Progress® DataDirect Connect® Series *for* JDBC™

Reference

Release 5.1.4

Notices

For details, see the following topics:

- Copyright

Copyright

Table of Contents

Preface..........11

About This Reference..........11
What Is Progress DataDirect Connect Series *for* JDBC?..........11
Using This Reference..........12
About the Product Documentation..........13
Typographical Conventions..........13
Contacting Technical Support..........14

Chapter 1: JDBC Support..........17

JDBC and JVM Compatibility..........17
Supported Functionality..........17
Array..........18
Blob..........18
CallableStatement..........19
Clob..........30
Connection..........31
ConnectionEventListener..........36
ConnectionPoolDataSource..........36
DatabaseMetaData..........36
DataSource..........45
Driver..........45
ParameterMetaData..........46
PooledConnection..........47
PreparedStatement..........47
Ref..........52
ResultSet..........52
ResultSetMetaData..........62
RowSet..........63
SavePoint..........64
Statement..........64
StatementEventListener..........68
Struct..........68
XAConnection..........68
XADataSource..........69
XAResource..........69

Chapter 2: JDBC Extensions..........71

Using JDBC Wrapper Methods to Access JDBC Extensions..........72

DatabaseMetaData Interface (Salesforce Driver)....73
DDBulkLoad Interface....73
ExtConnection Interface....80
ExtDatabaseMetaData Interface....85
ExtLogControl Class....85

Chapter 3: Supported SQL Functionality and Extensions for The Driver for Apache Hive....87

Data Definition Language (DDL)....87
Insert....88
Selecting Data With the Driver....89
Select List....89
From Clause....89
Group By Clause....90
Having Clause....90
Order By Clause....90
For Update Clause....90
Set Operators....90
Subqueries....91
SQL Expressions....91
Constants....91
Numeric Operators....92
Character Operator....92
Relational Operators....92
Logical Operators....93
Functions....93
Restrictions....95

Chapter 4: Supported SQL Statements and Extensions for the Salesforce Driver....97

Alter Cache (EXT)....98
Relational Caches....100
Alter Index....100
Alter Sequence....100
Alter Session (EXT)....101
Alter Table....102
Altering a Remote Table....102
Altering a Local Table....105
Checkpoint....108
Create Cache (EXT)....108
Relational Caches....109
Referencing Clause....110

- Refresh Interval Clause....110
- Initial Check Clause....111
- Persist Clause....111
- Enabled Clause....112
- Call Limit Clause....113
- Filter Clause....114

Create Index....115

Create Sequence....115
- Next Value For Clause....116

Create Table....116
- Creating a Remote Table....116
- Creating a Local Table....121

Create View....127

Delete....128

Drop Cache (EXT)....129

Drop Index....129

Drop Sequence....130

Drop Table....131

Drop View....131

Explain Plan....132

Insert....132
- Specifying an External ID Column....133

Refresh Cache (EXT)....134

Refresh Schema (EXT)....135

Select....135
- Select Clause....136
- From Clause....139

Set Checkpoint Defrag....145

Set Logsize....146

Update....146

SQL Expressions....147
- Column Names....148
- Literals....148

Operators....150
- Unary Operator....150
- Binary Operator....150
- Arithmetic Operators....151
- Concatenation Operator....151
- Comparison Operators....151
- Logical Operators....153
- Operator Precedence....153

Functions....154

Conditions....159

Subqueries....159
- IN Predicate....160

EXISTS Predicate....................160
UNIQUE Predicate....................160
Correlated Subqueries....................161

Chapter 5: getTypeInfo()....................163

DB2 Driver....................164
Informix Driver....................175
MySQL Driver....................184
Oracle Driver....................195
PostgreSQL Driver....................203
Progress OpenEdge Driver....................210
SQL Server Driver....................218
Sybase Driver....................232
The Driver for Apache Hive....................244
Greenplum Driver....................249
Salesforce Driver....................256

Chapter 6: Designing JDBC Applications for Performance Optimization.265

Using Database Metadata Methods....................266
Minimizing the Use of Database Metadata Methods....................266
Avoiding Search Patterns....................267
Using a Dummy Query to Determine Table Characteristics....................267
Returning Data....................268
Returning Long Data....................268
Reducing the Size of Returned Data....................269
Choosing the Right Data Type....................269
Retrieving Result Sets....................269
Selecting JDBC Objects and Methods 270
Using Parameter Markers as Arguments to Stored Procedures....................270
Using the Statement Object Instead of the PreparedStatement Object....................270
Using Batches Instead of Prepared Statements....................271
Choosing the Right Cursor....................272
Using get Methods Effectively....................272
Retrieving Auto Generated Keys....................273
Managing Connections and Updates....................273
Managing Connections....................274
Managing Commits in Transactions....................274
Choosing the Right Transaction Model....................275
Using updateXXX Methods....................275
Using getBestRowIdentifier....................275

Chapter 7: SQL Escape Sequences for JDBC....................277

Date, Time, and Timestamp Escape Sequences....................278

Scalar Functions....278
Outer Join Escape Sequences....287
LIKE Escape Character Sequence for Wildcards....289
Procedure Call Escape Sequences....289

Chapter 8: Using DataDirect Test....291

DataDirect Test Tutorial....291
Configuring DataDirect Test....292
Starting DataDirect Test....292
Connecting Using DataDirect Test....293
Executing a Simple Select Statement....296
Executing a Prepared Statement....297
Retrieving Database Metadata....300
Scrolling Through a Result Set....302
Batch Execution on a Prepared Statement....304
Returning ParameterMetaData....307
Establishing Savepoints....308
Updatable Result Sets....311
Retrieving Large Object Data....319

Chapter 9: Tracking JDBC Calls with DataDirect Spy....325

Enabling DataDirect Spy....325
Using the JDBC Driver Manager....326
Using JDBC Data Sources....327
DataDirect Spy Attributes....328

Chapter 10: Connection Pool Manager....331

About JDBC Connection Pools....332
Configuring the Connection Pool....332
Understanding the Maximum Pool Size....333
Using Reauthentication with the Pool Manager....333
Checking the Pool Manager Version....334
Enabling Pool Manager Tracing....335
Using a DataDirect Connection Pool....335
Creating a Driver DataSource Object....335
Creating the Connection Pool....336
Connecting Using a Connection Pool....338
Closing the Connection Pool....339
DataDirect Connection Pool Manager Interfaces....339
PooledConnectionDataSourceFactory....340
PooledConnectionDataSource....340
ConnectionPoolMonitor....344

Chapter 11: Statement Pool Monitor....347
Using DataDirect-Specific Methods to Access the Statement Pool Monitor....348
Using the poolEntries Method....348
Generating a List of Statements in the Statement Pool....349
Using JMX to Access the Statement Pool Monitor....350
Importing Statements into a Statement Pool....352
Clearing All Statements in a Statement Pool....352
Freezing and Unfreezing the Statement Pool....353
Generating a Statement Pool Export File....353
DataDirect Statement Pool Monitor Interfaces and Classes....353
ExtStatementPoolMonitor Class....354
ExtStatementPoolMonitorMBean Interface....354

Chapter 12: Troubleshooting....357
Troubleshooting Your Application....357
Turning On and Off DataDirect Spy Logging....358
DataDirect Spy Log Example....358
Troubleshooting Connection Pooling....360
Enabling Pool Manager Tracing....360
Pool Manager Trace File Example....360
Troubleshooting Statement Pooling....364
Generating a Statement Pool Export File....364
Statement Pool Export File Example....364
Using Java Logging (Salesforce)....365
Logging Components....365
Configuring Logging....367
Using the JVM....367
Using the Driver....368

Glossary....369

Index....373

Preface

For details, see the following topics:

- About This Reference
- What Is Progress DataDirect Connect Series for JDBC?
- Using This Reference
- About the Product Documentation
- Typographical Conventions
- Contacting Technical Support

About This Reference

This reference provides information on the Progress® DataDirect Connect® Series *for* JDBC™, which includes the following products:

- DataDirect Connect *for* JDBC
- DataDirect Connect XE *for* JDBC

What Is Progress DataDirect Connect Series *for* JDBC?

Progress DataDirect Connect Series *for* JDBC provides a suite of JDBC drivers that supports most leading databases. The drivers are compliant with Type 4 architecture, but provide advanced features that define them as Type 5 drivers. These features include:

- Application failover
- Distributed transactions
- Bulk load

The drivers consistently support the latest database features and are fully compliant with Java™ SE 8 and JDBC 4.0 functionality.

Using This Reference

This reference assumes that you are familiar with your operating system and its commands, the definition of directories, and accessing a database through an end-user application.

This reference contains the following information:

- JDBC Support on page 17 provides information about the JDBC interfaces and methods supported for DataDirect Connect Series *for* JDBC.
- JDBC Extensions on page 71 describes the JDBC extensions provided by the com.ddtek.jdbc.extensions package.
- Supported SQL Statements and Extensions for the Salesforce Driver on page 97 describes the standard SQL statements and the SQL extensions supported by the Salesforce driver.
- getTypeInfo() on page 163 provides results returned from the DataBaseMetaData.getTypeinfo() method for the drivers.
- Designing JDBC Applications for Performance Optimization on page 265 explains how you optimize your application code to improve performance.
- SQL Escape Sequences for JDBC on page 277 describes the scalar functions supported by the drivers. Your data store may not support all these functions.
- Using DataDirect Test on page 291 contains a tutorial that takes you through a step-by-step example of how to use DataDirect Test™ *for* JDBC, a tool that allows you to test and debug your JDBC applications during development.
- Tracking JDBC Calls with DataDirect Spy on page 325 describes how to use DataDirect Spy™ *for* JDBC for tracking JDBC calls in running applications.
- Connection Pool Manager on page 331 describes how to use the DataDirect Connection Pool Manager to create your own connection pooling mechanism.
- Statement Pool Monitor on page 347 describes how to use the DataDirect Statement Pool Monitor to import statements to and remove statements from the statement pool as well as generate information to help you troubleshoot statement pooling performance.
- Troubleshooting on page 357 provides information that can help you troubleshoot driver problems.

Note: This reference refers the reader to Web pages using URLs for more information about specific topics, including Web URLs not maintained by Progress DataDirect. Because it is the nature of Web content to change frequently, Progress DataDirect can guarantee only that the URLs in this reference were correct at the time of publishing.

About the Product Documentation

The Progress DataDirect Connect Series *for* JDBC library consists of the following guides:

- *Progress DataDirect Connect Series for JDBC Installation Guide* details requirements and procedures for installing the product.
- *Progress DataDirect Connect Series for JDBC User's Guide* provides information about customizing and using the product.
- *Progress DataDirect Connect Series for JDBC Reference* provides reference information for using the product.

Installed Documentation

The *User's Guide* and *Reference* are installed with the product as an HTML-based help system. This help system is located in the help subdirectory of the product installation directory. You can use the help system with any of the following browsers:

- Google Chrome 33.*x* or higher
- Internet Explorer 9.*x* or higher
- Mozilla Firefox 27.*x* or higher
- Safari 5.1.7 or higher
- Opera 20.*x* or higher

Online Documentation

The Progress DataDirect Connect Series *for* JDBC library is available online in HTML and PDF formats by searching the supported database system on the DataDirect Connectors Documentation Web Page.

Typographical Conventions

This guide uses the following typographical conventions:

Convention	Explanation
italics	Introduces new terms with which you may not be familiar, and is used occasionally for emphasis.
bold	Emphasizes important information. Also indicates button, menu, and icon names on which you can act. For example, click **Next**.
BOLD UPPERCASE	Indicates keys or key combinations that you can use. For example, press the **ENTER** key.
UPPERCASE	Indicates SQL reserved words.

Convention	Explanation
`monospace`	Indicates syntax examples, values that you specify, or results that you receive.
monospaced italics	Indicates names that are placeholders for values that you specify. For example, *filename*.
>	Separates menus and their associated commands. For example, Select **File > Copy** means that you should select **Copy** from the **File** menu.
/	The slash also separates directory levels when specifying locations under UNIX.
vertical rule |	Indicates an "OR" separator used to delineate items.
brackets [ ]	Indicates optional items. For example, in the following statement: `SELECT [DISTINCT]`, `DISTINCT` is an optional keyword. Also indicates sections of the Windows Registry.
braces { }	Indicates that you must select one item. For example, {`yes` | `no`} means that you must specify either `yes` or `no`.
ellipsis . . .	Indicates that the immediately preceding item can be repeated any number of times in succession. An ellipsis following a closing bracket indicates that all information in that unit can be repeated.

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.

1

JDBC Support

This section provides information, such as JDBC/JVM compatibility and how JDBC interfaces are supported, to help you develop JDBC applications for use with Progress DataDirect drivers.

Note: This section describes the behavior of multiple drivers across the spectrum of Progress DataDirect drivers. The functionality described may not necessarily apply to your driver or database system.

For details, see the following topics:

- JDBC and JVM Compatibility
- Supported Functionality

JDBC and JVM Compatibility

The drivers are compatible with JDBC 2.0, 3.0, 4.0, 4.1, and 4.2. The drivers are supported on Java SE 5 and higher JVMs.

Note: The Salesforce driver requires a Java SE 7 or higher JVM to comply with Salesforce security standards.

Supported Functionality

This section lists functionality supported for the following JDBC interfaces.

Array

Array Methods	Version Introduced	Supported	Comments
void free()	4.0	Yes	
Object getArray()	2.0 Core	Yes	
Object getArray(map)	2.0 Core	Yes	The drivers ignore the map argument.
Object getArray(long, int)	2.0 Core	Yes	
Object getArray(long, int, map)	2.0 Core	Yes	The drivers ignore the map argument.
int getBaseType()	2.0 Core	Yes	
String getBaseTypeName()	2.0 Core	Yes	
ResultSet getResultSet()	2.0 Core	Yes	
ResultSet getResultSet(map)	2.0 Core	Yes	The drivers ignore the map argument.
ResultSet getResultSet(long, int)	2.0 Core	Yes	
ResultSet getResultSet(long, int, map)	2.0 Core	Yes	The drivers ignore the map argument.

Blob

Blob Methods	Version Introduced	Supported	Comments
void free()	4.0	Yes	
InputStream getBinaryStream()	2.0 Core	Yes	The drivers support using data types that map to the JDBC LONGVARBINARY data type.
byte[] getBytes(long, int)	2.0 Core	Yes	The drivers support using data types that map to the JDBC LONGVARBINARY data type.
long length()	2.0 Core	Yes	The drivers support using data types that map to the JDBC LONGVARBINARY data type.

Blob Methods	Version Introduced	Supported	Comments
long position(Blob, long)	2.0 Core	Yes	The Informix driver requires that the pattern parameter (which specifies the Blob object designating the BLOB value for which to search) be less than or equal to a maximum value of 4096 bytes. All other drivers support using data types that map to the JDBC LONGVARBINARY data type.
long position(byte[], long)	2.0 Core	Yes	The Informix driver requires that the pattern parameter (which specifies the byte array for which to search) be less than or equal to a maximum value of 4096 bytes. All other drivers support using data types that map to the JDBC LONGVARBINARY data type.
OutputStream setBinaryStream(long)	3.0	Yes	The drivers support using data types that map to the JDBC LONGVARBINARY data type.
int setBytes(long, byte[])	3.0	Yes	The drivers support using data types that map to the JDBC LONGVARBINARY data type.
int setBytes(long, byte[], int, int)	3.0	Yes	The drivers support using data types that map to the JDBC LONGVARBINARY data type.
void truncate(long)	3.0	Yes	The drivers support using data types that map to the JDBC LONGVARBINARY data type.

CallableStatement

CallableStatement Methods	Version Introduced	Supported	Comments
Array getArray(int)	2.0 Core	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters. The Progress OpenEdge driver throws an "unsupported method" exception.
Array getArray(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw an "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
Reader getCharacterStream(int)	4.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
Reader getCharacterStream(String)	4.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "unsupported method" exception.
BigDecimal getBigDecimal(int)	2.0 Core	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
BigDecimal getBigDecimal(int, int)	1.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
BigDecimal getBigDecimal(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
Blob getBlob(int)	2.0 Core	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters. All other drivers support using data types that map to the JDBC LONGVARBINARY data type.
Blob getBlob(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
boolean getBoolean(int)	1.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
boolean getBoolean(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
byte getByte(int)	1.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.

CallableStatement Methods	Version Introduced	Supported	Comments
byte getByte(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
byte [] getBytes(int)	1.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
byte [] getBytes(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
Clob getClob(int)	2.0 Core	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters. All other drivers support using data types that map to the JDBC LONGVARBINARY data type.
Clob getClob(String)	3.0	Yes	Supported for the SQL Server driver only using with data types that map to the JDBC LONGVARCHAR data type. All other drivers throw "unsupported method" exception.
Date getDate(int)	1.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
Date getDate(int, Calendar)	2.0 Core	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
Date getDate(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
Date getDate(String, Calendar)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
double getDouble(int)	1.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
double getDouble(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
float getFloat(int)	1.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
float getFloat(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
int getInt(int)	1.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
int getInt(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
long getLong(int)	1.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
long getLong(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
Reader getNCharacterStream(int)	4.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "unsupported method" exception.
Reader getNCharacterStream(String)	4.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "unsupported method" exception.
NClob getNClob(int)	4.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
NClob getNClob(String)	4.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "unsupported method" exception.
String getNString(int)	4.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw "unsupported method" exception.
String getNString(String)	4.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "unsupported method" exception.
Object getObject(int)	1.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
Object getObject(int, Map)	2.0 Core	Yes	The drivers ignore the Map argument.
Object getObject(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
Object getObject(String, Map)	3.0	Yes	Supported for the SQL Server driver only. The SQL Server driver ignores the Map argument. All other drivers throw "unsupported method" exception.
Ref getRef(int)	2.0 Core	No	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters. All other drivers throw "unsupported method" exception.
Ref getRef(String)	3.0	No	The drivers throw "unsupported method" exception.
short getShort(int)	1.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
short getShort(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
SQLXML getSQLXML(int)	4.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "unsupported method" exception.
SQLXML getSQLXML(String)	4.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "unsupported method" exception.
String getString(int)	1.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
String getString(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
Time getTime(int)	1.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
Time getTime(int, Calendar)	2.0 Core	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
Time getTime(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
Time getTime(String, Calendar)	3.0	Yes	Supported for SQL Server driver only. All other drivers throw "unsupported method" exception.
Timestamp getTimestamp(int)	1.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
Timestamp getTimestamp(int, Calendar)	2.0 Core	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
Timestamp getTimestamp(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
Timestamp getTimestamp(String, Calendar)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
URL getURL(int)	3.0	No	The drivers throw "unsupported method" exception.
URL getURL(String)	3.0	No	The drivers throw "unsupported method" exception.
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
void registerOutParameter(int, int)	1.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
void registerOutParameter(int, int, int)	1.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters.
void registerOutParameter(int, int, String)	2.0 Core	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters. The Oracle driver supports the String argument. For all other drivers, the String argument is ignored.
void registerOutParameter(String, int)	3.0	Yes	Supported for the SQL Server driver only. The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters. All other drivers throw "unsupported method" exception.
void registerOutParameter(String, int, int)	3.0	Yes	Supported for the SQL Server driver only. The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters. All other drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
void registerOutParameter(String, int, String)	3.0	Yes	Supported for the SQL Server driver only. The drivers for Salesforce and Oracle Service Cloud throw an "invalid parameter bindings" exception when your application calls output parameters. All other drivers throw "unsupported method" exception. String/typename ignored.
void setArray(int, Array)	2.0 Core	Yes	Supported for the Oracle driver only. All other drivers throw "unsupported method" exception.
void setAsciiStream(String, InputStream)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setAsciiStream(String, InputStream, int)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setAsciiStream(String, InputStream, long)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setBigDecimal(String, BigDecimal)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setBinaryStream(String, InputStream)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setBinaryStream(String, InputStream, int)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setBinaryStream(String, InputStream, long)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setBlob(String, Blob)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
void setBlob(String, InputStream)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setBlob(String, InputStream, long)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setBoolean(String, boolean)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setByte(String, byte)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setBytes(String, byte [])	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setCharacterStream(String, Reader, int)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setCharacterStream(String, InputStream, long)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setClob(String, Clob)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setClob(String, Reader)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setClob(String, Reader, long)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setDate(String, Date)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
void setDate(String, Date, Calendar)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setDouble(String, double)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setFloat(String, float)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setInt(String, int)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setLong(String, long)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setNCharacterStream(String, Reader, long)	4.0	Yes	
void setNClob(String, NClob)	4.0	Yes	
void setNClob(String, Reader)	4.0	Yes	
void setNClob(String, Reader, long)	4.0	Yes	
void setNString(String, String)	4.0	Yes	
void setNull(int, int, String)	2.0 Core	Yes	
void setNull(String, int)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setNull(String, int, String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setObject(String, Object)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
void setObject(String, Object, int)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setObject(String, Object, int, int)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setShort(String, short)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setSQLXML(String, SQLXML)	4.0	Yes	The drivers for Salesforce and Oracle Service Cloud throw an "unsupported method" exception.
void setString(String, String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setTime(String, Time)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setTime(String, Time, Calendar)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setTimestamp(String, Timestamp)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setTimestamp(String, Timestamp, Calendar)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
<T> T unwrap(Class<T> iface)	4.0	Yes	
void setURL(String, URL)	3.0	No	The drivers throw "unsupported method" exception.
boolean wasNull()	1.0	Yes	

Clob

Clob Methods	Version Introduced	Supported	Comments
void free()	4.0	Yes	
InputStream getAsciiStream()	2.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.
Reader getCharacterStream()	2.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.
Reader getCharacterStream(long, long)	4.0	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.
String getSubString(long, int)	2.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.
long length()	2.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.
long position(Clob, long)	2.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type. The Informix driver requires that the searchStr parameter be less than or equal to a maximum value of 4096 bytes.
long position(String, long)	2.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type. The Informix driver requires that the searchStr parameter be less than or equal to a maximum value of 4096 bytes.
OutputStream setAsciiStream(long)	3.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.
Writer setCharacterStream(long)	3.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.

Clob Methods	Version Introduced	Supported	Comments
int setString(long, String)	3.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.
int setString(long, String, int, int)	3.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.
void truncate(long)	3.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.

Connection

Connection Methods	Version Introduced	Supported	Comments
void clearWarnings()	1.0	Yes	
void close()	1.0	Yes	When a connection is closed while a transaction is still active, that transaction is rolled back.
void commit()	1.0	Yes	
Blob createBlob()	4.0	Yes	
Clob createClob()	4.0	Yes	
NClob createNClob()	4.0	Yes	
createArrayOf(String, Object[])	4.0	No	The drivers throw an unsupported method exception.
createStruct(String, Object[])	4.0	Yes	Only the Oracle driver supports this method.
SQLXML createSQLXML()	4.0	Yes	
Statement createStatement()	1.0	Yes	

Connection Methods	Version Introduced	Supported	Comments
Statement createStatement(int, int)	2.0 Core	Yes	For the DB2 driver, ResultSet.TYPE_SCROLL_SENSITIVE is downgraded to TYPE_SCROLL_INSENSITIVE. For the drivers for Salesforce and Oracle Service Cloud, be aware that scroll-sensitive result sets are expensive from both a Web service call and a performance perspective. The drivers expend a network round trip for each row that is fetched.
Statement createStatement(int, int, int)	3.0	No	With the exception of the DB2 driver, the specified holdability must match the database default holdability. Otherwise, an "unsupported method" exception is thrown. For the DB2 driver, the method can be called regardless of whether the specified holdability matches the database default holdability.
Struct createStruct(String, Object[])	1.0	Yes	Supported for the Oracle driver only. All other drivers throw "unsupported method" exception.
boolean getAutoCommit()	1.0	Yes	
String getCatalog()	1.0	Yes	The drivers for the listed database systems return an empty string because they do not have the concept of a catalog: Oracle, PostgreSQL, Apache Cassandra, Apache Hive, Apache Spark SQL, Impala, Greenplum, Salesforce, Oracle Service Cloud, MongoDB, and Amazon Redshift.
String getClientInfo()	4.0	Yes	The drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB do not support storing or retrieving client information.
String getClientInfo(String)	4.0	Yes	The drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB do not support storing or retrieving client information.
int getHoldability()	3.0	Yes	
DatabaseMetaData getMetaData()	1.0	Yes	
int getTransactionIsolation()	1.0	Yes	

Connection Methods	Version Introduced	Supported	Comments
Map getTypeMap()	2.0 Core	Yes	Always returns empty java.util.HashMap.
SQLWarning getWarnings()	1.0	Yes	
boolean isClosed()	1.0	Yes	
boolean isReadOnly()	1.0	Yes	
boolean isValid()	4.0	Yes	
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
String nativeSQL(String)	1.0	Yes	Always returns the same String that was passed in from the application.
CallableStatement prepareCall(String)	1.0	Yes	
CallableStatement prepareCall(String, int, int)	2.0 Core	Yes	For the drivers for Apache Cassandra, DB2, Salesforce, Oracle Service Cloud, and MongoDB, ResultSet.TYPE_SCROLL_ SENSITIVE is downgraded to TYPE_SCROLL_INSENSITIVE.
CallableStatement prepareCall(String, int, int, int)	3.0	Yes	The DB2 driver allows this method whether or not the specified holdability is the same as the default holdability. The other drivers throw the exception "Changing the default holdability is not supported" when the specified holdability does not match the default holdability.
PreparedStatement prepareStatement (String)	1.0	Yes	
PreparedStatement prepareStatement (String, int)	3.0	Yes	
PreparedStatement prepareStatement (String, int, int)	2.0 Core	Yes	For the DB2 driver, ResultSet.TYPE_SCROLL_ SENSITIVE is downgraded to TYPE_SCROLL_INSENSITIVE. For the drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB, be aware that scroll-sensitive result sets are expensive from both a Web service call and a performance perspective. The drivers expend a network round trip for each row that is fetched.

Connection Methods	Version Introduced	Supported	Comments
PreparedStatement prepareStatement (String, int, int, int)	3.0	No	All drivers throw "unsupported method" exception.
PreparedStatement prepareStatement (String, int[])	3.0	Yes	Supported for the Oracle and SQL Server drivers. All other drivers throw "unsupported method" exception.
PreparedStatement prepareStatement (String, String [])	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void releaseSavepoint(Savepoint)	3.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. The drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB throw an "unsupported method" exception.
void rollback()	1.0	Yes	
void rollback(Savepoint)	3.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. The drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB throw an "unsupported method" exception.
void setAutoCommit(boolean)	1.0	Yes	The drivers for Apache Cassandra, Apache Hive, Apache Spark SQL, Impala, Salesforce, Oracle Service Cloud, and MongoDB throw "transactions not supported" exception if set to `false`.
void setCatalog(String)	1.0	Yes	The driver for the listed database systems ignore any value set by the String argument.The corresponding drivers return an empty string because they do not have the concept of a catalog: Oracle, PostgreSQL, Apache Cassandra, Apache Hive, Apache Spark SQL, Impala, Greenplum, Salesforce, Oracle Service Cloud, MongoDB, and Amazon Redshift.

Connection Methods	Version Introduced	Supported	Comments
String setClientInfo(Properties)	4.0	Yes	The drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB do not support storing or retrieving client information.
String setClientInfo(String, String)	4.0	Yes	The drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB do not support storing or retrieving client information.
void setHoldability(int)	3.0	Yes	The DB2 driver supports the Holdability parameter value. For other drivers, the Holdability parameter value is ignored.
void setReadOnly(boolean)	1.0	Yes	
Savepoint setSavepoint()	3.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. In addition, the DB2 driver only supports multiple nested savepoints for DB2 V8.2 and higher for Linux/UNIX/Windows. The drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB throw an "unsupported method" exception.
Savepoint setSavepoint(String)	3.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. In addition, the DB2 driver only supports multiple nested savepoints for DB2 V8.2 and higher for Linux/UNIX/Windows. The drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB throw an "unsupported method" exception.
void setTransactionIsolation(int)	1.0	Yes	The drivers for Apache Cassandra, Apache Hive, Apache Spark SQL, Impala, Salesforce, Oracle Service Cloud, and MongoDB ignore any specified transaction isolation level.
void setTypeMap(Map)	2.0 Core	Yes	The drivers ignore this connection method.
<T> T unwrap(Class<T> iface)	4.0	Yes	

ConnectionEventListener

ConnectionEventListener Methods	Version Introduced	Supported	Comments
void connectionClosed(event)	3.0	Yes	
void connectionErrorOccurred(event)	3.0	Yes	

ConnectionPoolDataSource

ConnectionPoolDataSource Methods	Version Introduced	Supported	Comments
int getLoginTimeout()	2.0 Optional	Yes	
PrintWriter getLogWriter()	2.0 Optional	Yes	
PooledConnection getPooledConnection()	2.0 Optional	Yes	
PooledConnection getPooledConnection (String, String)	2.0 Optional	Yes	
void setLoginTimeout(int)	2.0 Optional	Yes	
void setLogWriter(PrintWriter)	2.0 Optional	Yes	

DatabaseMetaData

DatabaseMetaData Methods	Version Introduced	Supported	Comments
boolean autoCommitFailureClosesAllResultSets()	4.0	Yes	
boolean allProceduresAreCallable()	1.0	Yes	
boolean allTablesAreSelectable()	1.0	Yes	
boolean dataDefinitionCausesTransactionCommit()	1.0	Yes	
boolean dataDefinitionIgnoredInTransactions()	1.0	Yes	
boolean deletesAreDetected(int)	2.0 Core	Yes	

DatabaseMetaData Methods	Version Introduced	Supported	Comments
boolean doesMaxRowSizeIncludeBlobs()	1.0	Yes	Not supported by the SQL Server and Sybase drivers.
getAttributes(String, String, String, String)	3.0	Yes	The Oracle driver may return results. All other drivers return an empty result set.
ResultSet getBestRowIdentifier(String, String, String, int, boolean)	1.0	Yes	
ResultSet getCatalogs()	1.0	Yes	
String getCatalogSeparator()	1.0	Yes	
String getCatalogTerm()	1.0	Yes	
String getClientInfoProperties()	4.0	Yes	The drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB do not support storing or retrieving client information.
ResultSet getColumnPrivileges(String, String, String, String)	1.0	Yes	Not supported by the drivers for Apache Hive, Apache Spark SQL, or Impala.
ResultSet getColumns(String, String, String, String)	1.0	Yes	
Connection getConnection()	2.0 Core	Yes	
ResultSet getCrossReference(String, String, String, String, String, String)	1.0	Yes	
ResultSet getFunctions()	4.0	Yes	The drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB return an empty result set. Not supported by the drivers for Apache Hive, Apache Spark SQL, or Impala.
ResultSet getFunctionColumns()	4.0	Yes	The drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB return an empty result set. Not supported by the drivers for Apache Hive, Apache Spark SQL, or Impala.
int getDatabaseMajorVersion()	3.0	Yes	
int getDatabaseMinorVersion()	3.0	Yes	
String getDatabaseProductName()	1.0	Yes	

DatabaseMetaData Methods	Version Introduced	Supported	Comments
String getDatabaseProductVersion()	1.0	Yes	
int getDefaultTransactionIsolation()	1.0	Yes	
int getDriverMajorVersion()	1.0	Yes	
int getDriverMinorVersion()	1.0	Yes	
String getDriverName()	1.0	Yes	
String getDriverVersion()	1.0	Yes	
ResultSet getExportedKeys(String, String, String)	1.0	Yes	
String getExtraNameCharacters()	1.0	Yes	
String getIdentifierQuoteString()	1.0	Yes	
ResultSet getImportedKeys(String, String, String)	1.0	Yes	
ResultSet getIndexInfo(String, String, String, boolean, boolean)	1.0	Yes	
int getJDBCMajorVersion()	3.0	Yes	
int getJDBCMinorVersion()	3.0	Yes	
int getMaxBinaryLiteralLength()	1.0	Yes	
int getMaxCatalogNameLength()	1.0	Yes	
int getMaxCharLiteralLength()	1.0	Yes	
int getMaxColumnNameLength()	1.0	Yes	
int getMaxColumnsInGroupBy()	1.0	Yes	
int getMaxColumnsInIndex()	1.0	Yes	
int getMaxColumnsInOrderBy()	1.0	Yes	
int getMaxColumnsInSelect()	1.0	Yes	
int getMaxColumnsInTable()	1.0	Yes	
int getMaxConnections()	1.0	Yes	
int getMaxCursorNameLength()	1.0	Yes	

DatabaseMetaData Methods	Version Introduced	Supported	Comments
int getMaxIndexLength()	1.0	Yes	
int getMaxProcedureNameLength()	1.0	Yes	
int getMaxRowSize()	1.0	Yes	
int getMaxSchemaNameLength()	1.0	Yes	
int getMaxStatementLength()	1.0	Yes	
int getMaxStatements()	1.0	Yes	
int getMaxTableNameLength()	1.0	Yes	
int getMaxTablesInSelect()	1.0	Yes	
int getMaxUserNameLength()	1.0	Yes	
String getNumericFunctions()	1.0	Yes	
ResultSet getPrimaryKeys(String, String, String)	1.0	Yes	
ResultSet getProcedureColumns(String, String, String, String)	1.0	Yes	For the drivers for Salesforce and Oracle Service Cloud, SchemaName and ProcedureName must be explicit values; they cannot be patterns. The drivers for Apache Cassandra and MongoDB return an empty result set. Not supported for the drivers for Apache Hive, Apache Spark SQL, or Impala.
ResultSet getProcedures(String, String, String)	1.0	Yes	Not supported for the drivers for Apache Hive, Apache Spark SQL, or Impala. The drivers for Apache Cassandra and MongoDB return an empty result set.
String getProcedureTerm()	1.0	Yes	
int getResultSetHoldability()	3.0	Yes	
ResultSet getSchemas()	1.0	Yes	
ResultSet getSchemas(catalog, pattern)	4.0	Yes	
String getSchemaTerm()	1.0	Yes	
String getSearchStringEscape()	1.0	Yes	

DatabaseMetaData Methods	Version Introduced	Supported	Comments
String getSQLKeywords()	1.0	Yes	
int getSQLStateType()	3.0	Yes	
String getStringFunctions()	1.0	Yes	
ResultSet getSuperTables(String, String, String)	3.0	Yes	Returns an empty result set.
ResultSet getSuperTypes(String, String, String)	3.0	Yes	Returns an empty result set.
String getSystemFunctions()	1.0	Yes	
ResultSet getTablePrivileges(String, String, String)	1.0	Yes	Not supported for the drivers for Apache Hive, Apache Spark SQL, or Impala.
ResultSet getTables(String, String, String, String [])	1.0	Yes	
ResultSet getTableTypes()	1.0	Yes	
String getTimeDateFunctions()	1.0	Yes	
ResultSet getTypeInfo()	1.0	Yes	
ResultSet getUDTs(String, String, String, int [])	2.0 Core	Yes	Supported for Oracle only.
String getURL()	1.0	Yes	
String getUserName()	1.0	Yes	
ResultSet getVersionColumns(String, String, String)	1.0	Yes	
boolean insertsAreDetected(int)	2.0 Core	Yes	
boolean isCatalogAtStart()	1.0	Yes	
boolean isReadOnly()	1.0	Yes	
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
boolean locatorsUpdateCopy()	3.0	Yes	
boolean nullPlusNonNullIsNull()	1.0	Yes	
boolean nullsAreSortedAtEnd()	1.0	Yes	

DatabaseMetaData Methods	Version Introduced	Supported	Comments
boolean nullsAreSortedAtStart()	1.0	Yes	
boolean nullsAreSortedHigh()	1.0	Yes	
boolean nullsAreSortedLow()	1.0	Yes	
boolean othersDeletesAreVisible(int)	2.0 Core	Yes	
boolean othersInsertsAreVisible(int)	2.0 Core	Yes	
boolean othersUpdatesAreVisible(int)	2.0 Core	Yes	
boolean ownDeletesAreVisible(int)	2.0 Core	Yes	
boolean ownInsertsAreVisible(int)	2.0 Core	Yes	
boolean ownUpdatesAreVisible(int)	2.0 Core	Yes	
boolean storesLowerCaseIdentifiers()	1.0	Yes	
boolean storesLowerCaseQuotedIdentifiers()	1.0	Yes	
boolean storesMixedCaseIdentifiers()	1.0	Yes	
boolean storesMixedCaseQuotedIdentifiers()	1.0	Yes	
boolean storesUpperCaseIdentifiers()	1.0	Yes	
boolean storesUpperCaseQuotedIdentifiers()	1.0	Yes	
boolean supportsAlterTableWithAddColumn()	1.0	Yes	
boolean supportsAlterTableWithDropColumn()	1.0	Yes	
boolean supportsANSI92EntryLevelSQL()	1.0	Yes	
boolean supportsANSI92FullSQL()	1.0	Yes	
boolean supportsANSI92IntermediateSQL()	1.0	Yes	
boolean supportsBatchUpdates()	2.0 Core	Yes	
boolean supportsCatalogsInDataManipulation()	1.0	Yes	

DatabaseMetaData Methods	Version Introduced	Supported	Comments
boolean supportsCatalogsInIndexDefinitions()	1.0	Yes	
boolean supportsCatalogsInPrivilegeDefinitions()	1.0	Yes	
boolean supportsCatalogsInProcedureCalls()	1.0	Yes	
boolean supportsCatalogsInTableDefinitions()	1.0	Yes	
boolean supportsColumnAliasing()	1.0	Yes	
boolean supportsConvert()	1.0	Yes	
boolean supportsConvert(int, int)	1.0	Yes	
boolean supportsCoreSQLGrammar()	1.0	Yes	
boolean supportsCorrelatedSubqueries()	1.0	Yes	
boolean supportsDataDefinitionAndData ManipulationTransactions()	1.0	Yes	
boolean supportsDataManipulationTransactionsOnly()	1.0	Yes	
boolean supportsDifferentTableCorrelationNames()	1.0	Yes	
boolean supportsExpressionsInOrderBy()	1.0	Yes	
boolean supportsExtendedSQLGrammar()	1.0	Yes	
boolean supportsFullOuterJoins()	1.0	Yes	
boolean supportsGetGeneratedKeys()	3.0	Yes	
boolean supportsGroupBy()	1.0	Yes	
boolean supportsGroupByBeyondSelect()	1.0	Yes	
boolean supportsGroupByUnrelated()	1.0	Yes	
boolean supportsIntegrityEnhancementFacility()	1.0	Yes	
boolean supportsLikeEscapeClause()	1.0	Yes	

DatabaseMetaData Methods	Version Introduced	Supported	Comments
boolean supportsLimitedOuterJoins()	1.0	Yes	
boolean supportsMinimumSQLGrammar()	1.0	Yes	
boolean supportsMixedCaseIdentifiers()	1.0	Yes	
boolean supportsMixedCaseQuotedIdentifiers()	1.0	Yes	
boolean supportsMultipleOpenResults()	3.0	Yes	
boolean supportsMultipleResultSets()	1.0	Yes	
boolean supportsMultipleTransactions()	1.0	Yes	
boolean supportsNamedParameters()	3.0	Yes	
boolean supportsNonNullableColumns()	1.0	Yes	
boolean supportsOpenCursorsAcrossCommit()	1.0	Yes	
boolean supportsOpenCursorsAcrossRollback()	1.0	Yes	
boolean supportsOpenStatementsAcrossCommit()	1.0	Yes	
boolean supportsOpenStatementsAcrossRollback()	1.0	Yes	
boolean supportsOrderByUnrelated()	1.0	Yes	
boolean supportsOuterJoins()	1.0	Yes	
boolean supportsPositionedDelete()	1.0	Yes	
boolean supportsPositionedUpdate()	1.0	Yes	
boolean supportsResultSetConcurrency(int, int)	2.0 Core	Yes	
boolean supportsResultSetHoldability(int)	3.0	Yes	
boolean supportsResultSetType(int)	2.0 Core	Yes	
boolean supportsSavePoints()	3.0	Yes	
boolean supportsSchemasInDataManipulation()	1.0	Yes	

DatabaseMetaData Methods	Version Introduced	Supported	Comments
boolean supportsSchemasInIndexDefinitions()	1.0	Yes	
boolean supportsSchemasInPrivilegeDefinitions()	1.0	Yes	
boolean supportsSchemasInProcedureCalls()	1.0	Yes	
boolean supportsSchemasInTableDefinitions()	1.0	Yes	
boolean supportsSelectForUpdate()	1.0	Yes	
boolean supportsStoredFunctionsUsingCallSyntax()	4.0	Yes	
boolean supportsStoredProcedures()	1.0	Yes	
boolean supportsSubqueriesInComparisons()	1.0	Yes	
boolean supportsSubqueriesInExists()	1.0	Yes	
boolean supportsSubqueriesInIns()	1.0	Yes	
boolean supportsSubqueriesInQuantifieds()	1.0	Yes	
boolean supportsTableCorrelationNames()	1.0	Yes	
boolean supportsTransactionIsolationLevel(int)	1.0	Yes	
boolean supportsTransactions()	1.0	Yes	
boolean supportsUnion()	1.0	Yes	
boolean supportsUnionAll()	1.0	Yes	
<T> T unwrap(Class<T> iface)	4.0	Yes	
boolean updatesAreDetected(int)	2.0 Core	Yes	
boolean usesLocalFilePerTable()	1.0	Yes	
boolean usesLocalFiles()	1.0	Yes	

DataSource

The DataSource interface implements the javax.naming.Referenceable and java.io.Serializable interfaces.

