Escapes are not used to connote embedded single quotation marks. For example:

```
pstmt.setString(
    2,
    "{name: 'Albert', email: 'aocampo@email.com', location: " +
       "{street: '12 North Street', city: 'Durham', state:'NC', " +
       "zip: 27704}}")
```

Data Types

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

The complex types List, Map, Set, Tuple, and user-defined types have no direct JDBC mapping. However, the driver normalizes these types to provide SQL access to Cassandra. For details on how complex types are mapped, see "Complex Type Normalization."

See "getTypeInfo()" for getTypeInfo() results of data types supported by the driver.

Table 12: Data Type Mapping

Apache Cassandra	JDBC
ASCII	VARCHAR
BIGINT	BIGINT
BLOB	LONGVARBINARY
BOOLEAN	BOOLEAN
COUNTER[2]	BIGINT
DATE[3]	DATE
DECIMAL	DECIMAL
DOUBLE	DOUBLE
DURATION[4]	VARCHAR
FLOAT	REAL
INET	VARCHAR
INT	INTEGER

[2] Update is supported for Counter columns when all the other columns in the row comprise that row's primary key. See Update on page 131 for details.
[3] Supported for Apache Cassandra 2.2 and higher.
[4] Currently, the Duration type is supported only in simple columns, and not in Collection types.

Apache Cassandra	JDBC
LIST[5]	LONGVARCHAR
MAP[5]	LONGVARCHAR
SET[5]	LONGVARCHAR
SMALLINT[3]	SMALLINT
TIME[3]	TIME
TIMESTAMP	TIMESTAMP
TIMEUUID	CHAR
TINYINT[3]	TINYINT
TUPLE[5]	LONGVARCHAR
USERTYPE[6]	LONGVARCHAR
UUID	CHAR
VARCHAR	VARCHAR
VARINT	DECIMAL

See also

Complex Type Normalization on page 13

getTypeInfo()

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

[5] See Complex Type Normalization on page 13 for details on how complex types are mapped.

[6] Also referred to as *user-defined types*. These are data types created with the CQL CREATE TYPE statement. See Complex Type Normalization on page 13 for details on how Usertype is mapped.

Table 13: getTypeInfo()

TYPE_NAME = ascii AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = ascii MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2000000000 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = bigint AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -5 (BIGINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = bigint MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 19 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = blob AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = X' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = blob MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = boolean 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 = boolean MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = counter[7] AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -5 (BIGINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = counter MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 19 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = date[8] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = false LITERAL_PREFIX = date' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = date MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[7] Update is supported for Counter columns when all the other columns in the row comprise that row's primary key. See Update on page 131 for details.
[8] Supported for Apache Cassandra 2.2 and higher.

TYPE_NAME = decimal AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = decimal MAXIMUM_SCALE = 100	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 38 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = double AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = double MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 15 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = duration AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = duration MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 50 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = float AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 7 (REAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = float MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 7 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = inet AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = inet MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 39 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = int AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = int MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = list[9] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = list MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = map[9] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = map MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = set[9] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = set MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[9] See Complex Type Normalization on page 13 for details on how complex types are mapped.

TYPE_NAME = smallint[8] AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 5 (SMALLINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = smallint MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 5 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = time[8] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 92 (TIME) FIXED_PREC_SCALE = false LITERAL_PREFIX = time' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = time MAXIMUM_SCALE = 9	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 18 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = timestamp AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = timestamp' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = timestamp MAXIMUM_SCALE = 3	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 23 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = timeuuid AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = timeuuid MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 36 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = tinyint[8] AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -6 (TINYINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = tinyint MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 3 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = tuple[9] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = tuple MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = usertype[10] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = usertype MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = uuid AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = uuid MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 36 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[10] Also referred to as *user-defined types*. These are data types created with the CQL CREATE TYPE statement. See Complex Type Normalization on page 13 for details on how Usertype is mapped.

TYPE_NAME = varchar AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = varchar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = varint AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = varint MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 100 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

SQL escape sequences

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

DataDirect tools

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

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

Troubleshooting

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

Additional information

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

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

Contacting Technical Support

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

https://www.progress.com/support

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

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

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

2

Getting started

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

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

For details, see the following topics:

- Driver and Data Source Classes
- Connecting Using the DriverManager
- Connecting using data sources

Driver and Data Source Classes

The driver class is:

```
com.ddtek.jdbc.cassandra.CassandraDriver
```

Two data source classes are provided with the driver. Which data source class you use depends on the JDBC functionality your application requires. The following table shows the recommended data source class to use with different JDBC specifications.

Table 14: Choosing a Data Source Class

If your application requires...	Data Source Class
JDBC 4.0 functionality and higher	`com.ddtek.jdbcx.cassandra.CassandraDataSource40`
JDBC 3.x functionality and earlier specifications	`com.ddtek.jdbcx.cassandra.CassandraDataSource`

See Connecting using data sources on page 40 for information about Progress DataDirect data sources.

Connecting Using the DriverManager

One way to connect to a Cassandra keyspace 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:cassandra://MyServer:9042;KeyspaceName=MyKS");
```

Note: A Cassandra keyspace is equivalent to a relational database.

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 `cassandra.jar` file as shown, where *`install_dir`* is the path to your product installation directory.

```
install_dir/lib/60/cassandra.jar
```

Windows Example

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

UNIX Example

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

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:

```
jdbc:datadirect:cassandra://server:port;
KeyspaceName=keyspace;[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:

server

specifies the name or the IP address of the server to which you want to connect.

port

specifies the port of the server that is listening for connections to the Cassandra keyspace. The default is `9042`.

keyspace

specifies the default name of the Cassandra keyspace to which the driver connects. This value is used as the default qualifier for unqualified table names in SQL queries. Note that a Cassandra keyspace is equivalent to a relational database.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon. For more information on connection properties, see Using Connection Properties on page 51.

This example shows how to establish a connection to a Cassandra data store:

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:cassandra://MyServer:9042;KeyspaceName=MyKS");
```

Testing the Connection

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

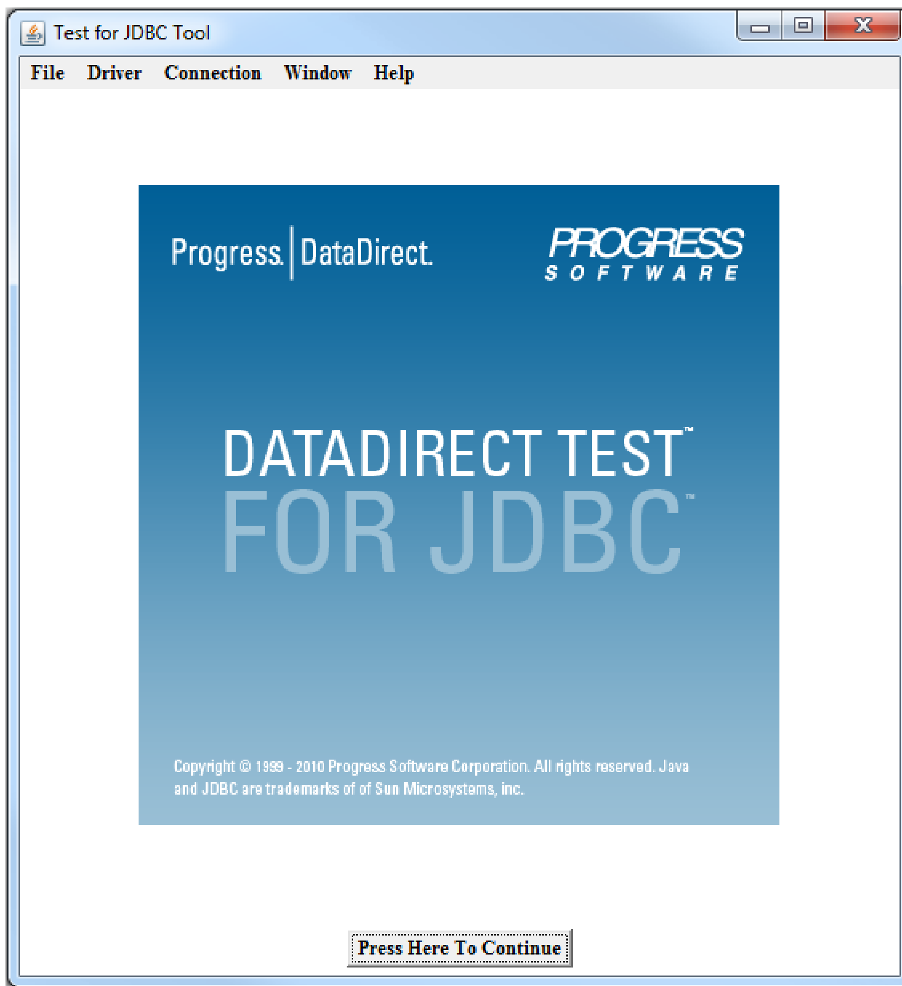
Take the following steps to establish a connection.

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

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

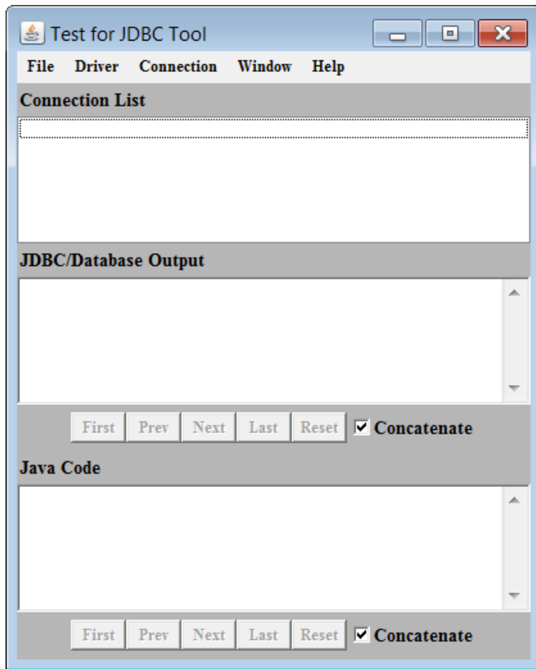
2. From the `testforjdbc` folder, run the platform-specific tool:
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

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



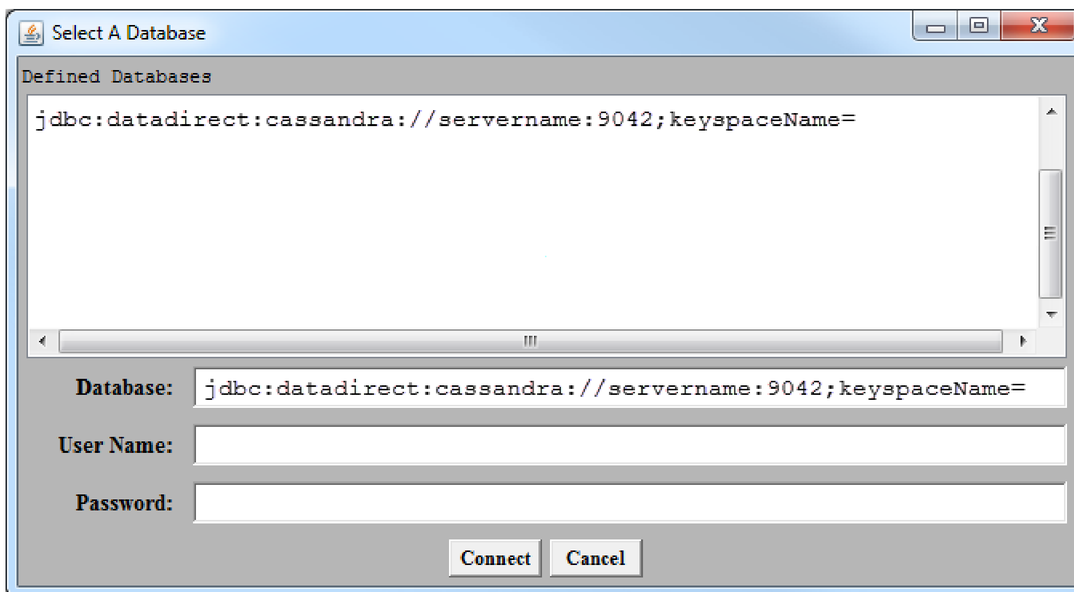
3. Click **Press Here to Continue**.

   The main dialog appears:



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

   The **Select A Database** dialog appears:



5. Select the appropriate database template from the **Defined Databases** field.
6. In the **Database** field, specify the ServerName, PortNumber, and KeyspaceName for your Apache Cassandra data store.

   For example:

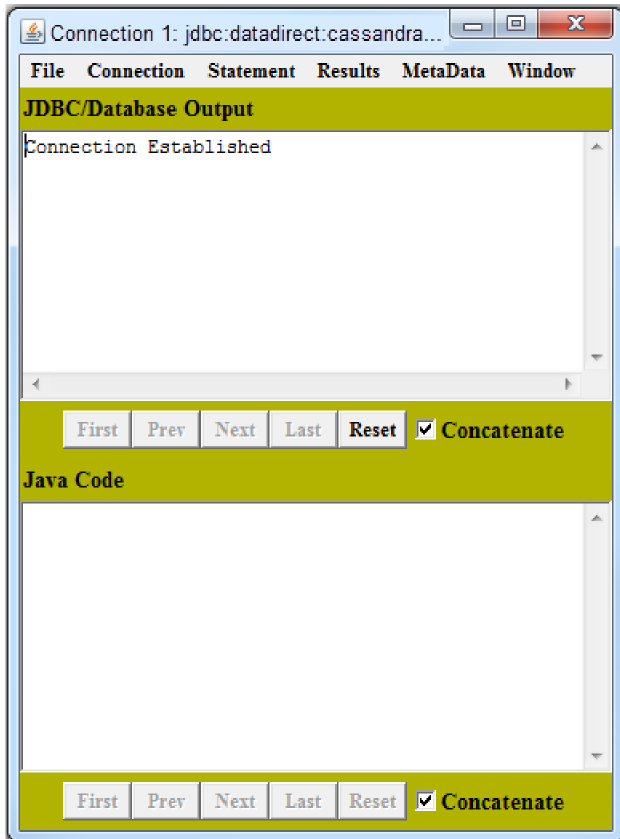
```
jdbc:datadirect:cassandra://MyServer:9042;keyspaceName=MyKS
```

7. If you are using user ID/password authentication, enter your user ID and password in the corresponding fields.

Attention: User ID/password authentication is not supported for the preview version of the driver.

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.

See also

Driver and Data Source Classes on page 12

Creating data sources

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

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

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

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

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.

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:

- Connecting from an application
- Using Connection Properties
- Performance Considerations
- Authentication
- Proxy server support
- Data Encryption
- Identifiers
- IP Addresses
- Parameter Metadata Support
- Isolation Levels
- Unicode support
- Error Handling
- Large Object (LOB) Support
- Rowset Support

- Executing CQL

Connecting from an application

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

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

Driver and Data Source Classes

The driver class is:

`com.ddtek.jdbc.cassandra.CassandraDriver`

Two data source classes are provided with the driver. Which data source class you use depends on the JDBC functionality your application requires. The following table shows the recommended data source class to use with different JDBC specifications.

Table 15: Choosing a Data Source Class

If your application requires...	Data Source Class
JDBC 4.0 functionality and higher	`com.ddtek.jdbcx.cassandra.CassandraDataSource40`
JDBC 3.x functionality and earlier specifications	`com.ddtek.jdbcx.cassandra.CassandraDataSource`

See Connecting using data sources on page 40 for information about Progress DataDirect data sources.

Connecting Using the DriverManager

One way to connect to a Cassandra keyspace 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:cassandra://MyServer:9042;KeyspaceName=MyKS");
```

Note: A Cassandra keyspace is equivalent to a relational database.

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 `cassandra.jar` file as shown, where *`install_dir`* is the path to your product installation directory.

```
install_dir/lib/60/cassandra.jar
```

Windows Example

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

UNIX Example

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

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:

```
jdbc:datadirect:cassandra://server:port;
KeyspaceName=keyspace;[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:

`server`

specifies the name or the IP address of the server to which you want to connect.

`port`

specifies the port of the server that is listening for connections to the Cassandra keyspace. The default is `9042`.

`keyspace`

specifies the default name of the Cassandra keyspace to which the driver connects. This value is used as the default qualifier for unqualified table names in SQL queries. Note that a Cassandra keyspace is equivalent to a relational database.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon. For more information on connection properties, see Using Connection Properties on page 51.

This example shows how to establish a connection to a Cassandra data store:

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:cassandra://MyServer:9042;KeyspaceName=MyKS");
```

Testing the Connection

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

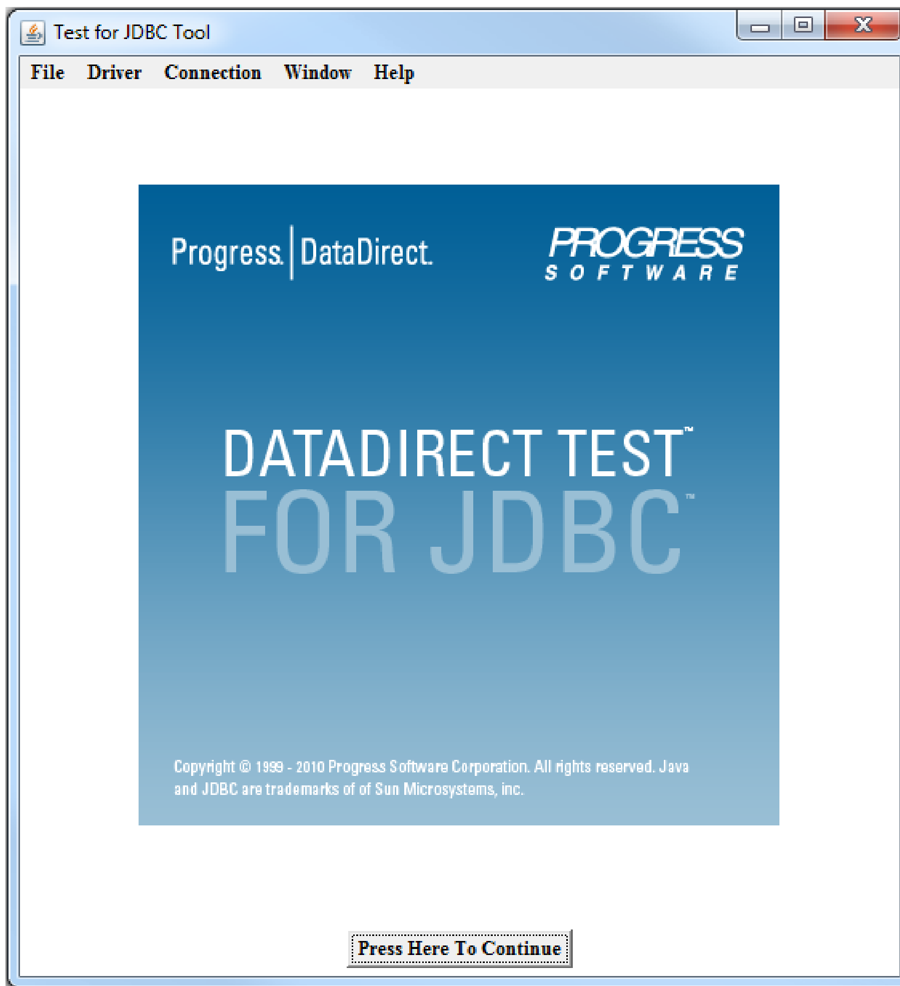
Take the following steps to establish a connection.

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

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

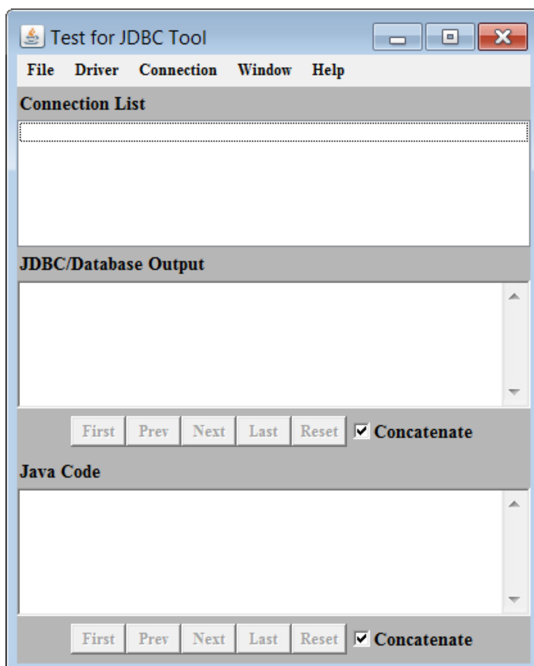
2. From the `testforjdbc` folder, run the platform-specific tool:
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

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



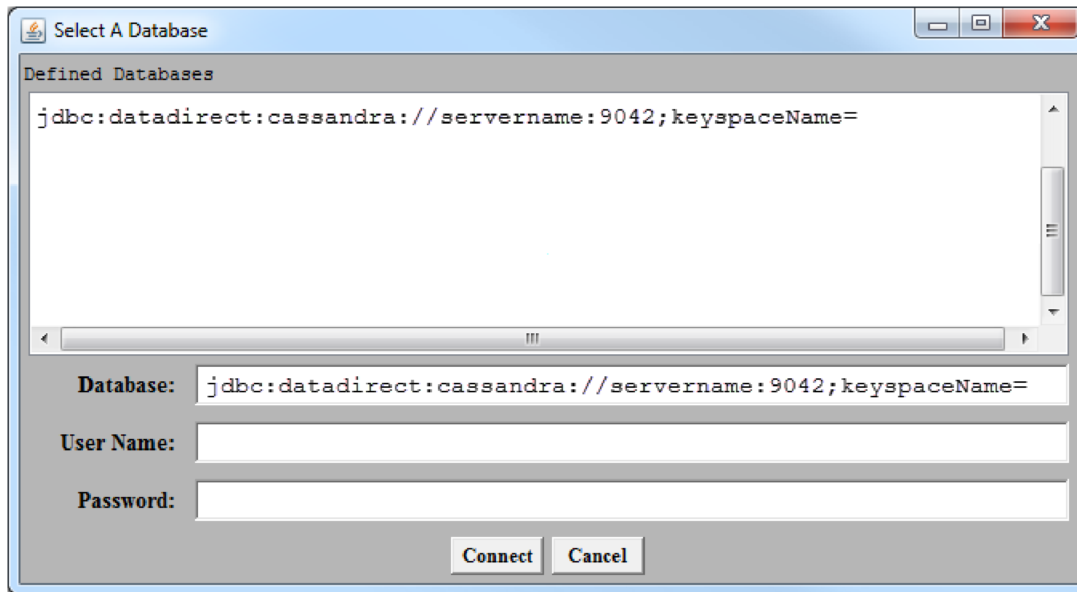
3. Click **Press Here to Continue**.

   The main dialog appears:



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

   The **Select A Database** dialog appears:



5. Select the appropriate database template from the **Defined Databases** field.
6. In the **Database** field, specify the ServerName, PortNumber, and KeyspaceName for your Apache Cassandra data store.