DataSource Methods	Version Introduced	Supported	Comments
Connection getConnection()	2.0 Optional	Yes	
Connection getConnection(String, String)	2.0 Optional	Yes	
int getLoginTimeout()	2.0 Optional	Yes	
PrintWriter getLogWriter()	2.0 Optional	Yes	
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
void setLoginTimeout(int)	2.0 Optional	Yes	
void setLogWriter(PrintWriter)	2.0 Optional	Yes	Enables DataDirect Spy, which traces JDBC information into the specified PrintWriter.
<T> T unwrap(Class<T> iface)	4.0	Yes	

Driver

Driver Methods	Version Introduced	Supported	Comments
boolean acceptsURL(String)	1.0	Yes	
Connection connect(String, Properties)	1.0	Yes	
int getMajorVersion()	1.0	Yes	
int getMinorVersion()	1.0	Yes	
DriverPropertyInfo [] getPropertyInfo(String, Properties)	1.0	Yes	

ParameterMetaData

ParameterMetaData Methods	Version Introduced	Supported	Comments
String getParameterClassName(int)	3.0	Yes	The DB2 driver supports parameter metadata for stored procedures for DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
int getParameterCount()	3.0	Yes	
int getParameterMode(int)	3.0	Yes	The DB2 driver supports parameter metadata for stored procedures for DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
int getParameterType(int)	3.0	Yes	The DB2 driver supports parameter metadata for stored procedures for DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
String getParameterTypeName(int)	3.0	Yes	The DB2 driver supports parameter metadata for stored procedures for DB2 V8.x and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
int getPrecision(int)	3.0	Yes	The DB2 driver supports parameter metadata for stored procedures for DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
int getScale(int)	3.0	Yes	The DB2 driver supports parameter metadata for stored procedures for DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
int isNullable(int)	3.0	Yes	The DB2 driver supports parameter metadata for stored procedures for DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.

ParameterMetaData Methods	Version Introduced	Supported	Comments
boolean isSigned(int)	3.0	Yes	The DB2 driver supports parameter metadata for stored procedures for DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
boolean jdbcCompliant()	1.0	Yes	
<T> T unwrap(Class<T> iface)	4.0	Yes	

PooledConnection

PooledConnection Methods	Version Introduced	Supported	Comments
void addConnectionEventListener(listener)	2.0 Optional	Yes	
void addStatementEventListener(listener)	4.0	Yes	
void close()	2.0 Optional	Yes	
Connection getConnection()	2.0 Optional	Yes	A pooled connection object can have only one Connection object open (the one most recently created). The purpose of allowing the server (PoolManager implementation) to invoke this a second time is to give an application server a way to take a connection away from an application and give it to another user (a rare occurrence). The drivers do not support the "reclaiming" of connections and will throw an exception.
void removeConnectionEventListener(listener)	2.0 Optional	Yes	
void removeStatementEventListener(listener)	4.0	Yes	

PreparedStatement

PreparedStatement Methods	Version Introduced	Supported	Comments
void addBatch()	2.0 Core	Yes	

PreparedStatement Methods	Version Introduced	Supported	Comments
void clearParameters()	1.0	Yes	
boolean execute()	1.0	Yes	
ResultSet executeQuery()	1.0	Yes	
int executeUpdate()	1.0	Yes	
ResultSetMetaData getMetaData()	2.0 Core	Yes	
ParameterMetaData getParameterMetaData()	3.0	Yes	
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
void setArray(int, Array)	2.0 Core	Yes	Supported for the Oracle driver only. All other drivers throw an "unsupported method" exception.
void setAsciiStream(int, InputStream)	4.0	Yes	
void setAsciiStream(int, InputStream, int)	1.0	Yes	
void setAsciiStream(int, InputStream, long)	4.0	Yes	
void setBigDecimal(int, BigDecimal)	1.0	Yes	
void setBinaryStream(int, InputStream)	4.0	Yes	When used with Blobs, the DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
void setBinaryStream(int, InputStream, int)	1.0	Yes	When used with Blobs, the DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
void setBinaryStream(int, InputStream, long)	4.0	Yes	When used with Blobs, the DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
void setBlob(int, Blob)	2.0 Core	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.

PreparedStatement Methods	Version Introduced	Supported	Comments
void setBlob(int, InputStream)	4.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void setBlob(int, InputStream, long)	4.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void setBoolean(int, boolean)	1.0	Yes	
void setByte(int, byte)	1.0	Yes	
void setBytes(int, byte [])	1.0	Yes	When used with Blobs, the DB2 driver only supports with DB2 V8.x and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
void setCharacterStream(int, Reader)	4.0	Yes	
void setCharacterStream(int, Reader, int)	2.0 Core	Yes	
void setCharacterStream(int, Reader, long)	4.0	Yes	
void setClob(int, Clob)	2.0 Core	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void setClob(int, Reader)	4.0	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void setClob(int, Reader, long)	4.0	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void setDate(int, Date)	1.0	Yes	
void setDate(int, Date, Calendar)	2.0 Core	Yes	
void setDouble(int, double)	1.0	Yes	
void setFloat(int, float)	1.0	Yes	

PreparedStatement Methods	Version Introduced	Supported	Comments
void setInt(int, int)	1.0	Yes	
void setLong(int, long)	1.0	Yes	
void setNCharacterStream(int, Reader)	4.0	Yes	For the drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB, N methods are identical to their non-N counterparts.
void setNCharacterStream(int, Reader, long)	4.0	Yes	For the drivers for Salesforce, Oracle Service Cloud, and MongoDB,N methods are identical to their non-N counterparts.
void setNClob(int, NClob)	4.0	Yes	For the drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB, N methods are identical to their non-N counterparts.
void setNClob(int, Reader)	4.0	Yes	For the drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB, N methods are identical to their non-N counterparts.
void setNClob(int, Reader, long)	4.0	Yes	For the drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB, N methods are identical to their non-N counterparts.
void setNull(int, int)	1.0	Yes	
void setNull(int, int, String)	2.0 Core	Yes	
void setNString(int, String)	4.0	Yes	
void setObject(int, Object)	1.0	Yes	
void setObject(int, Object, int)	1.0	Yes	
void setObject(int, Object, int, int)	1.0	Yes	

PreparedStatement Methods	Version Introduced	Supported	Comments
void setQueryTimeout(int)	1.0	Yes	The DB2 driver supports setting a timeout value, in seconds, for a statement with DB2 V8.*x* and higher for Linux/UNIX/Windows and DB2 V8.1 and higher for z/OS. If the execution of the statement exceeds the timeout value, the statement is timed out by the database server, and the driver throws an exception indicating that the statement was timed out. The DB2 driver throws an "unsupported method" exception with other DB2 versions. The Informix driver throws an "unsupported method" exception. The drivers for Salesforce and Oracle Service Cloud ignore any value set using this method. Use the WSTimeout connection property to set a timeout value. The drivers for Apache Cassandra and MongoDB ignore any value set using this method. All other drivers support setting a timeout value, in seconds, for a statement. If the execution of the statement exceeds the timeout value, the statement is timed out by the database server, and the driver throws an exception indicating that the statement was timed out.
void setRef(int, Ref)	2.0 Core	No	All drivers throw "unsupported method" exception.
void setShort(int, short)	1.0	Yes	
void setSQLXML(int, SQLXML)	4.0	Yes	
void setString(int, String)	1.0	Yes	
void setTime(int, Time)	1.0	Yes	
void setTime(int, Time, Calendar)	2.0 Core	Yes	
void setTimestamp(int, Timestamp)	1.0	Yes	
void setTimestamp(int, Timestamp, Calendar)	2.0 Core	Yes	
void setUnicodeStream(int, InputStream, int)	1.0	No	This method was deprecated in JDBC 2.0. All drivers throw "unsupported method" exception.

PreparedStatement Methods	Version Introduced	Supported	Comments
<T> T unwrap(Class<T> iface)	4.0	Yes	
void setURL(int, URL)	3.0	No	All drivers throw "unsupported method" exception.

Ref

Ref MethodsRef interface	Version Introduced	Supported	Comments
(all)	2.0 Core	No	

ResultSet

ResultSet Methods	Version Introduced	Supported	Comments
boolean absolute(int)	2.0 Core	Yes	
void afterLast()	2.0 Core	Yes	
void beforeFirst()	2.0 Core	Yes	
void cancelRowUpdates()	2.0 Core	Yes	
void clearWarnings()	1.0	Yes	
void close()	1.0	Yes	
void deleteRow()	2.0 Core	Yes	
int findColumn(String)	1.0	Yes	
boolean first()	2.0 Core	Yes	
Array getArray(int)	2.0 Core	Yes	
Array getArray(String)	2.0 Core	No	All drivers throw "unsupported method" exception.
InputStream getAsciiStream(int)	1.0	Yes	
InputStream getAsciiStream(String)	1.0	Yes	
BigDecimal getBigDecimal(int)	2.0 Core	Yes	

ResultSet Methods	Version Introduced	Supported	Comments
BigDecimal getBigDecimal(int, int)	1.0	Yes	
BigDecimal getBigDecimal(String)	2.0 Core	Yes	
BigDecimal getBigDecimal(String, int)	1.0	Yes	
InputStream getBinaryStream(int)	1.0	Yes	The DB2 driver supports for all DB2 versions when retrieving BINARY, VARBINARY, and LONGVARBINARY data. The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i when retrieving Blob data.
InputStream getBinaryStream(String)	1.0	Yes	The DB2 driver supports for all DB2 versions when retrieving BINARY, VARBINARY, and LONGVARBINARY data. The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i when retrieving Blob data.
Blob getBlob(int)	2.0 Core	Yes	The DB2 driver only supports with DB2 V8.x and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
Blob getBlob(String)	2.0 Core	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
boolean getBoolean(int)	1.0	Yes	
boolean getBoolean(String)	1.0	Yes	
byte getByte(int)	1.0	Yes	
byte getByte(String)	1.0	Yes	

ResultSet Methods	Version Introduced	Supported	Comments
byte [] getBytes(int)	1.0	Yes	The DB2 driver supports for all DB2 versions when retrieving BINARY, VARBINARY, and LONGVARBINARY data. The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i when retrieving Blob data.
byte [] getBytes(String)	1.0	Yes	The DB2 driver supports for all DB2 versions when retrieving BINARY, VARBINARY, and LONGVARBINARY data. The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i when retrieving Blob data.
Reader getCharacterStream(int)	2.0 Core	Yes	
Reader getCharacterStream(String)	2.0 Core	Yes	
Clob getClob(int)	2.0 Core	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
Clob getClob(String)	2.0 Core	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
int getConcurrency()	2.0 Core	Yes	
String getCursorName()	1.0	No	All drivers throw "unsupported method" exception.
Date getDate(int)	1.0	Yes	
Date getDate(int, Calendar)	2.0 Core	Yes	
Date getDate(String)	1.0	Yes	
Date getDate(String, Calendar)	2.0 Core	Yes	
double getDouble(int)	1.0	Yes	
double getDouble(String)	1.0	Yes	
int getFetchDirection()	2.0 Core	Yes	
int getFetchSize()	2.0 Core	Yes	
float getFloat(int)	1.0	Yes	

ResultSet Methods	Version Introduced	Supported	Comments
float getFloat(String)	1.0	Yes	
int getHoldability()	4.0	Yes	
int getInt(int)	1.0	Yes	
int getInt(String)	1.0	Yes	
long getLong(int)	1.0	Yes	
long getLong(String)	1.0	Yes	
ResultSetMetaData getMetaData()	1.0	Yes	
Reader getNCharacterStream(int)	4.0	Yes	
Reader getNCharacterStream(String)	4.0	Yes	
NClob getNClob(int)	4.0	Yes	
NClob getNClob(String)	4.0	Yes	
String getNString(int)	4.0	Yes	
String getNString(String)	4.0	Yes	
Object getObject(int)	1.0	Yes	The DB2 driver returns a Long object when called on Bigint columns.
Object getObject(int, Map)	2.0 Core	Yes	The Oracle and Sybase drivers support the Map argument. For all other drivers, the Map argument is ignored.
Object getObject(String)	1.0	Yes	
Object getObject(String, Map)	2.0 Core	Yes	The Oracle and Sybase drivers support the Map argument. For all other drivers, the Map argument is ignored.
Ref getRef(int)	2.0 Core	No	All drivers throw "unsupported method" exception.
Ref getRef(String)	2.0 Core	No	All drivers throw "unsupported method" exception.
int getRow()	2.0 Core	Yes	
short getShort(int)	1.0	Yes	
short getShort(String)	1.0	Yes	

ResultSet Methods	Version Introduced	Supported	Comments
SQLXML getSQLXML(int)	4.0	Yes	
SQLXML getSQLXML(String)	4.0	Yes	
Statement getStatement()	2.0 Core	Yes	
String getString(int)	1.0	Yes	
String getString(String)	1.0	Yes	
Time getTime(int)	1.0	Yes	
Time getTime(int, Calendar)	2.0 Core	Yes	
Time getTime(String)	1.0	Yes	
Time getTime(String, Calendar)	2.0 Core	Yes	
Timestamp getTimestamp(int)	1.0	Yes	
Timestamp getTimestamp(int, Calendar)	2.0 Core	Yes	
Timestamp getTimestamp(String)	1.0	Yes	
Timestamp getTimestamp(String, Calendar)	2.0 Core	Yes	
int getType()	2.0 Core	Yes	
InputStream getUnicodeStream(int)	1.0	No	This method was deprecated in JDBC 2.0. All drivers throw "unsupported method" exception.
InputStream getUnicodeStream(String)	1.0	No	This method was deprecated in JDBC 2.0. All drivers throw "unsupported method" exception.
URL getURL(int)	3.0	No	All drivers throw "unsupported method" exception.
URL getURL(String)	3.0	No	All drivers throw "unsupported method" exception.
SQLWarning getWarnings()	1.0	Yes	
void insertRow()	2.0 Core	Yes	
boolean isAfterLast()	2.0 Core	Yes	
boolean isBeforeFirst()	2.0 Core	Yes	

ResultSet Methods	Version Introduced	Supported	Comments
boolean isClosed()	4.0	Yes	
boolean isFirst()	2.0 Core	Yes	
boolean isLast()	2.0 Core	Yes	
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
boolean last()	2.0 Core	Yes	
void moveToCurrentRow()	2.0 Core	Yes	
void moveToInsertRow()	2.0 Core	Yes	
boolean next()	1.0	Yes	
boolean previous()	2.0 Core	Yes	
void refreshRow()	2.0 Core	Yes	
boolean relative(int)	2.0 Core	Yes	
boolean rowDeleted()	2.0 Core	Yes	
boolean rowInserted()	2.0 Core	Yes	
boolean rowUpdated()	2.0 Core	Yes	
void setFetchDirection(int)	2.0 Core	Yes	
void setFetchSize(int)	2.0 Core	Yes	
<T> T unwrap(Class<T> iface)	4.0	Yes	
void updateArray(int, Array)	3.0	No	All drivers throw "unsupported method" exception.
void updateArray(String, Array)	3.0	No	All drivers throw "unsupported method" exception.
void updateAsciiStream(int, InputStream, int)	2.0 Core	Yes	
void updateAsciiStream(int, InputStream, long)	4.0	Yes	
void updateAsciiStream(String, InputStream)	4.0	Yes	
void updateAsciiStream(String, InputStream, int)	2.0 Core	Yes	

ResultSet Methods	Version Introduced	Supported	Comments
void updateAsciiStream(String, InputStream, long)	4.0	Yes	
void updateBigDecimal(int, BigDecimal)	2.0 Core	Yes	
void updateBigDecimal(String, BigDecimal)	2.0 Core	Yes	
void updateBinaryStream(int, InputStream)	4.0	Yes	
void updateBinaryStream(int, InputStream, int)	2.0 Core	Yes	
void updateBinaryStream(int, InputStream, long)	4.0	Yes	
void updateBinaryStream(String, InputStream)	4.0	Yes	
void updateBinaryStream(String, InputStream, int)	2.0 Core	Yes	
void updateBinaryStream(String, InputStream, long)	4.0	Yes	
void updateBlob(int, Blob)	3.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateBlob(int, InputStream)	4.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateBlob(int, InputStream, long)	4.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.

ResultSet Methods	Version Introduced	Supported	Comments
void updateBlob(String, Blob)	3.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateBlob(String, InputStream)	4.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateBlob(String, InputStream, long)	4.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateBoolean(int, boolean)	2.0 Core	Yes	
void updateBoolean(String, boolean)	2.0 Core	Yes	
void updateByte(int, byte)	2.0 Core	Yes	
void updateByte(String, byte)	2.0 Core	Yes	
void updateBytes(int, byte [])	2.0 Core	Yes	
void updateBytes(String, byte [])	2.0 Core	Yes	
void updateCharacterStream(int, Reader)	4.0	Yes	
void updateCharacterStream(int, Reader, int)	2.0 Core	Yes	
void updateCharacterStream(int, Reader, long)	4.0	Yes	
void updateCharacterStream(String, Reader)	4.0	Yes	
void updateCharacterStream(String, Reader, int)	2.0 Core	Yes	
void updateCharacterStream(String, Reader, long)	4.0	Yes	

ResultSet Methods	Version Introduced	Supported	Comments
void updateClob(int, Clob)	3.0	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateClob(int, Reader)	4.0	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateClob(int, Reader, long)	4.0	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateClob(String, Clob)	3.0	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateClob(String, Reader)	4.0	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateClob(String, Reader, long)	4.0	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateDate(int, Date)	2.0 Core	Yes	
void updateDate(String, Date)	2.0 Core	Yes	
void updateDouble(int, double)	2.0 Core	Yes	
void updateDouble(String, double)	2.0 Core	Yes	
void updateFloat(int, float)	2.0 Core	Yes	
void updateFloat(String, float)	2.0 Core	Yes	
void updateInt(int, int)	2.0 Core	Yes	
void updateInt(String, int)	2.0 Core	Yes	
void updateLong(int, long)	2.0 Core	Yes	
void updateLong(String, long)	2.0 Core	Yes	
void updateNCharacterStream(int, Reader)	4.0	Yes	For the drivers for Salesforce, Oracle Service Cloud, and MongoDB, N methods are identical to their non-N counterparts.
void updateNCharacterStream(int, Reader, long)	4.0	Yes	For the drivers for Salesforce, Oracle Service Cloud, and MongoDB, N methods are identical to their non-N counterparts.

ResultSet Methods	Version Introduced	Supported	Comments
void updateNCharacterStream(String, Reader)	4.0	Yes	For the drivers for Salesforce, Oracle Service Cloud, and MongoDB, N methods are identical to their non-N counterparts.
void updateNCharacterStream(String, Reader, long)	4.0	Yes	For the drivers for Salesforce, Oracle Service Cloud, and MongoDB, N methods are identical to their non-N counterparts.
void updateNClob(int, NClob)	4.0	Yes	For the drivers for Salesforce, Oracle Service Cloud, and MongoDB, N methods are identical to their non-N counterparts.
void updateNClob(int, Reader)	4.0	Yes	For the drivers for Salesforce, Oracle Service Cloud, and MongoDB, N methods are identical to their non-N counterparts.
void updateNClob(int, Reader, long)	4.0	Yes	For the drivers for Salesforce, Oracle Service Cloud, and MongoDB, N methods are identical to their non-N counterparts.
void updateNClob(String, NClob)	4.0	Yes	For the drivers for Salesforce, Oracle Service Cloud, and MongoDB, N methods are identical to their non-N counterparts.
void updateNClob(String, Reader)	4.0	Yes	For the drivers for Salesforce, Oracle Service Cloud, and MongoDB, N methods are identical to their non-N counterparts.
void updateNClob(String, Reader, long)	4.0	Yes	For the drivers for Salesforce, Oracle Service Cloud, and MongoDB, N methods are identical to their non-N counterparts.
void updateNString(int, String)	4.0	Yes	For the drivers for Salesforce, Oracle Service Cloud, and MongoDB, N methods are identical to their non-N counterparts.
void updateNString(String, String)	4.0	Yes	For the drivers for Salesforce, Oracle Service Cloud, and MongoDB, N methods are identical to their non-N counterparts.
void updateNull(int)	2.0 Core	Yes	
void updateNull(String)	2.0 Core	Yes	
void updateObject(int, Object)	2.0 Core	Yes	
void updateObject(int, Object, int)	2.0 Core	Yes	
void updateObject(String, Object)	2.0 Core	Yes	
void updateObject(String, Object, int)	2.0 Core	Yes	

ResultSet Methods	Version Introduced	Supported	Comments
void updateRef(int, Ref)	3.0	No	All drivers throw "unsupported method" exception.
void updateRef(String, Ref)	3.0	No	All drivers throw "unsupported method" exception.
void updateRow()	2.0 Core	Yes	
void updateShort(int, short)	2.0 Core	Yes	
void updateShort(String, short)	2.0 Core	Yes	
void updateSQLXML(int, SQLXML)	4.0	Yes	
void updateSQLXML(String, SQLXML)	4.0	Yes	
void updateString(int, String)	2.0 Core	Yes	
void updateString(String, String)	2.0 Core	Yes	
void updateTime(int, Time)	2.0 Core	Yes	
void updateTime(String, Time)	2.0 Core	Yes	
void updateTimestamp(int, Timestamp)	2.0 Core	Yes	
void updateTimestamp(String, Timestamp)	2.0 Core	Yes	
boolean wasNull()	1.0	Yes	

ResultSetMetaData

ResultSetMetaData Methods	Version Introduced	Supported	Comments
String getCatalogName(int)	1.0	Yes	
String getColumnClassName(int)	2.0 Core	Yes	
int getColumnCount()	1.0	Yes	
int getColumnDisplaySize(int)	1.0	Yes	
String getColumnLabel(int)	1.0	Yes	
String getColumnName(int)	1.0	Yes	
int getColumnType(int)	1.0	Yes	

ResultSetMetaData Methods	Version Introduced	Supported	Comments
String getColumnTypeName(int)	1.0	Yes	
int getPrecision(int)	1.0	Yes	
int getScale(int)	1.0	Yes	
String getSchemaName(int)	1.0	Yes	
String getTableName(int)	1.0	Yes	
boolean isAutoIncrement(int)	1.0	Yes	
boolean isCaseSensitive(int)	1.0	Yes	
boolean isCurrency(int)	1.0	Yes	
boolean isDefinitelyWritable(int)	1.0	Yes	
int isNullable(int)	1.0	Yes	
boolean isReadOnly(int)	1.0	Yes	
boolean isSearchable(int)	1.0	Yes	
boolean isSigned(int)	1.0	Yes	
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
boolean isWritable(int)	1.0	Yes	
<T> T unwrap(Class<T> iface)	4.0	Yes	

RowSet

RowSet Methods	Version Introduced	Supported	Comments
(all)	2.0 Optional	No	

SavePoint

SavePoint Methods	Version Introduced	Supported	Comments
(all)	3.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS ((all versions), and DB2 for i.

Statement

Statement Methods	Version Introduced	Supported	Comments
void addBatch(String)	2.0 Core	Yes	All drivers throw "invalid method call" exception for PreparedStatement and CallableStatement.
void cancel()	1.0	Yes	The DB2 driver cancels the execution of the statement with DB2 V8.*x* and higher for Linux/UNIX/Windows and DB2 V8.1 and higher for z/OS. If the statement is canceled by the database server, the driver throws an exception indicating that it was canceled. The DB2 driver throws an "unsupported method" exception with other DB2 versions. The drivers for Apache Cassandra, Apache Hive, Apache Spark SQL, Impala, Informix, Progess OpenEdge, Oracle Service Cloud, Salesforce and MongoDB throw an "unsupported method" exception. The Greenplum, Oracle, PostgreSQL, SQL Server, Sybase, and Amazon Redshift drivers cancel the execution of the statement. If the statement is canceled by the database server, the driver throws an exception indicating that it was canceled.
void clearBatch()	2.0 Core	Yes	
void clearWarnings()	1.0	Yes	
void close()	1.0	Yes	
boolean execute(String)	1.0	Yes	All drivers throw "invalid method call" exception for PreparedStatement and CallableStatement.
boolean execute(String, int)	3.0	Yes	

Statement Methods	Version Introduced	Supported	Comments
boolean execute(String, int [])	3.0	Yes	Supported for the Oracle and SQL Server drivers. All other drivers throw "unsupported method" exception.
boolean execute(String, String [])	3.0	Yes	Supported for the Oracle and SQL Server drivers. All other drivers throw "unsupported method" exception.
int [] executeBatch()	2.0 Core	Yes	
ResultSet executeQuery(String)	1.0	Yes	All drivers throw "invalid method call" exception for PreparedStatement and CallableStatement.
int executeUpdate(String)	1.0	Yes	All drivers throw "invalid method call" exception for PreparedStatement and CallableStatement.
int executeUpdate(String, int)	3.0	Yes	
int executeUpdate(String, int [])	3.0	Yes	Supported for the Oracle and SQL Server drivers. All other drivers throw "unsupported method" exception.
int executeUpdate(String, String [])	3.0	Yes	Supported for the Oracle and SQL Server drivers. All other drivers throw "unsupported method" exception.
Connection getConnection()	2.0 Core	Yes	
int getFetchDirection()	2.0 Core	Yes	
int getFetchSize()	2.0 Core	Yes	
ResultSet getGeneratedKeys()	3.0	Yes	The DB2, SQL Server, and Sybase drivers return the last value inserted into an identity column. If an identity column does not exist in the table, the drivers return an empty result set. The Informix driver returns the last value inserted into a Serial or Serial8 column. If a Serial or Serial8 column does not exist in the table, the driver returns an empty result set.

Statement Methods	Version Introduced	Supported	Comments
			The Oracle driver returns the ROWID of the last row that was inserted. The drivers for Apache Cassandra, Salesforce, Oracle Service Cloud, and MongoDB return the ID of the last row that was inserted. Auto-generated keys are not supported in any of the other drivers.
int getMaxFieldSize()	1.0	Yes	
int getMaxRows()	1.0	Yes	
boolean getMoreResults()	1.0	Yes	
boolean getMoreResults(int)	3.0	Yes	
int getQueryTimeout()	1.0	Yes	The DB2 driver returns the timeout value, in seconds, set for the statement with DB2 V8.*x* and higher for Linux/UNIX/Windows and DB2 V8.1 and higher for z/OS. The DB2 driver returns `0` with other DB2 versions. The Informix and Progress OpenEdge drivers return `0`. The drivers for Apache Cassandra, Apache Hive, Apache Spark SQL, Impala, Greenplum, Oracle, PostgreSQL, SQL Server, Sybase, and Amazon Redshift return the timeout value, in seconds, set for the statement. The drivers for Salesforce and Oracle Service Cloud return an "unsupported method" exception.
ResultSet getResultSet()	1.0	Yes	
int getResultSetConcurrency()	2.0 Core	Yes	
int getResultSetHoldability()	3.0	Yes	
int getResultSetType()	2.0 Core	Yes	
int getUpdateCount()	1.0	Yes	
SQLWarning getWarnings()	1.0	Yes	
boolean isClosed()	4.0	Yes	

Statement Methods	Version Introduced	Supported	Comments
boolean isPoolable()	4.0	Yes	
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
void setCursorName(String)	1.0	No	Throws "unsupported method" exception.
void setEscapeProcessing(boolean)	1.0	Yes	Ignored.
void setFetchDirection(int)	2.0 Core	Yes	
void setFetchSize(int)	2.0 Core	Yes	
void setMaxFieldSize(int)	1.0	Yes	
void setMaxRows(int)	1.0	Yes	
void setPoolable(boolean)	4.0	Yes	
void setQueryTimeout(int)	1.0	Yes	The DB2 driver supports setting a timeout value, in seconds, for a statement with DB2 V8.*x* and higher for Linux/UNIX/Windows and DB2 V8.1 and higher for z/OS. If the execution of the statement exceeds the timeout value, the statement is timed out by the database server, and the driver throws an exception indicating that the statement was timed out. The DB2 driver throws an "unsupported method" exception with other DB2 versions. The Informix driver throws an "unsupported method" exception. The drivers for Greenplum, Apache Hive, Apache Spark SQL, Impala, Oracle, PostgreSQL, Progress OpenEdge, SQL Server, Sybase, and Amazon Redshift support setting a timeout value, in seconds, for a statement. If the execution of the statement exceeds the timeout value, the statement is timed out by the database server, and the driver throws an exception indicating that the statement was timed out. The drivers for Salesforce and Oracle Service Cloud ignore any value set using this method. Use the WSTimeout connection property to set a timeout.

Statement Methods	Version Introduced	Supported	Comments
			The drivers for Apache Cassandra and MongoDB driver ignore any value set using this method.
<T> T unwrap(Class<T> iface)	4.0	Yes	

StatementEventListener

StatementEventListener Methods	Version Introduced	Supported	Comments
void statementClosed(event)	4.0	Yes	
void statementErrorOccurred(event)	4.0	Yes	

Struct

Struct Methods	Version Introduced	Supported	Comments
(all)	2.0	Yes	Supported for the Oracle driver only. All other drivers throw "unsupported method" exception.

XAConnection

XAConnection Methods	Version Introduced	Supported	Comments
(all)	2.0 Optional	Yes	Supported for all drivers except DB2 V8.1 for z/OS, Greenplum, Apache Hive, Apache Spark SQL, Impala, PostgreSQL, and Amazon Redshift.

XADataSource

XADataSource Methods	Version Introduced	Supported	Comments
(all)	2.0 Optional	Yes	Supported for all drivers except DB2 V8.1 for z/OS, Greenplum, Apache Hive, Apache Spark SQL, Impala, PostgreSQL, and Amazon Redshift.

XAResource

XAResource Methods	Version Introduced	Supported	Comments
(all)	2.0 Optional	Yes	Supported for all drivers except DB2 V8.1 for z/OS, Greenplum, Apache Hive, Apache Spark SQL, Impala, PostgreSQL, and Amazon Redshift.

2

JDBC Extensions

This chapter describes the JDBC extensions provided by the com.ddtek.jdbc.extensions package. The interfaces in this package are:

Interface/Class	Description	For more information
DatabaseMetadata	The method in this interface is used with the Salesforce driver to extend the standard JDBC metadata results returned by the DatabaseMetaData.getColumns() method to include an additional column.	See DatabaseMetaData Interface (Salesforce Driver) on page 73.
DDBulkLoad	Methods that allow your application to perform bulk load operations.	See DDBulkLoad Interface on page 73. Refer to "Using DataDirect Bulk Load" in the *DataDirect Connect Series for JDBC User's Guide*.