   For example:

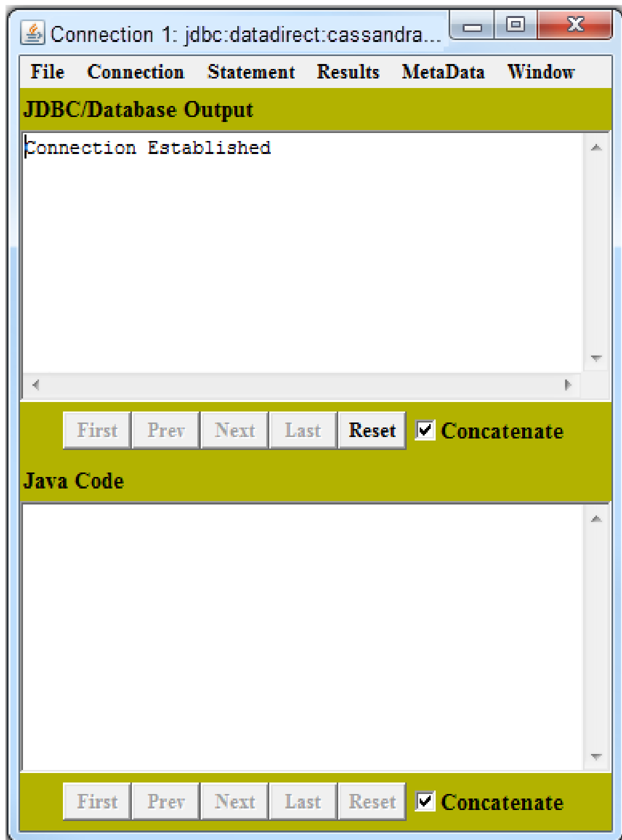
```
jdbc:datadirect:cassandra://MyServer:9042;keyspaceName=MyKS
```

7. If you are using user ID/password authentication, enter your user ID and password in the corresponding fields.

Attention: User ID/password authentication is not supported for the preview version of the driver.

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.

See also

Driver and Data Source Classes on page 12

Creating data sources

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

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

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

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

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.

Interactive SQL for JDBC (JDBCISQL)

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

To execute commands with JDBCISQL:

1. Start the ISQL tool. Do one of the following:
    - On Windows, double-click the jdbcisql.bat file in the *install_dir*\jdbcisql folder. Or, from a command prompt, navigate to the *install_dir*\jdbcisql directory and run the jdbcisql.bat file.

- On Linux and UNIX, change to the *install_dir*\isql directory and run jdbcisql.sh.

The Interactive SQL prompt appears.

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

```
com.ddtek.jdbc.cassandra.CassandraDriver
```

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

```
connect jdbc:datadirect:cassandra://MyServer:9042;AuthenticationMethod=none;
KeyspaceName=MyKS
```

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

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

```
SELECT * FROM FEATURES;
```

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

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

The results of the command are displayed in the terminal.

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

```
DISCONNECT;
```

6. Press any key to end the session.

Using Connection Properties

You can use connection properties to customize the driver for your environment. This section organizes connection properties according to functionality. You can use connection properties with either the JDBC DriverManager or a JDBC DataSource. For a DriverManager connection, a property is expressed as a key value pair and takes the form *property*=*value*. For a DataSource connection, a property is expressed as a JDBC method and takes the form set*property*(*value*). Connection property names are case-insensitive. For example, Password is the same as password.

Note: In a JDBC DataSource, string values must be enclosed in double quotation marks, for example, setKeyspaceName("MyKS").

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

See also

Connecting Using the DriverManager on page 36
Connecting using data sources on page 40

Connection Property Descriptions on page 75

Required Properties

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

Table 16: Required Properties

Property	Characteristic
KeyspaceName on page 89	Specifies the default name of the Cassandra keyspace to which the driver connects. This value is used as the default qualifier for unqualified table names in SQL queries. Note that a Cassandra keyspace is equivalent to a relational database.
PortNumber on page 97	Specifies the port of the server that is listening for connections to the Cassandra keyspace. The default is `9042`.
ServerName on page 107	Specifies the name or the IP address of the server to which you want to connect.
ClusterNodes on page 79	A comma-separated list of member nodes in your cluster to which the driver attempts to connect. Specifying a value for this option enables connection failover and load balancing.

See also

Connection Property Descriptions on page 75

Mapping Properties

The following table summarizes connection properties involved in mapping a native data model to a relational data model.

Table 17: Mapping Properties

Property	Characteristic
ConfigOptions on page 80	Determines how the mapping of the native data model to the relational data model is configured, customized, and updated.
CreateMap on page 82	Determines whether the driver creates the internal files required for a relational view of the native data when establishing a connection.
SchemaMap on page 103	Specifies the name and location of the configuration file used to create the relational map of native data. The driver looks for this file when connecting to the server. If the file does not exist, the driver creates one.

See also

Complex Type Normalization on page 13
Connection Property Descriptions on page 75

Authentication Properties

The following table summarizes connection properties related to authentication.

Table 18: Authentication Properties

Property	Characteristic
AuthenticationMethod on page 79	Determines which authentication method the driver uses when establishing a connection. The default is `userIdPassword`.
KeyspaceName on page 89	Specifies the default name of the Cassandra keyspace to which the driver connects. This value is used as the default qualifier for unqualified table names in SQL queries.
LoginConfigName on page 92	Specifies the name of the entry in the JAAS login configuration file that contains the authentication technology used by the driver to establish a Kerberos connection. The LoginModule-specific items found in the entry are passed on to the LoginModule. The default is `JDBC_DRIVER_01`.
Password on page 96	Specifies the password used to connect to your database for user ID/password authentication.
ServicePrincipalName on page 108	Specifies the three-part service principal name registered with the key distribution center (KDC) in a Kerberos configuration. When no value is specified, the driver builds a service principle name based on environment variables.
User on page 113	Specifies the user ID for user ID/password authentication or the user principal name for Kerberos authentication.

See also

Connection Property Descriptions on page 75

Data Encryption Properties

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

Table 19: Data Encryption Properties

Property	Characteristic
EncryptionMethod on page 83	Determines whether data is encrypted and decrypted when transmitted over the network between the driver and database server. To enable SSL, set EncryptionMethod to `SSL`. The default is `noEncryption`.

Property	Characteristic
HostNameInCertificate on page 85	Specifies a host name for certificate validation when SSL encryption is enabled (`EncryptionMethod=SSL`) and validation is enabled (`ValidateServerCertificate=true`). This property is optional and provides additional security against man-in-the-middle (MITM) attacks by ensuring that the server the driver is connecting to is the server that was requested.
KeyPassword on page 88	Specifies the password that is used to access the individual keys in the keystore file when SSL is enabled (`EncryptionMethod=SSL`) and SSL client authentication is enabled on the database server. This property is useful when individual keys in the keystore file have a different password than the keystore file.
KeyStore on page 90	Specifies the directory of the keystore file to be used when SSL is enabled (`EncryptionMethod=SSL`) and SSL client authentication is enabled on the database server. The keystore file contains the certificates that the client sends to the server in response to the server's certificate request.
KeyStorePassword on page 91	Specifies the password that is used to access the keystore file when SSL is enabled (`EncryptionMethod=SSL`) and SSL client authentication is enabled on the database server. The keystore file contains the certificates that the client sends to the server in response to the server's certificate request.
SecureConnectBundle on page 105	Specifies the name and location of the secure connect bundle that contains security certificates and credentials for your database.
TrustStore on page 112	Specifies the directory of the truststore file to be used when SSL is enabled (`EncryptionMethod=SSL`) and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts.
TrustStorePassword on page 113	Specifies the password that is used to access the truststore file when SSL is enabled (`EncryptionMethod=SSL`) and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts.
ValidateServerCertificate on page 114	Determines whether the driver validates the certificate that is sent by the database server when SSL encryption is enabled (`EncryptionMethod=SSL`). When using SSL server authentication, any certificate that is sent by the server must be issued by a trusted Certificate Authority (CA). The default is `true`.

See also

Data Encryption on page 67
Connection Property Descriptions on page 75
Authentication Properties on page 53

Statement Pooling Properties

The following table summarizes statement pooling connection properties.

Table 20: Statement Pooling Properties

Property	Characteristic
ImportStatementPool on page 86	Specifies the path and file name of the file to be used to load the contents of the statement pool. When this property is specified, statements are imported into the statement pool from the specified file.
MaxPooledStatements on page 94	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.
RegisterStatementPoolMonitorMBean on page 101	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.

See also

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

See also

Connection Property Descriptions on page 75

Additional Properties

The following table summarizes additional connection properties.

Table 21: Additional Properties

Property	Characteristic
FetchSize on page 84	Specifies the 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.

Property	Characteristic
InitializationString on page 87	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 87	Determines the amount of memory used by the driver to cache insensitive result set data.
LogConfigFile on page 91	Specifies the filename of the configuration file used to initialize driver logging.
LoginTimeout on page 93	The amount of time, in seconds, that the driver waits for a connection to be established before timing out the connection request.
NativeFetchSize on page 95	Specifies the number of rows of data the driver attempts to fetch from the native data source on each request submitted to the server.
ReadConsistency on page 99	Specifies how many replicas must respond to a read request before returning data to the client application.
ReadOnly on page 101	Specifies whether the connection supports read-only access to the data source.
ResultMemorySize on page 102	Specifies the maximum size, in megabytes, of an intermediate result set that the driver holds in memory. When this threshold is reached, the driver writes a portion of the result set to disk in temporary files.
SpyAttributes on page 109	Enables DataDirect Spy to log detailed information about calls issued by the driver on behalf of the application. DataDirect Spy is not enabled by default.
TransactionMode on page 111	Specifies how the driver handles manual transactions.
WriteConsistency on page 115	Determines the number of replicas on which the write must succeed before returning an acknowledgment to the client application.

See also

Connection Property Descriptions on page 75

Proxy Server Properties

The following table summarizes proxy server connection properties.

Table 22: Proxy Server Properties

Property	Characteristic
ProxyHost on page 97	Specifies a proxy server.

Property	Characteristic
ProxyPassword on page 98	Specifies the password needed to connect to a proxy server.
ProxyPort on page 98	Specifies the port number where the proxy server is listening for HTTP or HTTPS requests.
ProxyUser on page 99	Specifies the user name needed to connect to a proxy server.

See also

Connection Property Descriptions on page 75

Performance Considerations

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

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

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

For example, if an application attempts to fetch 100,000 rows from the native data source and NativeFetchSize is set to `500`, the driver must make 200 round trips across the network to get the 100,000 rows. If, however, NativeFetchSize is set to `10000`, the driver only needs to make 10 round trips to retrieve 100,000 rows. Network round trips are expensive, so generally, minimizing these round trips increases throughput.

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 NativeFetchSize to `200`. With these settings, the driver fetches all of the rows from the native data source that you would typically view in a single session and only processes the rows needed to display the first page.

Note: FetchSize 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.

InsensitiveResultSetBufferSize: To improve performance, insensitive result set data can be cached instead of written to disk. If the size of the result set data is greater than the size allocated for the cache, the driver writes the result set to disk. The maximum cache size setting is 2 GB.

JVM Heap Size: JVM heap size can be used to address memory and performance concerns. By increasing the max Java heap size, you increase the amount of data the driver may accumulate in memory. This can reduce the likelihood of out-of-memory errors and improve performance by ensuring that result sets fit easily within the JVM's free heap space. In addition, when a limit is imposed by setting ResultMemorySize to `-1`, increasing the max Java heap size can improve performance by reducing the need to write to disk, or removing it altogether.

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

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

ResultMemorySize: ResultMemorySize can affect performance in two main ways. First, if the size of the result set is larger than the value specified for ResultMemorySize, the driver writes a portion of the result set to disk. Since writing to disk is an expensive operation, performance losses will be incurred. Second, when you remove any limit on the size of an intermediate result set by setting ResultMemorySize to `0`, you can realize performance gains for result sets that easily fit within the JVM's free heap space. However, the same setting can diminish performance for result sets that barely fit within the JVM's free heap space.

See also

EncryptionMethod on page 83
FetchSize on page 84
InsensitiveResultSetBufferSize on page 87
MaxPooledStatements on page 94
ResultMemorySize on page 102

Authentication

The driver supports *user ID/password* and *Kerberos* authentication with the AuthenticationMethod connection property.

When `AuthenticationMethod=userIdPassword` (default), the driver uses `SCRAM-SHA-1` user ID/password authentication. The User property provides the user ID, and the Password property provides the password.

When `AuthenticationMethod=kerberos`, the driver uses Kerberos authentication.

When `AuthenticationMethod=none`, the driver does not attempt to authenticate with the server.

See also

AuthenticationMethod on page 79

Configuring User ID/Password Authentication

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

1. Set the AuthenticationMethod property to `userIdPassword`.
2. If necessary, set the KeyspaceName connection property.

   If authentication has not been enabled, client applications will have access to all keyspaces on the server. If authentication has been enabled, a client application will only have access to the keyspace specified by the KeyspaceName property assuming it has the required permissions.

   Even when authentication has not been enabled, KeyspaceName is strongly recommended because its value functions as the default qualifier for unqualified tables in SQL queries.

3. Set the User property to provide the user ID.
4. Set the Password property to provide the password.

See also

AuthenticationMethod on page 79
KeyspaceName on page 89
User on page 113
Password on page 96

Configuring the Driver for Kerberos Authentication

To configure the driver for Kerberos authentication, take the following steps.

1. Verify that your environment meets the requirements outlined in "Kerberos Authentication Requirements."
2. Use one of the following methods to integrate the JAAS configuration file into your Kerberos environment. (See "The JAAS Login Configuration File" for details about this file.)

   Option 1. Specify a login configuration file directly in your application with the `java.security.auth.login.config` system property. For example:

   ```
   System.setProperty("java.security.auth.login.config","install_dir/lib/JDBCDriverLogin.conf");
   ```

   Note: The *install_dir*/lib/JDBCDriverLogin.conf file is the JAAS login configuration file installed with the driver. You can use this file or another file as your JAAS login configuration file.

   Option 2. Set up a default configuration. Modify the Java security properties file to indicate the URL of the login configuration file with the `login.config.url.n` property where *n* is an integer connoting separate, consecutive login configuration files. When more than one login configuration file is specified, then the files are read and concatenated into a single configuration.

   a) Open the Java security properties file. The security properties file is the `java.security` file in the `/jre/lib/security` directory of your Java installation.

   b) Find the line `# Default login configuration file` in the security properties file.

   c) Below the `# Default login configuration file` line, add the URL of the login configuration file as the value for a `login.config.url.n` property. For example:

   ```
   # Default login configuration file
   login.config.url.1=file:${user.home}/.java.login.config
   login.config.url.2=file:install_dir/lib/JDBCDriverLogin.conf
   ```

3. Modify your JAAS login configuration file to include an entry with authentication technology that the driver can use to establish a Kerberos connection. (See "The JAAS Login Configuration File" for details about this file.)

   JAAS login configuration file entries begin with an entry name followed by one or more LoginModule items. Each LoginModule item contains information that is passed to the LoginModule. A login configuration file entry takes the following form.

   ```
   entry_name {
     login_module flag_value module_options
   };
   ```

where:

entry_name

is the name of the login configuration file entry. The driver's LoginConfigName connection property can be used to specify the name of this entry. `JDBC_DRIVER_01` is the default entry name for the `JDBCDriverLogin.conf` file installed with the driver.

login_module

is the fully qualified class name of the authentication technology used with the driver.

flag_value

specifies whether the success of the module is `required`, `requisite`, `sufficient`, or `optional`.

module_options

specifies available options for the LoginModule. These options vary depending on the LoginModule being used.

The following examples show that the LoginModule used for a Kerberos implementation depends on your JRE.

Oracle JRE

```
JDBC_DRIVER_01 {
  com.sun.security.auth.module.Krb5LoginModule required useTicketCache=true;
};
```

IBM JRE

```
JDBC_DRIVER_01 {
  com.ibm.security.auth.module.Krb5LoginModule required useDefaultCcache=true;
};
```

4. Set the Kerberos realm name and the KDC name for that realm using either of the following methods.

Note: If using Windows Active Directory, the Kerberos realm name is the Windows domain name and the KDC name is the Windows domain controller name.

Option 1. Modify the `krb5.conf` file to include the default realm name and the KDC name for that realm. (See "The krb5.conf File" for details about using and locating the `krb5.conf` file.)

For example, if the realm name is *XYZ.COM* and the KDC name is *kdc1*, your `krb5.conf` file would include the following entries.

```
[libdefaults]
default_realm = XYZ.COM

[realms]
XYZ.COM = {
kdc = kdc1
}
```

Option 2. Specify the Java system properties, `java.security.krb5.realm` and `java.security.krb5.kdc`, in your application. For example, if the realm name is *XYZ.COM* and the KDC name is *kdc1*, your application would include the following settings.

```
System.setProperty("java.security.krb5.realm","XYZ.COM");
System.setProperty("java.security.krb5.kdc","kdc1")
```

Note: Even if you do not use the `krb5.conf` file to specify the realm and KDC names, you may need to modify your `krb5.conf` file to suit your environment. Refer to your database vendor documentation for information.

If you do not specify a valid Kerberos realm and a valid KDC name, the following exception is thrown.

```
Message:[DataDirect][Cassandra JDBC Driver]Could not establish a connection using
integrated security: No valid credentials provided
```

5. Set the driver's AuthenticationMethod connection property to `kerberos`. (See "AuthenticationMethod" for details.)
6. If any of the following statements is valid, specify the service principal name with the ServicePrincipalName connection property. (See "ServicePrincipalName" for details on the composition of the service principal name.)

   Note: The ServicePrincipalName takes the following form.

   `Service_Name/Fully_Qualified_Domain_Name@REALM_NAME`

   - You are using a service name other than the default service name `cassandra`.
   - The fully qualified domain name (FQDN) in your connection string is different from the FQDN registered with the KDC.
   - You are using a Kerberos realm other than the default realm specified in the `krb5.conf` file.
7. If necessary, set the User connection property. (See "User" for details.)

   In most circumstances, there is no need to set the User connection property. By default, the driver uses the user principal name in the Kerberos Ticket Granting Ticket (TGT) as the value for the User property.
8. If necessary, set the KeyspaceName connection property. (See "KeyspaceName" for details.)

   If authentication has not been enabled, client applications will have access to all keyspaces on the server. If authentication has been enabled, a client application will only have access to the keyspace specified by the KeyspaceName property assuming it has the required permissions.

   Even when authentication has not been enabled, KeyspaceName is strongly recommended because its value functions as the default qualifier for unqualified tables in SQL queries.
9. If you want the driver to use user credentials other than the server user's operating system credentials, include code in your application to obtain and pass a `javax.security.auth.Subject` used for authentication. (See "Specifying User Credentials for Kerberos Authentication (Delegation of Credentials)" for details.)
10. Establish a procedure for obtaining a Kerberos Ticket Granting Ticket (TGT) for your environment. (See "Obtaining a Kerberos Ticket Granting Ticket" for details.)

    Scenario 1. For Windows Active Directory configurations, Active Directory automatically obtains a TGT.

Scenario 2. For non-Active Directory configurations, you can enable the application to obtain a TGT in either of the following ways.

a) Automate the method of obtaining the TGT as with a keytab. (See your Kerberos documentation for details.)

b) Require the application user to obtain the TGT with a `kinit` command when logging on.

A TGT can be obtained with a `kinit` command to the Kerberos server. For example, the following command requests a TGT from the server with a lifetime of 10 hours, which is renewable for 5 days.

```
kinit -l 10h -r 5d user@REALM
```

Note: The `klist` command can be used on Windows or UNIX/Linux systems to verify that a TGT has been obtained.

See also

Kerberos Authentication Requirements on page 62
The JAAS Login Configuration File on page 63
LoginConfigName on page 92
The krb5.conf File on page 64
AuthenticationMethod on page 79
ServicePrincipalName on page 108
User on page 113
KeyspaceName on page 89
Specifying User Credentials for Kerberos Authentication (Delegation of Credentials) on page 65
Obtaining a Kerberos Ticket Granting Ticket on page 66

Kerberos Authentication Requirements

Verify that your environment meets the requirements listed in the following table before you configure the driver for Kerberos authentication.

Note: For Windows Active Directory, the domain controller must administer both the database server and the client.

Table 23: Kerberos Configuration Requirements

Component	Requirements
Database server	The database server must be running Apache Cassandra 2.1 or higher.

Component	Requirements

Kerberos server	The Kerberos server is the machine where the user IDs for authentication are administered. The Kerberos server is also the location of the Kerberos key distribution center (KDC). Network authentication must be provided by one of the following methods.
	• Windows Active Directory on one of the following operating systems:
	  • Windows Server 2003 or higher
	  • Windows 2000 Server Service Pack 3 or higher
	• MIT Kerberos 1.5 or higher

Client	Java SE 8 or higher must be installed.

See also

Configuring the Driver for Kerberos Authentication on page 59

The JAAS Login Configuration File

The Java Authentication and Authorization Service (JAAS) login configuration file contains one or more entries that specify authentication technologies to be used by applications. To establish Kerberos connections with the driver, the login configuration file must be referenced either by setting the `java.security.auth.login.config` system property or by setting up a default configuration using the Java security properties file. In addition, the login configuration file must include an entry for the driver. (See "Configuring the Driver for Kerberos Authentication" for details.)

You can create your own JAAS login configuration file, or you can use the `JDBCDriverLogin.conf` file installed with the driver. This file is installed in the `/lib` directory of the product installation directory. In either case, the login configuration file must include an entry that specifies the authentication technology to be used by the driver to establish a Kerberos connection.

JAAS login configuration file entries begin with an entry name followed by one or more LoginModule items. Each LoginModule item contains information that is passed to the LoginModule. A login configuration file entry takes the following form.

```
entry_name {
  login_module flag_value module_options
};
```

where:

entry_name

is the name of the login configuration file entry. The driver's LoginConfigName connection property can be used to specify the name of this entry. `JDBC_DRIVER_01` is the default entry name for the `JDBCDriverLogin.conf` file installed with the driver.

login_module

is the fully qualified class name of the authentication technology used with the driver.

flag_value

specifies whether the success of the module is `required`, `requisite`, `sufficient`, or `optional`.

module_options

specifies available options for the LoginModule. These options vary depending on the LoginModule being used.

The following examples show that the LoginModule used for a Kerberos implementation depends on your JRE.

Oracle JRE

```
JDBC_DRIVER_01 {
  com.sun.security.auth.module.Krb5LoginModule required useTicketCache=true;
};
```

IBM JRE

```
JDBC_DRIVER_01 {
  com.ibm.security.auth.module.Krb5LoginModule required useDefaultCcache=true;
};
```

Refer to Java Authentication and Authorization Service documentation for information about the JAAS login configuration file and implementing authentication technologies.