<table>
<tr><th>Interface/Class</th><th>Description</th><th>For more information</th></tr>
<tr><td>ExtConnection</td><td>Methods that allow you to perform the following actions:
Store and return client information.
Switch the user associated with a connection to another user to minimize the number of connections that are required in a connection pool.
Access the DataDirect Statement Pool Monitor from a connection.</td><td>See ExtConnection Interface on page 80.
Refer to "Using Client Information" in the DataDirect Connect Series for JDBC User's Guide.
Refer to "Using Reauthentication" in the DataDirect Connect Series for JDBC User's Guide.
See Statement Pool Monitor on page 347.</td></tr>
<tr><td>ExtLogControl</td><td>Methods that allow you to determine if DataDirect Spy logging is enabled and turning on and off DataDirect Spy logging if enabled.</td><td>See ExtLogControl Class on page 85.
See Tracking JDBC Calls with DataDirect Spy on page 325.</td></tr>
</table>

For details, see the following topics:

- Using JDBC Wrapper Methods to Access JDBC Extensions
- DatabaseMetaData Interface (Salesforce Driver)
- DDBulkLoad Interface
- ExtConnection Interface
- ExtDatabaseMetaData Interface
- ExtLogControl Class

Using JDBC Wrapper Methods to Access JDBC Extensions

The Wrapper methods allow an application to access vendor-specific classes. The following example shows how to access the DataDirect-specific ExtConnection class using the Wrapper methods:

```
ExtStatementPoolMonitor monitor = null;
Class<ExtConnection> cls = ExtConnection.class;
if (con.isWrapperFor(cls)) {
   ExtConnection extCon = con.unwrap(cls);
   extCon.setClientUser("Joe Smith");
   monitor = extCon.getStatementPoolMonitor();
}
...
if(monitor != null) {
   long hits = monitor.getHitCount();
   long misses = monitor.getMissCount();
}
...
```

DatabaseMetaData Interface (Salesforce Driver)

The Salesforce driver extends the standard JDBC metadata results returned by the DatabaseMetaData.getColumns() method to include an additional column.

Column	Data Type	Description
IS_EXTERNAL_ID	VARCHAR (3), NOT NULL	Provides an indication of whether the column can be used as an External ID. External ID columns can be used as the lookup column for insert and upsert operations and foreign-key relationship values. Valid values are: • `YES`: The column can be used as an external ID. • `NO`: The column cannot be used as an external ID. The standard catalog table SYSTEM_COLUMNS is also extended to include the IS_EXTERNAL_ID column.

DDBulkLoad Interface

Interface Methods	Description
void clearWarnings()	Clears all warnings that were generated by this DDBulkLoad object.
void close()	Releases a DDBulkLoad object's resources immediately instead of waiting for the connection to close.
long export(File)	Exports all rows from the table into the specified CSV file specified by a file reference. The table is specified using the setTableName() method. If the CSV file does not already exist, the driver creates it when the export() method is executed. In addition, the driver creates a bulk load configuration file matching the CSV file. Refer to "Exporting Data to a CSV File" in the *DataDirect Connect Series for JDBC User's Guide* for more information. This method also returns the number of rows that were successfully exported from the table.
long export(ResultSet, File)	Exports all rows from the specified ResultSet into the CSV file specified by a file reference. If the CSV file does not already exist, the driver creates it when the export() method is executed. In addition, the driver creates a bulk load configuration file matching the CSV file. Refer to "Exporting Data to a CSV File" in the *DataDirect Connect Series for JDBC User's Guide* for more information. This method also returns the number of rows that were successfully exported from the ResultSet object.

Interface Methods	Description
long export(String)	Exports all rows from the table into the CSV file specified by name. The table is specified using the setTableName() method. If the CSV file does not already exist, the driver creates it when the export() method is executed. In addition, the driver creates a bulk load configuration file matching the CSV file. Refer to "Exporting Data to a CSV File" in the *DataDirect Connect Series for JDBC User's Guide* for more information. This method also returns the number of rows that were successfully exported from the table.
long getBatchSize()	Returns the number of rows that the driver sends at a time when bulk loading data.
long getBinaryThreshold()	Returns the maximum size (in bytes) of binary data that can be exported to the CSV file. Once this size is reached, binary data is written to one or multiple external overflow files. Refer to "External Overflow Files" in the *DataDirect Connect Series for JDBC User's Guide* for more information.
long getCharacterThreshold()	Returns the maximum size (in bytes) of character data that can be exported to the CSV file. Once this size is reached, character data is written to one or multiple external overflow files. Refer to "External Overflow Files" in the *DataDirect Connect Series for JDBC User's Guide* for more information.
String getCodePage()	Returns the code page that the driver uses for the CSV file. Refer to "Character Set Conversions" in the *DataDirect Connect Series for JDBC User's Guide* for more information.
String getConfigFile()	Returns the name of the bulk load configuration file. Refer to "Bulk Load Configuration File" in the *DataDirect Connect Series for JDBC User's Guide* for more information.
String getDiscardFile()	Returns the name of the discard file. The discard file contains rows that were unable to be loaded as the result of a bulk load operation. Refer to "Discard File" in the *DataDirect Connect Series for JDBC User's Guide* for more information.
long getErrorTolerance()	Returns the number of errors that can occur before this DDBulkLoad object ends the bulk load operation.
String getLogFile()	Returns the name of the log file. The log file records information about each bulk load operation. Refer to "Logging" in the *DataDirect Connect Series for JDBC User's Guide* for more information.
long getNumRows()	Returns the maximum number of rows from the CSV file or ResultSet object the driver will load when the load() method is executed.
Properties getProperties()	Returns the properties specified for a DDBulkLoad object. Properties are specified using the setProperties() method.
long getReadBufferSize()	Returns the size (in KB) of the buffer that is used to read the CSV file.

Interface Methods	Description
long getStartPosition()	Returns the position (number of the row) in a CSV file or ResultSet object from which the driver starts loading. The position is specified using the setStartPosition() method.
void getTableName()	Returns the name of the table to which the data is loaded into or exported from. Refer to "Loading Data From a ResultSet Object," "Loading Data From a CSV File," and "Exporting Data to a CSV File" in the *DataDirect Connect Series for JDBC User's Guide* for more information.
long getTimeout()	Returns the number of seconds the bulk load operation requires to complete before it times out. The timeout is specified using the setTimeout() method.
SQLWarning getWarnings()	Returns any warnings generated by this DDBulkLoad object.
long getWarningTolerance()	Returns the maximum number of warnings that can occur. Once the maximum number is reached, the bulk load operation ends.
long load(File)	Loads data from the CSV file specified by a file reference into a table. The table is specified using the setTableName() method. This method also returns the number of rows that have been successfully loaded. If logging is enabled using the setLogFile() method, information about the bulk load operation is recorded in the log file. If a discard file is created using the setDiscardFile() method, rows that were unable to be loaded are recorded in the discard file. Refer to "Logging" and "Discard File" in the *DataDirect Connect Series for JDBC User's Guide* for more information. Before the bulk load operation is performed, your application can verify that the data in the CSV file is compatible with the structure of the target table using the validateTableFromFile() method.
long load(String)	Loads data from the CSV file specified by file name into a table. The table is specified using the setTableName() method. This method also returns the number of rows that have been successfully loaded. If logging is enabled using the setLogFile() method, information about the bulk load operation is recorded in the log file. If a discard file is created using the setDiscardFile() method, rows that were unable to be loaded are recorded in the discard file. Refer to "Logging" and "Discard File" in the *DataDirect Connect Series for JDBC User's Guide* for more information. Before the bulk load operation is performed, your application can verify that the data in the CSV file is compatible with the structure of the target table using the validateTableFromFile() method.

Interface Methods	Description
long load(ResultSet)	Loads data from a ResultSet object into the table specified using the setTableName() method. This method also returns the number of rows that have been successfully loaded. If logging is enabled using the setLogFile() method, information about the bulk load operation is recorded in the log file. Refer to "Logging" in the *DataDirect Connect Series for JDBC User's Guide* for more information. The structure of the table that produced the ResultSet object must match the structure of the target table. If not, the driver throws an exception.
void setBatchSize(long)	Specifies the number of rows that the driver sends at a time when bulk loading data. Performance can be improved by increasing the number of rows the driver loads at a time because fewer network round trips are required. Be aware that increasing the number of rows that are loaded also causes the driver to consume more memory on the client. If unspecified, the driver uses a value of `2048`.

void setBinaryThreshold(long)	Specifies the maximum size (in bytes) of binary data to be exported to the CSV file. Any column with data over this threshold is exported into individual external overflow files and a marker of the format `{DD LOBFILE "`*filename*`"}` is placed in the CSV file to signify that the data for this column is located in an external file. The format for overflow file names is:
	csv_filename_xxxxxx.lob
	where:
	csv_filename
	is the name of the CSV file.
	xxxxxx
	is a 6-digit number that increments the overflow file.
	For example, if multiple overflow files are created for a CSV file named CSV1, the file names would look like this:
	`CSV1.000001.lob`
	`CSV1.000002.lob`
	`CSV1.000003.lob`
	`...`
	If set to `-1`, the driver does not overflow binary data to external files. If unspecified, the driver uses a value of `4096`.
	Refer to "External Overflow Files" in the *DataDirect Connect Series for JDBC User's Guide* for more information.

Interface Methods	Description

void setCharacterThreshold(long)	Specifies the maximum size (in bytes) of character data to be exported to the CSV file. Any column with data over this threshold is exported into individual external overflow files and a marker of the format `{DD LOBFILE "`*filename*`"}` is placed in the CSV file to signify that the data for this column is located in an external file. The format for overflow file names is:
	csv_filename_xxxxxx.lob
	where:
	csv_filename
	is the name of the CSV file.
	xxxxxx
	is a 6-digit number that increments the overflow file.
	For example, if multiple overflow files are created for a CSV file named CSV1, the file names would look like this:
	`CSV1.000001.lob`
	`CSV1.000002.lob`
	`CSV1.000003.lob`
	`...`
	If set to `-1`, the driver does not overflow character data to external files.If unspecified, the driver uses a value of `4096`.
	Refer to "External Overflow Files" in the *DataDirect Connect Series for JDBC User's Guide* for more information.

void setCodePage(String)	Specifies the code page the driver uses for the CSV file. Refer to "Character Set Conversions" in the *DataDirect Connect Series for JDBC User's Guide* for more information.
void setConfigFile(String)	Specifies the fully qualified directory and file name of the bulk load configuration file. If the Column Info section in the bulk load configuration file is specified, the driver uses it to map the columns in the CSV file to the columns in the target table when performing a bulk load operation. If unspecified, the name of the bulk load configuration file is assumed to be *csv_filename*.xml, where *csv_filename* is the file name of the CSV file. If set to an empty string, the driver does not try to use the bulk load configuration file and reads all data from the CSV file as character data. Refer to "Bulk Load Configuration File" in the *DataDirect Connect Series for JDBC User's Guide* for more information.

<table>
<tr><th>Interface Methods</th><th>Description</th></tr>
<tr><td>void setDiscardFile(String)</td><td>Specifies the fully qualified directory and file name of the discard file. The discard file contains rows that were unable to be loaded from a CSV file as the result of a bulk load operation. After fixing the reported issues in the discard file, the bulk load can be reissued, using the discard file as the CSV file. If unspecified, a discard file is not created. Refer to "Discard File" in the DataDirect Connect Series for JDBC User’s Guide for more information.</td></tr>
<tr><td>void setErrorTolerance(long)</td><td>Specifies the maximum number of errors that can occur. Once the maximum number is reached, the bulk load operation ends. Errors are written to the log file. If set to <code>0</code>, no errors are tolerated; the bulk load operation fails if any error is encountered. Any rows that were processed before the error occurred are loaded. If unspecified or set to <code>-1</code>, an infinite number of errors are tolerated.</td></tr>
<tr><td>void setLogFile(String)</td><td>Specifies the fully qualified directory and file name of the log file. The log file records information about each bulk load operation.If unspecified, a log file is not created. Refer to "Logging" in the DataDirect Connect Series for JDBC User’s Guide for more information.</td></tr>
<tr><td>void setNumRows()</td><td>Specifies the maximum number of rows from the CSV file or ResultSet object the driver will load.</td></tr>
<tr><td>void setProperties(Properties)</td><td>Specifies one or more of the following properties for a DDBulkLoad object:
<pre>
tableName   numRows
codePage   binaryThreshold
timeout   characterThreshold
logFile   errorTolerance
discardFile   warningTolerance
configFile   readBufferSize
startPosition   batchSize
operation
</pre>
Except for the operation property, these properties also can be set using the corresponding setxxx() methods, which provide a description of the values that can be set.
The operation property defines which type of bulk operation will be performed when a load method is called. The operation property accepts the following values: <code>insert</code>, <code>update</code>, <code>delete</code>, or <code>upsert</code>. The default value is <code>insert</code>. Refer to "Specifying the Bulk Load Operation" in the DataDirect Connect Series for JDBC User’s Guide for more information. </td></tr>
<tr><td>void setReadBufferSize(long)</td><td>Specifies the size (in KB) of the buffer that is used to read the CSV file. If unspecified, the driver uses a value of <code>2048</code>.</td></tr>
<tr><td>void setStartPosition()</td><td>Specifies the position (number of the row) in a CSV file or ResultSet object from which the bulk load operation starts. For example, if a value of <code>10</code> is specified, the first 9 rows of the CSV file are skipped and the first row loaded is row 10.</td></tr>
</table>

Interface Methods	Description

void setTableName(*tablename* ([*destinationColumnList*]))	When loading data into a table, specifies the name of the table into which the data is loaded (*tablename*).
	Optionally, for the Salesforce driver, you can specify the column names that identify which columns to update in the table (*destinationColumnList*). Specifying column names is useful when loading data from a CSV file into a table. The column names used in the column list must be the names reported by the driver for the columns in the table. For example, if you are loading data into the Salesforce system column NAME, the column list must identify the column as SYS_NAME.
	If *destinationColumnList* is not specified, a one-to-one mapping is performed between the columns in the CSV file and the columns in the table.
	destinationColumnList has the following format:
	`(destColumnName [,destColumnName]...)`
	where:
	`destColumnName`
	is the name of the column in the table to be updated.
	The number of specified columns must match the number of columns in the CSV file. For example, the following call tells the driver to update the Name, Address, City, State, PostalCode, Phone, and Website columns:
	`bulkload.setTableName("account(Name, Address, City,State, PostalCode, Phone, Website)")`
	When exporting data from a table, specifies the name of the table from which the data is exported. If the specified table does not exist, the driver throws an exception. Refer to "Loading Data From a ResultSet Object," "Loading Data From a CSV File," and "Exporting Data to a CSV File" in the *DataDirect Connect Series for JDBC User's Guide* for more information.

void setTimeout(long)	Sets the maximum number of seconds that can elapse for this bulk load operation to complete. Once this number is reached, the bulk load operation times out.

Interface Methods	Description
void setWarningTolerance(long)	Specifies the maximum number of warnings that can occur. Once the maximum is reached, the bulk load operation ends. Warnings are written to the log file. If set to `0`, no warnings are tolerated; the bulk load operation fails if any warning is encountered. If unspecified or set to `-1`, an infinite number of warnings are tolerated.
Properties validateTableFromFile()	Verifies the metadata in the bulk load configuration file against the structure of the table to which the data is loaded. This method is used to ensure that the data in a CSV file is compatible with the structure of the target table before the actual bulk load operation is performed. The driver performs checks to detect mismatches of the following types: Data types Column sizes Code pages Column info This method returns a Properties object with an entry for each of these checks: • If no mismatches are found, the Properties object does not contain any messages. • If minor mismatches are found, the Properties object lists the problems. • If problems are detected that would prevent a successful bulk load operation, for example, if the target table does not exist, the driver throws an exception. Refer to "Verifying the Bulk Load Configuration File for Database Connections" in the *DataDirect Connect Series for JDBC User's Guide* for more information.

ExtConnection Interface

The methods of this interface are supported for all drivers.

ExtConnection Interface Methods	Description
void abortConnection()	Closes the current connection and marks the connection as closed. This method does not attempt to obtain any locks when closing the connection. If subsequent operations are performed on the connection, the driver throws an exception.
Connection createArray(String, Object[])	Supported by the Oracle driver only for use with Oracle VARRAY and TABLE data types. Creates an array object.

ExtConnection Interface Methods	Description
String getClientAccountingInfo()	Returns the accounting client information on the connection or an empty string if the accounting client information value or the connection has not been set. If getting accounting client information is supported by the database and this operation fails, the driver throws an exception.
String getClientApplicationName()	Returns the name of the client application on the connection or an empty string if the client name value for the connection has not been set. If getting client name information is supported by the database and this operation fails, the driver throws an exception.
String getClientHostname()	Returns the name of the host used by the client application on the connection or an empty string if the client hostname value in the database has not been set. If getting host name information is supported by the database and this operation fails, the driver throws an exception.
String getClientUser()	Returns the user ID of the client on the connection or an empty string if the client user ID value for the connection has not been set. The user ID may be different from the user ID establishing the connection. If getting user ID application information is supported by the database and this operation fails, the driver throws an exception.
String getCurrentUser()	Returns the current user of the connection. If reauthentication was performed on the connection, the current user may be different than the user that created the connection. For the DB2 and Oracle drivers, the current user is the same as the user reported by DatabaseMetaData.getUserName(). For the SQL Server driver, the current user is the login user name. DatabaseMetaData.getUserName() reports the user name the login user name is mapped to in the database. Refer to "Using Reauthentication" in the *DataDirect Connect Series for JDBC User's Guide* for more information.
int getNetworkTimeout()	Supported by the SQL Server driver to return the network timeout. The network timeout is the maximum time (in milliseconds) that a connection, or objects created by a connection, will wait for the database to reply to an application request. A value of 0 means that no network timeout exists. See void setNetworkTimeout(int) for details about setting a network timeout.
ExtStatementPoolMonitor getStatementPoolMonitor()	Returns an ExtStatementPoolMonitor object for the statement pool associated with the connection. If the connection does not have a statement pool, this method returns null. See Using DataDirect-Specific Methods to Access the Statement Pool Monitor on page 348 for more information.

ExtConnection Interface Methods	Description
void resetUser(String)	Specifies a non-null string that resets the current user on the connection to the user that created the connection. It also restores the current schema, current path, or current database to the original value used when the connection was created. If reauthentication was performed on the connection, this method is useful to reset the connection to the original user. For the SQL Server driver, the current user is the login user name.The driver throws an exception in the following circumstances: • The driver cannot change the current user to the initial user. • A transaction is active on the connection.
void setClientAccountingInfo(String)	Specifies a non-null string that sets the accounting client information on the connection. Some databases include this information in their usage reports. The maximum length allowed for accounting information for a particular database can be determined by calling the ExtDatabaseMetaData.getClientAccountingInfoLength() method. If the length of the information specified is longer than the maximum length allowed, the information is truncated to the maximum length, and the driver generates a warning. If setting accounting client information is supported by the database and this operation fails, the driver throws an exception.
void setClientApplicationName(String)	Specifies a non-null string that sets the name of the client application on the connection. The maximum client name length allowed for a particular database can be determined by calling the ExtDatabaseMetaData.getClientApplicationNameLength() method. If the length of the client application name specified is longer than the maximum name length allowed, the name is truncated to the maximum length allowed, and the driver generates a warning. If setting client name information is supported by the database and this operation fails, the driver throws an exception.
void setClientHostname(String)	Specifies a non-null string that sets the name of the host used by the client application on the connection. The maximum hostname length allowed for a particular database can be determined by calling the ExtDatabaseMetaData.getClientHostnameLength() method. If the length of the hostname specified is longer than the maximum hostname length allowed, the hostname is truncated to the maximum hostname length, and the driver generates a warning. If setting hostname information is supported by the database and this operation fails, the driver throws an exception.

ExtConnection Interface Methods	Description
void setClientUser(String)	Specifies a non-null string that sets the user ID of the client on the connection. This user ID may be different from the user ID establishing the connection. The maximum user ID length allowed for a particular database can be determined by calling the ExtDatabaseMetaData.getClientUserLength() method. If the length of the user ID specified is longer than the maximum length allowed, the user ID is truncated to the maximum user ID length, and the driver generates a warning. If setting user ID information is supported by the database and this operation fails, the driver throws an exception.
void setCurrentUser(String)	Specifies a non-null string that sets the current user on the connection. This method is used to perform reauthentication on a connection. For the SQL Server driver, the current user is the login user name. The driver throws an exception in the following circumstances: • The driver is connected to a database server that does not support reauthentication. • The database server rejects the request to change the user on the connection. • A transaction is active on the connection.
void setCurrentUser(String, Properties)	Specifies a non-null string that sets the current user on the connection. This method is used to perform reauthentication on a connection. In addition, this method sets options that control how the driver handles reauthentication. The options that are supported depend on the driver. See the DB2 driver, Oracle driver, and SQL Server driver chapters for information on which options are supported by each driver. For the SQL Server driver, the current user is the login user name. The driver throws an exception in the following circumstances: • The driver is connected to a database server that does not support reauthentication. • The database server rejects the request to change the user on the connection. • A transaction is active on the connection.

ExtConnection Interface Methods	Description
void setCurrentUser(javax.security.auth.Subject)	Specifies a non-null string that sets the current user on the connection to the user specified by the javax.security.auth.Subject object. This method is used to perform reauthentication on a connection. For the SQL Server driver, the current user is the login user name. The driver throws an exception in the following circumstances: • The driver does not support reauthentication. • The driver is connected to a database server that does not support reauthentication. • The database server rejects the request to change the user on the connection. • A transaction is active on the connection.
void setCurrentUser(javax.security.auth.Subject, Properties)	Specifies a non-null string that sets the current user on the connection to the user specified by the javax.security.auth.Subject object. This method is used to perform reauthentication on a connection. In addition, this method sets options that control how the driver handles reauthentication. The options that are supported depend on the driver. See the DB2 driver, Oracle driver, and SQL Server driver chapters for information on which options are supported by each driver. For the SQL Server driver, the current user is the login user name. The driver throws an exception in the following circumstances: • The driver does not support reauthentication. • The driver is connected to a database server that does not support reauthentication. • The database server rejects the request to change the user on the connection. • A transaction is active on the connection.
void setNetworkTimeout(int)	Supported by the SQL Server driver to set the network timeout. The network timeout is the maximum time (in milliseconds) that a connection, or objects created by a connection, will wait for the database to reply to an application request. If this limit is exceeded, the connection or objects are closed and the driver returns an exception indicating that a timeout occurred. A value of 0 means that no network timeout exists. Note that if a query timeout occurs before a network timeout, the execution of the statement is cancelled. Both the connection and the statement can be used. If a network timeout occurs before a query timeout or if the query timeout fails because of network problems, the connection is closed and neither the connection or the statement can be used.
boolean supportsReauthentication()	Indicates whether the connection supports reauthentication. If true is returned, you can perform reauthentication on the connection. If false is returned, any attempt to perform reauthentication on the connection throws an exception.

ExtDatabaseMetaData Interface

ExtDatabaseMetaData Interface Methods	Description
int getClientApplicationNameLength()	Returns the maximum length of the client application name. A value of `0` indicates that the client application name is stored locally in the driver, not in the database. There is no maximum length if the application name is stored locally.
int getClientUserLength()	Returns the maximum length of the client user ID. A value of `0` indicates that the client user ID is stored locally in the driver, not in the database. There is no maximum length if the client user ID is stored locally.
int getClientHostnameLength()	Returns the maximum length of the hostname. A value of `0` indicates that the hostname is stored locally in the driver, not in the database. There is no maximum length if the hostname is stored locally.
int getClientAccountingInfoLength()	Returns the maximum length of the accounting information. A value of `0` indicates that the accounting information is stored locally in the driver, not in the database. There is no maximum length if the hostname is stored locally.

ExtLogControl Class

Class Methods	Description
void setEnableLogging(boolean enable|disable)	If DataDirect Spy was enabled when the connection was created, you can turn on or off DataDirect Spy logging at runtime using this method. If true, logging is turned on. If false, logging is turned off. If DataDirect Spy logging was not enabled when the connection was created, calling this method has no effect.
boolean getEnableLogging()	Indicates whether DataDirect Spy logging was enabled when the connection was created and whether logging is turned on. If the returned value is true, logging is turned on. If the returned value is false, logging is turned off.

3

Supported SQL Functionality and Extensions for The Driver for Apache Hive

The DataDirect Connect® XE *for* JDBC™ driver for Apache Hive™ supports an extended set of SQL-92 in addition to the syntax for Apache HiveQL, which is a subset of SQL-92. Refer to the Hive Language Manual for information about using HiveQL.

For details, see the following topics:

- Data Definition Language (DDL)
- Insert
- Selecting Data With the Driver
- SQL Expressions
- Restrictions

Data Definition Language (DDL)

The Driver for Apache Hive supports a broad set of DDL, including (but not limited to) the following:

- CREATE Database and DROP Database
- CREATE Table and DROP Table
- ALTER Table and ALTER Partition statements
- CREATE View and DROP View

- CREATE Function and DROP Function

Refer to the Hive Data Definition Language manual for information about using HiveQL.

Insert

Purpose

Adds new rows to a table.

Syntax

```
INSERT INTO TABLE table_name VALUES (expression [,expression]...)
```

where:

table_name

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

expression

is a literal, a parameterized array, or null.

Notes

- The following conditions apply for the successful execution of an insert:
  - Values for all columns must be specified in order.
  - Column lists cannot be used.
  - Casts and other functions cannot be used.
  - String values must be enclosed in single quotation marks (').
- By default, the driver supports multirow inserts for parameterized arrays. For a multirow insert, the driver attempts to execute a single insert for all the rows contained in a parameter array. If the size of the insert statement exceeds the available buffer memory of the driver, the driver executes multiple statements. This behavior provides substantial performance gains for batch inserts.
- The driver modifies the HQL statement to perform a multirow insert. Therefore, the default multirow insert behavior may not be desirable in all scenarios. You can disable this behavior by setting the BatchMechanism connection property to `nativeBatch`. When `BatchMechanism=nativeBatch`, Hive's native batch mechanism is used to execute batch operations, and an insert statement is executed for each row contained in a parameter array.

Selecting Data With the Driver

Select List

The following sections discuss how the Select list can be used with the driver.

Between Clause

The BETWEEN clause is only supported in Apache Hive 0.9 or higher.

Column Name Qualification

A column can only be qualified with a single name, which must be a table alias. Furthermore, a table can be qualified with a database (JDBC schema) name in the FROM clause, and in some cases, must also be aliased. Aliasing may not be necessary if the database qualifier is not the current database.

The driver can work around these limitations using the Remove Column Qualifiers connection option.

- If set to 1, the driver removes three-part column qualifiers and replaces them with alias.column qualifiers.
- If set to 0, the driver does not do anything with the request.

Suppose you have the following ANSI SQL query:

```
SELECT schema.table1.col1,schema.table2.col2 FROM schema.table1,schema.table2

WHERE schema.table1.col3=schema.table2.col3
```

If the Remove Column Qualifiers connection option is enabled, the driver replaces the three-part column qualifiers:

```
SELECT table1.col1, table2.col2 FROM schema.table1 table1 JOIN schema.table2 table2
WHERE table1.col3 = table2.col3
```

From Clause

LEFT, RIGHT, and FULL OUTER JOINs are supported, as are LEFT SEMI JOINs and CROSS JOINs using the equal comparison operator, as shown in the following examples.

```
SELECT a.* FROM a JOIN b ON (a.id = b.id AND a.department = b.department)

SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1) JOIN c ON (c.key =
b.key2)

SELECT a.val, b.val FROM a LEFT OUTER JOIN b ON (a.key=b.key) WHERE
a.ds='2009-07-07' AND b.ds='2009-07-07'
```

However, the following syntax fails because of the use of non-equal comparison operators.

```
SELECT a.* FROM a JOIN b ON (a.id <> b.id)
```

HiveQL does not support join syntax in the form of a comma-separated list of tables. The driver, however, overcomes this limitation by translating the SQL into HiveQL, as shown in the following examples.

ANSI SQL-92 Query	Driver for Apache Hive HiveQL Translation
`SELECT * FROM t1, t2 WHERE a = b`	`SELECT * FROM t1 t1 JOIN t2 t2 WHERE a = b`
`SELECT * FROM t1 y, t2 x WHERE a = b`	`SELECT * FROM t1 y JOIN t2 x WHERE a = b`
`SELECT * FROM t2, (SELECT * FROM t1) x`	`SELECT * FROM t2 t2 JOIN (SELECT * FROM t1 t1) x`

Group By Clause

The GROUP BY clause is supported, with the following Entry SQL level restrictions:

- The COLLATE clause is not supported.
- SELECT DISTINCT is not supported for queries which also have a GROUP BY clause.
- The grouping column reference cannot be an alias. The following queries fail because `fc` is an alias for the `intcol` column:

  ```
  SELECT intcol AS fc, COUNT (*) FROM p_gtable GROUP BY fc
  ```

  ```
  SELECT f(col) as fc, COUNT (*) FROM table_name GROUP BY fc
  ```

Having Clause

The Having Clause is supported, with the following Entry SQL level restriction: a GROUP BY clause is required.

Order By Clause

The Order By clause is supported, with the following Entry SQL level restrictions:

- An integer sort key is not allowed.
- The COLLATE clause is not supported.

For Update Clause

Not supported in this release. If present, the driver strips the For Update clause from the query.

Set Operators

Supported, with the following Entry SQL level restrictions:

- UNION is not supported.

  Therefore, the following query fails:

```
SELECT * FROM t1 UNION SELECT * FROM t2
```

- UNION ALL is supported.

  Therefore, the following query works:

  ```
  SELECT * FROM t1 UNION ALL SELECT * FROM t2
  ```

  Note: For versions of Apache Hive 0.12 and earlier, UNION ALL is supported only in a subquery.

In addition, INTERSECT or EXCEPT are not supported.

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.

Subqueries are supported, with the following Entry SQL level restriction: subqueries can only exist in the FROM clause, that is, in a derived table. In the following example, the second Select statement is a subquery:

```
SELECT * FROM (SELECT * FROM t1 UNION ALL SELECT * FROM t2) sq
```

Although Apache Hive currently does not support IN or EXISTS subqueries, you can efficiently implement the semantics by rewriting queries to use LEFT SEMI JOIN.

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 clauses of Select statements.

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

Valid expression elements are:

- Constants on page 91
- Numeric Operators on page 92
- Character Operator on page 92
- Relational Operators on page 92
- Logical Operators on page 93
- Functions on page 93

Constants

Apache Hive uses binary literals for internal functions. Although the driver supports binary literals, no useful information is returned.

Apache Hive servers prior to Apache Hive 0.8 do not support literal values expressed in scientific notation.

Numeric Operators

You can use a numeric 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 1: Numeric Operators

Entry SQL Level Operator	HiveQL Operator
*	Supported
+	Supported
-	Supported
/	Supported
^ (XOR)	N/A
% (Mod)	N/A
& (bitwise AND)	N/A

Character Operator

The concatenation operator (||) is not supported; however, the CONCAT function is supported by HiveQL.

```
SELECT CONCAT('Name is', '(ename FROM emp)')
```

Relational Operators

Relational operators compare one expression to another.

The following table lists the supported relational operators.

Table 2: Relational Operators Supported with Apache Hive

Entry SQL Level Operator	Support in HiveQL
<>	Supported
<	Supported
<=	Supported
=	Supported
<=>	Supported (Hive versions 0.9 and higher)

Entry SQL Level Operator	Support in HiveQL
>	Supported
>=	Supported
IS [NOT] NULL	Supported
[NOT] BETWEEN x AND y	Supported
[NOT] IN	Supported
EXISTS	Supported
[NOT] LIKE	Supported, except that no collate clause is allowed
RLIKE	Supported
REGEXP	Supported

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 3: Logical Operators

Operator	Support in HiveQL
NOT !	Supported
AND &&	Supported
OR ||	Supported

Functions

The following tables show how SQL-92 functions are supported in HiveQL. Additional methods may be supported with Escapes. See SQL Escape Sequences for JDBC on page 277 for more information.

Table 4: Set Functions Supported

Set Function	Support in HiveQL
Count	Supported
AVG	Supported
MIN	Supported

Set Function	Support in HiveQL
MAX	Supported
SUM	Supported
DISTINCT	Supported
ALL	Supported

Table 5: Numeric Functions Supported

Numeric Function	Support in HiveQL
CHAR_LENGTH CHARACTER_LENGTH	Not supported. Use LENGTH(string) instead.
Position...In	Not supported
BIT_LENGTH(s)	Not supported
OCTET_LENGTH(str)	Not supported
EXTRACT...FROM	Not supported

Table 6: String Functions Supported

String Function	Support in HiveQL
Substring	Supported
Convert … using	Not supported
TRIM	Supported
Leading	Not supported. Use LTRIM.
Trailing	Not supported. Use RTRIM.
Both	Not supported (default behavior of TRIM)

Table 7: Date/Time Functions Supported

Date/Time Function	Support in HiveQL
CURRENT_DATE( )	Not supported
CURRENT_TIME( )	Not supported
CURRENT_TIMESTAMP	Not supported

Table 8: System Functions Supported

System Function	Support in HiveQL
CASE ... END	Supported
COALESCE	Supported
NULLIF	Not supported
CAST	Supported

Restrictions

Apache Hive has the following SQL restrictions:

- Column values and parameters are always nullable.
- No support for stored procedures
- No ROWID support
- No support for materialized views
- No support for synonyms
- Primary and foreign keys are not supported.
- Support for indexes is incomplete.
- Join support is limited to equality joins.
- A single quote within a string literal must be escaped using a \ instead of using a single quote. Because string literals can be expressed with either single or double quotation marks, *Apache's* would be written as *'Apache\'s'* or *"Apache\'s"*.

4

Supported SQL Statements and Extensions for the Salesforce Driver

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

For details, see the following topics:

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

- Drop Cache (EXT)
- Drop Index
- Drop Sequence
- Drop Table
- Drop View
- Explain Plan
- Insert
- Refresh Cache (EXT)
- Refresh Schema (EXT)
- Select
- Set Checkpoint Defrag
- Set Logsize
- Update
- SQL Expressions
- Operators
- Functions
- Conditions
- Subqueries

Alter Cache (EXT)

Purpose

Changes the definition of a cache on a remote table or view. An error is returned if the remote table or view specified does not exist.

Syntax

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

where:

remote_table

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

`view`

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

`REFERENCING`

is an optional clause that specifies the name of the remote table(s) for which a relationship cache is to be created. See Relational Caches on page 100 and Referencing Clause on page 110 for a complete explanation.

`REFRESH_INTERVAL`

is an optional clause that specifies the length of time the data in the cached table can be used before being refreshed. See Refresh Interval Clause on page 110 for a complete explanation.

`INITIAL_CHECK`

is an optional clause that specifies when the driver initially checks whether the data in the cache needs refreshed. See Initial Check Clause on page 111 for a complete explanation.

`PERSIST`

is an optional clause that specifies the life span of the data in the cached table or view. See Persist Clause on page 111 for a complete explanation.

`ENABLED`

is an optional clause that specifies whether the cache is enabled or disabled for use with SQL statements. See Enabled Clause on page 112 for a complete explanation.

`CALL_LIMIT`

is an optional clause that specifies the maximum number of Web service calls that can be used to populate or refresh the cache. See Call Limit Clause on page 113 for a complete explanation.

`FILTER`

is an optional clause that specifies a filter for the primary table to limit the number of rows that are cached in the primary table. See Filter Clause on page 114 for a complete explanation.

Notes

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

Relational Caches

If the Referencing clause is specified, the Alter Cache statement drops the existing cache and any referenced caches and creates a new set of related caches, one for each of the tables specified in the statement. The cache attributes for the existing cache are the default cache attributes for the new relational cache. Any attributes specified in the Alter Cache statement override the default attributes. If the Referencing clause is not specified, the existing cache references, if any, are used.

If the cache being altered is a relational cache, the attributes specified in the Alter Cache statement apply to all of the caches that comprise the relational cache.

Alter Index

Purpose

Changes the name of an existing index.

Syntax

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

where:

index_name

specifies an existing index name.

new_name

specifies the new index name.

Notes

- Index names must not conflict with other user-defined or system-defined names.
- Indexes on remote tables cannot be created, altered or dropped. Indexes can only be defined on local tables.

Alter Sequence

Purpose

Resets the next value of an existing sequence.

Syntax

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

where:

sequence_name

specifies an existing sequence.

value

specifies the next value to be returned through the Next Value For clause (see Next Value For Clause on page 116).

Alter Session (EXT)

Purpose

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

Syntax

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

where:

attribute_name

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

value

specifies the value for that attribute.