See also

Configuring the Driver for Kerberos Authentication on page 59
LoginConfigName on page 92

The krb5.conf File

The `krb5.conf` file contains Kerberos configuration information. Typically, the default realm name and the KDC name for that realm are specified in the `krb5.conf` file. However, you can specify the realm and KDC names directly in your application with the `java.security.krb5.realm` and `java.security.krb5.kdc` system properties. Setting these system properties will override the settings in the `krb5.conf` file.

When a client application does not use the `java.security.krb5.realm` and `java.security.krb5.kdc` system properties, the JVM looks for a `krb5.conf` file that contains the realm and KDC names. The JVM first looks for the `krb5.conf` file in the location specified with the `java.security.krb5.conf` system property. If this system property has not been used, then the JVM continues looking for the `krb5.conf` file using an internal algorithm. The JVM must be able to locate the `krb5.conf` to establish a Kerberos connection. Refer to your vendor's JVM documentation for the list of directories that the JVM searches in order to find the `krb5.conf` file.

Depending on your environment, other modifications may need to be made to your `krb5.conf` file. Refer to the following resources for more information on configuring Kerberos and the `krb5.conf` file.

- Your database vendor documentation
- "Keberos Requirements" in *Java™ Documentation*
- "krb5.conf" in *MIT Kerberos Documentation*

See also

Configuring the Driver for Kerberos Authentication on page 59

Specifying User Credentials for Kerberos Authentication (Delegation of Credentials)

By default, when Kerberos authentication is used, the driver takes advantage of the user name and password maintained by the operating system to authenticate users to the database. By allowing the database to share the user name and password used for the operating system, users with a valid operating system account can log into the database without supplying a user name and password.

Many application servers or Web servers act on behalf of the client user logged on the machine on which the application is running, rather than the server user. If you want the driver to use user credentials other than the operating system user name and password, include code in your application to obtain and pass a `javax.security.auth.Subject` used for authentication as shown in the following example.

```
import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import java.sql.*;
//  The following code creates a javax.security.auth.Subject instance
//  used for authentication. Refer to the Java Authentication
//  and Authorization Service documentation for details on using a
//  LoginContext to obtain a Subject.
LoginContext lc = null;
Subject subject = null;
try {
    lc = new LoginContext("JaasSample", new TextCallbackHandler());
    lc.login();
    subject = lc.getSubject();
}
catch (Exception le) {
    ... // display login error
}//  This application passes the javax.security.auth.Subject
//  to the driver by executing the driver code as the subject
Connection con =
   (Connection) Subject.doAs(subject, new PrivilegedExceptionAction() {

   public Object run() {

         Connection con = null;
    try {
         Class.forName("com.ddtek.jdbc.cassandra.CassandraDriver");
         String url = "jdbc:datadirect:cassandra://myServer:27017";
         con = DriverManager.getConnection(url);
        }
     catch (Exception except) {
     ... //log the connection error
            Return null;
        }

         return con;
    }
});
```

```
//  This application now has a connection that was authenticated with
//  the subject. The application can now use the connection.
Statement   stmt = con.createStatement();
String      sql = "SELECT * FROM employee";
ResultSet   rs = stmt.executeQuery(sql);
... // do something with the results
```

See also

Configuring the Driver for Kerberos Authentication on page 59

Obtaining a Kerberos Ticket Granting Ticket

Kerberos uses the credentials in a Ticket Granting Ticket (TGT) to verify the identity of users and control access to services. Depending on your environment, you will need to establish a procedure for obtaining a TGT.

For Windows Active Directory configurations, Active Directory automatically obtains a TGT.

For non-Active Directory configurations, you can enable the application to obtain a TGT in one of two ways. First, you can automate the method of obtaining the TGT as with a keytab. Second, you can require the application user to obtain the TGT with a `kinit` command when logging on.

A TGT can be obtained directly with a `kinit` command to the Kerberos server. For example, the following command requests a TGT from the server with a lifetime of 10 hours, which is renewable for 5 days.

```
kinit -l 10h -r 5d user@REALM
```

Note: The `klist` command can be used on Windows or UNIX/Linux systems to verify that a TGT has been obtained.

Refer to your Kerberos documentation for more information on automating the process of obtaining a TGT.

See also

Configuring the Driver for Kerberos Authentication on page 59

Proxy server support

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

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

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

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

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

Alternatively, you can use the ProxyHost, ProxyPort, ProxyUser, and ProxyPassword connection properties. See Connection Property Descriptions on page 75 and Proxy Server Properties on page 56 for details about these properties.

Data Encryption

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

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

Configuring SSL Encryption

The following steps outline how to configure SSL encryption.

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

To configure SSL encryption:

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

---

1. Set the EncryptionMethod property to `SSL`.
2. Specify the location and password of the truststore file used for SSL server authentication. Either set the TrustStore and TrustStorePassword properties or their corresponding Java system properties (javax.net.ssl.trustStore and javax.net.ssl.trustStorePassword, respectively).
3. To validate certificates sent by the database server, set the ValidateServerCertificate property to `true`.
4. Optionally, set the HostNameInCertificate property to a host name to be used to validate the certificate. The HostNameInCertificate property provides additional security against man-in-the-middle (MITM) attacks by ensuring that the server the driver is connecting to is the server that was requested.
5. If your database server is configured for SSL client authentication, configure your keystore information:
   a) Specify the location and password of the keystore file. Either set the KeyStore and KeyStorePassword properties or their corresponding Java system properties (javax.net.ssl.keyStore and javax.net.ssl.keyStorePassword, respectively).
   b) If any key entry in the keystore file is password-protected, set the KeyPassword property to the key password.

See also

Data Encryption Properties on page 53
Configuring SSL Server Authentication on page 68

Configuring SSL Client Authentication on page 68

Configuring SSL Server Authentication

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

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

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

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

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

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

  ```
  jdbc:datadirect:cassandra://MyServer:9042;KeyspaceName=MyKS;
  Password=secr3t;TrustStore=C:\Certficates\MyTruststore.jks;
  TrustStorePassword=MyTruststorePassword;User=jsmith;
  ```

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

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

Configuring SSL Client Authentication

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

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

The drivers can use the following types of keystores:

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

- PKCS #12 keystores. To gain access to the certificate and its private key, the driver must provide the keystore password. The file extension of the keystore must be .pfx or .p12.

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

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

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

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

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

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

```
jdbc:datadirect:cassandra://MyServer:9042;KeyspaceName=MyKS;
KeyStore=C:\Certficates\MyTruststore.jks;KeyStorePassword=MyKeystorePassword;
Password=secr3t;User=jsmith;
```

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

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

FIPS (Federal Information Processing Standard)

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

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

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

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

Identifiers

Identifiers are used to refer to objects exposed by the driver, such as tables and columns. The driver supports both quoted and unquoted identifiers for naming objects. The maximum length of both quoted and unquoted identifiers is 48 characters for table names and 128 characters for column names. Quoted identifiers must be enclosed in double quotation marks (""). The characters supported in quoted identifiers depends on the version of Cassandra being used. For details on valid characters, refer to the Cassandra documentation for your database version.

Naming conflicts can arise from restrictions imposed by third party applications, from the normalization of native data, or from the truncation of object names. The driver avoids naming conflicts by appending an underscore separator and integer (for example, `_1`) to identifiers with the same name. For example, if a third party application restricts the naming of three columns such that each column retains the name `address`, the driver would expose the columns in the following manner:

- `address`
- `address_1`
- `address_2`

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:cassandra://MyServer:9042;KeyspaceName=MyKS
```

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:cassandra://123.456.78.90:9042;KeyspaceName=MyKS
```

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:

```
CassandraDataSource mds = new CassandraDataSource();
mds.setDescription("My CassandraDataSource");
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:cassandra://[2001:DB8:0:0:8:800:200C:417A]:9042;KeyspaceName=MyKS
```

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

```
jdbc:datadirect:cassandra://[0000:0000:0000:0000:0000:FFFF:123.456.78.90]:54321;KeyspaceName=MyKS
```

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

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

Parameter Metadata Support

The driver supports returning parameter metadata as described in Insert and Update Statements on page 71 and Select Statements on page 71.

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 = ?
```

Isolation Levels

Apache Cassandra does not support transactions. However, manual transactions can be handled to some degree with the TransactionMode connection property. See TransactionMode on page 111 for details.

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][Cassandra 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][Cassandra JDBC Driver][Cassandra]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 driver allows you to retrieve LONGVARBINARY data, using JDBC methods designed for Binary Large Objects (BLOBs). When using these methods to update long data as BLOBs, the updates are made to the local copy of the data contained in the BLOB object.

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

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

To provide the benefits normally associated with BLOBs, 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.

Rowset Support

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

- CachedRowSets
- FilteredRowSets
- WebRowSets
- JoinRowSets
- JDBCRowSets

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

Executing CQL

The driver supports new Native and Refresh escape sequences to embed CQL commands in SQL-92 statements. The Native escape sequence allows you to execute CQL directly through the client application. The Refresh escape sequence is then used to incorporate any changes introduced by the Native escape into the driver's relational map of the data.

Note: The Native and Refresh escape sequences are mainly intended for the execution of DDL commands, such as ALTER, CREATE, and DROP. A returning clause for the Native escape is not currently supported by the driver. Therefore, results cannot be retrieved using the Native escape sequence.

See also

Native and Refresh Escape Sequences on page 118

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 Driver Manager or a JDBC data source. For a Driver Manager 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 24: Driver Properties

Connection String Property	Data Source Method	Default Value
AuthenticationMethod on page 79	setAuthenticationMethod	`userIdPassword`

Connection String Property	Data Source Method	Default Value
ClusterNodes on page 79	setClusterNodes	Empty string
ConfigOptions on page 80	setConfigOptions	Empty string
CreateMap on page 82	setCreateMap	notExist
EncryptionMethod on page 83	setEncryptionMethod	noEncryption
FetchSize on page 84	setFetchSize	100 (rows)
HostNameInCertificate on page 85	setHostNameInCertificate	Empty string
ImportStatementPool on page 86	setImportStatementPool	No default value
InitializationString on page 87	setInitializationString	No default value
InsensitiveResultSetBufferSize on page 87	setInsensitiveResultSetBufferSize	2048 (KB)
KeyPassword on page 88	setKeyPassword	No default value
KeyspaceName on page 89	setKeyspaceName	No default value
KeyStore on page 90	setKeyStore	No default value
KeyStorePassword on page 91	setKeyStorePassword	No default value
LogConfigFile on page 91	setLogConfigFile	ddlogging.properties
LoginConfigName on page 92	setLoginConfigName	JDBC_DRIVER_01
LoginTimeout on page 93	setLoginTimeout	0
MaxPooledStatements on page 94	setMaxPooledStatements	0
NativeFetchSize on page 95	setNativeFetchSize	10000 (rows)
Password on page 96	setPassword	No default value
PortNumber on page 97	setPortNumber	9042
ProxyHost on page 97	setProxyHost	No default value
ProxyPassword on page 98	setProxyPassword	No default value
ProxyPort on page 98	setProxyPort	0
ProxyUser on page 99	setProxyUser	No default value
ReadConsistency on page 99	setReadConsistency	quorum

Connection String Property	Data Source Method	Default Value
ReadOnly on page 101	setReadOnly	`false`
RegisterStatementPoolMonitorMBean on page 101	setRegisterStatementPoolMonitorMBean	`false`
ResultMemorySize on page 102	setResultMemorySize	`-1`
SchemaMap on page 103	setSchemaMap	Default value depends on environment
SecureConnectBundle on page 105	setSecureConnectBundle	`UseConnInfo`
ServerName on page 107	setServerName	No default value
ServicePrincipalName on page 108	setServicePrincipalName	Driver builds value based on environment
SpyAttributes on page 109	setSpyAttributes	No default value
TransactionMode on page 111	setTransactionMode	`noTransactions`
TrustStore on page 112	setTrustStore	No default value
TrustStorePassword on page 113	setTrustStorePassword	No default value
User on page 113	setUser	No default value
ValidateServerCertificate on page 114	setValidateServerCertificate	No default value
WriteConsistency on page 115	setWriteConsistency	`quorum`

For details, see the following topics:

- AuthenticationMethod
- ClusterNodes
- ConfigOptions
- CreateMap
- EncryptionMethod
- FetchSize
- HostNameInCertificate
- ImportStatementPool
- InitializationString
- InsensitiveResultSetBufferSize

- KeyPassword
- KeyspaceName
- KeyStore
- KeyStorePassword
- LogConfigFile
- LoginConfigName
- LoginTimeout
- MaxPooledStatements
- NativeFetchSize
- Password
- PortNumber
- ProxyHost
- ProxyPassword
- ProxyPort
- ProxyUser
- ReadConsistency
- ReadOnly
- RegisterStatementPoolMonitorMBean
- ResultMemorySize
- SchemaMap
- SecureConnectBundle
- ServerName
- ServicePrincipalName
- SpyAttributes
- TransactionMode
- TrustStore
- TrustStorePassword
- User
- ValidateServerCertificate
- WriteConsistency

AuthenticationMethod

Purpose

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

Valid Values

`none` | `kerberos` | `userIdPassword`

Behavior

If set to `none`, the driver does not attempt to authenticate with the server.

If set to `kerberos`, the driver uses Kerberos authentication.

If set to `userIdPassword`, the driver uses `SCRAM-SHA-1` user ID/password authentication. The User property provides the user ID, and the Password property provides the password.

Notes

- In a scenario where AuthenticationMethod has been set to `userIdPassword` but the server has not been configured for user ID/password authentication, the driver will still connect to the database server.
- If authentication has not been enabled, client applications will have access to all keyspaces on the server. If authentication has been enabled, a client application will only have access to the keyspace specified by the KeyspaceName property assuming it has the required permissions.

Data Source Method

setAuthenticationMethod

Default

`userIdPassword`

Data Type

String

See also

- Authentication on page 58
- Authentication Properties on page 53

ClusterNodes

Purpose

A comma-separated list of member nodes in your cluster to which the driver attempts to connect. Specifying a value for this property enables connection failover and load balancing when connecting to a cluster.

Valid Values

server_name | *port_number* [, ...]

where:

server_name

is the name of a server for a member node or the IP address of the server for a member node in the replica-set cluster to which you want to connect.

port_number

is the port number of the server listener for the corresponding member node.

For example:

```
server1:9042,255.125.1.11:9043,"2001:DB8:0000:0000:8:800:200C:417A":9044
```

Notes

- When specifying a value for this property, the driver randomly selects from the list for a server node to first attempt a connection. If that connection fails, the driver again randomly selects another server node from this list until all servers in the list have been tried or a connection is successfully established.

Data Source Method

setClusterNodes

Default

Empty string

Data Type

String

ConfigOptions

Purpose

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

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:

- `SchemaFormat`

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=(SchemaFormat=NormalizeNonFrozen)
```

Data Source Method

`setConfigOptions`

Default

Empty string

Data Type

String

SchemaFormat (configuration option)

Purpose

Determines how the driver maps collections to the relational view of your data.

Valid Values

`NormalizeAll` | `NormalizeNonFrozen` | `NormalizeNone`

Behavior

If set to `NormalizeAll`, the driver normalizes all collection types when generating a relational view of your data. Collections are mapped to as child tables with foreign key relationships to parent tables. See "Complex Type Normalization" for more details.

If set to `NormalizeNonFrozen`, the driver normalizes collections not labeled FROZEN when generating a relational view of your data. Collection fields that are labeled FROZEN are returned as JSON strings in the table in which they are defined.

If set to `NormalizeNone`, the driver does not normalize any collection types when generating a relational view of your data. All collection fields are returned as JSON formatted strings in the table in which they are defined.

Default

`NormalizeAll`

See also

- Complex Type Normalization on page 13

CreateMap

Purpose

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

Valid Values

`forceNew` | `no` | `notExist`

Behavior

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

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

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

Notes

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

Data Source Method

setCreateMap

Default

`notExist`

Data Type

String

See also

- SchemaMap on page 103
- Refresh Map (EXT) on page 120
- Mapping Properties on page 52

EncryptionMethod

Purpose

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

Valid Values

`noEncryption` | `SSL`

Behavior

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

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

Notes

When SSL is enabled, the following properties also apply:

- HostNameInCertificate
- KeyStore (for SSL client authentication)
- KeyStorePassword (for SSL client authentication)
- KeyPassword (for SSL client authentication)
- TrustStore
- TrustStorePassword
- ValidateServerCertificate

Data Source Method

setEncryptionMethod

Default

`noEncryption`

Data Type

String

See also

- Data Encryption on page 67
- Performance Considerations on page 57

FetchSize

Purpose

Specifies the 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 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.
- The ResultMemorySize connection property can also be used to reduce demands on memory and decrease the likelihood of out-of-memory errors.
- FetchSize is related to, but different than, NativeFetchSize. NativeFetchSize specifies the number of rows of raw data that the driver fetches from the native data source, while FetchSize specifies how many of these rows the driver processes before returning control to the application. Processing the data includes converting native data types to SQL data types used by the application. If FetchSize is greater than NativeFetchSize, the driver may make multiple round trips to the data source to get the requested number of rows before returning control to the application.

Data Source Method

setFetchSize

Default

`100` (rows)

Data Type

Int

See also

- Additional Properties on page 55
- Performance Considerations on page 57
- NativeFetchSize on page 95
- ResultMemorySize on page 102

HostNameInCertificate

Purpose

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

Valid Values

`host_name` | `#SERVERNAME#`

where:

`host_name`

is a valid host name.

Behavior

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

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

Notes

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

Data Source Method

setHostNameInCertificate

Default

Empty string

Data Type

String

See also

Data Encryption on page 67

ImportStatementPool

Purpose

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

Valid Values

`string`

where:

`string`

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

Notes

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

Data Source Method

setImportStatementPool

Default

No default value

Data Type

String

See also

- Refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference* for further details.
- Statement Pooling Properties on page 55
- MaxPooledStatements on page 94

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.

Data Source Method

setInitializationString

Default

No default value

Data Type

String

See also

Additional Properties on page 55

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 in KB.

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` (KB)

Data Type

Int

See also

Additional Properties on page 55

KeyPassword

Purpose

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

Valid Values

string

where:

string

is a valid password.

Data Source Method

setKeyPassword

Default

None

Data Type

String

See also

Data Encryption on page 67

KeyspaceName

Purpose

Specifies the default name of the Cassandra keyspace to which the driver connects. This value is used as the default qualifier for unqualified table names in SQL queries.

Valid Values

`keyspace_name`

where:

`keyspace_name`

is the name of a valid CQL keyspace. If the driver cannot find the specified keyspace, the connection fails.

Notes

- A Cassandra keyspace is equivalent to a relational database.
- If KeyspaceName is not specified, Cassandra's internal keyspace name `system` will be used.
- If authentication has not been enabled, client applications will have access to all keyspaces on the server. If authentication has been enabled, a client application will only have access to the keyspace specified by the KeyspaceName property assuming it has the required permissions.

Data Source Method

setKeyspaceName

Default

No default value

Data Type

String

See also

Required Properties on page 52

KeyStore

Purpose

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

Valid Values

`string`

where:

`string`

is a valid directory of a keystore file.

Notes

- This value overrides the directory of the keystore file that is specified by the javax.net.ssl.keyStore Java system property. If this property is not specified, the keystore directory is specified by the javax.net.ssl.keyStore Java system property.
- The keystore and truststore files can be the same file.

Data Source Method

setKeyStore

Default

None

Data Type

String

See also

Data Encryption on page 67

KeyStorePassword

Purpose

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

Valid Values

string

where:

string

is a valid password.

Notes

- This value overrides the password of the keystore file that is specified by the javax.net.ssl.keyStorePassword Java system property. If this property is not specified, the keystore password is specified by the javax.net.ssl.keyStorePassword Java system property.
- The keystore and truststore files can be the same file.

Data Source Method

setKeyStorePassword

Default

None

Data Type

String

See also

Data Encryption on page 67

LogConfigFile

Purpose

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

Valid Values

string

where:

`string`

is the relative or fully qualified path of the configuration file used to initialize driver logging. If the specified file does not exist, the driver continues searching for an appropriate configuration file.

Data Source Method

setLogConfigFile

Default

`ddlogging.properties`

Data Type

String

See also

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

LoginConfigName

Purpose

Specifies the name of the entry in the JAAS login configuration file that contains the authentication technology used by the driver to establish a Kerberos connection. The LoginModule-specific items found in the entry are passed on to the LoginModule.

Valid Values

`entry_name`

where:

`entry_name`

is the name of the entry that contains the authentication technology used with the driver.

Example

In the following example, `JDBC_DRIVER_01` is the entry name while the authentication technology and related settings are found in the brackets.

```
JDBC_DRIVER_01 {
  com.sun.security.auth.module.Krb5LoginModule required useTicketCache=true;
};
```

Data Source Method

setLoginConfigName

Default

`JDBC_DRIVER_01`

Data Type

String

See also

- Authentication on page 58
- The JAAS Login Configuration File on page 63
- Authentication Properties on page 53

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

See also

Additional Properties on page 55

MaxPooledStatements

Purpose

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

Valid Values

`0` | *x*

where:

`x`

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

Behavior

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

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

Notes

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

Example

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

Data Source Method

setMaxPooledStatements

Default

`0`

Data Type

Int

See also

- Refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference* for further details.
- Statement Pooling Properties on page 55
- Performance Considerations on page 57
- RegisterStatementPoolMonitorMBean on page 101

NativeFetchSize

Purpose

Specifies the number of rows of data the driver attempts to fetch from the native data source on each request submitted to the server.

Valid Values

`0` | *x*

where:

x

is a positive integer that defines the number of rows.

Behavior

If set to `0`, the driver requests that the server return all rows for each request submitted to the server. Block fetching is not used.

If set to *x*, the driver attempts to fetch up to a maximum of the specified number of rows on each request submitted to the server.

Notes

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

Data Source Method

setNativeFetchSize

Default

`10000` (rows)

Data Type

Int

See also

- Additional Properties on page 55
- Performance Considerations on page 57
- FetchSize on page 84
- ResultMemorySize on page 102

Password

Description

Specifies the password used to connect to your database for user ID/password authentication.

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

Valid Values

password

where:

password

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

Notes

When `AuthenticationMethod=kerberos`, the driver ignores the Password property.

Data Source Method

setPassword

Default

No default value

Data Type

String

See also

- Authentication on page 58
- Authentication Properties on page 53

PortNumber

Purpose

Specifies the port number of the server that is listening for connections to the Cassandra keyspace.

Valid Values

port

where:

port

is the number port of the server listener.

Data Source Method

setPortNumber

Default

9042

Data Type

Int

See also

Required Properties on page 52

ProxyHost

Description

Identifies a proxy server to use for the first connection.

Valid Values

server_name | *IP_address*

where:

server_name

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

IP_address

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

Data Source Method

`setProxyHost`

Default

empty string

See also

- Proxy Server Properties on page 56

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

Proxy Server Properties on page 56

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

Proxy Server Properties on page 56

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

Proxy Server Properties on page 56

ReadConsistency

Purpose

Specifies how many replicas must respond to a read request before returning data to the client application.

Valid Values

all | quorum | one | all | quorum | local_quorum | one | two | three | local_one | serial | local_serial

Behavior

If set to all, data is returned to the application after all replicas have responded. This setting provides the highest consistency and lowest availability.

If set to quorum, data is returned after a quorum of replicas has responded from any data center.

If set to local_quorum, data is returned after a quorum of replicas in the same data center as the coordinator node has responded. This setting voids latency of inter-data center communication.

If set to one, data is returned from the closest replica. This setting provides the highest availability, but increases the likelihood of stale data being read.

If set to two, data is returned from two of the closest replicas.

If set to three, data is returned from three of the closest replicas.

If set to local_one, data is returned from the closest replica in the local data center.

If set to serial, the data is read without proposing a new addition or update. Uncommitted transactions are committed as part of the read.

If set to local_serial, the data within a data center is read without proposing a new addition or update. Uncommitted transactions within the data center are committed as part of the read.

Notes

- If the server does not support the ReadConsistency value specified, the connection attempt fails and the driver returns a consistency level error.
- Refer to Apache Cassandra documentation for more information about configuring consistency levels, including usage scenarios.
- If you wish to specify a ReadConsistency value besides the values documented in this section, contact Technical Support.

Data Source Method

setReadConsistency

Default

quorum

Data Type

String

See also

- Additional Properties on page 55
- Performance Considerations on page 57
- ReadOnly on page 101
- WriteConsistency on page 115

ReadOnly

Purpose

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

Valid Values

`true` | `false`

Behavior

If set to `true`, the connection supports read-only access.

If set to `false`, the connection supports read-write access.

Data Source Method

setReadOnly

Default

`false`

Data Type

Boolean

See also

Additional Properties on page 55

RegisterStatementPoolMonitorMBean

Purpose

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

Valid Values

`true` | `false`

Behavior

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

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

Notes

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

Data Source 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 55
- MaxPooledStatements on page 94

ResultMemorySize

Purpose

Specifies the maximum size, in megabytes, of an intermediate result set that the driver holds in memory. When this threshold is reached, the driver writes a portion of the result set to disk in temporary files.

Valid Values

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

where:

`x`

is the maximum size, in MB, of an intermediate result set that the driver holds in memory.

Behavior

If set to `-1`, the maximum size of an intermediate result that the driver holds in memory is a percentage of the max Java heap size. When this threshold is reached, the driver writes a portion of the result set to disk.

If set to `0`, the driver holds the entire intermediate result set in memory regardless of size. No portion of the result set is written to disk. Setting ResultMemorySize to `0` can improve performance for result sets that easily fit within the JVM's free heap space, but can diminish performance for result sets that barely fit within the JVM's free heap space.

If set to *`x`*, the driver holds intermediate results in memory that are no larger than the size specified. When this threshold is reached, the driver writes a portion of the result set to disk.

Notes

- By default, ResultMemorySize is set to `-1`. When set to `-1`, the maximum size of an intermediate result that the driver holds in memory is a percentage of the max Java heap size. When processing large sets of data, out-of-memory errors can occur when the size of the result set exceeds the available memory allocated to the JVM. In this scenario, you can tune ResultMemorySize to suit your environment. To begin, set ResultMemorySize equal to the max Java heap size divided by 4. Proceed by decreasing this value until out-of-memory errors are eliminated. As a result, the maximum size of an intermediate result set the driver holds in memory will be reduced, and some portion of the result set will be written to disk. Be aware that while writing to disk reduces the risk of out-of-memory errors, it also negatively impacts performance. For optimal performance, decrease this value only to a size necessary to avoid errors.
- You can also adjust the max Java heap size to address memory and performance concerns. By increasing the max Java heap size, you increase the amount of data the driver accumulates in memory. This can reduce the likelihood of out-of-memory errors and improve performance by ensuring that result sets fit easily within the JVM's free heap space. In addition, when a limit is imposed by setting ResultMemorySize to `-1`, increasing the max Java heap size can improve performance by reducing the need to write to disk, or removing it altogether.
- The FetchSize connection property can also be used to reduce demands on memory and decrease the likelihood of out-of-memory errors.

Data Source Method

setResultMemorySize

Default

`-1`

Data Type

Int

See also

- Additional Properties on page 55
- Performance Considerations on page 57
- FetchSize on page 84

SchemaMap

Purpose

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

Valid Values

`string`

where:

`string`

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

Notes

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

Example

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

Driver Manager Example

```
jdbc:datadirect:cassandra://MyServer:9042;KeyspaceName=MyKS;
SchemaMap=C:\Users\Default\AppData\Local\Progress\DataDirect
   \Cassandra_Schema\MyServer.config
```

Data Source Example

```
CassandraDataSource mds = new CassandraDataSource();
mds.setDescription("My CassandraDataSource");
mds.setServerName("MyServer");
mds.setPortNumber(9042);
mds.setKeyspaceName("MyKS");
mds.setSchemaMap("C:\\Users\\Default\\AppData\\Local\\Progress
   \\DataDirect\\Cassandra_Schema\\MyServer.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\Cassandra_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/Cassandra_schema/*user_name*.config

Data Type

String

See also

- Complex Type Normalization on page 13
- Refresh Map (EXT) on page 120

SecureConnectBundle

Purpose

Specifies the name and location of the secure connect bundle that contains security certificates and credentials for your database. Provide a value for this property only if you are connecting to a database that uses a secure connect bundle.

Valid Values

bundle_path | UseConnInfo

where:

bundle_path

is the path and filename of the zip file containing your security . For example, C:\secure-connect-*database_name*.zip.

Behavior

If set to *bundle_path*, the driver uses the security certificates and credentials contained in the specified secure connect bundle .zip file to connect.

If set to `UseConnInfo`, the driver uses credential information stored in the connection string or data source, instead of a `.zip` file, to connect to databases that employ a secure connect bundle. The following properties are obtained from the connection string in lieu of the `.zip` file:

- KeyStore
- KeyStorePassword
- KeySpaceName
- PortNumber
- ServerName
- TrustStore
- TrustStorePassword

Notes

- Escapes are required when specifying a path for this property in a data source. For example, `C:\\path\\secure-connect-`*`database_name`*`.zip`.
- When the value of SecureConnectBundle is set to `UseConnInfo`, the values for the host (ServerName) and port (PortNumber) in the string must be the same as those in the `config.json` of the `.zip` file.
- With the exception of KeySpaceName, the values stored in the secure connect bundle override corresponding values for connection properties specified in the connection string or data source. The value for KeyspaceName in the connection string or data source always takes precedent.

Data Source Method

setSecureConnectBundle

Default

No default value

Data Type

String

See also

- KeyStore on page 90
- KeyStorePassword on page 91
- KeyspaceName on page 89
- PortNumber on page 97
- ServerName on page 107
- TrustStore on page 112
- TrustStorePassword on page 113

ServerName

Purpose

Specifies the name or the IP address of the server to which you want to connect.

Important: In a Kerberos configuration, an IP address cannot be used for the ServerName connection property. The value of the ServerName property must be the same as the fully qualified host name used in the Kerberos service principal name.

Valid Values

`server_name` | *`IP_address`*

where:

`server_name`

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

`IP_address`

is the IP address of the server to which you want to connect.

The IP address can be specified in either IPv4 or IPv6 format, or a combination of the two.

Example

If your network supports named servers, you can specify a server name such as `MyServer`, or you can specify an IP address such as `199.226.224.34`.

Data Source Method

setServerName

Default

No default value

Data Type

String

See also

- Required Properties on page 52
- IP Addresses on page 70

ServicePrincipalName

Purpose

Specifies the three-part service principal name registered with the key distribution center (KDC) in a Kerberos configuration.

Valid Values

Service_Name/Fully_Qualified_Domain_Name@REALM_NAME

where:

Service_Name

is the name of the service hosting the instance. The default value is `cassandra`.

Fully_Qualified_Domain_Name

is the fully qualified domain name (FQDN) of the host machine. (The FQDN consists of a host name and a domain name. For the example `myserver.test.com`, `myserver` is the host name and `test.com` is the domain name.) The FQDN registered with the KDC must match the FQDN in your connection string.

REALM_NAME

is the Kerberos realm name. This part of the value must be specified in upper-case characters, for example, `EXAMPLE.COM`. For Windows Active Directory, the Kerberos realm name is the Windows domain name.

Notes

- If the ServicePrincipalName property is unspecified, the driver builds a service principal name in the following manner.
  - `cassandra` is used as the service name.
  - The FQDN is obtained from the connection string.
  - The default realm in the `krb5.conf` file is used as the realm name.
- You must specify a value for the ServicePrincipalName property if any of the following statements are valid.
  - You are using a service name other than the default service name `cassandra`.
  - The FQDN in your connection string is different from the FQDN registered with the KDC.
  - You are using a Kerberos realm other than the default realm specified in the `krb5.conf` file.
- In a Kerberos configuration, an IP address cannot be used as a FQDN.
- If AuthenticationMethod is set to `userIdPassword`, the value of the ServicePrincipalName property is ignored.

Example

The following is an example of a valid service principal name.

```
cassandra/myserver.test.com@EXAMPLE.COM
```

Default

Driver builds value based on environment

See also

- Authentication on page 58
- Authentication Properties on page 53
- The krb5.conf File on page 64

SpyAttributes

Purpose

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

Valid Values

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

where:

`spy_attribute`

is any valid DataDirect Spy attribute.

Behavior

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

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

Notes

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

Example

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

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

Data Source Method

setSpyAttributes

Default

No default value

Data Type

String

See also

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

TransactionMode

Purpose

Specifies how the driver handles manual transactions.

Valid Values

`noTransactions` | `ignore`

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 return an error 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

See also

Additional Properties on page 55

TrustStore

Purpose

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

Valid Values

`string`

where:

`string`

is the directory of the truststore file.

Notes

- This value overrides the directory of the truststore file that is specified by the javax.net.ssl.trustStore Java system property. If this property is not specified, the truststore directory is specified by the javax.net.ssl.trustStore Java system property.
- This property is ignored if ValidateServerCertificate=`false`.

Data Source Method

setTrustStore

Default

None

Data Type

String

See also

Data Encryption on page 67

TrustStorePassword

Purpose

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

Valid Values

string

where:

string

is a valid password for the truststore file.

Notes

- This value overrides the password of the truststore file that is specified by the javax.net.ssl.trustStorePassword Java system property. If this property is not specified, the truststore password is specified by the javax.net.ssl.trustStorePassword Java system property.
- This property is ignored if ValidateServerCertificate=`false`.

Data Source Method

setTrustStorePassword

Default

None

Data Type

String

See also

Data Encryption on page 67

User

Purpose

Specifies the user ID for user ID/password authentication or the user principal name for Kerberos authentication.

Valid Values

userid | *user_principal_name*

where:

userid

is a valid user ID with permissions to access the keyspace using user ID/password authentication.

user_principal_name

is a valid Kerberos user principal name with permissions to access the keyspace using Kerberos authentication.

Notes

When `AuthenticationMethod=kerberos`, the User property does not have to be specified. The driver uses the user principal name in the Kerberos Ticket Granting Ticket (TGT) as the value for the User property. Any value specified must be a valid Kerberos user principal name used in the Kerberos authentication protocol.

Data Source Method

setUser

Default

No default value

Data Type

String

See also

- Authentication on page 58
- Authentication Properties on page 53

ValidateServerCertificate

Purpose

Determines whether the driver validates the certificate that is sent by the database server when SSL encryption is enabled (`EncryptionMethod=SSL`). When using SSL server authentication, any certificate that is sent by the server must be issued by a trusted Certificate Authority (CA).

Valid Values

`true` | `false`

Behavior

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

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

Notes

- Truststore information is specified using the TrustStore and TrustStorePassword properties or by using Java system properties.
- Allowing the driver to trust any certificate that is returned from the server even if the issuer is not a trusted CA is useful in test environments because it eliminates the need to specify truststore information on each client in the test environment.

Data Source Method

setValidateServerCertificate

Default

`true`

Data Type

boolean

See also

Data Encryption on page 67

WriteConsistency

Purpose

Determines the number of replicas on which the write must succeed before returning an acknowledgment to the client application.

Valid Values

`all` | `quorum` | `one` | `all` | `each_quorum` | `quorum` | `local_quorum` | `one` | `two` | `three` | `local_one` | `any` | `serial` | `local_serial`

Behavior

If set to `all`, a write must succeed on all replica nodes in the cluster for that partition key. This setting provides the highest consistency and lowest availability.

If set to `each_quorum`, a write must succeed on a quorum of replica nodes across a data center.

If set to `quorum`, a write must succeed on a quorum of replica nodes.

If set to `local_quorum`, a write must succeed on a quorum of replica nodes in the same data center as the coordinator node. This setting voids latency of inter-data center communication.

If set to `one`, a write must succeed on at least one replica node.

If set to `two`, a write must succeed on at least two replica nodes.

If set to `three`, a write must succeed on at least three replica nodes.

If set to `local_one`, a write must succeed on at least one replica node in the local data center.

If set to `any`, a write must succeed on at least one node. Even if all replica nodes for the given partition key are down, the write can succeed after a hinted handoff has been written. This setting provides the lowest consistency and highest availability.

If set to `serial`, the driver prevents unconditional updates to achieve linearizable consistency for lightweight transactions.

If set to `local_serial`, the driver prevents unconditional updates to achieve linearizable consistency for lightweight transactions within the data center.

Notes

- An update operation can result in a consistency level error if the server does not support the WriteConsistency value specified.
- Refer to Apache Cassandra documentation for more information about configuring consistency levels, including usage scenarios.
- If you wish to specify a ReadConsistency value besides the values documented in this section, contact Technical Support.

Data Source Method

setWriteConsistency

Default

`quorum`

Data Type

String

See also

- Additional Properties on page 55
- Performance Considerations on page 57
- ReadConsistency on page 99
- ReadOnly on page 101

5

Supported SQL Functionality

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

For details, see the following topics:

- Data Definition Language (DDL)
- Delete
- Insert
- Refresh Map (EXT)
- Select
- Update
- SQL Expressions
- Subqueries

Data Definition Language (DDL)

The driver supports data store-specific DDL through the Native and Refresh escape sequences. See "Native and Refresh Escape Sequences" for details.

Native and Refresh Escape Sequences

The driver supports the Native and Refresh escape sequences to embed data store-specific commands in SQL-92 statements. The Native escape sequence allows you to execute native commands directly through the client application, while the Refresh escape sequence is used to incorporate any changes introduced by the Native escape into the driver's relational map of the data.

Note: The Native and Refresh escape sequences are mainly intended for the execution of DDL commands, such as ALTER, CREATE, and DROP. A returning clause for the Native escape is not currently supported by the driver. Therefore, results cannot be retrieved using the Native escape sequence.

The Native escape sequences can be used with the following syntax:

`{native(`*`command_text`*`)}`

where:

`command_text`

is a data store-specific command.

The Refresh escape sequence has no additional argument and takes the form:

`{refresh}`

The following example shows the execution of two data store-specific commands with a refresh of the driver's relational map of the data. Note that each Native escape sequence must have its own execute method. The Refresh escape, however, can be used in the same execute statement as the Native escape.

```
stmt.executeUpdate ("{native (CREATE TABLE emp (empid int, title varchar))}");
stmt.executeUpdate ("{native (CREATE TABLE dept (deptid int, city varchar))}{refresh}");
```

See also

Supported SQL Functionality on page 117

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 on page 126 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

- Delete is supported for primitive types and non-nested complex types. See "Complex Type Normalization" for details.
- To enable Insert, Update, and Delete, set the ReadOnly connection property to `false`.
- A Where clause can be used to restrict which rows are deleted.

See also

Complex Type Normalization on page 13

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

In Cassandra, Inserts are in effect Upserts. When an Insert is performed on a row that already exists, the row will be updated.

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 on page 134 for more information.

select_statement

is a query that returns values for each *column_name* value specified in the column list. Using a Select statement instead of a list of value expressions lets you select a set of rows from one table and insert it into another table using a single Insert statement. The Select statement is evaluated before any values are inserted. This query cannot be made on the table into which values are inserted. See Select on page 121 for information about Select statements.

Notes

- Insert is supported for primitive types and non-nested complex types. See "Complex Type Normalization" for details.
- The driver supports an insert on a child table prior to an insert on a parent table, circumventing referential integrity constraints associated with traditional RDBMS. To maintain integrity between parent and child tables, it is recommended that an insert be performed on the parent table for each foreign key value added to the child. If such an insert is not first performed, the driver automatically inserts a row into the parent tables that contains only the primary key values and NULL values for all non-primary key columns. See "Complex Type Normalization" for details.
- To enable Insert, Update, and Delete, set the ReadOnly connection property to `false`.

See also

Complex Type Normalization on page 13

Refresh Map (EXT)

Purpose

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

Syntax

```
REFRESH MAP
```

Notes

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

Select

Purpose

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

Syntax

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

where:

select_clause

specifies the columns from which results are to be returned by the query. See Select Clause on page 122 for a complete explanation.

from_clause

specifies one or more tables on which the other clauses in the query operate. See From Clause on page 124 for a complete explanation.

where_clause

is optional and restricts the results that are returned by the query. See Where Clause on page 126 for a complete explanation.

groupby_clause

is optional and allows query results to be aggregated in terms of groups. See Group By Clause on page 127 for a complete explanation.

having_clause

is optional and specifies conditions for groups of rows (for example, display only the departments that have salaries totaling more than $200,000). See Having Clause on page 127 for a complete explanation.

UNION

is an optional operator that combines the results of the left and right Select statements into a single result. See Union Operator on page 128 for a complete explanation.

`INTERSECT`

is an optional operator that returns a single result by keeping any distinct values from the results of the left and right Select statements. See Intersect Operator on page 129 for a complete explanation.

`EXCEPT | MINUS`

are synonymous optional operators that returns a single result by taking the results of the left Select statement and removing the results of the right Select statement. See Except and Minus Operators on page 129 for a complete explanation.

`orderby_clause`

is optional and sorts the results that are returned by the query. See Order By Clause on page 130 for a complete explanation.

`limit_clause`

is optional and places an upper bound on the number of rows returned in the result. See Limit Clause on page 131 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 offset number | 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 on page 133 for details. *`column_expression`* can also include aggregate functions. See Aggregate Functions on page 123 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`.

Aggregate Functions

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

The following table lists supported aggregate functions.

Table 25: Aggregate Functions

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

Example A

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

```
COUNT (DISTINCT last_name)
```

Example B

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

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

From Clause

Purpose

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

Syntax

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

where:

table_name

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

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

Subqueries can be used instead of table names. Subqueries must be enclosed in parentheses. See Subquery in a From Clause on page 126 for an example.

table_alias

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

Example

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

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

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

```
FROM emp E
```

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

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

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

Outer Join Escape Sequences

Purpose

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

Syntax

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

where *outer-join* is

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

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

For example:

```
SELECT customers.custid, customers.name, orders.orderid, orders.status FROM {oj customers
 LEFT OUTER JOIN orders ON customers.custid=orders.custid} WHERE orders.status='OPEN'
```

The following outer join escape sequences are supported:

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

Join in a From Clause

Purpose

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

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

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

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

Syntax

```
FROM table_name {RIGHT OUTER | INNER | LEFT OUTER | CROSS | 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

For more information about subqueries, see Subqueries on page 141.

Where Clause

Purpose

Specifies the conditions that rows must meet to be retrieved.

Syntax

```
WHERE expr1 rel_operator expr2
```

where:

expr1

is either a column name, literal, or expression.

expr2

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

rel_operator

is the relational operator that links the two expressions.

Example

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

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

See also

- SQL Expressions on page 133
- Subqueries on page 141

Group By Clause

Purpose

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

Syntax

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

where:

column_expression

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

Example

The following example totals the salaries in each department:

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

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

See also

- SQL Expressions on page 133
- Subqueries on page 141

Having Clause

Purpose

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

Syntax

```
HAVING expr1 rel_operator expr2
```

where:

expr1 | *expr2*

> can be column names, constant values, or expressions. These expressions do not have to match a column expression in the Select clause. See SQL Expressions on page 133 for details regarding SQL expressions.

rel_operator

is the relational operator that links the two expressions.

Example

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

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

See also

- SQL Expressions on page 133
- Subqueries on page 141

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 133

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

- Update is supported for primitive types, non-nested Tuple types, and non-nested user-defined types. Update is also supported for values in non-nested Map types. The driver does not support updates on List types, Set types, or keys in Map types because the values in each are part of the primary key of their respective child tables and primary key columns cannot be updated. If an Update is attempted when not allowed, the driver issues the following error message:

  ```
  [DataDirect][Cassandra JDBC Driver][Cassandra]syntax error or access rule
  violation: UPDATE not permitted for column: column_name
  ```

  See "Complex Type Normalization" for details.

- Update is supported for Counter columns when all the other columns in the row comprise that row's primary key. The Counter column itself is the only updatable field in the row. When updating a Counter column on an existing row, the Counter column is updated according to the increment (or decrement) specified in the SQL statement. When updating a Counter column for which there is no existing row, the values of the columns that comprise the row's primary key are inserted into the table alongside the value of the Counter column.

  For example, consider the following table.

  ```
  CREATE TABLE page_view_counts (
    counter_value counter,
    url_name varchar,
    page_name varchar,
    PRIMARY KEY (url_name, page_name));
  ```

The following Update can be performed on the page_view_counts table.

```
UPDATE page_view_counts
   SET counter_value=counter_value + 1
   WHERE url_name = 'www.progress.com' AND page_name = 'home'
```

This Update would provide the following output.

Note: Cassandra initially assigns a value of 0 (zero) to Counter columns. An increment or decrement can be specified in the SQL statement.

```
 url_name         | page_name | counter_value
------------------+-----------+---------------
 www.progress.com |      home |             1
```

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

See also

- Where Clause on page 126
- Subqueries on page 141
- Complex Type Normalization on page 13
- Native and Refresh Escape Sequences on page 118

SQL Expressions

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

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

The driver supports both unquoted and quoted identifiers. An unquoted identifier must start with an ASCII alpha character and can be followed by zero or more ASCII alphanumeric characters. Unquoted identifiers are converted to uppercase before being used.

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

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

Valid expression elements are:

- Column names
- Literals
- Operators
- Functions

Column Names

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

Literals

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

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

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

Table 26: 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`

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

Character String Literals

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

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

Example

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

Numeric Literals

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

Example

```
+1894.1204
```

Binary Literals

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

Example

```
'00af123d'
```

Date/Time Literals

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

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

Integer Literals

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

Notes

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

Example

```
1994 or -2
```

Operators

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

Unary Operator

A unary operator operates on only one operand.

operator operand

Binary Operator

A binary operator operates on two operands.

operand1 operator operand2

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

Arithmetic Operators

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

Table 27: Arithmetic Operators

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

Concatenation Operator

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

Table 28: Concatenation Operator

<table>
<tr><th>Operator</th><th>Purpose</th><th>Example</th></tr>
<tr><td>||</td><td>Concatenates character strings.</td><td><code>SELECT 'Name is' || ename FROM emp</code></td></tr>
</table>

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 29: 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

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 30: 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 31: 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 JDBC core functions. For details, refer to "JDBC support" in the *Progress DataDirect for JDBC Drivers Reference*.

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 32: 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)
```