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

Table 9: Alter Session Attributes

Attribute Name	Session Type	Description
Current_Schema	Database	Sets the current schema for the database session. The current schema is the schema used when an identifier in a SQL statement is unqualified. The string value must be the name of a schema visible in the database session. For example: `ALTER SESSION SET CURRENT_SCHEMA=sforce`

Attribute Name	Session Type	Description
Stmt_Call_Limit	Database	Sets the maximum number of Web service calls the driver can make in executing a statement. Setting the Stmt_Call_Limit attribute has the same effect as setting the StmtCallLimit connection option. It sets the default Web service call limit used by any statement on the connection. Executing this command on a statement overrides the previously set StmtCallLimit for the connection. The value specified must be a positive integer or `0`. The value `0` means that no call limit exists. For example: `ALTER SESSION SET STMT_CALL_LIMIT=10`
Ws_Call_Count	Remote	Resets the Web service call count of a remote session to the value specified. The value must be `0` or a positive integer. WS_Call_Count represents the total number of Web service calls made to the remote data source instance for the current session. For example: `ALTER SESSION SET sforce.WS_CALL_COUNT=0` The current value of WS_Call_Count can be obtained by referring to the System_Remote_Sessions system table (see SYSTEM_REMOTE_SESSIONS Catalog Table for details). For example: `SELECT * from information_schema.system_remote_sessions WHERE session_id = cursessionid()`

Alter Table

For information on...	See
Altering a remote table	Altering a Remote Table on page 102
Altering a local table	Altering a Local Table on page 105

Altering a Remote Table

Purpose

Adds a column, removes a column, or redefines a column in a table. The table being altered can be either a remote or local table. A remote table is a Salesforce object and is exposed in the SFORCE schema. A local table is maintained by the driver and is local to the machine on which the driver is running. A local table is exposed in the PUBLIC schema.

Syntax

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

where:

table_name

specifies an existing remote table.

add_clause

specifies a column or a foreign key constraint to be added to the table. See Add Clause: Columns on page 103 and Add Clause: Constraints on page 104 for a complete explanation.

drop_clause

specifies a column to be dropped from the table. See Drop Clause: Columns on page 104 for a complete explanation.

Notes

- You cannot drop a constraint from a remote table.

Add Clause: Columns

Purpose

Adds a column to an existing table. It is optional.

Syntax

```
ADD [COLUMN] column_name Datatype ...
[DEFAULT default_value] [[NOT]NULL] [EXT_ID] [PRIMARY KEY]
[START WITH starting_value]
```

default_value

is the default value to be assigned to the column. See Column Definition for Remote Tables on page 117 for details.

starting_value

is the starting value for the Identity column. The default start value is `0`.

Notes

- If `NOT NULL` is specified and the table is not empty, a default value must be specified. In all other respects, this command is the equivalent of a column definition in a Create Table statement.
- You cannot specify ANYTYPE, BINARY, COMBOBOX, or TIME data types in the column definition of Alter Table statements.
- If a SQL view includes `SELECT * FROM` for the table to which the column was added in the view's Select statement, the new column is added to the view.

Example A

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

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

Example B

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

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

Add Clause: Constraints

Purpose

Adds a constraint to an existing table. It is optional.

Syntax

ADD [CONSTRAINT *constraint_name*] ...

Notes

- The only type of constraint you can add is a foreign key constraint.
- When adding a foreign key constraint, the table that contains the foreign key must be empty.

Example

Assuming the current schema is SFORCE, a foreign key constraint is added to the `deptId` column of the `test` table, referencing the `rowId` of the `dept` table. For the operation to succeed, the `dept` table must be empty.

```
ALTER TABLE test ADD FOREIGN KEY (deptId) REFERENCES dept(rowId)
```

Drop Clause: Columns

Purpose

Drops a column from an existing table. It is optional.

Syntax

DROP {[COLUMN] *column_name*}

where:

column_name

specifies an existing column in an existing table.

Notes

- The column being dropped cannot have a constraint defined on it.
- Drop fails if a SQL view includes the column.

Example

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

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

Altering a Local Table

Purpose

Adds a column, removes a column, or redefines a column in a table. The table being altered can be either a remote or local table. A remote table is a Salesforce object and is exposed in the SFORCE schema. A local table is maintained by the driver and is local to the machine on which the driver is running. A local table is exposed in the PUBLIC schema.

Syntax

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

where:

table_name

specifies an existing local table.

add_clause

specifies a column or constraint to be added to the table. See Add Clause: Columns on page 105 and Add Clause: Constraints on page 106for a complete explanation.

drop_clause

specifies a column or constraint to be dropped from the table. See Drop Clause: Columns on page 106 and Drop Clause: Constraints on page 107 for a complete explanation.

rename_clause

specifies a new name for the table. See Rename Clause on page 107 for a complete explanation.

Add Clause: Columns

Purpose

Adds a column to an existing table. It is optional. The column is added to the end of the column list.

Syntax

```
ADD [COLUMN] column_name Datatype ... [BEFORE existing_column]
```

Notes

- If `NOT NULL` is specified and the table is not empty, a default value must be specified. In all other respects, this command is the equivalent of a column definition in a Create Table statement.
- You cannot specify ANYTYPE, BINARY, COMBOBOX, or TIME data types in the column definition of Alter Table statements.
- Using the `Before` *existing_column* clause, you can specify the name of an existing column so that the new column is inserted in a position just before the existing column.

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

Example A

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

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

Example B

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

```
ALTER TABLE test ADD COLUMN deptId INT
```

Add Clause: Constraints

Purpose

Adds a constraint to an existing table. It is optional.

This command adds a constraint using the same syntax as the Create Table command (see Constraint Definition for Local Tables on page 123).

Syntax

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

Notes

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

Example

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

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

Drop Clause: Columns

Purpose

Drops a column from an existing table. It is optional.

Syntax

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

where:

column_name

specifies an existing column in an existing table.

Notes

- Drop fails if a SQL view includes the column.

Example

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

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

Drop Clause: Constraints

Purpose

Drops a constraint from an existing table. It is optional.

Syntax

DROP {[CONSTRAINT] *constraint_name*}

where:

constraint_name

specifies an existing constraint.

Notes

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

Example

This example drops the `test_fk` constraint.

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

Rename Clause

Purpose

Renames an existing table. It is optional.

Syntax

RENAME TO *new_name*

where:

new_name

specifies the new name for the table.

Example

This example renames the table to `test2`.

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

Checkpoint

Purpose

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

Syntax

```
CHECKPOINT [DEFRAG]
```

where:

`DEFRAG`

if specified, this statement evaluates abandoned space in the database data file (.data) and shrinks the data file to its minimum size.

Create Cache (EXT)

Purpose

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

Syntax

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

where:

remote_table

is the name of the remote table from which data is to be cached on the client. The name of the cached table is the same as the name of the remote table. When the table name is specified in a query, the cached table is accessed, not the remote table. The remote table name can be a two-part name: *schemaname.tablename*. When specifying a two-part name, the specified remote table must be defined in the specified schema, and you must have the privilege to create objects in the specified schema.

`REFERENCING`

is an optional clause that specifies the name of the remote table(s) for which a relationship cache is to be created. See Relational Caches on page 109 and Referencing Clause on page 110 for a complete explanation.

`REFRESH_INTERVAL`

is an optional clause that specifies the length of time the data in the cached table can be used before being refreshed. See Refresh Interval Clause on page 110 for a complete explanation.

`INITIAL_CHECK`

is an optional clause that specifies when the driver initially checks whether the data in the cache needs refreshed. See Initial Check Clause on page 111 for a complete explanation.

`PERSIST`

is an optional clause that specifies the life span of the data in the cached table or view. See Persist Clause on page 111 for a complete explanation.

`ENABLED`

is an optional clause that specifies whether the cache is enabled or disabled for use with SQL statements. See Enabled Clause on page 112 for a complete explanation.

`CALL_LIMIT`

is an optional clause that specifies the maximum number of Web service calls that can be used to populate or refresh the cache. See Call Limit Clause on page 113 for a complete explanation.

`FILTER`

is an optional clause that specifies a filter for the primary table to limit the number of rows that are cached in the primary table. See Filter Clause on page 114 for a complete explanation.

Notes

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

Relational Caches

If the Referencing clause is specified, the Create Cache statement creates a set of related caches, one for each of the tables specified in the statement. This set of caches is referred to as a related or relational cache. The set of caches in a relational cache is treated as a single entity. They are refreshed, altered, and dropped as a unit. Any attributes specified in the Create Cache statement apply to the cache created for the primary table and to the caches created for all of the referenced tables specified.

A database session can have both standalone and relational caches defined, but only one cache can be defined on a table. If a table is referenced in a relational cache definition, a standalone cache cannot be created on that table.

Referencing Clause

Purpose

Specifies the name of the remote table(s) for which a relationship cache is to be created; it is optional. The specified remote table must be related to either the primary table being cached or one of the other specified related tables. The remote table name cannot include a schema name. The referenced tables must exist in the same schema as the primary table.

Syntax

```
REFERENCING (remote_table_ref[,remote_table_ref]...)]
```

where:

remote_table_ref

represents *remote_table*[.*foreign_key_name*]

remote_table

specifies one or more tables related to the primary table that are to be cached in conjunction with the primary table.

foreign_key_name

specifies the name of the foreign key relationship between the remote table and the primary table (or, optionally, another related table). If a foreign key name is not specified, the driver attempts to find a relationship between the remote table and one of the other tables specified in the relational cache. The driver first looks for a relationship to the primary table. If a relationship to the primary table does not exist, the driver then looks for a relationship to other referenced tables.

See also

Creating a Cache in the *DataDirect Connect Series for JDBC User's Guide*

Refresh Interval Clause

Purpose

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

Syntax

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

where:

`0`

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

`-1`

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

`interval_value`

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

Initial Check Clause

Purpose

Specifies when the driver performs its initial check of the data in the cache to determine whether it needs to be refreshed; it is optional.

Syntax

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

where:

`ONFIRSTCONNECT`

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

`FIRSTUSE`

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

`DEFAULT`

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

Persist Clause

Purpose

Specifies the life span of the data in the cached table or view; it is optional.

Syntax

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

where:

`TEMPORARY`

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

`MEMORY`

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

`DISK`

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

`DEFAULT`

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

Notes

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

Enabled Clause

Purpose

Specifies whether the cache is enabled or disabled for use with SQL statements; it is optional.

Syntax

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

where:

`YES | TRUE`

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

The driver does not check whether the cache needs to be refreshed when the cache is enabled. The check occurs the next time that the cache is accessed.

`NO | FALSE`

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

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

Default

The default behavior is `TRUE`.

Call Limit Clause

Purpose

Specifies the maximum number of Web service calls that can be used to populate or refresh the cache; it is optional.

Syntax

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

where:

`0`

specifies no call limit.

`-1`

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

`max_calls`

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

Default

The default behavior is `0`.

Notes

- The call limit for a cache is independent of the Stmt_Call_Limit set on a database session. See Alter Session (EXT) on page 101 for details.

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

Filter Clause

Purpose

Specifies a filter for the primary table to limit the number of rows that are cached in the primary table; it is optional. This clause is not supported for views.

Syntax

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

where:

expression

is any valid Where clause. See Where Clause on page 140 for details. Do not include the Where keyword in the clause. The filter for an existing cache can be removed by specifying an empty string for the filter expression, for example, `FILTER()`.

Default

The default behavior is that cached data is not filtered.

Example A

Referencing clause allows multiple related tables to be cached as a single entity. This example creates a cache on the remote table account. The cache is populated with all accounts that have had activity in 2010. Additionally, caches are created for the following remote tables: `opportunity`, `contact`, and `opportunitylineitem`. These caches are populated with the opportunities and contacts that are associated with the accounts stored in the accounts cache and the opportunity line items associated with the opportunities stored in the opportunity cache.

```
CREATE CACHE ON account
  REFERENCING (opportunity, contact, opportunitylineitem)
  FILTER (lastactivitydate >= {d'2010-01-01'})
```

Example B

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

```
CREATE CACHE ON account
```

Example C

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

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

Create Index

Purpose

Creates an index on one or more columns in a local table.

Syntax

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

where:

`UNIQUE`

means that key columns cannot have duplicate values.

`index_name`

specifies the name of the index to be created.

`table_name`

specifies an existing local table.

`column_name`

specifies an existing column.

Notes

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

Create Sequence

Purpose

Creates an auto-incrementing sequence for a local table.

Syntax

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

where:

`sequence_name`

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

start_value

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

increment_value

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

Next Value For Clause

Purpose

Specifies the next value for a sequence that is used in a Select, Insert, or Update statement.

Syntax

`NEXT VALUE FOR` *sequence_name*

where:

sequence_name

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

Example

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

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

Create Table

For information on...	See...
Creating a remote table	Creating a Remote Table on page 116
Creating a local table	Creating a Local Table on page 121

Creating a Remote Table

Purpose

Creates a new table. You can create either a remote or local table. A remote table is a Salesforce object and is exposed in the SFORCE schema. Creating a table in the SFORCE schema creates a remote table. A local table is maintained by the driver and is local to the machine on which the driver is running. A local table is exposed in the PUBLIC schema. Creating a table in the PUBLIC schema creates a local table.

Syntax

```
CREATE TABLE table_name (column_definition [, ...] [, constraint_definition...])
```

where:

table_name

specifies the name of the new remote table. The table name can be qualified by a schema name using the format *schema.table*. If the schema is not specified, the table is created in the current schema. See Alter Session (EXT) on page 101 for information about changing the current schema.

column_definition

specifies the definition of a column in the new table. See Column Definition for Remote Tables on page 117 for a complete explanation.

constraint_definition

specifies constraints on the columns of the new table. See Constraint Definition for Remote Tables on page 119 for a complete explanation.

Notes

- Creating tables in Salesforce is not a quick operation. It can take several minutes for Salesforce to create the table and its relationships.

Column Definition for Remote Tables

Purpose

Defines a column for remote tables.

Syntax

```
column_name Datatype [(precision[,scale])...]
[DEFAULT default_value][[NOT]NULL][EXT_ID][PRIMARY KEY]
[START WITH starting_value]
```

where:

column_name

is the name to be assigned to the column.

Datatype

is the data type of the column to be created. See Data Types in the *DataDirect Connect Series for JDBC User's Guide* for a list of supported Salesforce data types. You cannot specify ANYTYPE, BINARY, COMBOBOX, ENCRYPTEDTEXT, or TIME data types in the column definition of Create Table statements.

precision

is the total number of digits for NUMBER, CURRENCY, and PERCENT columns, and the length of HTML, LONGTEXTAREA, and TEXT columns.

scale

is the number of digits to the right of the decimal point for NUMBER, CURRENCY, and PERCENT columns.

default_value

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

- For character columns, a single-quoted string or NULL.
- For datetime columns, a single-quoted Date, Time, or Timestamp value or NULL. You can also use the following datetime SQL functions: CURRENT_DATE, CURRENT_ TIMESTAMP, TODAY, or NOW.
- For boolean columns, the literals FALSE, TRUE, NULL.
- For numeric columns, any valid number or NULL.

starting_value

is the starting value for the Identity column. The default start value is `0`.

`[NOT]NULL`

is used to specify whether NULL values are allowed or not allowed in a column. If `NOT NULL` is specified, all rows in the table must have a column value. If `NULL` is specified or if neither `NULL` or `NOT NULL` is specified, NULL values are allowed in the column.

`EXT_ID`

is used to specify that the column is an external ID column.

`PRIMARY KEY`

can only be specified when the data type of the column is ID. ID columns are always the primary key column for Salesforce.

`START WITH`

specifies the sequence of numbers generated for the Identity column. It can only be used when the data type of the column definition is AUTONUMBER.

Example A

Assuming the current schema is SFORCE, the remote table `Test` is created in the SFORCE schema. The `id` column has a starting value of `1000`.

```
CREATE TABLE Test (id AUTONUMBER START WITH 1000, Name TEXT(30))
```

Example B

The table name is qualified with a schema name that is not the current schema, creating the `Test` table in the `SFORCE` schema. The table is created with the following columns: `id`, `Name`, and `Status`. The `Status` column contains a default value of `ACTIVE`.

```
CREATE TABLE SFORCE.Test (id NUMBER(9, 0), Name TEXT(30), Status TEXT(10) DEFAULT
'ACTIVE')
```

Example C

Assuming the current schema is SFORCE, the remote table `dept` is created with the `name` and `deptId` columns. The `deptId` column can be used as an external ID column.

```
CREATE TABLE dept (name TEXT(30), deptId NUMBER(9, 0) EXT_ID)
```

Constraint Definition for Remote Tables

Purpose

Defines a constraint for a remote table.

Syntax

```
[CONSTRAINT [constraint_name] {foreign_key_constraint}]
```

where:

constraint_name

is ignored. The driver uses the Salesforce relationship naming convention to generate the constraint name.

foreign_key_constraint

defines a link between related tables. See Foreign Key Clause on page 120 for the syntax.

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

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

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

Referenced Table			**Main Table**		
DEPT			EMP		
					(Foreign Key)

Referenced Table			Main Table		
id	name		id	name	dept id
1	Dev		1	Mark	1
2	Finance		1	Jim	3
3	Sales		1	Mike	4

Foreign Key Clause

Purpose

Specifies a foreign key for a constraint.

Syntax

```
FOREIGN KEY (fcolumn_name)
   REFERENCES ref_table (pcolumn_name)
```

where:

fcolumn_name

specifies the foreign key column to which the constraint is applied. The data type of this column must be the same as the data type of the column it references.

ref_table

specifies the table to which the foreign key refers.

pcolumn_name

specifies the primary key column in the referenced table. For Salesforce, the primary key column is always the rowId column.

Example

Assuming the current schema is SFORCE, the remote table `emp` is created with the `name`, `empId`, and `deptId` columns. The table contains a foreign key constraint on the `deptId` column, referencing the `rowId` in the `dept` table created in Example C. For the operation to succeed, the data type of the `deptId` column must be the same as that of the `rowId` column.

```
CREATE TABLE emp (name TEXT(30), empId NUMBER(9, 0) EXT_ID, deptId TEXT(18),
FOREIGN KEY(deptId) REFERENCES dept(rowId))
```

Creating a Local Table

Purpose

Creates a new table. You can create either a remote or local table. A remote table is a Salesforce object and is exposed in the SFORCE schema. Creating a table in the SFORCE schema creates a remote table. A local table is maintained by the driver and is local to the machine on which the driver is running. A local table is exposed in the PUBLIC schema. Creating a table in the PUBLIC schema creates a local table.

Syntax

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

where:

`MEMORY`

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

`DISK`

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

`TEMPORARY | TEMP`

> creates the new table as a global temporary table. The `GLOBAL` qualifier is optional. The definition of a global temporary table is visible to all connections. The data written to a global temporary table is visible only to the connection used to write the data.

`table_name`

> specifies the name of the new table.

`column_definition`

> specifies the definition of a column in the new table. See Column Definition for Local Tables on page 122 for a complete explanation.

`constraint_definition`

> specifies constraints on the columns of the new table. See Constraint Definition for Local Tables on page 123 for a complete explanation.

`ON COMMIT PRESERVE ROWS`

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

`ON COMMIT DELETE ROWS`

> empties row values on each commit or rollback.

Notes

- If `MEMORY`, `DISK`, or `TEMPORARY|TEMP` is not specified, the new table is created in memory.

Column Definition for Local Tables

Purpose

Defines a column for local tables.

Syntax

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

where:

column_name

is the name to be assigned to the column.

Datatype

is the data type of the column to be created. See Data Types in the *DataDirect Connect Series for JDBC User's Guide* for a list of supported Salesforce data types. You cannot specify ANYTYPE, BINARY, COMBOBOX, or TIME data types in the column definition of Create Table statements.

precision

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

scale

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

default_value

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

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

`IDENTITY | GENERATED BY DEFAULT AS IDENTITY`

define an auto-increment column. You can only specify these clauses on INTEGER and BIGINT columns. Identity columns are considered primary key columns, so a table can have only one Identity column.

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

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

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

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

Example A

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

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

This example is equivalent to the previous example.

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

Example B

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

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

Constraint Definition for Local Tables

Purpose

Defines a constraint for a local table.

Syntax

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

where:

constraint_name

specifies a name for the constraint.

unique_constraint

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

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

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

A single table can have multiple columns with unique constraints.

primary_key_constraint

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

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

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

Only one primary key constraint is allowed in the table.

foreign_key_constraint

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

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

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

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

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

Unique Clause

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

where:

column_name

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

Primary Key Clause

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

where:

column_name

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

Foreign Key Clause

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

where:

fcolumn_name

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

ref_table

specifies the table to which a foreign key refers.

pcolumn_name

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

ON DELETE

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

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

ON UPDATE

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

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

Notes

- You must specify at least one constraint.
- Both the ON DELETE and ON UPDATE clauses can be used in a single foreign key definition.

Example

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

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

Create View

Purpose

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

Syntax

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

where:

view_name

specifies the name of the view.

view_column

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

The other commands used for Create View are the same as those used for Select (see Select on page 135).

Notes

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

Example A

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

```
CREATE VIEW myOpportunities AS
SELECT a.name AS AccountName,
       o.name AS OpportunityName,
       o.amount AS Amount,
       o.description AS Description
```

```
FROM Opportunity o INNER JOIN Account a
       ON o.AccountId = a.id
       INNER JOIN User u
       ON o.OwnerId = u.id
WHERE u.name = 'MyName'
       AND o.isClosed = 'false'
ORDER BY Amount desc
```

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

```
SELECT * FROM myOpportunities;
```

Example B

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

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

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

Delete

Purpose

Deletes 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 140 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.

Drop Cache (EXT)

Purpose

Drops the cache defined on a remote table. To drop a relational cache, the specified table must be the primary table of the relational cache. If a relational cache is specified, the cache for the primary table and all referenced caches are dropped.

Syntax

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

where:

remote_table

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

`IF EXISTS`

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

Notes

- Caches on views are not supported.

Drop Index

Purpose

Drops an index for a local table.

Syntax

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

where:

index_name

specifies an existing index.

`IF EXISTS`

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

Notes

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

Drop Sequence

Purpose

Drops a sequence for a local table.

Syntax

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

where:

sequence_name

specifies the name of a sequence to drop.

`IF EXISTS`

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

`RESTRICT`

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

`CASCADE`

silently drops all dependent database objects.

Drop Table

Purpose

Drops (removes) a remote or local table, its data, and its indexes. A remote table is a Salesforce object and is exposed in the SFORCE schema. Dropping a table in the SFORCE schema drops a remote table. A local table is maintained by the driver and is local to the machine on which the driver is running. A local table is exposed in the PUBLIC schema. Dropping a table in the PUBLIC schema drops a local table.

Syntax

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

where:

table_name

specifies the name of an existing table to drop.

`IF EXISTS`

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

`RESTRICT`

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

`CASCADE`

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

Drop View

Purpose

Drops a view.

Syntax

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

where:

view_name

specifies the name of a view.

`IF EXISTS`

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

`RESTRICT`

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

`CASCADE`

silently drops all dependent views.

Explain Plan

Purpose

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

Syntax

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

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

Insert

Purpose

Adds new rows to a table. You can specify either of the following options:

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

Syntax

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

table_name

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

column_name

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

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 (’).

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 also

Specifying an External ID Column on page 133

Literals on page 148

Select on page 135

Specifying an External ID Column

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

Syntax

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

where:

`EXT_ID`

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

schema_name

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

table_name

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

ext_id_column

is the external ID column.

Example A

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

```
INSERT INTO emp (last_name,
                 first_name,
```

```
                    emp_id,
                    salary,
                    hire_date)
VALUES ('Smith', 'John', 'E22345', 27500, {1999-04-06})
```

Example B

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

```
INSERT INTO emp1 (first_name,
                  last_name,
                  emp_id,
                  dept,
                  salary)
SELECT first_name, last_name, emp_id, dept, salary FROM emp
WHERE dept = 'D050'
```

Example C

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

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

Refresh Cache (EXT)

Purpose

Forces the data in the cache for the specified remote table to be refreshed.

Syntax

`REFRESH CACHE ON {`*`remote_table`*` | ALL} [CLEAN]`

where:

`remote_table`

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

`ALL`

forces all caches to be refreshed.

`CLEAN`

is optional and discards the data in the cache for the specified table or view, or all cache data if `ALL` is specified, and repopulates the cache with the data in the remote table or view.

Notes

- Caches on views are not supported.

Refresh Schema (EXT)

Purpose

Updates the remote object mapping and other information contained in a remote schema.

Syntax

```
REFRESH SCHEMA schema_name
```

where:

schema_name

is the name of the schema to be refreshed.

Select

Purpose

Fetches results from one or more tables. It can operate on local and remote tables in any combination.

Syntax

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

where:

select_clause

specifies the columns from which results are to be returned by the query. See Select Clause on page 136 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 139 for a complete explanation.

where_clause

is optional and restricts the results that are returned by the query. See Where Clause on page 140 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 141 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 141 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 142 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 143 for a complete explanation.

`EXCEPT | MINUS`

are synonymous optional operators that return 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 143 for a complete explanation.

orderby_clause

is optional and sorts the results that are returned by the query. See Order By Clause on page 144 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 145 for a complete explanation.

Select Clause

Purpose

Specifies a list of column expressions that identify columns of values that you want to retrieve or an asterisk (*) to retrieve the value of all columns.

Syntax

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

where:

`LIMIT` *`offset number`*

creates the result set for the Select statement first and then discards the first number of rows specified by *offset* and returns the number of remaining rows specified by *number*. To not discard any of the rows, specify `0` for *offset*, for example, LIMIT 0 *number.* To discard the first *offset* number of rows and return all the remaining rows, specify `0` for *number*, for example, LIMIT *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 147 for details. *column_expression* can also include aggregate functions. See Aggregate Functions on page 138 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 dept AS department FROM emp
```

`DISTINCT`

eliminates duplicate rows from the result of a query. For example:

```
SELECT DISTINCT dept FROM emp
```

`INTO`

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

Notes

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

Aggregate Functions

The result of a query can be the result of one or more aggregate functions. 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 describes the supported aggregate functions.

Table 10: 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 column expression. For example, `COUNT(`*name*`)` returns the number of name values. When using `COUNT` with a column 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.

Except for `COUNT(*)`, all aggregate functions exclude NULL values. The returned value type for `COUNT` is INTEGER and for `MIN`, `MAX`, and `AVG` it is the same type as the column.

Example A

In this 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

This 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

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 140 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

This 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
```

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

Join in a From Clause

Purpose

Associates multiple tables within a Select statement. Joins may be either explicit or implicit.

Example A

This is the example from the previous section restated as an explicit inner join:

```
SELECT e.name, d.deptName
FROM emp e INNER JOIN dep d ON e.deptId = d.id;
FROM table_name {RIGHT OUTER | INNER | LEFT OUTER | CROSS} JOIN table.key ON
search-condition
```

Example B

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

Purpose

Used in place of table references (*table_name*).

Syntax

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

See also

Subqueries on page 159

Where Clause

Purpose

Specifies the conditions that rows must meet to be retrieved.

Syntax

`WHERE` *`expr1rel_operatorexpr2`*

where:

`expr1`

is either a column name, literal, or expression.

`expr2`

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

`rel_operator`

is the relational operator that links the two expressions.

Example

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

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

See also

Subqueries on page 159

SQL Expressions on page 147

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

This 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 147

Having Clause

Purpose

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

Syntax

```
HAVING expr1rel_operatorexpr2
```

where:

expr1

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

expr2

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

rel_operator

is the relational operator that links the two expressions.

Example

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

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

See also

SQL Expressions on page 147

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

This 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

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

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

Intersect Operator

Purpose

Returns a single result set. The result set contains rows that are returned by both Select statements. Duplicates are returned unless the `DISTINCT` operator is added.

Syntax

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

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

This 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

This 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

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

Syntax

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

where:

`DISTINCT`

eliminates duplicate rows from the results.

Notes

- When using either 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

This 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

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

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

Order By Clause

Purpose

Specifies how the rows are to be sorted.

Syntax

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

where:

`sort_expression`

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

The default behavior is 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 147

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

This example returns a maximum of 20 rows.

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

Set Checkpoint Defrag

Purpose

Sets the threshold for triggering a Checkpoint Defrag. It is used in conjunction with the Checkpoint statement.

Syntax

`SET CHECKPOINT DEFRAG size`

where:

size

specifies the threshold size.

Notes

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

See also

Checkpoint on page 108

Set Logsize

Purpose

Sets the maximum size to which the driver's embedded database log file can grow before a Checkpoint statement is performed. When the log file exceeds the specified size, the Checkpoint statement closes and then reopens the database files, resetting the .log file.

Syntax

```
SET LOGSIZE size
```

where:

size

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

See also

Checkpoint on page 108

Update

Purpose

Changes the value of columns in selected rows of a table.

Syntax

```
UPDATE table_name SET column_name = expression [, column_name = expression] [WHERE
conditions]
```

where:

table_name

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

column_name

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

expression

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

Notes

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

Example A

This 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

This 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'
```

See also

Subqueries on page 159

Where Clause on page 140

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, Having, and Order By clauses of Select statements; and in the Set clauses of Update statements.

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

The Salesforce driver supports both unquoted and quoted identifiers. An unquoted identifier must start with an ASCII alpha character and can be followed by zero 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 Salesforce driver recognizes the Unicode escape sequence \uxxxx as a Unicode character. You can specify a double quotation mark in a quoted identifier by escaping it with a double quotation mark.

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

Valid expression elements are:

- Column names
- Literals
- Operators
- Functions

Column Names

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

Literals

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

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

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

Table 11: 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`

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

Character String Literals

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

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

Example

```
'Hello'

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

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`

Numeric Literals

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

Example

```
+1894.1204
```

Binary Literals

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

Example

```
'00af123d'
```

Date/Time Literals

Date and time literal values are:

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

Operators

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

Unary Operator

```
operator operand
```

Binary Operator

```
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 12: 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 13: 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 Salesforce driver considers the UNKNOWN result as FALSE. The following table lists the supported comparison operators.

Table 14: 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`

[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`

ESCAPE clause in LIKE operatorLIKE ’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.	`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 '\'`

Operator	Purpose	Example
	The default escape character is backslash (\).	This matches only records with name ’JOE_JOHN’.

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 15: 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 16: Operator Precedence

Precedence	Operator
1	+ (Positive), - (Negative)
2	*(Multiply), / (Division)
3	+ (Add), - (Subtract)

Precedence	Operator
4	|| (Concatenate)
5	=, >, <, >=, <=, <>, != (Comparison operators)
6	NOT, IN, LIKE
7	AND
8	OR

Example A

The query in this 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 this 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 Salesforce driver supports a number of functions that you may use in expressions, as listed and described in the tables in this section.

Table 17: Numerical Functions Supported

Numerical Function	Description
ABS(*d*)	Returns the absolute value of a double value.
ACOS(*d*)	Returns the arc cosine of an angle.
ASIN(*d*)	Returns the arc sine of an angle.
ATAN(*d*)	Returns the arc tangent of an angle.
ATAN2(*a*,*b*)	Returns the tangent of *a*/*b*.
BITAND(*a*,*b*)	Returns *a* and *b*.

Numerical Function	Description
BITOR(*a*,*b*)	Returns *a* or *b*.
CEILING(*d*)	Returns the smallest integer that is not less than *d*.
COS(*d*)	Returns the cosine of an angle.
COT(*d*)	Returns the cotangent of an angle.
DEGREES(*d*)	Converts radians to degrees.
EXP(*d*)	Returns e (2.718... raised to the power of *d*).
FLOOR(*d*)	Returns the largest integer that is not greater than *d*.
LOG(*d*)	Returns the natural logarithm (base *e*).
LOG10(*d*)	Returns the logarithm (base 10).
MOD(*a*,*b*)	Returns a modulo *b*.
PI( )	Returns pi (3.1415...).
POWER(*a*,*b*)	Returns a raised to the power of *b*.
RADIANS(*d*)	Converts degrees to radians.
RAND( )	Returns a random number *x* bigger or equal to 0.0 and smaller than 1.0.
ROUND(*a*,*b*)	Rounds *a* to *b* digits after the decimal point.
ROUNDMAGIC(*d*)	Solves rounding problems such as 3.11-3.1-0.01.
SIGN(*d*)	Returns -1 if *d* is smaller than 0, 0 if *d*==0 and 1 if *d* is bigger than 0.
SIN(*d*)	Returns the sine of an angle.
SQRT(*d*)	Returns the square root.
TAN(*A*)	Returns the trigonometric tangent of an angle.
TRUNCATE(*a*,*b*)	Truncates a to b digits after the decimal point.

Table 18: String Functions Supported

String Function	Description
ASCII(*s*)	Returns the ASCII code of the leftmost character of *s*.
BIT_LENGTH(*str*)	Returns the length of the string in bits.

String Function	Description
CHAR(*c*)	Returns a character that has the ASCII code c.
CHAR_LENGTH(*str*)	Returns the length of the string in characters.
CONCAT(*str1*,*str2*)	Returns the string that results from concatenating *str1* + *str2*.
DIFFERENCE(*s1*,*s2*)	Returns the difference between the sound of *s1* and *s2*.
HEXTORAW(*s1*)	Returns a translated string/.
INSERT(*s*,*start*,*len*,*s2*)	Returns a string where *len* number of characters beginning at *start* has been replaced by *s2*.
LCASE(*s*)	Converts s to lower case.
LEFT(*s*,count)	Returns the leftmost count of characters of *s*. If *s* requires double quoting, use SUBSTRING( ) instead.
LENGTH(*s*)	Returns the number of characters in *s*.
LOCATE(*search*,*s*,[*start*])	Returns the first index (1=left, 0=not found) where *search* is found in *s*, starting at *start*.
LTRIM(*s*)	Removes all leading blanks in *s*.
OCTET_LENGTH(*str*)	Returns the length of the string in bytes (twice the number of characters).
RAWTOHEX(*s1*)	Returns translated string.
REPEAT(*s*,*count*)	Returns s repeated count times.
REPLACE(*s*,*replace*,*s2*)	Returns *s* with all occurrences of *replace* replaced with *s2*.
RIGHT(*s*,*count*)	Returns the right-most count of characters of *s*.
RTRIM(*s*)	Removes all trailing spaces in *s*.
SOUNDEX(*s*)	Returns a 4-character code representing the sound of *s*.
SPACE(*count*)	Returns a string consisting of count spaces.
SUBSTR(*s*,*start*[,*len*])	Alias for substring.
SUBSTRING(*s*,*start*[,*len*])	Returns the substring starting at *start* (1=left) with length *len*.
UCASE(*s*)	Converts *s* to uppercase.
LOWER(*s*)	Converts *s* to lowercase.
UPPER(*s*)	Converts *s* to uppercase.

Table 19: Date/Time Functions Supported

Date/Time Function	Description
CURDATE( )	Returns the current date.
CURTIME( )	Returns the current time.
DATEDIFF(*string*, *datetime1*, *datetime2*)	Returns the count of units of time elapsed from *datetime1* to *datetime2*. The string indicates the unit of time and can have the following values: • 'ms'='millisecond' • 'ss'='second' • 'mi'='minute' • 'hh'='hour' • 'dd'='day' • 'mm'='month' • 'yy' = 'year' Both the long and short form of the strings can be used.
DAYNAME(*date*)	Returns the name of the day.
DAYOFMONTH(*date*)	Returns the day of the month (1-31).
DAYOFWEEK(*date*)	Returns the day of the week (1 means Sunday).
DAYOFYEAR(*date*	Returns the day of the year (1-366).
HOUR(*time*)	Returns the hour (0-23).
MINUTE(*time*)	Returns the minute (0-59).
MONTH(*date*)	Returns the month (1-12).
MONTHNAME(*date*)	Returns the name of the month.
NOW( )	Returns the current date and time as a timestamp.
QUARTER(*date*)	Returns the quarter (1-4).
SECOND(*time*)	Returns the second (0-59).
WEEK(*date*)	Returns the week of this year (1-53).
YEAR(*date*)	Returns the year.
CURRENT_DATE	Returns the current date.
CURRENT_TIME	Returns the current time.
CURRENT_TIMESTAMP	Returns the current timestamp.

Table 20: System/Connection Functions Supported

System/Connection Function	Description
DATABASE( )	Returns the name of the database of this connection.
USER( )	Returns the user name of this connection.
CURRENT_USER	SQL standard function, returns the user name of this connection.
CURSESSIONID( )	Returns the ID of the session (connection) on which this function was invoked.
IDENTITY( )	Returns the last identity value that was inserted by this connection.

Table 21: System Functions Supported

System Function	Description
IFNULL(*expr*,*value*)	If *expr* is NULL, then *value* is returned; otherwise the result of *expr* is returned. See COALESCE(*expr1*, *expr2*, ...) in this table for evaluating multiple expressions.
CONVERT(*term*,*type*)	Converts *term* to another data type.
CAST(term AS *type*)	Converts *term* to another data type.
COALESCE(expr1,expr2, ...)	If *expr1* is not Null, then it is returned; otherwise, *expr2* is evaluated and, if not Null, it is returned, and so on. This is an ANSISQL standard system function.
NULLIF(*value1*,*value2*)	If *value1* equals *value2*, then Null is returned; otherwise, *value1* is returned.
CASE *value1* WHEN *value2* THEN *value3* [ELSE *value4*] END	When *value1* equals *value2*, then *value3* is returned; otherwise, *value4* or Null is returned in the absence of ELSE.
CASE WHEN *expr1* THEN *value1* [WHEN *expr2* THEN *value2*] [ELSE *value4*] END	When *expr1* is true, then *value1* is returned (optionally repeated for more cases); otherwise *value4* or Null is returned in the absence of ELSE.
EXTRACT ({YEAR | MONTH | DAY | HOUR | MINUTE| SECOND} FROM *datetime_value*)	Any of the date and time terms can be extracted from *datetime_value*.
POSITION(*string_expression1* IN *string_expression2*)	If *string_expression1* is a sub-string of *string_expression2*, then the position of the sub-string, counting from one, is returned; otherwise, 0 is returned.
SUBSTRING(*string_expression* FROM *numeric_expression1* [FOR *numeric_expression2*])	*string_expression* is returned from the *numeric_expression1* starting location. Optionally, *numeric_expression2* specifies the length of the substring.
TRIM([{LEADING | TRAILING | BOTH}] FROM *string_expression*)	When returned, either the leading or trailing spaces, or both, are trimmed from *string_expression*.

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 table describes supported conditions.

Table 22: Conditions

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

Subqueries

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

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

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

IN Predicate

Purpose

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

Tests the cardinality of a subquery. It 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

Determines 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 table2t_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

This statement returns data about employees whose salaries exceed their department average. It 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 example specifies 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 example finds the department number (`deptno`) with multiple employees:

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

Example D

This example correlates a table with itself:

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

5

getTypeInfo()

This chapter provides results returned from the DataBaseMetaData.getTypeInfo() method for the drivers. The getTypeInfo() method returns information about data types supported by a particular database. The information in this chapter is organized by driver, and within each section, the results are organized alphabetically for each TYPE_NAME column.

For details, see the following topics:

- DB2 Driver
- Informix Driver
- MySQL Driver
- Oracle Driver
- PostgreSQL Driver
- Progress OpenEdge Driver
- SQL Server Driver
- Sybase Driver
- The Driver for Apache Hive
- Greenplum Driver
- Salesforce Driver

DB2 Driver

The following table provides getTypeInfo() results for all DB2 databases supported by the DB2 driver. Refer to "DB2 Driver" in the *DataDirect Connect Series for JDBC User's Guide* for more information.

Table 23: getTypeInfo() for DB2

TYPE_NAME = bigint [1] 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 = NULL PRECISION = 19 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = binary [1] AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = -2 (BINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = BINARY(X' LITERAL_SUFFIX = ') LOCAL_TYPE_NAME = binary MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[1] Supported only for DB2 v9.1 and higher for z/OS.

TYPE_NAME = blob [2] AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = (*length*) DATA_TYPE = 2004 (BLOB) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = BLOB MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 1 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = char AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = (*length*) DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = char MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 254 (DB2 for Linux/UNIX/Windows), 255 (DB2 for z/OS), 32765 (DB2 for i) SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[2] Supported only for DB2 v8.1 and higher for Linux/UNIX/Windows, DB2 for z/OS, and DB2 for i.

TYPE_NAME = char() for bit data AUTO_INCREMENT = NULL NULL CASE_SENSITIVE = false CREATE_PARAMS = (*length*) DATA_TYPE = -2 (BINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = X' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = char() for bit data MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 254 (DB2 for Linux/UNIX/Windows), 254 (DB2 for z/OS), 32765 (DB2 for i) SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = clob AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = (*length*) DATA_TYPE = 2005 (CLOB) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = clob MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 1 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = date AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = false LITERAL_PREFIX = {d ' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = date MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = dbclob [3] AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = (*length*) (DB2 for Linux/UNIX/Windows and DB2 for z/OS), (*length*) CCSID 13488 (DB2 for i) DATA_TYPE = 2005 (CLOB) [4] FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = dbclob MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 1 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = decfloat [5] AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision* DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = NULL MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 34 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

[3] Supported only for DB2 v8.1 and higher for Linux/UNIX/Windows, DB2 for z/OS, and DB2 for i.
[4] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: 2011 (NCLOB) (if using Java SE 6 or higher) or 2005 (CLOB) (if using another JVM).
[5] Supported only for DB2 V9.5 and higher for Linux/UNIX/Windows, DB2 v9.1 for z/OS, and DB2 for i 6.1.

TYPE_NAME = decimal AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = (*precision*,*scale*) DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = decimal MAXIMUM_SCALE = 31	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 31 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = double AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = double MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 15 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = float AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 6 (FLOAT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = float MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 15 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = graphic AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *length* DATA_TYPE = 1 (CHAR) [6] FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = char MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 127 (DB2 for Linux/UNIX/Windows), 127 (DB2 for z/OS), 16352 (DB2 for i) SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = integer AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = integer MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

[6] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: -15 (NCHAR) (if using Java SE 6 or higher) or 1 (CHAR) (if using another JVM).

TYPE_NAME = long varchar	
AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = long varchar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32700 (DB2 for Linux/UNIX/Windows), 32704 (DB2 for z/OS), 32700 (DB2 for i) SEARCHABLE = 1 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = long varchar for bit data	
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 = long varchar for bit data MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32700 (DB2 for Linux/UNIX/Windows), 32698 (DB2 for z/OS), 32739 (DB2 for i) SEARCHABLE = 1 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = long vargraphic AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *length* DATA_TYPE = -1 (LONGVARCHAR) [7] FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = longvarchar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16352 SEARCHABLE = 1 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = numeric AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = (*precision*,*scale*) DATA_TYPE = 2 (NUMERIC) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = numeric MAXIMUM_SCALE = 31	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX =10 PRECISION = 31 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = real 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(4) MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 7 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

[7] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: -16 (LONGNVARCHAR) (if using Java SE 6 or higher) or -1 (LONGVARCHAR) (if using another JVM).

TYPE_NAME = rowid [8] AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = not null generated always DATA_TYPE = -2 (Binary) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = rowid MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 0 NUM_PREC_RADIX = NULL PRECISION = 40 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true
TYPE_NAME = smallint AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 5 (SMALLINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = smallint MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 5 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = time AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 92 (TIME) FIXED_PREC_SCALE = false LITERAL_PREFIX = {t ' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = time MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 8 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[8] Supported only for DB2 for z/OS and DB2 for i5/OS V5R2 and higher.

TYPE_NAME = timestamp AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = {ts ' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = timestamp MAXIMUM_SCALE = 6	MINIMUM_SCALE = 6 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 26 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = varbinary [9] AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = -3 (VARVINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = VARBINARY(X' LITERAL_SUFFIX = ') LOCAL_TYPE_NAME = varbinary MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32703 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[9] Supported only for DB2 v9.1 for z/OS.

TYPE_NAME = varchar AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = (*max length*) DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = varchar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32762 (DB2 for Linux/UNIX/Windows), 32698 (DB2 for z/OS), 32739 (DB2 for i) SEARCHABLE = 3 (DB2 for Linux/UNIX/Windows), 1 (DB2 for z/OS), 1 (DB2 for i) SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = varchar() for bit data AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = (*max length*) DATA_TYPE = -3 (VARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = X' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = varchar() for bit data MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32762 (DB2 for Linux/UNIX/Windows), 32698 (DB2 for z/OS), 32739 (DB2 for i) SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = vargraphic AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *length* DATA_TYPE = 12 (VARCHAR) [10] 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 = 16352 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = xml [11] AUTO_INCREMENT = false CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = 2005 (CLOB) or 2009 (SQLXML) [12] FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = xml MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 1 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

Informix Driver

The following table provides getTypeInfo() results for all Informix databases supported by the Informix driver. Refer to "Informix Driver" in the *DataDirect Connect Series for JDBC User's Guide* for more information.

[10] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: -9 (NVARCHAR) (if using Java SE 6 or higher) or 12 (VARCHAR) (if using another JVM).
[11] Supported only for DB2 V9.1 and higher for Linux/UNIX/Windows and DB2 v9.1 for z/OS.
[12] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: 2009 (SQLXML) (if using Java SE 6 or higher) or 2005 (CLOB) (if using another JVM). In addition, the XMLDescribeType property can override driver mappings.

Table 24: getTypeInfo() for Informix

TYPE_NAME = blob AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 2004 (BLOB) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = blob MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = boolean AUTO_INCREMENT = 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 = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 1 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = byte AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = byte 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 = char AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *length* DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = char MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32766 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = clob AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = 2005 (CLOB) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = clob 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 = date AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = false LITERAL_PREFIX = {d ' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = date MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = datetime hour to second AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 92 (TIME) FIXED_PREC_SCALE = false LITERAL_PREFIX = {t ' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = datetime hour to second MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 8 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = datetime year to day AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = false LITERAL_PREFIX = {d ' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = datetime year to day MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = datetime year to fraction(5) AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = {ts ' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = datetime year to fraction(5) MAXIMUM_SCALE = 5	MINIMUM_SCALE = 5 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 25 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = datetime year to second AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = {ts ' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = datetime year to second MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 19 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = decimal AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision*,*scale* DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = decimal MAXIMUM_SCALE = 32	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 32 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = float AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 6 (FLOAT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = float MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 15 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = int8 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 = int8 MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 19 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = integer AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = integer MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = lvarchar AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL (Informix 9.2, 9.3), *max length* (Informix 9.4, 10) DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = lvarchar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2048 (Informix 9.2, 9.3), 32739 (Informix 9.4, 10) SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = money AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision*,*scale* DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = true LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = money MAXIMUM_SCALE = 32	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 32 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = nchar AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *length* DATA_TYPE = 1 (CHAR) [13] FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = nchar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32766 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = nvarchar AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *max length* DATA_TYPE = 12 (VARCHAR) [14] FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = nvarchar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 254 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[13] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: -15 (NCHAR) (if using Java SE 6 or higher) or 1 (CHAR) (if using another JVM).

[14] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: -9 (NVARCHAR) (if using Java SE 6 or higher) or 12 (VARCHAR) (if using another JVM).

TYPE_NAME = serial AUTO_INCREMENT = true CASE_SENSITIVE = false CREATE_PARAMS = start DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = serial MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = serial8 AUTO_INCREMENT = true CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -5 (BIGINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = serial8 MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 19 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = smallfloat 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 = smallfloat MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 7 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = smallint AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 5 (SMALLINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = smallint MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 5 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = text AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = text MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = varchar AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *max length* DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = varchar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 254 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

MySQL Driver

The following table provides getTypeInfo() results for MySQL 5.0.*x* and 5.1. Refer to "MySQL Driver" in the *DataDirect Connect Series for JDBC User's Guide* for more information.

Table 25: getTypeInfo() for MySQL

TYPE_NAME = bigint AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision* 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 = bigint unsigned AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision* DATA_TYPE = -5 (BIGINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = BIGINT UNSIGNED MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 20 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true

TYPE_NAME = binary AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = -2 (BINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = 0x LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = BINARY MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = bit AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = -2 (BINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = b' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = BIT MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 64 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = blob AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = 0x LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = BLOB MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 65535 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = char AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = CHAR MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = date AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = DATE MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = datetime AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = DATETIME MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 19 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = decimal AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision,scale* DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = DECIMAL MAXIMUM_SCALE = 30	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 65 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = decimal unsigned AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision,scale* DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = DECIMAL UNSIGNED MAXIMUM_SCALE = 30	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 65 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true
TYPE_NAME = double AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = DOUBLE MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 15 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = double unsigned AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = DOUBLE UNSIGNED MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 15 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true
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 = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 7 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = float unsigned 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 UNSIGNED MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 7 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true

TYPE_NAME = integer AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision* DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = INTEGER MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = integer unsigned AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision* DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = INTEGER UNSIGNED MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true
TYPE_NAME = longblob AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = 0x LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = LONGBLOB 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 = longtext AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = LONGTEXT MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = mediumblob AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = 0x LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = MEDIUMBLOB MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777215 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = mediumint AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision* DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = MEDIUMINT MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 8 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = mediumint unsigned AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision* DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = MEDIUMINT UNSIGNED MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 8 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true
TYPE_NAME = mediumtext AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = MEDIUMTEXT MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777215 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = smallint AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision* 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 = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = smallint unsigned AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision* DATA_TYPE = 5 (SMALLINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = SMALLINT UNSIGNED MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 5 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true
TYPE_NAME = text AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = TEXT MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 65535 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = time AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 92 (TIME) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = TIME MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 8 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 = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = TIMESTAMP MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 19 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = tinyblob AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = 0x LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = TINYBLOB MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = tinyint AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision* 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 = tinyint unsigned AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision* DATA_TYPE = -6 (TINYINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = TINYINT UNSIGNED 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 = true
TYPE_NAME = tinytext AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = TINYTEXT MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = varbinary AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = -3 (VARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = 0x LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = VARBINARY MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = varchar AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* 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 = 255 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = year AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision* DATA_TYPE = 5 (SMALLINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = YEAR MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 4 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true

Oracle Driver

The following table provides getTypeInfo() results for all Oracle databases supported by the Oracle driver. Refer to "Oracle Driver" in the *DataDirect Connect Series for JDBC User's Guide* for more information.

Table 26: getTypeInfo() for Oracle

TYPE_NAME = bfile AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 2004 (BLOB) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = bfile 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 = binary_double [15] AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = binary_double MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 15 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = binary_float [15] 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 = binary_float MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 7 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

[15] Supported only for Oracle 10*g* and higher.

TYPE_NAME = blob AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 2004 (BLOB) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = blob 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 = char AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *length* DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = char MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2000 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = clob AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = 2005 (CLOB) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = clob 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 = date AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = {ts ' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = date MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 19 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = long AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = long 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 = long raw AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = long raw 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 = nchar AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *length* DATA_TYPE = 1 (CHAR) [16] FIXED_PREC_SCALE = false LITERAL_PREFIX = N' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = nchar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32766 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = nclob AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = 2005 (CLOB) [17] FIXED_PREC_SCALE = false LITERAL_PREFIX = N' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = nclob 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 = number AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision*,*scale* DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = number MAXIMUM_SCALE = 127	MINIMUM_SCALE = -84 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 38 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

[16] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: -15 (NCHAR) (if using Java SE 6 or higher) or 1 (CHAR) (if using another JVM).

[17] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: 2001 (NCLOB) (if using Java SE 6 or higher) or 2005 (CLOB) (if using another JVM).

TYPE_NAME = number 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 = number MAXIMUM_SCALE = 127	MINIMUM_SCALE = -84 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 38 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = nvarchar2[18] AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *max length* DATA_TYPE = 12 (VARCHAR) [19] FIXED_PREC_SCALE = false LITERAL_PREFIX = N' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = nvarchar2 MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 4000 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = raw[18] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *max length* DATA_TYPE = -3 (VARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = raw MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2000 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[18] Supported as an extended data type for Oracle 12*c* and higher. Refer to "Using Extended Data Types" in the Oracle chapter of the *Connect for JDBC User's Guide* for details.

[19] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: -9 (NVARCHAR) (if using Java SE 6 or higher) or 12 (VARCHAR) (if using another JVM).

TYPE_NAME = timestamp [20] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *fractional_seconds_precision* DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = {ts ' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = timestamp MAXIMUM_SCALE = 9	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 19 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = timestamp with local time zone [20] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *fractional_seconds_precision* DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = {ts ' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = timestamp with local time zone MAXIMUM_SCALE = 9	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 19 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = timestamp with time zone [20] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *fractional_seconds_precision* DATA_TYPE = 12 (VARCHAR) or 93 (TIMESTAMP) [21] FIXED_PREC_SCALE = false LITERAL_PREFIX = {ts ' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = timestamp with time zone MAXIMUM_SCALE = 9	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 19 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[20] Supported only for Oracle 9*i* and higher.

[21] When FetchTSWTZasTimestamp=false (default), this data type is mapped to the JDBC VARCHAR data type; when FetchTSWTZasTimestamp=true, it is mapped to the JDBC TIMESTAMP data type.

TYPE_NAME = urowid [20] AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *max length* DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = urowid MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 4000 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = varchar2[20,18] AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *max length* DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = varchar2 MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 4000 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = xmltype [22] AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = 2005 (CLOB) [23] FIXED_PREC_SCALE = false LITERAL_PREFIX = xmltype(' LITERAL_SUFFIX = ') LOCAL_TYPE_NAME = xmltype 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

[22] Supports XMLType columns, except those with binary or object relational storage.

[23] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: 2009 (SQLXML) (if using Java SE 6 or higher) or 2005 (CLOB) (if using another JVM).

PostgreSQL Driver

The following table provides getTypeInfo() results for PostgreSQL databases supported by the driver. Refer to "PostgreSQL Driver" in the *DataDirect Connect Series for JDBC User's Guide* for more information.

Table 27: getTypeInfo() for PostgreSQL

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 = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 19 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = bigserial AUTO_INCREMENT = true CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -5 (BIGINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Bigserial MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 0 NUM_PREC_RADIX = 10 PRECISION = 19 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = bit [24] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -2 (BINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Bit MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 83886080 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = bit varying AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *max length* DATA_TYPE = -3 (VARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Bit varying MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 83886080 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 = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Boolean MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[24] Bit maps to -7 (BIT) when the length for the bit is 1. If the length is greater than 1, the driver maps the column to BINARY.

TYPE_NAME = bytea AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Bytea 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 = character AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Character MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10485760 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = character varying [25] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *max length* DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Character varying MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10485760 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[25] Columns of this type will be described as VARCHAR when precision is 4000 or less. If precision is greater than 4000, columns will be described as LONGVARCHAR.

TYPE_NAME = date AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = false LITERAL_PREFIX = {d' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = DATE MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = double precision AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = precision DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Double precision MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 2 PRECISION = 53 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = integer AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Integer MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = numeric AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *precision, scale* DATA_TYPE = 2 (NUMERIC) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Numeric MAXIMUM_SCALE = 0	MINIMUM_SCALE = 999 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1000 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = real AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision* DATA_TYPE = 7 (REAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Real MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 2 PRECISION = 24 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = serial AUTO_INCREMENT = true CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Serial MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 0 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = smallint AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 5 (SMALLINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Smallint MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 5 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = text AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Text MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1073741823 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = time AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *fractional_seconds_precision* DATA_TYPE = 93 (TIME) FIXED_PREC_SCALE = false LITERAL_PREFIX = {t' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = Time MAXIMUM_SCALE = 6	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 15 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = time with time zone AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *fractional_seconds_precision* DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = {t' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = Time with time zone MAXIMUM_SCALE = 6	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 22 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = timestamp AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *fractional_seconds_precision* DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = {ts' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = Timestamp MAXIMUM_SCALE = 6	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 26 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = timestamp with time zone AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *fractional_seconds_precision* DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = {ts' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = Timestamp with time zone MAXIMUM_SCALE = 6	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 33 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = XML [26] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 2009 (SQLXML) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = XML MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 10485760 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

Progress OpenEdge Driver

The following table provides getTypeInfo() results for the Progress OpenEdge® databases supported by the Progress OpenEdge driver. Refer to "Progress OpenEdge Driver" in the *DataDirect Connect Series for JDBC User's Guide* for more information.

[26] The XML data type is supported in PostgreSQL versions 8.3 and higher.

Table 28: getTypeInfo() for Progress OpenEdge

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 = binary AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = -2 (BINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = BINARY MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2000 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = bit AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -7 (BIT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = BIT MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 1 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = blob AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = LONGVARBINARY MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1000000000 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = character AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *length* DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = CHAR MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2000 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = clob AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = LONGVARCHAR MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1000000000 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = date AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = DATE MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = double precision AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = DOUBLE PRECISION MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 15 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = float AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = FLOAT MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 15 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = integer AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = INTEGER MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = lvarbinary AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = LONGVARBINARY MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2000000 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = lvarchar AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = LONGVARCHAR MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2000000 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = numeric AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *precision*, *scale* DATA_TYPE = 2 (NUMERIC) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = NUMERIC MAXIMUM_SCALE = 50	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 50 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = real 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 = REAL MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 7 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = smallint AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 5 (SMALLINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = SMALLINT MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 5 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = time AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 92 (TIME) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = TIME MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 12 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 = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = TIMESTAMP MAXIMUM_SCALE = 0	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 = timestamp with time zone AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = CHAR MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2000 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = tinyint 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 = varbinary AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = -3 (VARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = VARBINARY MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 31960 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = varchar AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *length* 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 = 31960 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

SQL Server Driver

The following table provides getTypeInfo() results for all Microsoft SQL Server and Microsoft Windows Azure SQL Database databases supported by the SQL Server driver. Refer to "Microsoft SQL Server Driver" in the *DataDirect Connect Series for JDBC User's Guide* for more information.

Table 29: getTypeInfo() for SQL Server

TYPE_NAME = bigint[27] 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 = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = bigint identity[27] AUTO_INCREMENT = true CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -5 (BIGINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = bigint identity MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 0 NUM_PREC_RADIX = 10 PRECISION = 19 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

[27] Supported only for Microsoft SQL Server 2000 and higher and Microsoft Windows Azure SQL Database.

TYPE_NAME = binary AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = -2 (BINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = 0x LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = binary MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 8000 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = bit AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -7 (BIT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = bit MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = char AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = char MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 8000 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = date AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = date MAXIMUM_SCALE = 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
TYPE_NAME = datetime AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = datetime MAXIMUM_SCALE = 3	MINIMUM_SCALE = 3 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 23 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = datetime2 AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = datetime2 MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 27 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = datetimeoffset AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) or 93 (TIMESTAMP)[28] FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = datetimeoffset MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 34 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = decimal AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision*,*scale* DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = decimal MAXIMUM_SCALE = 28 (SQL Server 7), 38 (SQL Server 2000 and higher)[29] , 38 (Microsoft Windows Azure SQL Database)[29]	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 28 (SQL Server 7), 38 (SQL Server 2000 and higher)[29] , 38 (Microsoft Windows Azure SQL Database)[29] SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

[28] When FetchTSWTZasTimestamp=false, the data type that is returned by DATA_TYPE is VARCHAR; when FetchTSWTZasTimestamp=true, the data type that is returned is TIMESTAMP.

[29] Configurable server option for Microsoft SQL Server 2000 and higher and Microsoft Windows Azure SQL Database.

TYPE_NAME = decimal() identity AUTO_INCREMENT = true CASE_SENSITIVE = false CREATE_PARAMS = *precision* DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = decimal() identity MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 0 NUM_PREC_RADIX = 10 PRECISION = 28 (SQL Server 7), 38 (SQL Server 2000 and higher) SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = float AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 6 (FLOAT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = float MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 2 PRECISION = 53 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = image AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = 0x LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = image 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 = 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 = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = int identity AUTO_INCREMENT = true CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = int identity MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 0 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = money AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = true LITERAL_PREFIX = $ LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = money MAXIMUM_SCALE = 4	MINIMUM_SCALE = 4 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 19 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = nchar AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = 1 (CHAR)[30] FIXED_PREC_SCALE = false LITERAL_PREFIX = N' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = nchar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 4000 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = ntext AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR)[31] FIXED_PREC_SCALE = false LITERAL_PREFIX = N' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = ntext MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1073741823 SEARCHABLE = 1 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[30] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: -15 (NCHAR) (if using Java SE 6 or higher) or 1 (CHAR) (if using another JVM).

[31] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: -16 (LONGNVARCHAR) (if using Java SE 6 or higher) or -1 (LONGVARCHAR) (if using another JVM).

TYPE_NAME = numeric AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision*,*scale* DATA_TYPE = 2 (NUMERIC) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = numeric MAXIMUM_SCALE = 28 (SQL Server 7), 38 (SQL Server 2000 and higher)[32] , 38 (Microsoft Windows Azure SQL Database)[32]	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 28 (SQL Server 7), 38 (SQL Server 2000 and higher)[32] , 38 (Microsoft Windows Azure SQL Database)[32] SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = numeric() identity AUTO_INCREMENT = true CASE_SENSITIVE = false CREATE_PARAMS = *precision* DATA_TYPE = 2 (NUMERIC) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = numeric() identity MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 0 NUM_PREC_RADIX = 10 PRECISION = 28 (SQL Server 7.0), 38 (SQL Server 2000 and higher), 38 (Microsoft Windows Azure SQL Database) SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

[32] Configurable server option for Microsoft SQL Server 2000 and higher and Microsoft Windows Azure SQL Database.

TYPE_NAME = nvarchar AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *max length* DATA_TYPE = 12 (VARCHAR)[33] FIXED_PREC_SCALE = false LITERAL_PREFIX = N' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = nvarchar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 4000 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = nvarchar(max)[34] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR)[35] FIXED_PREC_SCALE = false LITERAL_PREFIX = N' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = nvarchar(max) MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1073741823 SEARCHABLE = 1 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = real 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 = real MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 2 PRECISION = 24 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

[33] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: -9 (NVARCHAR) (if using Java SE 6 or higher) or 12 (VARCHAR) (if using another JVM).

[34] Supported only for Microsoft SQL Server 2005 and higher and Microsoft Windows Azure SQL Database.

[35] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: -16 (LONGNVARCHAR) (if using Java SE 6 or higher) or -1 (LONGVARCHAR) (if using another JVM).

TYPE_NAME = smalldatetime AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = smalldatetime MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = smallint AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 5 (SMALLINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = smallint MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 5 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = smallint identity AUTO_INCREMENT = true CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 5 (SMALLINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = smallint identity MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 0 NUM_PREC_RADIX = 10 PRECISION = 5 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = smallmoney AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = true LITERAL_PREFIX = $ LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = smallmoney MAXIMUM_SCALE = 4	MINIMUM_SCALE = 4 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = sql_variant[36] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = sql_variant MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 8000 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = sysname AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = N' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = sysname MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 0 NUM_PREC_RADIX = NULL PRECISION = 128 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[36] Supported only for Microsoft SQL Server 2000 and higher and Microsoft Windows Azure SQL Database.

TYPE_NAME = text AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = text MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 1 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = time AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = time MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16 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 = -2 (BINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = 0x LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = timestamp MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 0 NUM_PREC_RADIX = NULL PRECISION = 8 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = tinyint 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 = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true
TYPE_NAME = tinyint identity AUTO_INCREMENT = true CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -6 (TINYINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = tinyint identity MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 0 NUM_PREC_RADIX = 10 PRECISION = 3 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true
TYPE_NAME = uniqueidentifier AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 1(CHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = uniqueidentifier MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 36 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = varbinary AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *max length* DATA_TYPE = -3 (VARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = 0x LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = varbinary MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 8000 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = varbinary(max)[37] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = 0x LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = varbinary(max) 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 = varchar AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *max length* 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 = 8000 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[37] Supported only for Microsoft SQL Server 2005 and higher and Microsoft Windows Azure SQL Database.

TYPE_NAME = varchar(max)[38] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = varchar(max) MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 1 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = xml[38] AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) or 2009 (SQLXML)[39] FIXED_PREC_SCALE = false LITERAL_PREFIX = N' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = xml MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1073741823 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

Sybase Driver

The following table provides getTypeInfo() results for all Sybase databases supported by the Sybase driver. Refer to "Sybase Driver" in the *DataDirect Connect Series for JDBC User's Guide* for more information.

[38] Supported only for Microsoft SQL Server 2005 and higher and Microsoft Windows Azure SQL Database.
[39] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: 2009 (SQLXML) (if using Java SE 6 or higher) or -1 (LONGVARCHAR) (if using another JVM). In addition, the XMLDescribeType property can override driver mappings.

Table 30: getTypeInfo() for Sybase

TYPE_NAME = bigint [40] 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 = NULL PRECISION = 19 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = bigdatetime [41] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = bigdatetime MAXIMUM_SCALE = 6	MINIMUM_SCALE = 6 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 26 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[40] Supported only for Sybase 15.0 and higher.
[41] Supported only for Sybase 15.5 and higher.

TYPE_NAME = bigtime [42] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 92 (TIME), 93 (TIMESTAMP) [43] FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = bigtime MAXIMUM_SCALE = 6	MINIMUM_SCALE = 6 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 15 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = binary AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = -2 (BINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = 0x LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = binary MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 (Sybase 11.*x*, 12.0) [44] , 2048 (Sybase 12.5 and higher) [44] SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

42 Supported only for Sybase 15.5 and higher.

43 When FetchTWFSasTime=true, this Sybase data type is mapped to the JDBC TIME data type. When FetchTWFSasTime=false, this Sybase data type is mapped to the JDBC TIMESTAMP data type.

44 For Sybase 12.5.1 and higher, precision is determined by the server page size.

TYPE_NAME = bit AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -7 (BIT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = bit MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 0 NUM_PREC_RADIX = NULL PRECISION = 1 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = char AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *length* DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = char MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 (Sybase 11.*x*, 12.0) [45] , 2048 (Sybase 12.5 and higher) [45] SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = date [46] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = date MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[45] For Sybase 12.5.1 and higher, precision is determined by the server page size.
[46] Supported only for Sybase 12.5.1 and higher.

TYPE_NAME = datetime AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = datetime MAXIMUM_SCALE = 3	MINIMUM_SCALE = 3 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 23 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = decimal AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision*,*scale* DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = decimal MAXIMUM_SCALE = 38	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 38 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = float AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 6 (FLOAT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = float MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 15 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = image AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = 0x LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = image MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 1 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 = NULL PRECISION = 10 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = money AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = true LITERAL_PREFIX = $ LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = money MAXIMUM_SCALE = 4	MINIMUM_SCALE = 4 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 19 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = numeric AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision,scale* DATA_TYPE = 2 (NUMERIC) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = numeric MAXIMUM_SCALE = 38	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 38 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = real 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 = real MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 7 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = smalldatetime AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = smalldatetime MAXIMUM_SCALE = 3	MINIMUM_SCALE = 3 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = smallint AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 5 (SMALLINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = smallint MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 5 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = smallmoney AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = true LITERAL_PREFIX = $ LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = smallmoney MAXIMUM_SCALE = 4	MINIMUM_SCALE = 4 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = sysname AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *max length* DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = sysname MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 30 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = text AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = text MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 1 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = time [47] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 92 (TIME) or 93 (TIMESTAMP) [48] FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = time MAXIMUM_SCALE = 3	MINIMUM_SCALE = 3 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 12 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 = -3 (VARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = 0x LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = timestamp MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 8 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[47] Supported only for Sybase 12.5.1 and higher.
[48] When FetchTWFSasTime=time, this Sybase data type is mapped to the JDBC TIME data type. When FetchTWFSasTime=false, this Sybase data type is mapped to the JDBC TIMESTAMP data type.

TYPE_NAME = tinyint 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 = NULL PRECISION = 3 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true
TYPE_NAME = unsigned bigint [49] 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 = unsigned bigint MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 20 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true
TYPE_NAME = unsigned int [49] 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 = unsigned int MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true

[49] Supported only for Sybase 15.0 and higher.

TYPE_NAME = unsigned smallint [49] 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 = unsigned smallint MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 5 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true
TYPE_NAME = unichar [50] AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *length* DATA_TYPE = 1 (CHAR) or -15 (NCHAR) [51] FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = unichar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION =2048 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[50] Supported only for Sybase 12.5 and higher.

[51] If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: -15 (NCHAR) (if using Java SE 6 or higher) or 1 (CHAR) (if using another JVM).

TYPE_NAME = unitext [52] AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) or 2011 LONGNVARCHAR [53] FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = unitext MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 1 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = univarchar [54] AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *max length* DATA_TYPE = 12 (VARCHAR) or -9 (NVARCHAR)[55] FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = univarchar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2048 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

52 Supported only for Sybase 15.0 and higher.
53 If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: 2011 (LONGNVARCHAR) (if using Java SE 6 or higher) or -1 (LONGVARCHAR) (if using another JVM).
54 Supported only for Sybase 12.5 and higher.
55 If JDBCBehavior=0, the value returned for DATA_TYPE depends on the JVM used by the application: -9 (NVARCHAR) (if using Java SE 6 or higher) or 12 (VARCHAR) (if using another JVM).

TYPE_NAME = varbinary AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *max length* DATA_TYPE = -3 (VARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = 0x LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = varbinary MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 (Sybase 11.*x*, 12.0) [56] , 2048 (Sybase 12.5 and higher) [56] SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = varchar AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *max length* DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = varchar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 (Sybase 11.*x*, 12.0), [57] 2048 (Sybase 12.5 and higher) [57] SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

The Driver for Apache Hive

The following table provides getTypeInfo() results for all sources supported by The Driver for Apache Hive. Refer to "The Driver for Apache Hive" in the *DataDirect Connect Series for JDBC User's Guide* for more information.

[56] For Sybase 12.5.1 and higher, precision is determined by the server page size.
[57] For Sybase 12.5.1 and higher, precision is determined by the server page size.

Table 31: getTypeInfo() for The Driver for Apache Hive

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 = L 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 = binary [58] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -3 (VARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = binary MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 214748647 SEARCHABLE = 0 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 = 10 PRECISION = 1 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[58] Supported only for HiveServer1.

TYPE_NAME = char[59] AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = true LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = char MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = date[60] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = true LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = date MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = decimal[61] 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 = 38	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 38 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

[59] Supported for Apache Hive 0.13 and higher.
[60] Supported for Apache Hive 0.12 and higher.
[61] For Apache Hive 0.13 and higher, supported as either a user-defined, variable precision data type or as a fixed precision data type. For Apache Hive 0.11 and 0.12, supported as a fixed precision data type only.

TYPE_NAME = double	
AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = double MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 15 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
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 = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 7 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
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 = smallint	
AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 5 (SMALLINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = S 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 = string[62]	
AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) *or* -1 (LONGVARCHAR)[63] FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = string MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 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 = {ts' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = timestamp MAXIMUM_SCALE = 9	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 29 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[62] Maximum of 2 GB

[63] If the StringDescribeType connection property is set to `varchar` (the default), the String data type maps to VARCHAR. If StringDescribeType is set to `longvarchar`, String maps to LONGVARCHAR.

TYPE_NAME = tinyint	
AUTO_INCREMENT = false	MINIMUM_SCALE = 0
CASE_SENSITIVE = false	NULLABLE = 1
CREATE_PARAMS = NULL	NUM_PREC_RADIX = 10
DATA_TYPE = -6 (TINYINT)	PRECISION = 3
FIXED_PREC_SCALE = false	SEARCHABLE = 3
LITERAL_PREFIX = NULL	SQL_DATA_TYPE = NULL
LITERAL_SUFFIX = Y	SQL_DATETIME_SUB = NULL
LOCAL_TYPE_NAME = tinyint	UNSIGNED_ATTRIBUTE = false
MAXIMUM_SCALE = 0	
TYPE_NAME = varchar[60]	
AUTO_INCREMENT = NULL	MINIMUM_SCALE = 0
CASE_SENSITIVE = true	NULLABLE = 1
CREATE_PARAMS = NULL	NUM_PREC_RADIX = NULL
DATA_TYPE = 12 (VARCHAR)	PRECISION = 2147483647
FIXED_PREC_SCALE = false	SEARCHABLE = 3
LITERAL_PREFIX = '	SQL_DATA_TYPE = NULL
LITERAL_SUFFIX = '	SQL_DATETIME_SUB = NULL
LOCAL_TYPE_NAME = varchar	UNSIGNED_ATTRIBUTE = NULL
MAXIMUM_SCALE = 0	

Greenplum Driver

The following table provides getTypeInfo() results for Greenplum databases supported by the driver. Refer to "Greenplum Driver" in the *DataDirect Connect Series for JDBC User's Guide* for more information.

Table 32: getTypeInfo() for Greenplum

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 = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 19 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = bigserial AUTO_INCREMENT = true CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -5 (BIGINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Bigserial MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 0 NUM_PREC_RADIX = 10 PRECISION = 19 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = bit [64] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -2 (BINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Bit MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 83886080 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[64] Bit maps to -7 (BIT) when the length for the bit is 1. If the length is greater than 1, the driver maps the column to BINARY.

TYPE_NAME = bit varying AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *max length* DATA_TYPE = -3 (VARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Bit varying MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 83886080 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 = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Boolean MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = bytea AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Bytea 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 = character AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Character MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10485760 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = character varying [65] AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *max length* DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Character varying MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10485760 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = date AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = false LITERAL_PREFIX = {d' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = DATE MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

[65] Columns of this type will be described as VARCHAR when precision is 4000 or less. If precision is greater than 4000, columns will be described as LONGVARCHAR.

TYPE_NAME = double precision AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = precision DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Double precision MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 2 PRECISION = 53 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = integer AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Integer MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = numeric AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *precision, scale* DATA_TYPE = 2 (NUMERIC) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Numeric MAXIMUM_SCALE = 0	MINIMUM_SCALE = 999 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1000 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = real AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision* DATA_TYPE = 7 (REAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Real MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 2 PRECISION = 24 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = serial AUTO_INCREMENT = true CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Serial MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 0 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = smallint AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 5 (SMALLINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Smallint MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 5 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = text AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Text MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1073741823 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = time AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *fractional_seconds_precision* DATA_TYPE = 93 (TIME) FIXED_PREC_SCALE = false LITERAL_PREFIX = {t' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = Time MAXIMUM_SCALE = 6	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 15 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = time with time zone AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *fractional_seconds_precision* DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = {t' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = Time with time zone MAXIMUM_SCALE = 6	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 22 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = timestamp AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *fractional_seconds_precision* DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = {ts' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = Timestamp MAXIMUM_SCALE = 6	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 26 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = timestamp with time zone AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *fractional_seconds_precision* DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = {ts' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = Timestamp with time zone MAXIMUM_SCALE = 6	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 33 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

Salesforce Driver

The following table provides getTypeInfo() results for all sources supported by the Salesforce driver. Refer to "Salesforce Driver" in the *DataDirect Connect Series for JDBC User's Guide* for more information.

Table 33: getTypeInfo() for Salesforce

TYPE_NAME = AnyType AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = ANYTYPE MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = AutoNumber AUTO_INCREMENT = true CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = AUTONUMBER MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 0 NUM_PREC_RADIX = NULL PRECISION = 30 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = Binary AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = BINARY MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 5242880 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = CheckBox AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 16 (BOOLEAN) FIXED_PREC_SCALE = false LITERAL_PREFIX = LITERAL_SUFFIX = LOCAL_TYPE_NAME = CHECKBOX MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 0 NUM_PREC_RADIX = NULL PRECISION = 1 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = ComboBox AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = COMBOBOX MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = Currency AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision*, *scale* DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = false LITERAL_PREFIX = LITERAL_SUFFIX = LOCAL_TYPE_NAME = CURRENCY MAXIMUM_SCALE = 18	MINIMUM_SCALE = NULL NULLABLE = 0 NUM_PREC_RADIX = NULL PRECISION = 18 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = DataCategoryGroupReference AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = DATE MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = Date AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = DATE MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = DateTime AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = DATETIME MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 19 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = Email AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = EMAIL MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 80 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = HTML AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = HTML MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32000 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = ID AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = ID MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 0 NUM_PREC_RADIX = NULL PRECISION = 18 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = LongTextArea AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = LONGTEXTAREA MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32000 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = MultiSelectPickList AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = MULTISELECTPICKLIST MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = Number AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision*, *scale* DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = NUMBER MAXIMUM_SCALE = 18	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 18 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = Percent AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision*, *scale* DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = false LITERAL_PREFIX = LITERAL_SUFFIX = LOCAL_TYPE_NAME = PERCENT MAXIMUM_SCALE = 18	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 18 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = Phone AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = PHONE MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 40 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = PickList AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = PICKLIST MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = Reference AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = REFERENCE MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 18 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = Text AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *length* DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = TEXT MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = TextArea AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = TEXTAREA MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = Time AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 92 (TIME) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = TIME MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 8 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = URL AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = URL MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

6

Designing JDBC Applications for Performance Optimization

Developing performance-oriented JDBC applications is not easy. JDBC drivers do not throw exceptions to tell you when your code is running too slow. This chapter presents some general guidelines for improving JDBC application performance that have been compiled by examining the JDBC implementations of numerous shipping JDBC applications. These guidelines include:

- Use DatabaseMetaData methods appropriately
- Return only required data
- Select functions that optimize performance
- Manage connections and updates

Following these general guidelines can help you solve some common JDBC system performance problems, such as those listed in the following table.

Problem	Solution	See guidelines in...
Network communication is slow.	Reduce network traffic.	Using Database Metadata Methods on page 266
Evaluation of complex SQL queries on the database server is slow and can reduce concurrency.	Simplify queries.	Using Database Metadata Methods on page 266 Selecting JDBC Objects and Methods on page 270

Problem	Solution	See guidelines in...
Excessive calls from the application to the driver slow performance.	Optimize application-to-driver interaction.	Returning Data on page 268 Selecting JDBC Objects and Methods on page 270
Disk I/O is slow.	Limit disk I/O.	Managing Connections and Updates on page 273

In addition, most JDBC drivers provide options that improve performance, often with a trade-off in functionality. If your application is not affected by functionality that is modified by setting a particular option, significant performance improvements can be realized.

Note: The section describes functionality across a spectrum of data stores. In some cases, the functionality described may not apply to the driver or data store you are using. In addition, examples are drawn from a variety of drivers and data stores.

For details, see the following topics:

- Using Database Metadata Methods
- Returning Data
- Selecting JDBC Objects and Methods
- Managing Connections and Updates

Using Database Metadata Methods

Because database metadata methods that generate ResultSet objects are slow compared to other JDBC methods, their frequent use can impair system performance. The guidelines in this section will help you optimize system performance when selecting and using database metadata.

Minimizing the Use of Database Metadata Methods

Compared to other JDBC methods, database metadata methods that generate ResultSet objects are relatively slow. Applications should cache information returned from result sets that generate database metadata methods so that multiple executions are not needed.

Although almost no JDBC application can be written without database metadata methods, you can improve system performance by minimizing their use. To return all result column information *mandated* by the JDBC specification, a JDBC driver may have to perform complex queries or multiple queries to return the necessary result set for a single call to a database metadata method. These particular elements of the SQL language are performance-expensive.

Applications should cache information from database metadata methods. For example, call getTypeInfo() once in the application and cache the elements of the result set that your application depends on. It is unlikely that any application uses all elements of the result set generated by a database metadata method, so the cache of information should not be difficult to maintain.

Avoiding Search Patterns

Using null arguments or search patterns in database metadata methods results in generating time-consuming queries. In addition, network traffic potentially increases due to unwanted results. Always supply as many non-null arguments as possible to result sets that generate database metadata methods.

Because database metadata methods are slow, invoke them in your applications as efficiently as possible. Many applications pass the fewest non-null arguments necessary for the function to return success. For example:

```
ResultSet WSrs = WSdbmd.getTables(null, null, "WSTable", null);
```

In this example, an application uses the getTables() method to determine if the WSTable table exists. A JDBC driver interprets the request as: return all tables, views, system tables, synonyms, temporary tables, and aliases named "WSTable" that exist in any database schema inside the database catalog.

In contrast, the following request provides non-null arguments as shown:

```
String[] tableTypes = {"TABLE"};
WSdbmd.getTables("cat1", "johng", "WSTable", "tableTypes");
```

Clearly, a JDBC driver can process the second request more efficiently than it can process the first request.

Sometimes, little information is known about the object for which you are requesting information. Any information that the application can send the driver when calling database metadata methods can result in improved performance and reliability.

Using a Dummy Query to Determine Table Characteristics

Avoid using the getColumns() method to determine characteristics about a database table. Instead, use a dummy query with getMetadata().

Consider an application that allows the user to choose the columns to be selected. Should the application use getColumns() to return information about the columns to the user or instead prepare a dummy query and call getMetadata()?

Case 1: GetColumns() Method

```
ResultSet WSrc = WSc.getColumns(... "UnknownTable" ...);
// This call to getColumns will generate a query to
// the system catalogs... possibly a join
// which must be prepared, executed, and produce
// a result set
. . .
WSrc.next();
string Cname = getString(4);
. . .
// user must return N rows from the server
// N = # result columns of UnknownTable
// result column information has now been obtained
```

Case 2: GetMetadata() Method

```
// prepare dummy query
PreparedStatement WSps = WSc.prepareStatement
    ("SELECT * FROM UnknownTable WHERE 1 = 0");
// query is never executed on the server - only prepared
ResultSetMetaData WSsmd=WSps.getMetaData();
int numcols = WSrsmd.getColumnCount();
...
```

```
int ctype = WSrsmd.getColumnType(n)
...
// result column information has now been obtained
// Note we also know the column ordering within the
// table!  This information cannot be
// assumed from the getColumns example.
```

In both cases, a query is sent to the server. However, in Case 1, the potentially complex query must be prepared and executed, result description information must be formulated, and a result set of rows must be sent to the client. In Case 2, we prepare a simple query where we only return result set information. Clearly, Case 2 is the better performing model.

To somewhat complicate this discussion, let us consider a DBMS server that does not natively support preparing a SQL statement. The performance of Case 1 does not change but the performance of Case 2 improves slightly because the dummy query must be evaluated in addition to being prepared. Because the Where clause of the query always evaluates to FALSE, the query generates no result rows and should execute without accessing table data. For this situation, Case 2 still outperforms Case 1.

In summary, always use result set metadata to return table column information, such as column names, column data types, and column precision and scale. Only use the getColumns() method when the requested information cannot be obtained from result set metadata (for example, using the table column default values).

Returning Data

To return data efficiently, return only the data that you need and choose the most efficient method of doing so. The guidelines in this section will help you optimize system performance when retrieving data with JDBC applications.

Returning Long Data

Because retrieving long data across a network is slow and resource intensive, applications should not request long data unless it is necessary.

Most users do not want to see long data. If the user does want to see these result items, then the application can query the database again, specifying only the long columns in the Select list. This method allows the average user to return the result set without having to pay a high performance penalty for network traffic.

Although the best method is to exclude long data from the Select list, some applications do not formulate the Select list before sending the query to the JDBC driver (that is, some applications `SELECT * FROM` *`table_name`* `...`). If the Select list contains long data, most drivers are forced to return that long data at fetch time, even if the application does not ask for the long data in the result set. When possible, the designer should attempt to implement a method that does not return all columns of the table.

For example, consider the following code:

```
ResultSet rs = stmt.executeQuery(
   "SELECT * FROM Employees WHERE SSID = '999-99-2222'");
rs.next();
string name = rs.getString(1);
```

Remember that a JDBC driver cannot interpret an application's final intention. When a query is executed, the driver has no way to know which result columns an application will use. A driver anticipates that an application can request any of the result columns that are returned. When the JDBC driver processes the rs.next request, it will probably return at least one, if not more, result rows from the database server across the network. In this case, a result row contains all the column values for each row, including an employee photograph if the Employees table contains such a column. If you limit the Select list to contain only the employee name column, it results in decreased network traffic and a faster performing query at runtime. For example:

```
ResultSet rs = stmt.executeQuery(
   "SELECT name FROM Employees WHERE SSID = '999-99-2222'");
rs.next();
string name = rs.getString(1);
```

Additionally, although the getClob() and getBlob() methods allow the application to control how long data is returned in the application, the designer must realize that in many cases, the JDBC driver emulates these methods due to the lack of true Large Object (LOB) locator support in the DBMS. In such cases, the driver must return all the long data across the network before exposing the getClob() and getBlob() methods.

Reducing the Size of Returned Data

Sometimes long data must be returned. When this is the case, remember that most users do not want to see 100 KB, or more, of text on the screen.

To reduce network traffic and improve performance, you can reduce the size of any data being returned to some manageable limit by calling setMaxRows(), setMaxFieldSize(), and the driver-specific setFetchSize(). Another method of reducing the size of the data being returned is to decrease the column size.

In addition, be careful to return only the rows you need. If you return five columns when you only need two columns, performance is decreased, especially if the unnecessary rows include long data.

Choosing the Right Data Type

Retrieving and sending certain data types can be expensive. When you design a schema, select the data type that can be processed most efficiently. For example, integer data is processed faster than floating-point data. Floating-point data is defined according to internal database-specific formats, usually in a compressed format. The data must be decompressed and converted into a different format so that it can be processed by the database wire protocol.

Retrieving Result Sets

Most JDBC drivers cannot implement scrollable cursors because of limited support for scrollable cursors in the database system. Unless you are certain that the database supports using a scrollable result set, rs, for example, do not call rs.last and rs.getRow() methods to find out how many rows the result set contains. For JDBC drivers that emulate scrollable cursors, calling rs.last results in the driver retrieving all results across the network to reach the last row. Instead, you can either count the rows by iterating through the result set or get the number of rows by submitting a query with a Count column in the Select clause.

In general, do not write code that relies on the number of result rows from a query because drivers must fetch all rows in a result set to know how many rows the query will return.

Selecting JDBC Objects and Methods

The guidelines in this section will help you to select which JDBC objects and methods will give you the best performance.

Using Parameter Markers as Arguments to Stored Procedures

When calling stored procedures, always use parameter markers for argument markers instead of using literal arguments. JDBC drivers can call stored procedures on the database server either by executing the procedure as a SQL query or by optimizing the execution by invoking a Remote Procedure Call (RPC) directly on the database server. When you execute a stored procedure as a SQL query, the database server parses the statement, validates the argument types, and converts the arguments into the correct data types.

Remember that SQL is always sent to the database server as a character string, for example, `{call getCustName(12345)}`. In this case, even though the application programmer may have assumed that the only argument to getCustName() was an integer, the argument is actually passed inside a character string to the server. The database server parses the SQL query, isolates the single argument value `12345`, and converts the string `12345` into an integer value before executing the procedure as a SQL language event.

By invoking a RPC on the database server, the overhead of using a SQL character string is avoided. Instead, the JDBC driver constructs a network packet that contains the parameters in their native data type formats and executes the procedure remotely.

Case 1: Not Using a Server-Side RPC

In this example, the stored procedure getCustName() cannot be optimized to use a server-side RPC. The database server must treat the SQL request as a normal language event, which includes parsing the statement, validating the argument types, and converting the arguments into the correct data types before executing the procedure.

```
CallableStatement cstmt = conn.prepareCall("call getCustName(12345)");
ResultSet rs = cstmt.executeQuery();
```

Case 2: Using a Server-Side RPC

In this example, the stored procedure getCustName() can be optimized to use a server-side RPC. Because the application avoids literal arguments and calls the procedure by specifying all arguments as parameters, the JDBC driver can optimize the execution by invoking the stored procedure directly on the database as an RPC. The SQL language processing on the database server is avoided and execution time is greatly improved.

```
CallableStatement cstmt = conn.prepareCall("call getCustName(?)}");
cstmt.setLong(1,12345);
ResultSet rs = cstmt.executeQuery();
```

Using the Statement Object Instead of the PreparedStatement Object

JDBC drivers are optimized based on the perceived use of the functions that are being executed. Choose between the PreparedStatement object and the Statement object depending on how you plan to use the object. The Statement object is optimized for a single execution of a SQL statement. In contrast, the PreparedStatement object is optimized for SQL statements to be executed two or more times.

The overhead for the initial execution of a PreparedStatement object is high. The advantage comes with subsequent executions of the SQL statement. For example, suppose we are preparing and executing a query that returns employee information based on an ID. Using a PreparedStatement object, a JDBC driver would process the prepare request by making a network request to the database server to parse and optimize the query. The execute results in another network request. If the application will only make this request once during its life span, using a Statement object instead of a PreparedStatement object results in only a single network roundtrip to the database server. Reducing network communication typically provides the most performance gains.

This guideline is complicated by the use of prepared statement pooling because the scope of execution is longer. When using prepared statement pooling, if a query will only be executed once, use the Statement object. If a query will be executed infrequently, but may be executed again during the life of a statement pool inside a connection pool, use a PreparedStatement object. Under similar circumstances without statement pooling, use the Statement object.

Using Batches Instead of Prepared Statements

Updating large amounts of data typically is done by preparing an Insert statement and executing that statement multiple times, resulting in numerous network roundtrips. To reduce the number of JDBC calls and improve performance, you can send multiple queries to the database at a time using the addBatch method of the PreparedStatement object. For example, let us compare the following examples, Case 1 and Case 2.

Case 1: Executing Prepared Statement Multiple Times

```
PreparedStatement ps = conn.prepareStatement(
   "INSERT INTO employees VALUES (?, ?, ?)");
for (n = 0; n < 100; n++) {
   ps.setString(name[n]);
   ps.setLong(id[n]);
   ps.setInt(salary[n]);
   ps.executeUpdate();
}
```

Case 2: Using a Batch

```
PreparedStatement ps = conn.prepareStatement(
   "INSERT INTO employees VALUES (?, ?, ?)");
for (n = 0; n < 100; n++) {
   ps.setString(name[n]);
   ps.setLong(id[n]);
   ps.setInt(salary[n]);
   ps.addBatch();
}
ps.executeBatch();
```

In Case 1, a prepared statement is used to execute an Insert statement multiple times. In this case, 101 network roundtrips are required to perform 100 Insert operations: one roundtrip to prepare the statement and 100 additional roundtrips to execute its iterations. When the addBatch method is used to consolidate 100 Insert operations, as demonstrated in Case 2, only two network roundtrips are required—one to prepare the statement and another to execute the batch. Although more database CPU cycles are involved by using batches, performance is gained through the reduction of network roundtrips. Remember that the biggest gain in performance is realized by reducing network communication between the JDBC driver and the database server.

Choosing the Right Cursor

Choosing the appropriate type of cursor allows maximum application flexibility. This section summarizes the performance issues of three types of cursors: forward-only, insensitive, and sensitive.

A *forward-only cursor* provides excellent performance for sequential reads of all rows in a table. For retrieving table data, there is no faster way to return result rows than using a forward-only cursor; however, forward-only cursors cannot be used when the rows to be returned are not sequential.

Insensitive cursors are ideal for applications that require high levels of concurrency on the database server and require the ability to scroll forwards and backwards through result sets. The first request to an insensitive cursor fetches all the rows and stores them on the client. In most cases, the first request to an insensitive cursor fetches all the rows and stores them on the client. If a driver uses "lazy" fetching (fetch-on-demand), the first request may include many rows, if not all rows.The initial request is slow, especially when long data is returned. Subsequent requests do not require any network traffic (or, when a driver uses "lazy" fetching, requires limited network traffic) and are processed quickly.

Because the first request is processed slowly, insensitive cursors should not be used for a single request of one row. Developers should also avoid using insensitive cursors when long data or large result sets are returned because memory can be exhausted. Some insensitive cursor implementations cache the data in a temporary table on the database server and avoid the performance issue, but most cache the information local to the application.

Sensitive cursors, or keyset-driven cursors, use identifiers such as a ROWID that already exist in the database. When you scroll through the result set, the data for these identifiers is returned. Because each request generates network traffic, performance can be very slow. However, returning non-sequential rows does not further affect performance.

To illustrate this point further, consider an application that normally returns 1000 rows to an application. At execute time, or when the first row is requested, a JDBC driver does not execute the Select statement that was provided by the application. Instead, the JDBC driver replaces the Select list of the query with a key identifier, for example, ROWID. This modified query is then executed by the driver and all 1000 key values are returned by the database server and cached for use by the driver. Each request from the application for a result row directs the JDBC driver to look up the key value for the appropriate row in its local cache, construct an optimized query that contains a Where clause similar to WHERE ROWID=?, execute the modified query, and return the single result row from the server.

Sensitive cursors are the preferred scrollable cursor model for dynamic situations when the application cannot afford to buffer the data associated with an insensitive cursor.

Using get Methods Effectively

JDBC provides a variety of methods to return data from a result set (for example, getInt(), getString(), and getObject()). The getObject() method is the most generic and provides the worst performance when the non-default mappings are specified because the JDBC driver must perform extra processing to determine the type of the value being returned and generate the appropriate mapping. Always use the specific method for the data type.

To further improve performance, provide the column number of the column being returned, for example, `getString(1)`, `getLong(2)`, and `getInt(3)`, instead of the column name. If the column names are not specified, network traffic is unaffected, but costly conversions and lookups increase. For example, suppose you use:

```
getString("foo")...
```

The JDBC driver may need to convert foo to uppercase and then compare foo with all columns in the column list, which is costly. If the driver is able to go directly to result column 23, a large amount of processing is saved.

For example, suppose you have a result set that has 15 columns and 100 rows, and the column names are not included in the result set. You are interested in only three columns: EMPLOYEENAME (string), EMPLOYEENUMBER (long integer), and SALARY (integer). If you specify `getString("EmployeeName")`, `getLong("EmployeeNumber")`, and `getInt("Salary")`, each column name must be converted to the appropriate case of the columns in the database metadata and lookups would increase considerably. Performance improves significantly if you specify `getString(1)`, `getLong(2)`, and `getInt(15)`.

Retrieving Auto Generated Keys

Many databases have hidden columns (pseudo-columns) that represent a unique key for each row in a table. Typically, using these types of columns in a query is the fastest way to access a row because the pseudo-columns usually represent the physical disk address of the data. Prior to JDBC 3.0, an application could only return the value of the pseudo-columns by executing a Select statement immediately after inserting the data. For example:

```
//insert row
int rowcount = stmt.executeUpdate (
   "INSERT INTO LocalGeniusList (name)
   VALUES ('Karen')");
// now get the disk address - rowid -
// for the newly inserted row
ResultSet rs = stmt.executeQuery (
   "SELECT rowid FROM LocalGeniusList
   WHERE name = 'Karen'");
```

Retrieving pseudo-columns this way has two major flaws. First, retrieving the pseudo-column requires a separate query to be sent over the network and executed on the server. Second, because there may not be a primary key over the table, the search condition of the query may be unable to uniquely identify the row. In the latter case, multiple pseudo-column values can be returned, and the application may not be able to determine which value is actually the value for the most recently inserted row.

An optional feature of the JDBC 3.0 specification is the ability to return auto-generated key information for a row when the row is inserted into a table. For example:

```
int rowcount = stmt.executeUpdate(
   "INSERT INTO LocalGeniusList(name) VALUES('Karen')",
// insert row AND return key
Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
// key is automatically available
```

Now, the application contains a value that can be used in a search condition to provide the fastest access to the row and a value that uniquely identifies the row, even when a primary key doesn't exist on the table.

The ability to return keys provides flexibility to the JDBC developer and creates performance boosts when accessing data.

Managing Connections and Updates

The guidelines in this section will help you to manage connections and updates to improve system performance for your JDBC applications.

Managing Connections

Connection management is important to application performance. Optimize your application by connecting once and using multiple Statement objects, instead of performing multiple connections. Avoid connecting to a data source after establishing an initial connection.

Although gathering driver information at connect time is a good practice, it is often more efficient to gather it in one step rather than two steps. For example, some applications establish a connection and then call a method in a separate component that reattaches and gathers information about the driver. Applications that are designed as separate entities should pass the established connection object to the data collection routine instead of establishing a second connection.

Another bad practice is to connect and disconnect several times throughout your application to perform SQL statements. Connection objects can have multiple Statement objects associated with them. Statement objects, which are defined to be memory storage for information about SQL statements, can manage multiple SQL statements.

You can improve performance significantly with connection pooling, especially for applications that connect over a network or through the World Wide Web. Connection pooling lets you reuse connections. Closing connections does not close the physical connection to the database. When an application requests a connection, an active connection is reused, thus avoiding the network round trips needed to create a new connection.

Typically, you can configure a connection pool to provide scalability for connections. The goal is to maintain a reasonable connection pool size while ensuring that each user who needs a connection has one available within an acceptable response time. To achieve this goal, you can configure the minimum and maximum number of connections that are in the pool at any given time, and how long idle connections stay in the pool. In addition, to help minimize the number of connections required in a connection pool, you can switch the user associated with a connection to another user, a process known as *reauthentication*. Not all databases support reauthentication.

In addition to connection pooling tuning options, JDBC also specifies semantics for providing a prepared statement pool. Similar to connection pooling, a prepared statement pool caches PreparedStatement objects so that they can be re-used from a cache without application intervention. For example, an application may create a PreparedStatement object similar to the following SQL statement:

```
SELECT name, address, dept, salary FROM personnel
WHERE empid = ? or name = ? or address = ?
```

When the PreparedStatement object is created, the SQL query is parsed for semantic validation and a query optimization plan is produced. The process of creating a prepared statement can be extremely expensive in terms of performance with some database systems. Once the prepared statement is closed, a JDBC 3.0-compliant driver places the prepared statement into a local cache instead of discarding it. If the application later attempts to create a prepared statement with the same SQL query, a common occurrence in many applications, the driver can simply retrieve the associated statement from the local cache instead of performing a network roundtrip to the server and an expensive database validation.

Connection and statement handling should be addressed before implementation. Thoughtfully handling connections and statements improves application performance and maintainability.

Managing Commits in Transactions

Committing transactions is slow because of the amount of disk I/O and potentially network round trips that are required. Always turn off Autocommit by using `Connection.setAutoCommit(false)`.

What does a commit actually involve? The database server must flush back to disk every data page that contains updated or new data. This is usually a sequential write to a journal file, but nevertheless, it involves disk I/O. By default, Autocommit is on when connecting to a data source, and Autocommit mode usually impairs performance because of the significant amount of disk I/O needed to commit every operation.

Furthermore, most database servers do not provide a native Autocommit mode. For this type of server, the JDBC driver must explicitly issue a COMMIT statement and a BEGIN TRANSACTION for every operation sent to the server. In addition to the large amount of disk I/O required to support Autocommit mode, a performance penalty is paid for up to three network requests for every statement issued by an application.

Although using transactions can help application performance, do not take this tip too far. Leaving transactions active can reduce throughput by holding locks on rows for longer than necessary, preventing other users from accessing the rows. Commit transactions in intervals that allow maximum concurrency.

Choosing the Right Transaction Model

Many systems support distributed transactions; that is, transactions that span multiple connections. Distributed transactions are at least four times slower than normal transactions due to the logging and network round trips necessary to communicate between all the components involved in the distributed transaction (the JDBC driver, transaction monitor, and DBMS). Unless distributed transactions are required, avoid using them. Instead, use local transactions when possible. Many Java application servers provide a default transaction behavior that uses distributed transactions.

For the best system performance, design the application to run using a single Connection object.

Using updateXXX Methods

Although programmatic updates do not apply to all types of applications, developers should attempt to use programmatic updates and deletes. Using the updateXXX methods of the ResultSet object allows the developer to update data without building a complex SQL statement. Instead, the developer simply supplies the column in the result set that is to be updated and the data that is to be changed. Then, before moving the cursor from the row in the result set, the updateRow() method must be called to update the database as well.

In the following code fragment, the value of the Age column of the ResultSet object rs is returned using the getInt() method, and the updateInt() method is used to update the column with an int value of `25`. The updateRow() method is called to update the row in the database with the modified value.

```
int n = rs.getInt("Age");
// n contains value of Age column in the resultset rs
...
rs.updateInt("Age", 25);
rs.updateRow();
```

In addition to making the application more easily maintainable, programmatic updates usually result in improved performance. Because the database server is already positioned on the row for the Select statement in process, performance-expensive operations to locate the row that needs to be changed are unnecessary. If the row must be located, the server usually has an internal pointer to the row available (for example, ROWID).

Using getBestRowIdentifier

Use getBestRowIdentifier() to determine the optimal set of columns to use in the Where clause for updating data. Pseudo-columns often provide the fastest access to the data, and these columns can only be determined by using getBestRowIdentifier().

Some applications cannot be designed to take advantage of positioned updates and deletes. Some applications formulate the Where clause by calling getPrimaryKeys() to use all searchable result columns or by calling getIndexInfo() to find columns that may be part of a unique index. These methods usually work, but can result in fairly complex queries.

Consider the following example:

```
ResultSet WSrs = WSs.executeQuery
     ("SELECT first_name, last_name, ssn, address, city, state, zip FROM emp");
// fetchdata
...
WSs.executeQuery (
   "UPDATE emp SET address = ?
    WHERE first_name = ? AND last_name = ? AND ssn = ?
    AND address = ? AND city = ? AND state = ? AND zip = ?");
// fairly complex query
```

Applications should call getBestRowIdentifier() to return the optimal set of columns (possibly a pseudo-column) that identifies a specific record. Many databases support special columns that are not explicitly defined by the user in the table definition, but are "hidden" columns of every table (for example, ROWID and TID). These pseudo-columns generally provide the fastest access to the data because they typically are pointers to the exact location of the record. Because pseudo-columns are not part of the explicit table definition, they are not returned from getColumns(). To determine if pseudo-columns exist, call getBestRowIdentifier().

Consider the previous example again:

```
...
ResultSet WSrowid = getBestRowIdentifier()
   (... "emp", ...);
...
WSs.executeUpdate("UPDATE EMP SET ADDRESS = ? WHERE ROWID = ?");
// fastest access to the data!
```

If your data source does not contain special pseudo-columns, the result set of getBestRowIdentifier() consists of the columns of the most optimal unique index on the specified table (if a unique index exists). Therefore, your application does not need to call getIndexInfo() to find the smallest unique index.

7

SQL Escape Sequences for JDBC

Language features, such as outer joins and scalar function calls, are commonly implemented by database systems. The syntax for these features is often database-specific, even when a standard syntax has been defined. JDBC defines escape sequences that contain the standard syntax for the following language features:

- Date, time, and timestamp literals
- Scalar functions such as numeric, string, and data type conversion functions
- Outer joins
- Escape characters for wildcards used in LIKE clauses
- Procedure calls

The escape sequence used by JDBC is:

`{extension}`

The escape sequence is recognized and parsed by the drivers, which replaces the escape sequences with data store-specific grammar.

For details, see the following topics:

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

Date, Time, and Timestamp Escape Sequences

The escape sequence for date, time, and timestamp literals is:

`{literal-type 'value'}`

where:

literal-type

is one of the following:

literal-type	Description	Value Format
d	Date	`yyyy-mm-dd`
t	Time	`hh:mm:ss []`
ts	Timestamp	`yyyy-mm-dd hh:mm:ss[.f...]`

Example:

```
UPDATE Orders SET OpenDate={d '1995-01-15'} WHERE OrderID=1023
```

Scalar Functions

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

`{fn scalar-function}`

where:

scalar-function

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

Example:

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

Table 34: Supported Scalar Functions

Data Store	String Functions	Numeric Functions	Timedate Functions	System Functions
DB2	ASCII BLOB CHAR CHR CLOB	ABS or ABSVAL ACOS ASIN ATAN	CURDATE CURTIME DATE DAY DAYNAME	COALESCE DEREF DLCOMMENT DLLINKTYPE DLURLCOMPLETE

Data Store	String Functions	Numeric Functions	Timedate Functions	System Functions
	CONCAT DAYNAMEDBCLOB DECFLOAT_FORMAT DIFFERENCE GRAPHIC HEX INITCAPINSERT INSTR LCASE or LOWER LCASE [66] LEFT LENGTH LOCATE LOCATE_IN_STRING LONG_VARCHAR LONG_VARGRAPHIC LPAD LTRIM LTRIM MONTHNAME POSSTR REPEAT REPLACE RIGHT RPADRTRIM RTRIM RTRIM SOUNDEX SPACE SUBSTR TO_CLOB TO_NUMBER TRUNCATE or TRUNC UCASE or UPPER	ATANH ATAN2 BIGINT CEILING or CEIL COS COSH COT DECIMAL DEGREES DIGITS DOUBLE EXP FLOAT FLOOR INTEGER LN LOG LOG10 MOD POWER RADIANS RAND REAL ROUND SIGN SIN SINH SMALLINT SQRT TAN TANH TRUNCATE	DAYOFWEEK DAYOFYEAR DAYS HOUR JULIAN_DAY MICROSECOND MIDNIGHT_SECONDS MINUTE MONTH MONTHNAME NOW QUARTER SECOND TIME TIMESTAMP TIMESTAMP_ISO TIMESTAMPDIFF WEEK YEAR	DLURLPATH DLURLPATHONLY DLURLSCHEME DLURLSERVER DLVALUE EVENT_MON_STATE GENERATE_UNIQUE NODENUMBER NULLIF PARTITION RAISE_ERROR TABLE_NAME TABLE_SCHEMA TRANSLATE TYPE_ID TYPE_NAME TYPE_SCHEMA VALUE

[66] SYSFUN schema.

Data Store	String Functions	Numeric Functions	Timedate Functions	System Functions
	VARCHAR VARGRAPHIC			
Informix	CONCAT LEFT LENGTH LTRIM REPLACE RTRIM SUBSTRING	ABS ACOS ASIN ATAN ATAN2 COS COT EXP FLOOR LOG LOG10 MOD PI POWER ROUND SIN SQRT TAN TRUNCATE	CURDATE CURTIME DAYOFMONTH DAYOFWEEK MONTH NOW TIMESTAMPADD TIMESTAMPDIFF YEAR	DATABASE USER
MySQL	ASCII CHAR CONCAT INSERT LCASE LEFT LENGTH LOCATE LOCATE_2 LTRIM REPEAT	ABSA COS ASIN ATAN ATAN2 CEILING COS COT DEGREES EXP FLOOR	CURDATE CURRENT_DATE CURRENT_TIME CURRENT_TIMESTAMP CURTIME DAYNAME DAYOFMONTH DAYOFWEEK DAYOFYEAR EXTRACT HOUR	DATABASE IFNULL USER

Data Store	String Functions	Numeric Functions	Timedate Functions	System Functions
	REPLACE RIGHT RTRIM SOUNDEX SPACE SUBSTRING UCASE	LOG LOG10 MOD PI POWER RADIANS RAND ROUND SIGN SIN SQRT TAN TRUNCATE	MINUTE MONTH MONTHNAME NOW QUARTER SECOND TIMESTAMPADD TIMESTAMPDIFF WEEK YEAR	
Oracle	ASCII BIT_LENGTH CHAR CONCAT INSERT LCASE LEFT LENGTH LOCATE LOCATE2 LTRIM OCTET_LENGTH REPEAT REPLACE RIGHT RTRIM SOUNDEX SPACE SUBSTRING UCASE	ABS ACOS ASIN ATAN ATAN2 CEILING COS COT EXP FLOOR LOG LOG10 MOD PI POWER ROUND SIGN SIN SQRT TAN TRUNCATE	CURDATE DAYNAME DAYOFMONTH DAYOFWEEK DAYOFYEAR HOUR MINUTE MONTH MONTHNAME NOW QUARTER SECOND WEEK YEAR	IFNULL USER

Data Store	String Functions	Numeric Functions	Timedate Functions	System Functions
PostgreSQL	ASCII BIT_LENGTH CHAR CHAR_LENGTH CHARACTER_LENGTH CONCAT LCASE LENGTH LEFT [67] LOCATE LTRIM OCTET_LENGTH POSITION REPEAT REPLACE RIGHT RTRIM SUBSTRING UCASE	ABS ACOS ASIN ATAN ATAN2 CEILING COS COT DEGREES EXP FLOOR LOG LOG10 MOD PI POWER RADIANS RAND ROUND SIGN SIN SQRT TAN TRUNCATE	CURDATE CURRENT_DATE CURRENT_TIME CURRENT_TIMESTAMP CURTIME EXTRACT NOW	USERNAME DBNAME IFNULL
Progress OpenEdge	ASCII CHAR CONCAT DIFFERENCE LCASE LEFT LENGTH LOCATE	ABS ACOS ASIN ATAN ATAN2 CEILING COS DEGREES	CURDATE CURTIME DAYNAME DAYOFMONTH DAYOFWEEK HOUR MINUTE MONTH	DATABASE IFNULL USER

[67] Supported for PostgreSQL 9.1 and higher

Data Store	String Functions	Numeric Functions	Timedate Functions	System Functions
	LTRIM REPEAT REPLACE RIGHT RTRIM SPACE SUBSTRING UCASE	EXP FLOOR LOG10 MOD PI POWER RADIANS ROUND SIN SQRT TAN	MONTHNAME NOW QUARTER SECOND TIMESTAMPADD TIMESTAMPDIFF WEEK YEAR	
SQL Server	ASCII CHAR CONCAT DIFFERENCE INSERT LCASE LEFT LENGTH LOCATE LTRIM REPEAT REPLACE RIGHT RTRIM SOUNDEX SPACE SUBSTRING UCASE	ABS ACOS ASIN ATAN ATAN2 CEILING COS COT DEGREES EXP FLOOR LOG LOG10 MOD PI POWER RADIANS RAND ROUND SIGN SIN SQRT TAN	DAYNAME DAYOFMONTH DAYOFWEEK DAYOFYEAR EXTRACT HOUR MINUTE MONTH MONTHNAME NOW QUARTER SECOND TIMESTAMPADD TIMESTAMPDIFF WEEK YEAR	DATABASE IFNULL USER

Data Store	String Functions	Numeric Functions	Timedate Functions	System Functions
		TRUNCATE		
Sybase	ASCII CHAR CONCAT DIFFERENCE INSERT LCASE LEFT LENGTH LOCATE LTRIM REPEAT RIGHT RTRIM SOUNDEX SPACE SUBSTRING UCASE	ABS ACOS ASIN ATAN ATAN2 CEILING COS COT DEGREES EXP FLOOR LOG LOG10 MOD PI POWER RADIANS RAND ROUND SIGN SIN SQRT TAN	DAYNAME DAYOFMONTH DAYOFWEEK DAYOFYEAR HOUR MINUTE MONTH MONTHNAME NOW QUARTER SECOND TIMESTAMPADD TIMESTAMPDIFF WEEK YEAR	DATABASE IFNULL USER
Apache Hive	ASCII CONCAT INSERT LCASE LEFT LENGTH LOCATE LOCATE2	ABS ACOS ASIN ATAN CEILING COS COT DEGREES	CURDATE CURRENT_DATE CURRENT_TIME CURRENT_TIMESTAMP CURTIME DAYOFMONTH EXTRACT HOUR	DBNAME IFNULL

Data Store	String Functions	Numeric Functions	Timedate Functions	System Functions
	LTRIM REPEAT REPLACE RIGHT RTRIM SPACE SUBSTRING UCASE	EXP FLOOR LOG LOG10 MODP PI POWER RADIANS RAND ROUND SIGN SIN SQRT TAN	MINUTE MONTH NOW QUARTER SECOND TIMESTAMPADD [68] TIMESTAMPDIFF WEEK YEAR	
Greenplum	ASCII BIT_LENGTH	ABS ACOS	CURDATE CURRENT_DATE	USERNAME DBNAME

[68] Apache Hive is limited to adding only days to a timestamp.

Data Store	String Functions	Numeric Functions	Timedate Functions	System Functions
	CHAR CHAR_LENGTH CHARACTER_LENGTH CONCAT LCASE LENGTH LOCATE LTRIM OCTET_LENGTH POSITION REPEAT REPLACE RIGHT RTRIM SUBSTRING UCASE	ASIN ATAN ATAN2 CEILING COS COT DEGREES EXP FLOOR LOG LOG10 MOD PI POWER RADIANS RAND ROUND SIGN SIN SQRT TAN TRUNCATE	CURRENT_TIME CURRENT_TIMESTAMP CURTIME EXTRACT NOW	IFNULL
Salesforce	ASCII BITLENGTH CHAR CHAR_LENGTH CHARACTER_LENGTH CONCAT DIFFERENCE HEXTORAW INSERT LCASE LEFT LENGTH	ABS ACOS ASIN ATAN ATAN2 BITAND BITOR BITXOR CEILING COS COT DEGREES	CURDATE CURTIME DATEDIFF DAY DAYNAME DAYOFMONTH DAYOFWEEK DAYOFYEAR HOUR MINUTE MONTH MONTHNAME	CURSESSIONID DATABASE IDENTITY USER

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

Outer Join Escape Sequences

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

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

where:

`outer-join`

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

`table-reference`

is a database table name.

`search-condition`

is the join condition you want to use for the tables.

Example:

```
SELECT Customers.CustID, Customers.Name, Orders.OrderID, Orders.Status
   FROM {oj Customers LEFT OUTER JOIN
```

```
   Orders ON Customers.CustID=Orders.CustID}
WHERE Orders.Status='OPEN'
```

The following table lists the outer join escape sequences supported by the drivers for each data store.

Table 35: Outer Join Escape Sequences Supported

Data Store	Outer Join Escape Sequences
DB2	Left outer joins Right outer joins Full outer joins Nested outer joins
Informix	Left outer joins Right outer joins Full outer joins Nested outer joins
MySQL	Left outer joins Right outer joins Nested outer joins
Oracle	Left outer joins Right outer joins Full outer joins Nested outer joins
PostgreSQL	Left outer joins Right outer joins Full outer joins Nested outer joins
Progress OpenEdge	Left outer joins Nested outer joins
SQL Server	Left outer joins Right outer joins Full outer joins Nested outer joins

Data Store	Outer Join Escape Sequences
Sybase	Left outer joins Right outer joins Nested outer joins
Apache Hive	Left outer joins Right outer joins Full outer joins
Greenplum	Left outer joins Right outer joins Full outer joins Nested outer joins
Salesforce	Left outer joins Right outer joins Nested outer joins

LIKE Escape Character Sequence for Wildcards

You can specify the character to be used to escape wildcard characters (% and _, for example) in LIKE clauses. The escape sequence for escape characters is:

`{escape 'escape-character'}`

where:

escape-character

is the character used to escape the wildcard character.

For example. the following SQL statement specifies that an asterisk (*) be used as the escape character in the LIKE clause for the wildcard character %:

```
SELECT col1 FROM table1 WHERE col1 LIKE '*%%' {escape '*'}
```

Procedure Call Escape Sequences

A procedure is an executable object stored in the data store. Generally, it is one or more SQL statements that have been precompiled. The escape sequence for calling a procedure is:

`{[?=]call procedure-name[(parameter[,parameter]...)]}`

where:

procedure-name

specifies the name of a stored procedure.

parameter

specifies a stored procedure parameter.

Note: For DB2 for Linux/UNIX/Windows, a catalog name cannot be used when calling a stored procedure. Also, for DB2 V8.1 and V8.2 for Linux/UNIX/Windows, literal parameter values are supported for stored procedures. Other supported DB2 versions do not support literal parameter values for stored procedures.

8

Using DataDirect Test

Use DataDirect Test to test your JDBC applications and learn the JDBC API. DataDirect Test contains menu selections that correspond to specific JDBC functions, for example, connecting to a database or passing a SQL statement. DataDirect Test allows you to perform the following tasks:

- Execute a single JDBC method or execute multiple JDBC methods simultaneously, so that you can easily perform some common tasks, such as returning result sets
- Display the results of all JDBC function calls in one window, while displaying fully commented, JDBC code in an alternate window

DataDirect Test works only with JDBC drivers from Progress DataDirect.

For details, see the following topics:

- DataDirect Test Tutorial

DataDirect Test Tutorial

This DataDirect Test tutorial explains how to use the most important features of DataDirect Test (and the JDBC API) and assumes that you can connect to a database with the standard available demo table or fine-tune the sample SQL statements shown in this example as appropriate for your environment.

Note: The tutorial describes functionality across a spectrum of data stores. In some cases, the functionality described may not apply to the driver or data store you are using. Additionally, examples are drawn from a variety of drivers and data stores.

Note: The step-by-step examples used in this tutorial do not show typical clean-up routines (for example, closing result sets and connections). These steps have been omitted to simplify the examples. Do not forget to add these steps when you use equivalent code in your applications.

Configuring DataDirect Test

The default DataDirect Test configuration file is:

`install_dir/testforjdbc/Config.txt`

where:

`install_dir`

is your product installation directory.

The DataDirect Test configuration file can be edited as appropriate for your environment using any text editor. All parameters are configurable, but the most commonly configured parameters are:

Drivers	A list of colon-separated JDBC driver classes.
DefaultDriver	The default JDBC driver that appears in the **Get Driver URL** window.
Databases	A list of comma-separated JDBC URLs. The first item in the list appears as the default in the **Database Selection** window. You can use one of these URLs as a template when you make a JDBC connection. The default Config.txt file contains example URLs for most databases.
InitialContextFactory	Set to `com.sun.jndi.fscontext.RefFSContextFactory` if you are using file system data sources, or `com.sun.jndi.ldap.LdapCtxFactory` if you are using LDAP.
ContextProviderURL	The location of the .bindings file if you are using file system data sources, or your LDAP Provider URL if you are using LDAP.
Datasources	A list of comma-separated JDBC data sources. The first item in the list appears as the default in the **Data Source Selection** window.

To connect using a data source, DataDirect Test needs to access a JNDI data store to persist the data source information. By default, DataDirect Test is configured to use the JNDI File System Service Provider to persist the data source. You can download the JNDI File System Service Provider from the Oracle Java Platform Technology Downloads page.

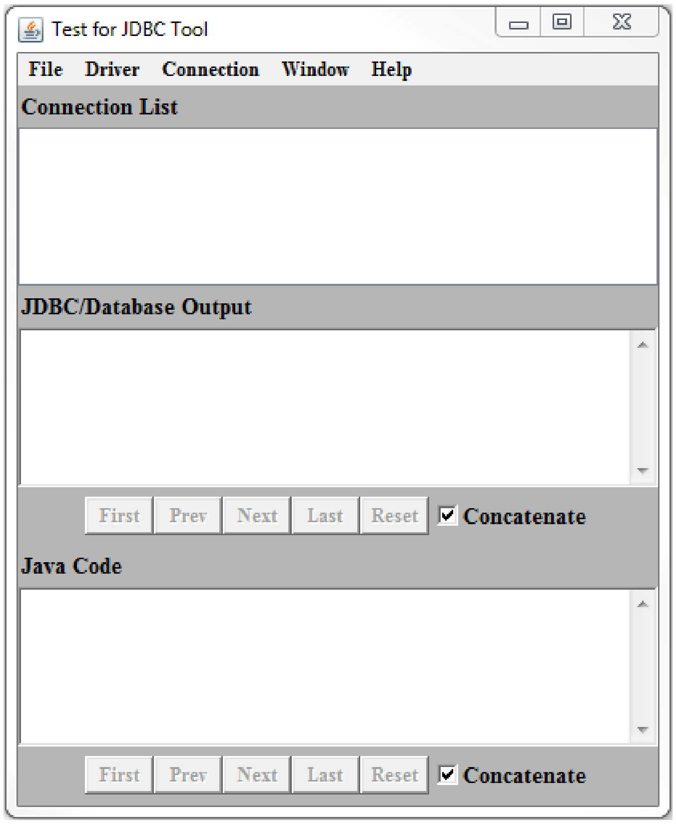
Make sure that the `fscontext.jar` and `providerutil.jar` files from the download are on your classpath.

Starting DataDirect Test

How you start DataDirect Test depends on your platform:

- **As a Java application on Windows**. Run the `testforjdbc.bat` file located in the `testforjdbc` subdirectory of your product installation directory.
- **As a Java application on Linux/UNIX**. Run the `testforjdbc.sh` shell script located in the `testforjdbc` subdirectory in the installation directory.

After you start DataDirect Test, the **Test for JDBC Tool** window appears.



The main **Test for JDBC Tool** window shows the following information:

- In the **Connection List** box, a list of available connections.
- In the **JDBC/Database Output** scroll box, a report indicating whether the last action succeeded or failed.
- In the **Java Code** scroll box, the actual Java code used to implement the last action.

Tip: DataDirect Test windows contain two **Concatenate** check boxes. Select a **Concatenate** check box to see a cumulative record of previous actions; otherwise, only the last action is shown. Selecting **Concatenate** can degrade performance, particularly when displaying large result sets.

Connecting Using DataDirect Test

You can use either of the following methods to connect using DataDirect Test:

- Using a data source
- Using a driver/database selection

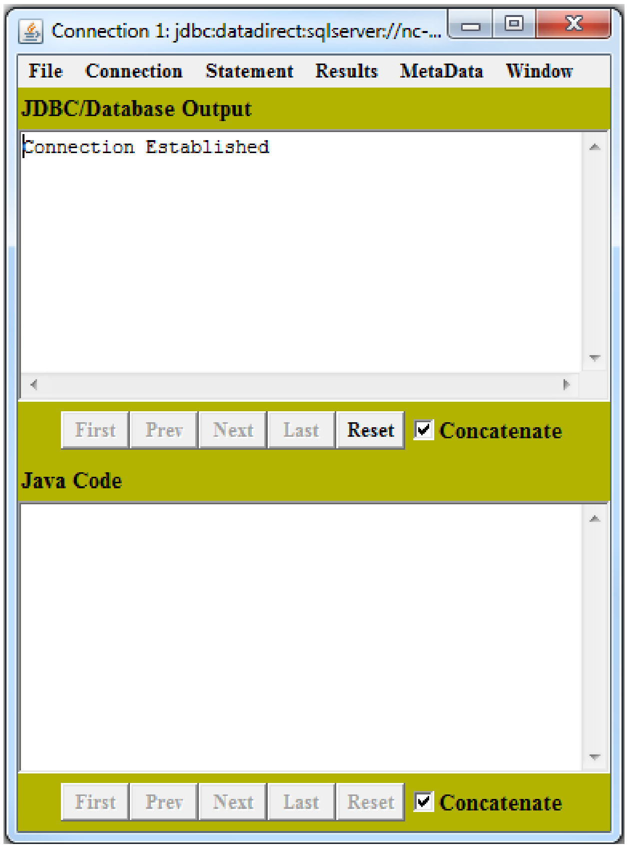
Connecting Using a Data Source

To connect using a data source, DataDirect Test needs to access a JNDI data store to persist the data source information. By default, DataDirect Test is configured to use the JNDI File System Service Provider to persist the data source. You can download the JNDI File System Service Provider from the Oracle Java Platform Technology Downloads page.

Make sure that the `fscontext.jar` and `providerutil.jar` files from the download are on your classpath.

To connect using a data source:

1. From the main **Test for JDBC Tool** window menu, select **Connection / Connect to DB via Data Source**. The **Select A Datasource** window appears.
2. Select a data source from the **Defined Datasources** pane. In the User Name and Password fields, type values for the User and Password connection properties; then, click **Connect**. For information about JDBC connection properties, refer to your driver's connection property descriptions.
3. If the connection was successful, the **Connection** window appears and shows the `Connection Established` message in the JDBC/Database Output scroll box.

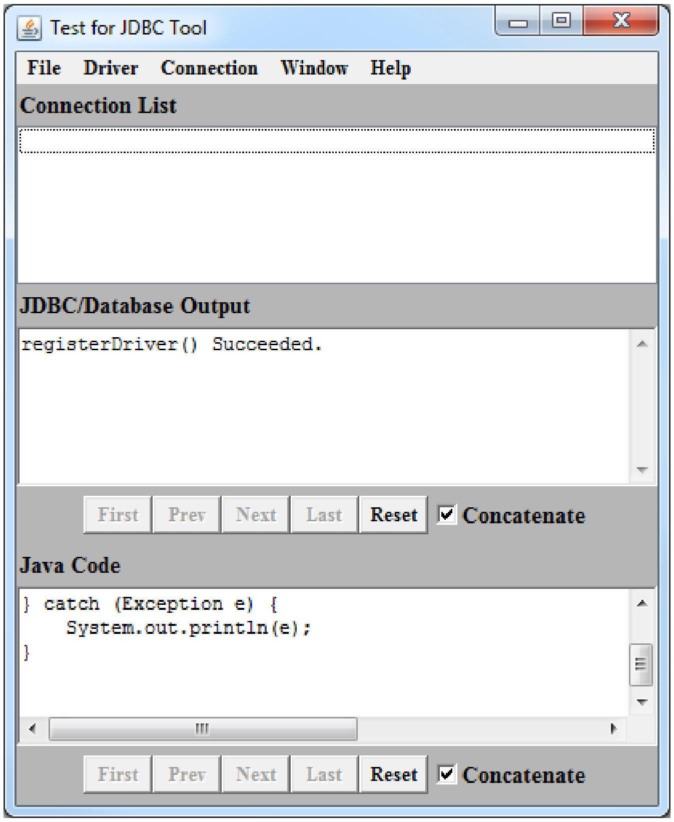


Connecting Using Database Selection

To connect using database selection:

1. From the **Test for JDBC Tool** window menu, select **Driver / Register Driver**. DataDirect Test prompts for a JDBC driver name.
2. In the Please Supply a Driver URL field, specify a driver (for example `com.ddtek.jdbc.sqlserver.SQLServerDriver`); then, click **OK**.

   If the driver was registered successfully, the **Test for JDBC Tool** window appears with a confirmation in the JDBC/Database Output scroll box.

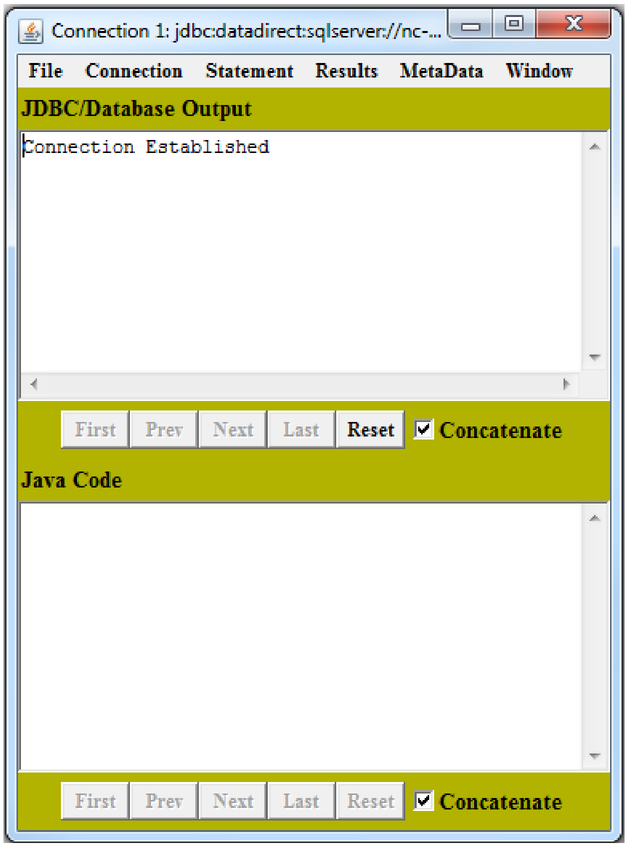


3. From the **Test for JDBC Tool** window, select **Connection / Connect to DB**. The **Select A Database** window appears with a list of default connection URLs.
4. Select one of the default driver connection URLs. In the Database field, modify the default values of the connection URL appropriately for your environment.

Note: There are two entries for DB2: one with locationName and another with databaseName. If you are connecting to DB2 for Linux/UNIX/Windows, select the entry containing `databaseName`. If you are connecting to DB2 for z/OS or DB2 for i, select the entry containing `locationName`.

---

5. In the User Name and Password fields, type the values for the User and Password connection properties; then, click **Connect**. For information about JDBC connection properties, refer to your driver's connection property descriptions.
6. If the connection was successful, the **Connection** window appears and shows the `Connection Established` message in the JDBC/Database Output scroll box.

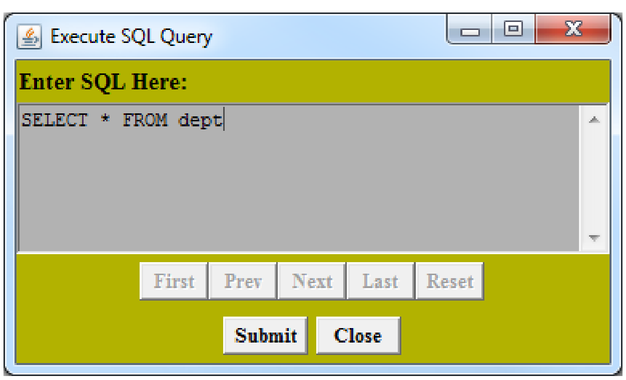


Executing a Simple Select Statement

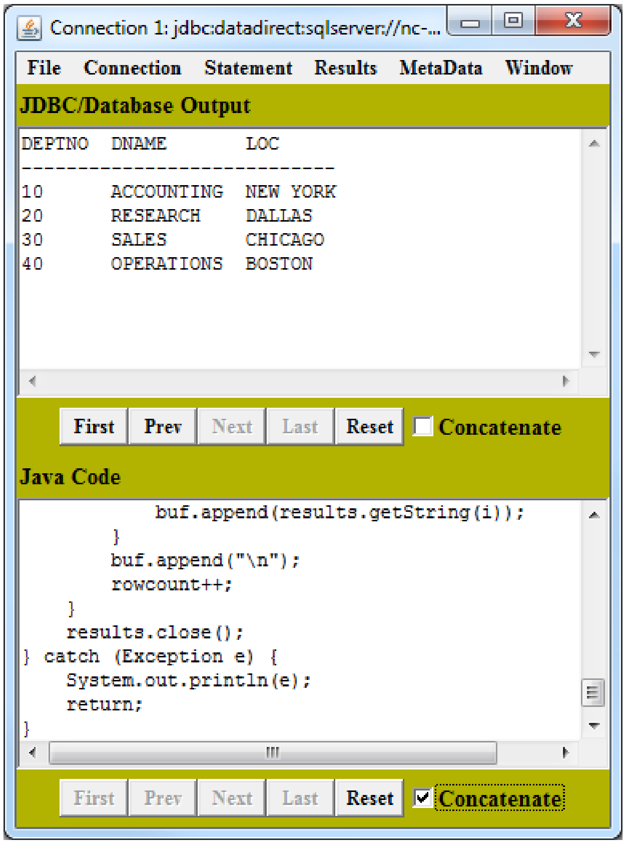
This example explains how to execute a simple Select statement and return the results.

To Execute a Simple Select Statement:

1. From the **Connection** window menu, select **Connection / Create Statement**. The **Connection** window indicates that the creation of the statement was successful.
2. Select **Statement / Execute Stmt Query**. DataDirect Test displays a dialog box that prompts for a SQL statement.



3. Type a Select statement and click **Submit**. Then, click **Close**.
4. Select **Results / Show All Results**. The data from your result set displays in the JDBC/Database Output scroll box.



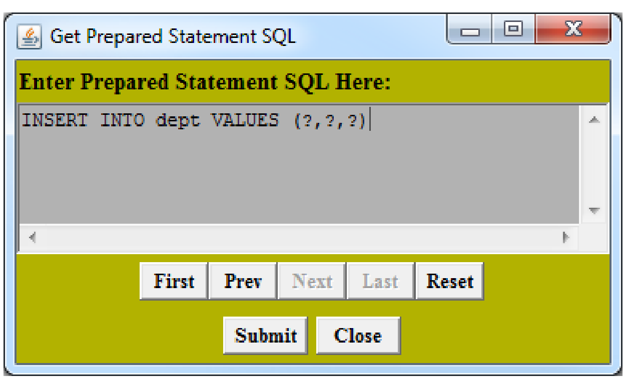
5. Scroll through the code in the Java Code scroll box to see which JDBC calls have been implemented by DataDirect Test.

Executing a Prepared Statement

This example explains how to execute a parameterized statement multiple times.

To Execute a Prepared Statement:

1. From the **Connection** window menu, select **Connection / Create Prepared Statement**. DataDirect Test prompts you for a SQL statement.
2. Type an Insert statement and click **Submit**. Then, click **Close**.



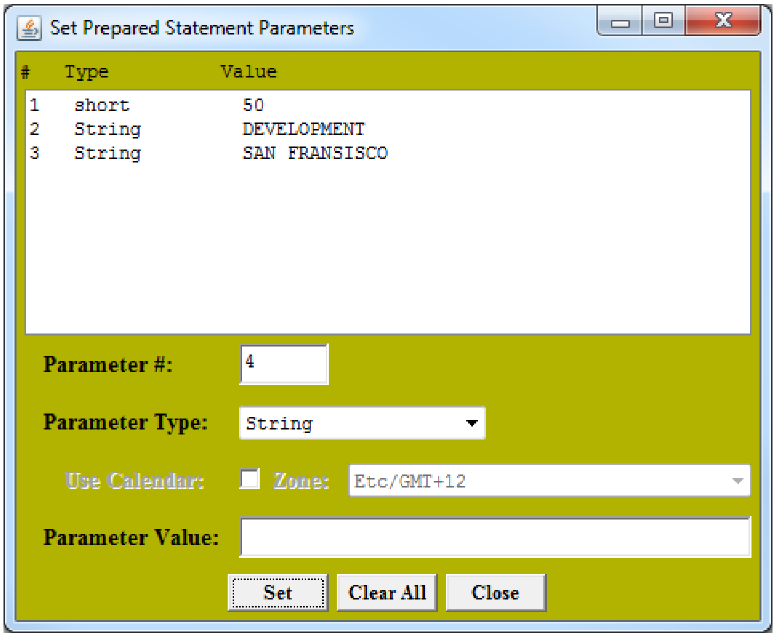
3. Select **Statement / Set Prepared Parameters**. To set the value and type for each parameter:

a) Type the parameter number.

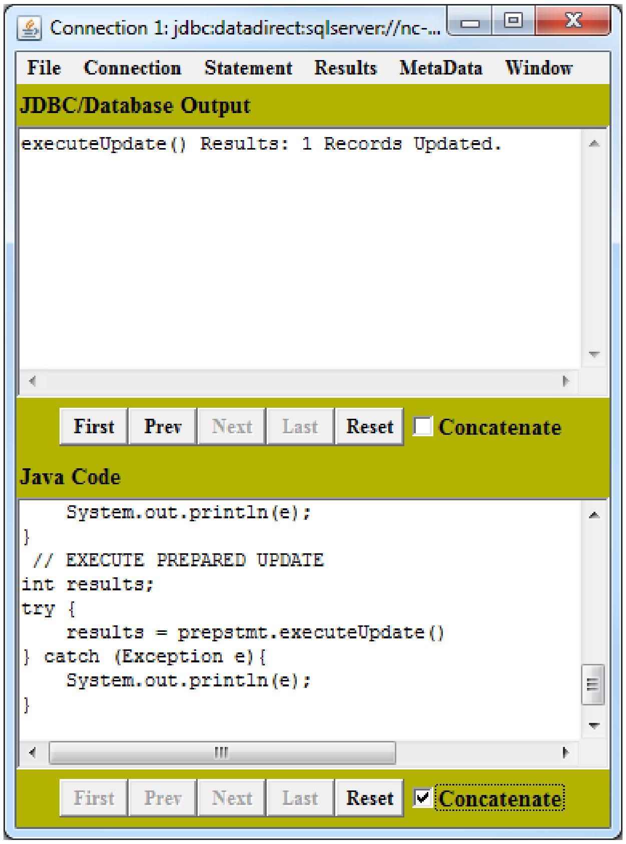
b) Select the parameter type.

c) Type the parameter value.

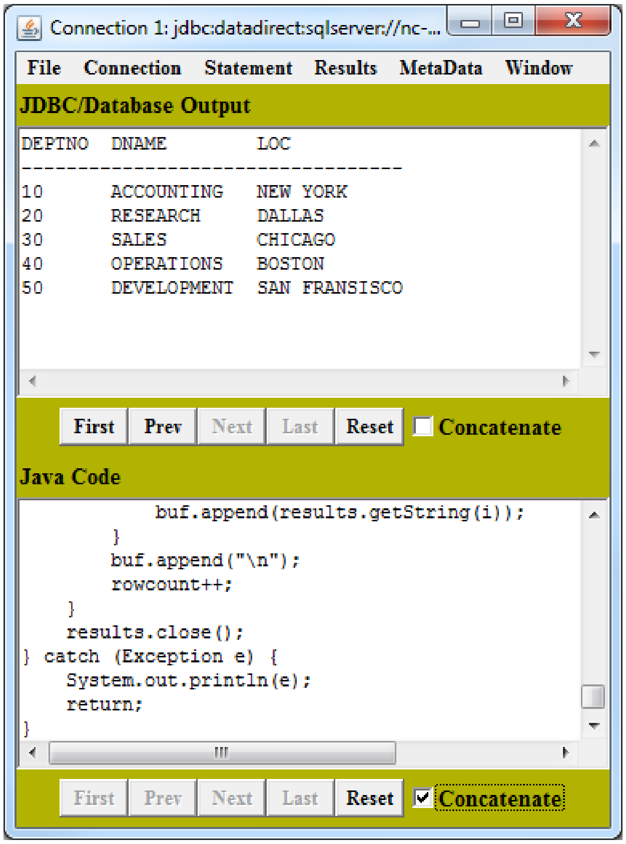
d) Click **Set** to pass this information to the JDBC driver.



4. When you are finished, click **Close**.
5. Select **Statement / Execute Stmt Update**. The JDBC/Database Output scroll box indicates that one row has been inserted.

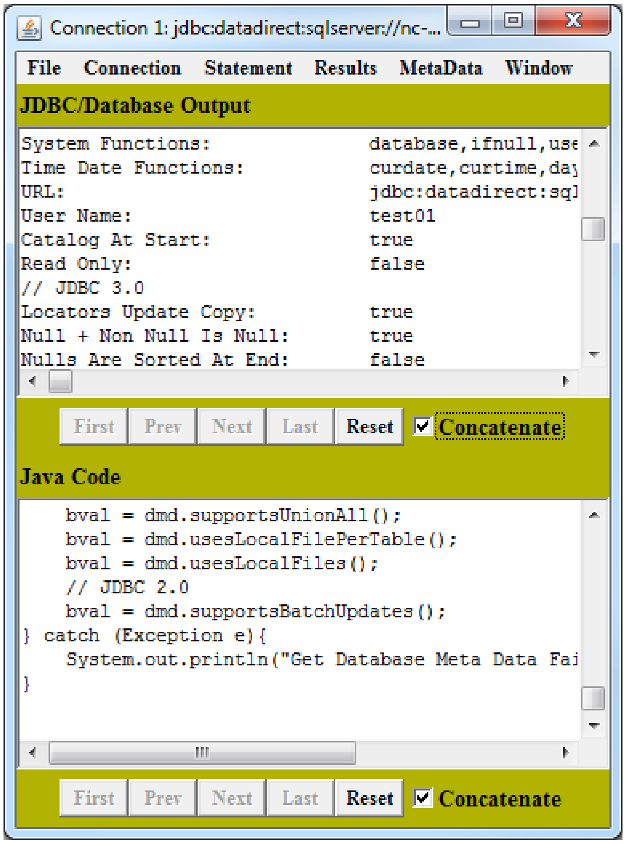


6. If you want to insert multiple records, repeat Step 3 on page 297 and Step 5 on page 298 for each record.
7. If you repeat the steps described in Executing a Simple Select Statement on page 296, you will see that the previously inserted records are also returned.



Retrieving Database Metadata

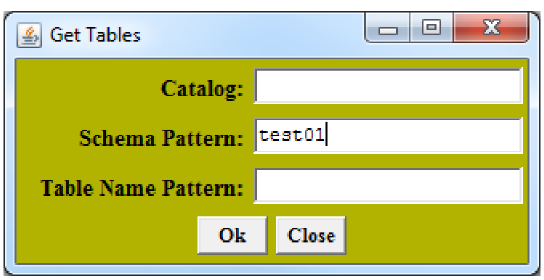
1. From the **Connection** window menu, select **Connection / Get DB Meta Data**.
2. Select **MetaData / Show Meta Data**. Information about the JDBC driver and the database to which you are connected is returned.



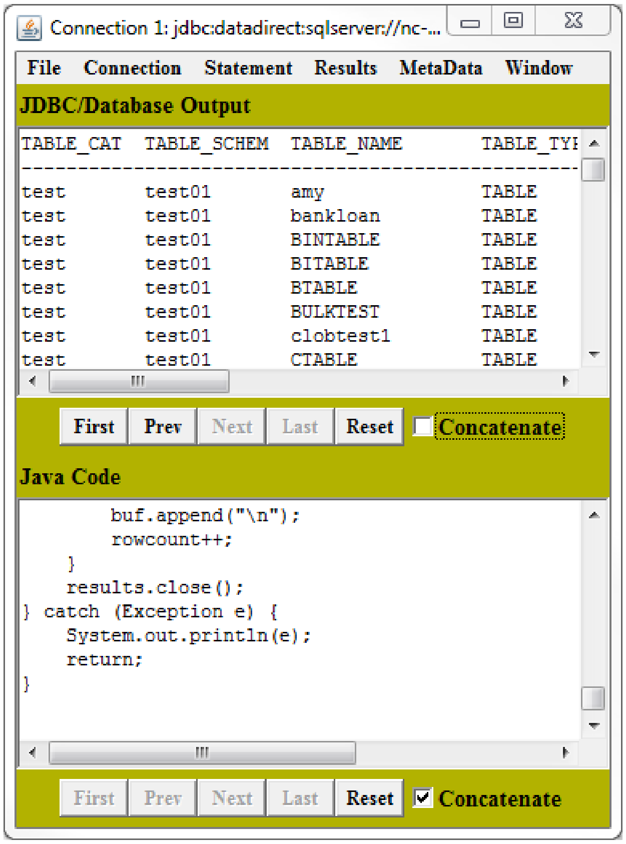
3. Scroll through the Java code in the Java Code scroll box to find out which JDBC calls have been implemented by DataDirect Test.

   Metadata also allows you to query the database catalog (enumerate the tables in the database, for example). In this example, we will query all tables with the schema pattern `test01`.

4. Select **MetaData / Tables**.
5. In the Schema Pattern field, type `test01`.

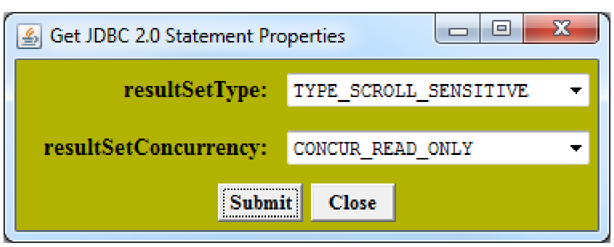


6. Click **Ok**. The **Connection** window indicates that getTables() succeeded.
7. Select **Results / Show All Results**. All tables with a `test01` schema pattern are returned.

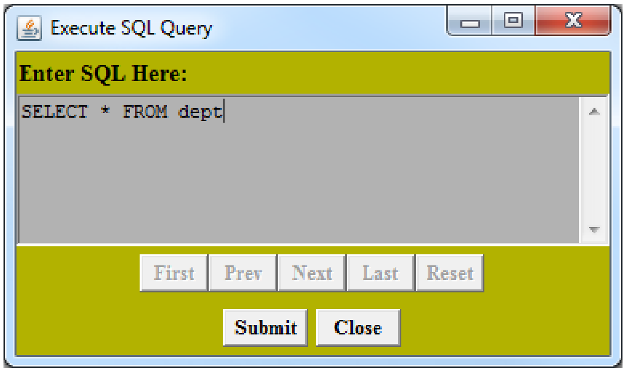


Scrolling Through a Result Set

1. From the **Connection** window menu, select **Connection / Create JDBC 2.0 Statement**. DataDirect Test prompts for a result set type and concurrency.
2. Complete the following fields:
   a) In the resultSetType field, select **TYPE_SCROLL_SENSITIVE**.
   b) In the resultSetConcurrency field, select **CONCUR_READ_ONLY**.
   c) Click **Submit**; then, click **Close**.

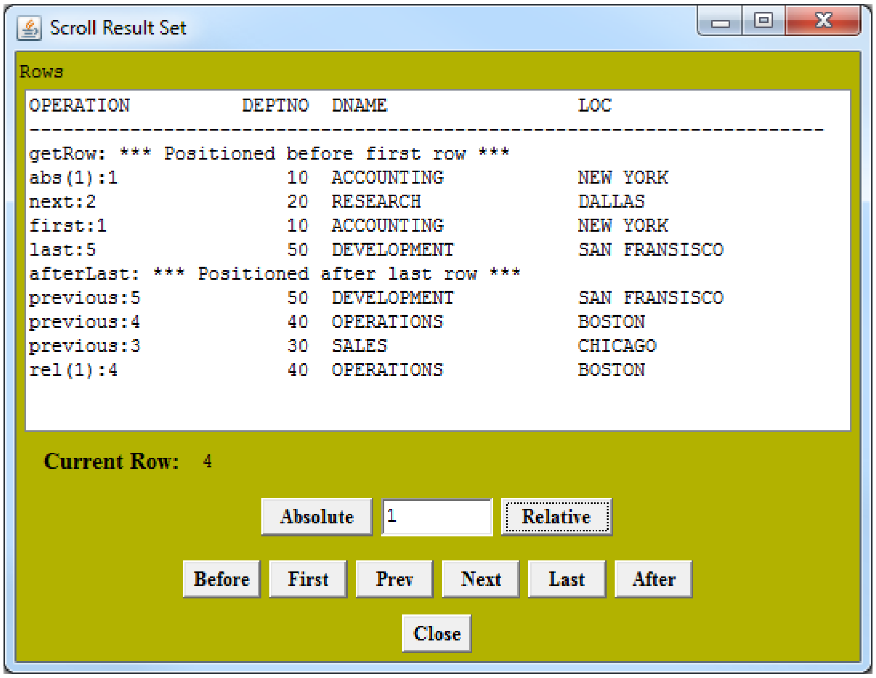


3. Select **Statement / Execute Stmt Query**.
4. Type a Select statement and click **Submit**. Then, click **Close**.



5. Select **Results / Scroll Results**. The **Scroll Result Set** window indicates that the cursor is positioned before the first row.

6. Click the **Absolute**, **Relative**, **Before**, **First**, **Prev**, **Next**, **Last**, and **After** buttons as appropriate to navigate through the result set. After each action, the **Scroll Result Set** window displays the data at the current position of the cursor.



7. Click **Close**.

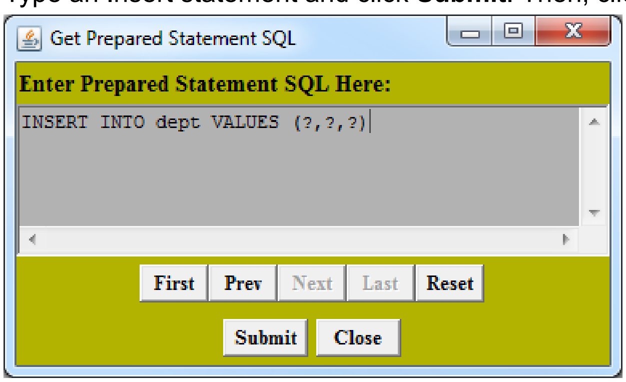
Batch Execution on a Prepared Statement

Batch execution on a prepared statement allows you to update or insert multiple records simultaneously. In some cases, this can significantly improve system performance because fewer round trips to the database are required.

To Execute a Batch on a Prepared Statement:

1. From the **Connection** window menu, select **Connection / Create Prepared Statement**.

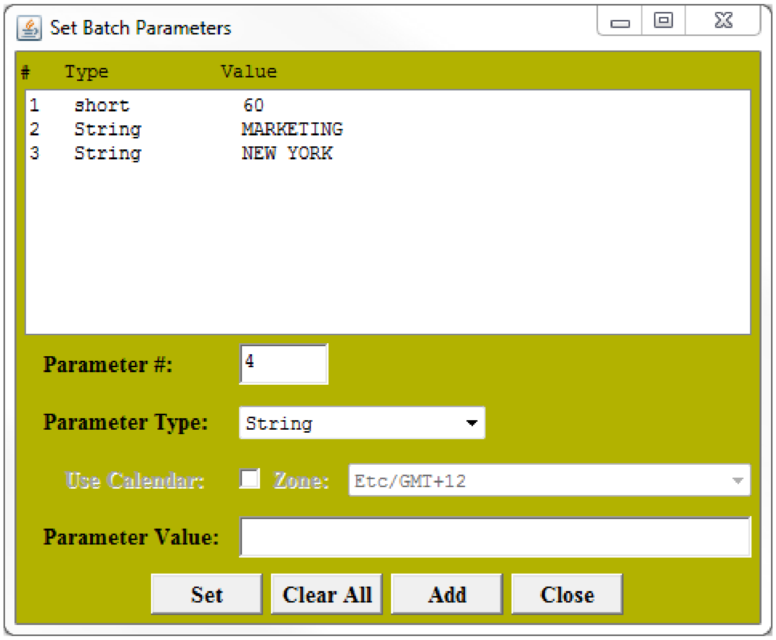
   Type an Insert statement and click **Submit**. Then, click **Close**.



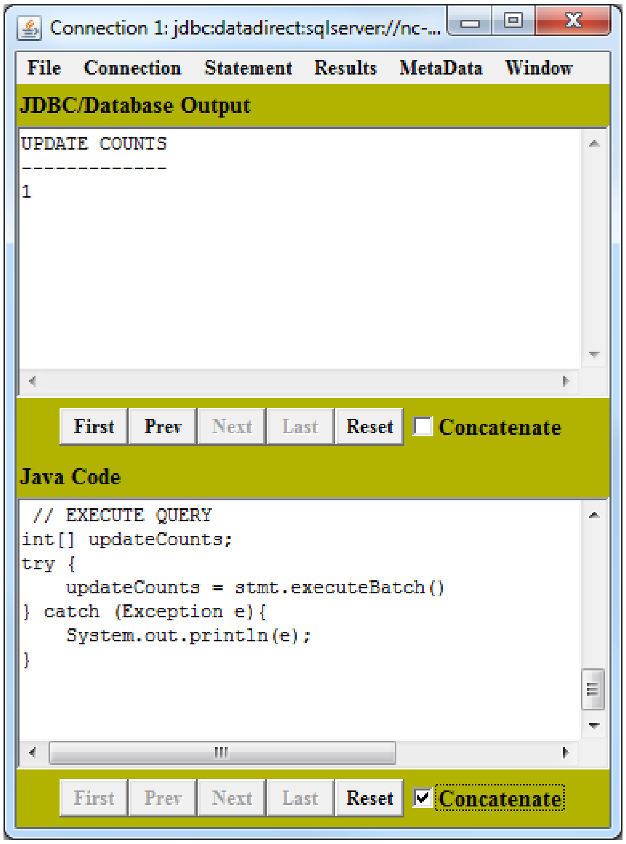
2. Select **Statement / Add Stmt Batch**.
3. For each parameter:
   a) Type the parameter number.
   b) Select the parameter type.

c) Type the parameter value.

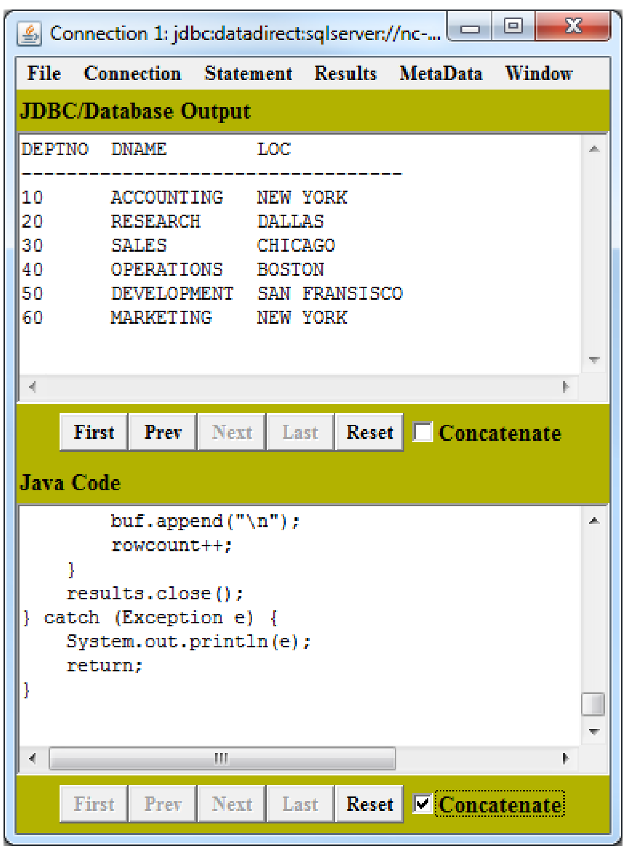
d) Click **Set**.



4. Click **Add** to add the specified set of parameters to the batch. To add multiple parameter sets to the batch, repeat Step 2 on page 304 through Step 4 on page 305 as many times as necessary. When you are finished adding parameter sets to the batch, click **Close**.
5. Select **Statement** / **Execute Stmt Batch**. DataDirect Test displays the rowcount for each of the elements in the batch.



6. If you re-execute the Select statement from Executing a Simple Select Statement on page 296, you see that the previously inserted records are returned.

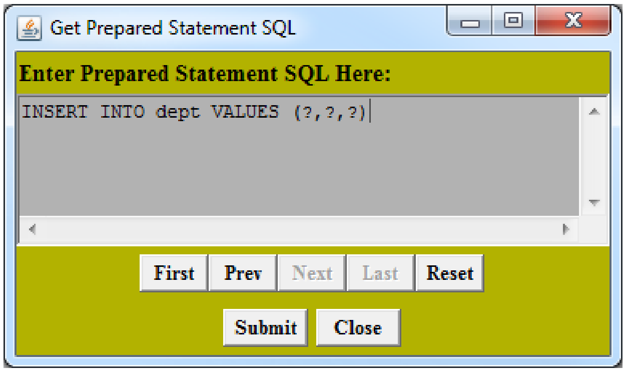


Returning ParameterMetaData

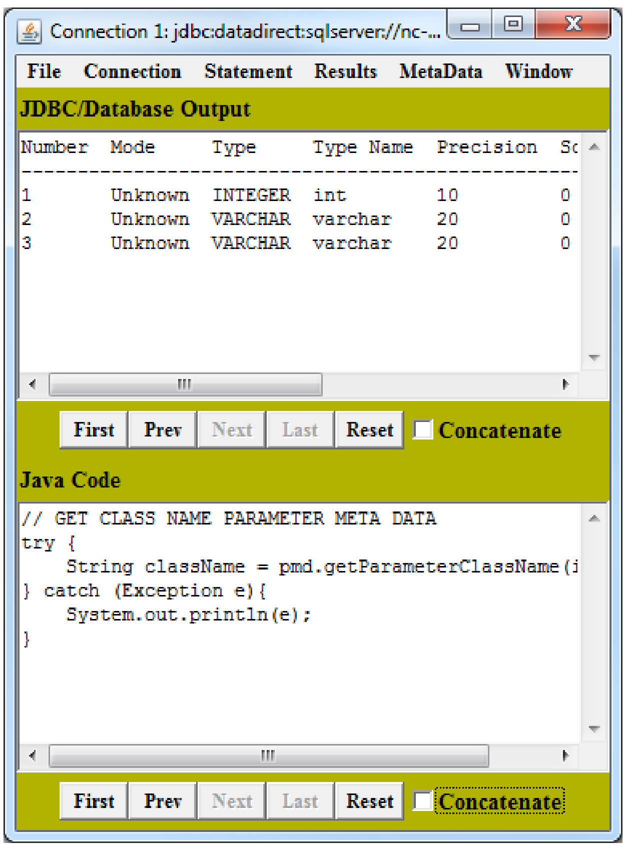
Note: Returning ParameterMetaData requires a Java SE 5 or higher JVM.

1. From the **Connection** window menu, select **Connection** / **Create Prepared Statement**.

   Type the prepared statement and click **Submit**. Then, click **Close**.



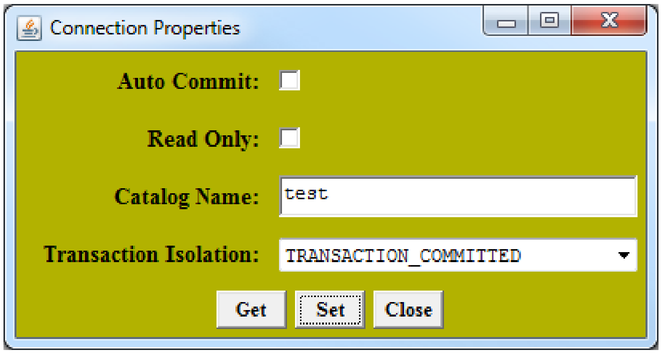
2. Select **Statement** / **Get ParameterMetaData**. The **Connection** window displays ParameterMetaData.



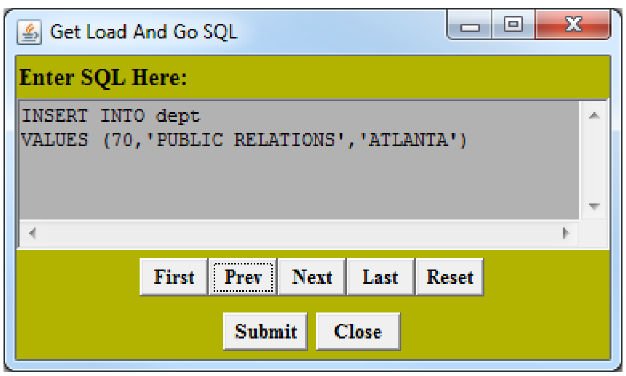
Establishing Savepoints

Note: Savepoints require a Java SE 5 or higher JVM.

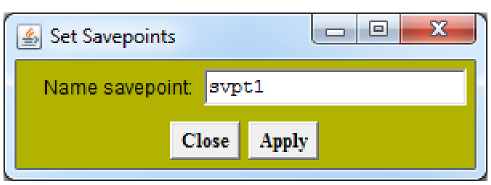
1. From the **Connection** window menu, select **Connection** / **Connection Properties**.
2. Select **TRANSACTION_COMMITTED** from the Transaction Isolation drop-down list. Do not select the Auto Commit check box.



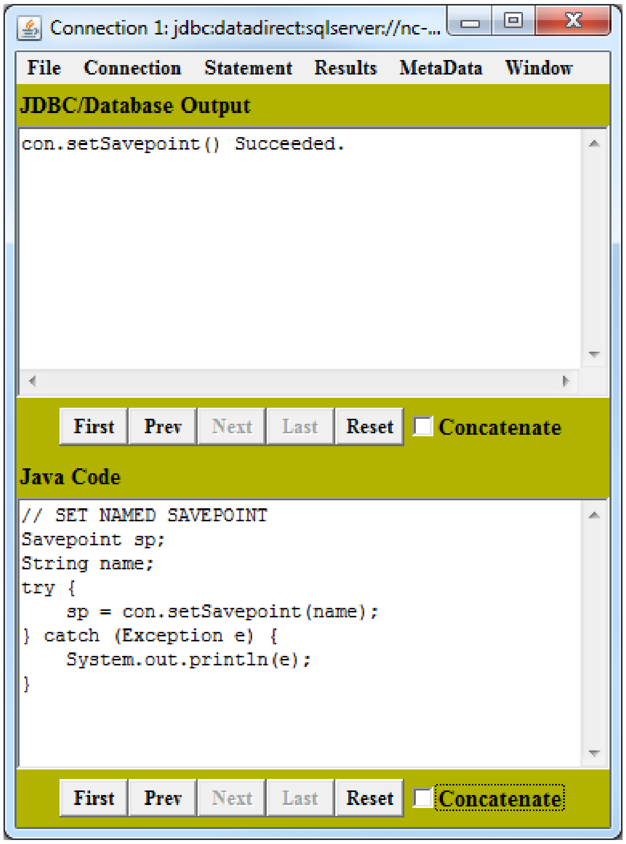
3. Click **Set**; then, click **Close**.
4. From the **Connection** window menu, select **Connection** / **Load and Go**. The **Get Load And Go SQL** window appears.
5. Type a statement and click **Submit**.



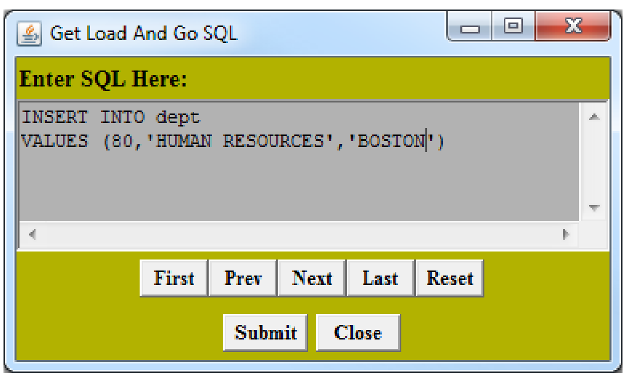
6. Select **Connection** / **Set Savepoint**.
7. In the **Set Savepoints** window, type a savepoint name.



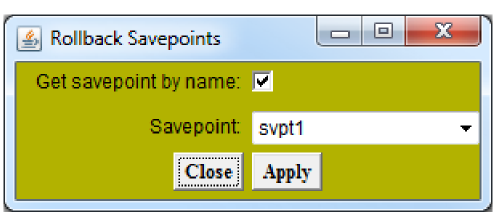
8. Click **Apply**; then, click **Close**. The **Connection** window indicates whether or not the savepoint succeeded.



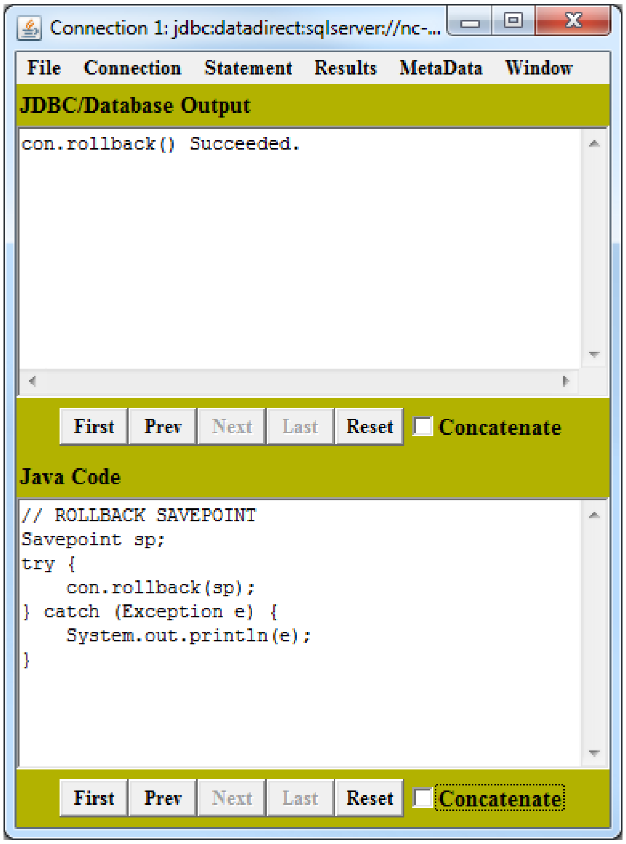
9. Return to the **Get Load And Go SQL** window and specify another statement. Click **Submit**.



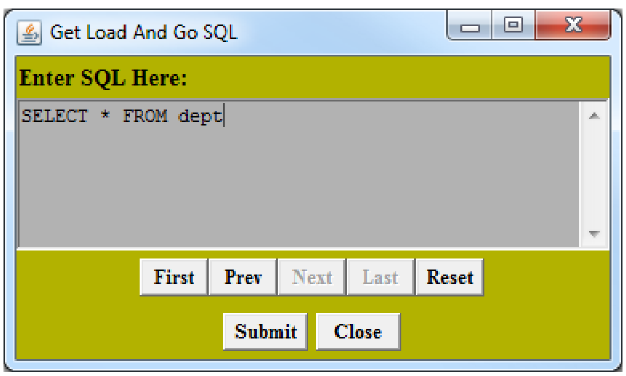
10. Select **Connection** / **Rollback Savepoint**. In the **Rollback Savepoints** window, specify the savepoint name.



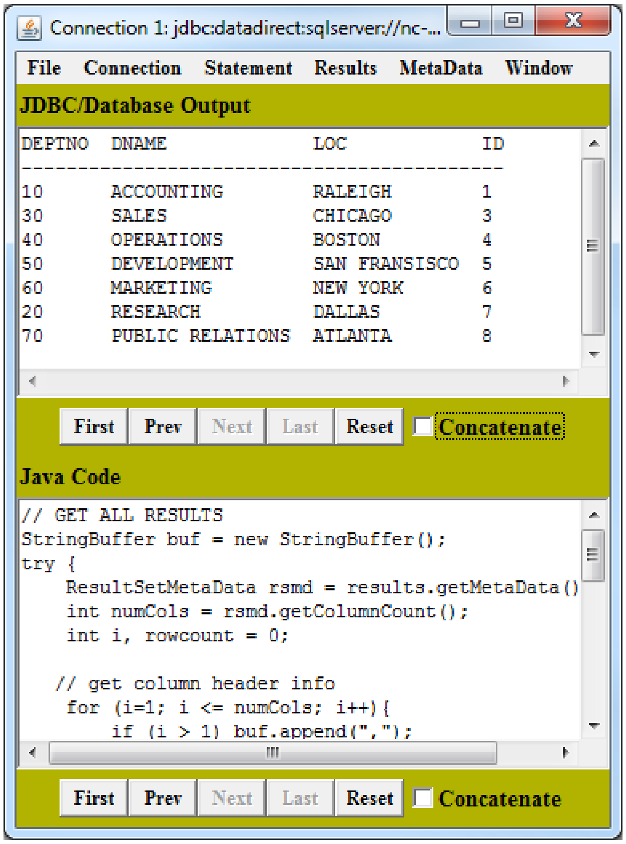
11. Click **Apply**; then, click **Close**. The **Connection** window indicates whether or not the savepoint rollback succeeded.



12. Return to the **Get Load And Go SQL** window and specify another statement.



Click **Submit**; then, click **Close**. The **Connection** window displays the data inserted before the first Savepoint. The second insert was rolled back.

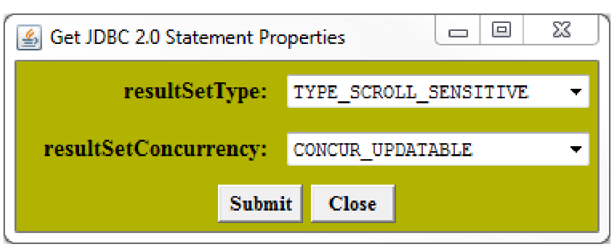


Updatable Result Sets

The following examples explain the concept of updatable result sets by deleting, inserting, and updating a row.

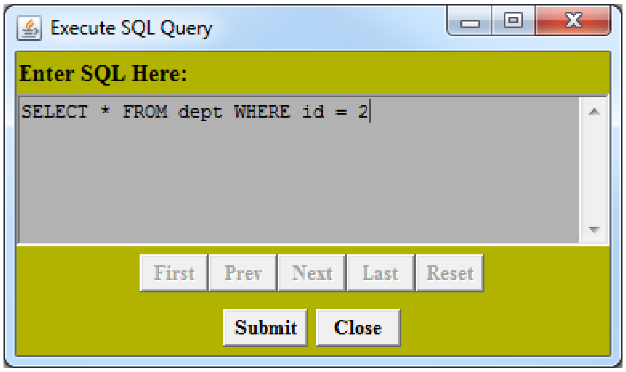
Deleting a Row

1. From the **Connection** window menu, select **Connection** / **Create JDBC 2.0 Statement**.
2. Complete the following fields:
   a) In the resultSetType field, select **TYPE_SCROLL_SENSITIVE**.
   b) In the resultSetConcurrency field, select **CONCUR_UPDATABLE**.



3. Click **Submit**; then, click **Close**.
4. Select **Statement** / **Execute Stmt Query**.

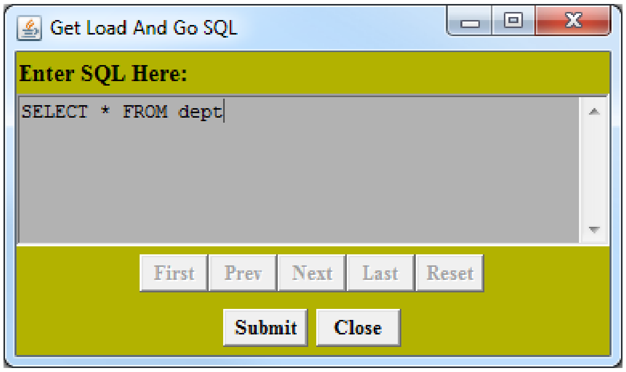
5. Specify the Select statement and click **Submit**. Then, click **Close**.



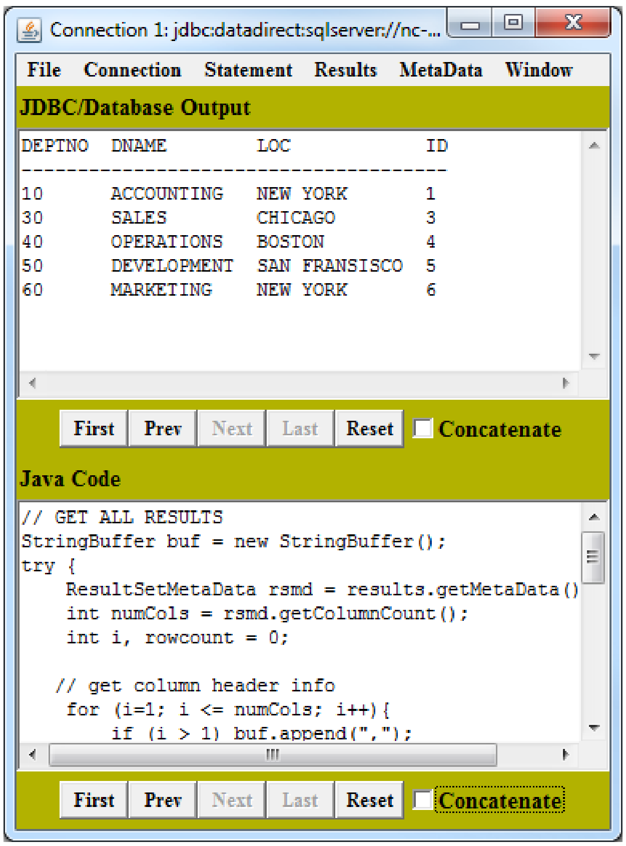
6. Select **Results** / **Inspect Results**. The **Inspect Result Set** window appears.

7. Click **Next**. Current Row changes to 1.
8. Click **Delete Row**.

9. To verify the result, return to the Connection menu and select **Connection** / **Load and Go**. The **Get Load And Go SQL** window appears.
10. Specify the statement that you want to execute and click **Submit**. Then, click **Close**.

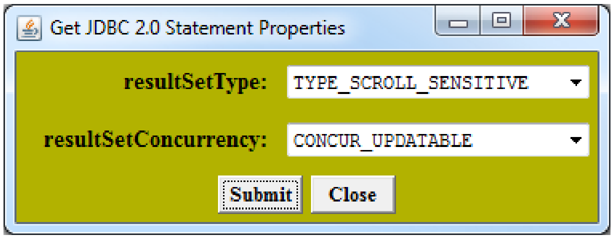


11. The **Connection** window shows that the row has been deleted.

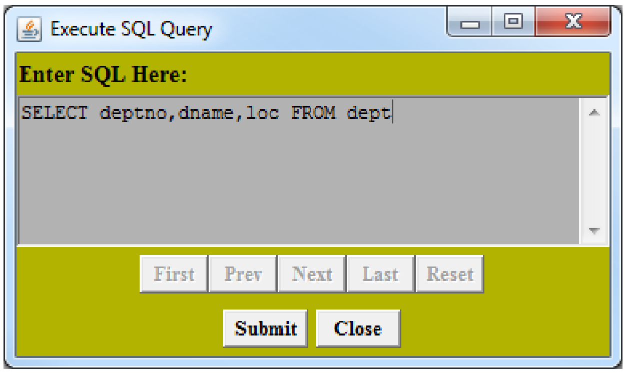


Inserting a Row

1. From the **Connection** window menu, select **Connection** / **Create JDBC 2.0 Statement**.
2. Complete the following fields:
   a) In the resultSetType field, select **TYPE_SCROLL_SENSITIVE**.
   b) In the resultSetConcurrency field, select **CONCUR_UPDATABLE**.

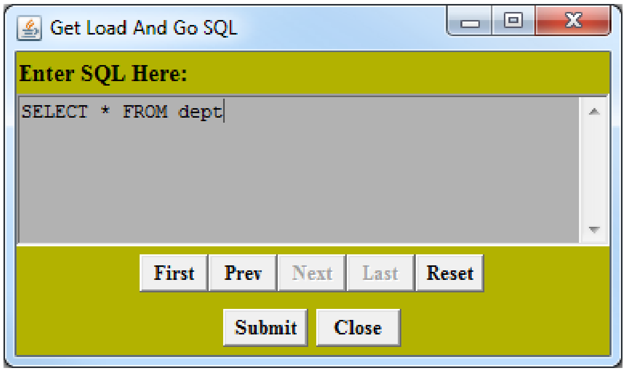


3. Click **Submit**; then, click **Close**.
4. Select **Statement** / **Execute Stmt Query**.
5. Specify the Select statement that you want to execute and click **Submit**. Then, click **Close**.

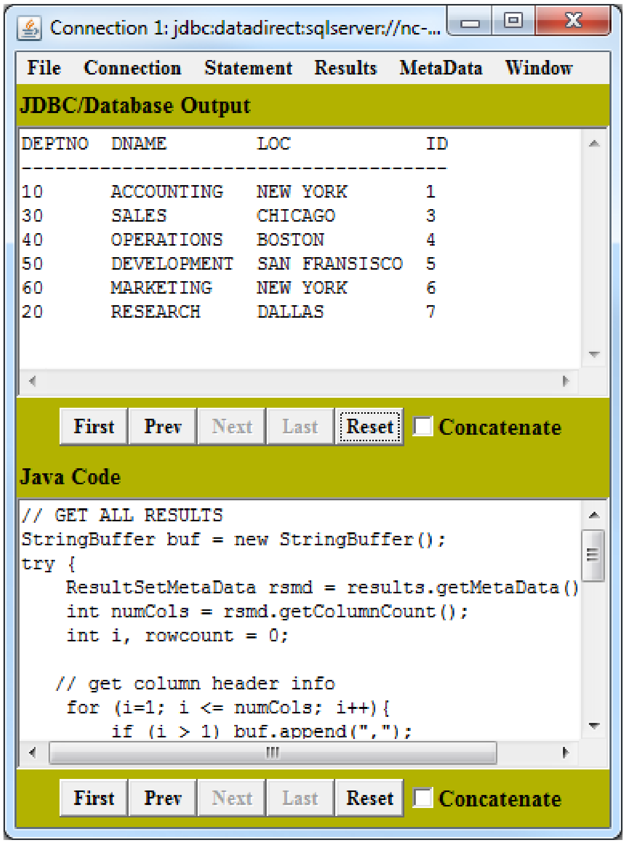


6. Select **Results** / **Inspect Results**. The **Inspect Result Set** window appears.

7. Click **Move to insert row**; Current Row is now Insert row.
8. Change Data Type to int. In Set Cell Value, enter `20`. Click **Set Cell**.
9. Select the second row in the top pane. Change the Data Type to String. In Set Cell Value, enter `RESEARCH`. Click **Set Cell**.
10. Select the third row in the top pane. In Set Cell Value, enter `DALLAS`. Click **Set Cell**.
11. Click **Insert Row**.
12. To verify the result, return to the Connection menu and select **Connection** / **Load and Go**. The **Get Load And Go SQL** window appears.
13. Specify the statement that you want to execute and click **Submit**. Then, click **Close**.



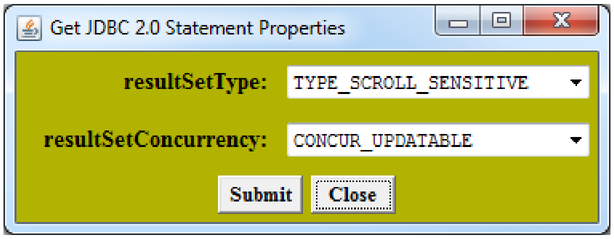
14. The **Connection** window shows the newly inserted row.



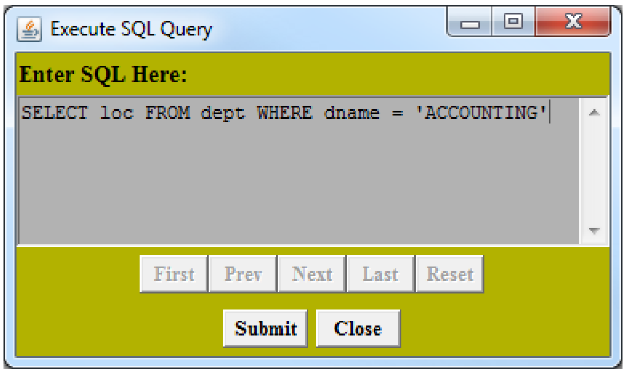
Caution: The ID will be 3 for the row you just inserted because it is an auto increment column.

Updating a Row

1. From the **Connection** window menu, select **Connection** / **Create JDBC 2.0 Statement**.
2. Complete the following fields:
   a) In the resultSetType field, select **TYPE_SCROLL_SENSITIVE**.
   b) In the resultSetConcurrency field, select **CONCUR_UPDATABLE**.

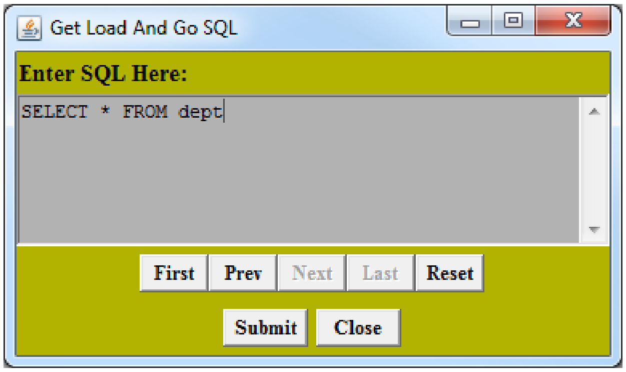


3. Click **Submit**; then, click **Close**.
4. Select **Statement** / **Execute Stmt Query**.
5. Specify the Select statement that you want to execute.

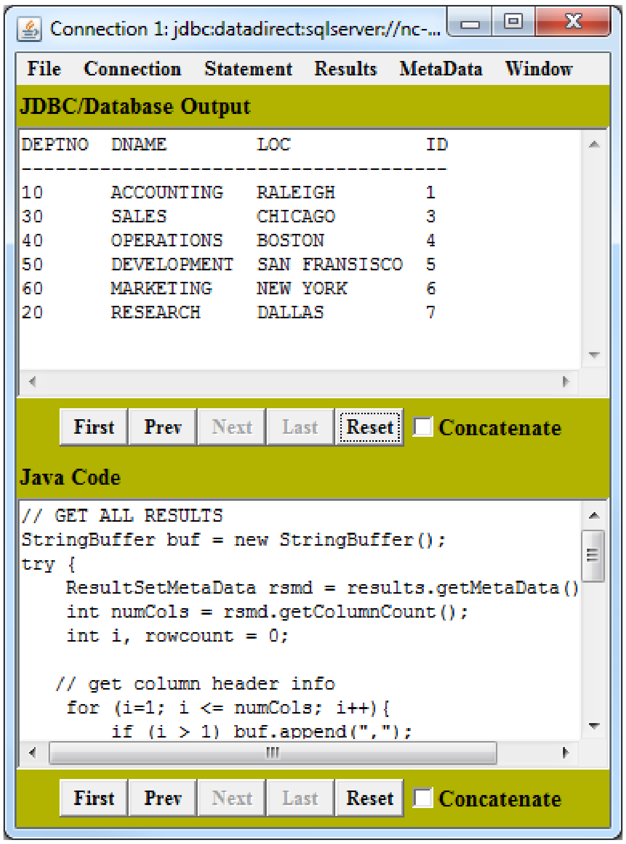


6. Click **Submit**; then, click **Close**.
7. Select **Results** / **Inspect Results**. The **Inspect Result Set** window appears.

8. Click **Next**. Current Row changes to 1.
9. In Set Cell Value, type `RALEIGH`. Then, click **Set Cell**.
10. Click **Update Row**.
11. To verify the result, return to the Connection menu and select **Connection** / **Load and Go**. The **Get Load And Go SQL** window appears.
12. Specify the statement that you want to execute.



13. Click **Submit**; then, click **Close**.
14. The **Connection** window shows LOC for accounting changed from NEW YORK to RALEIGH.



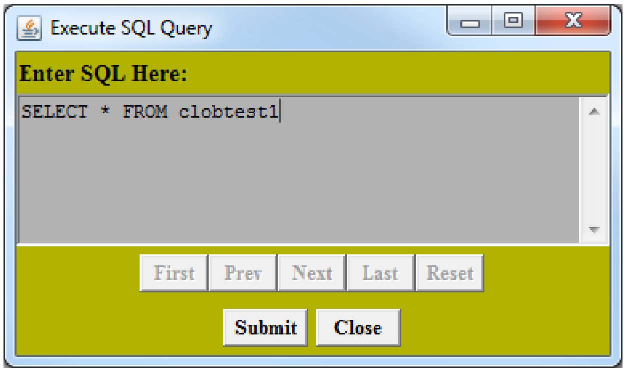
Retrieving Large Object Data

Note: LOB support (Blobs and Clobs) requires a Java SE 5 or higher JVM.

The following example uses Clob data; however, this procedure also applies to Blob data. This example illustrates only one of multiple ways in which LOB data can be processed.

1. From the **Connection** window menu, select **Connection** / **Create Statement**.
2. Select **Statement** / **Execute Stmt Query**.

3. Specify the Select statement that you want to execute.

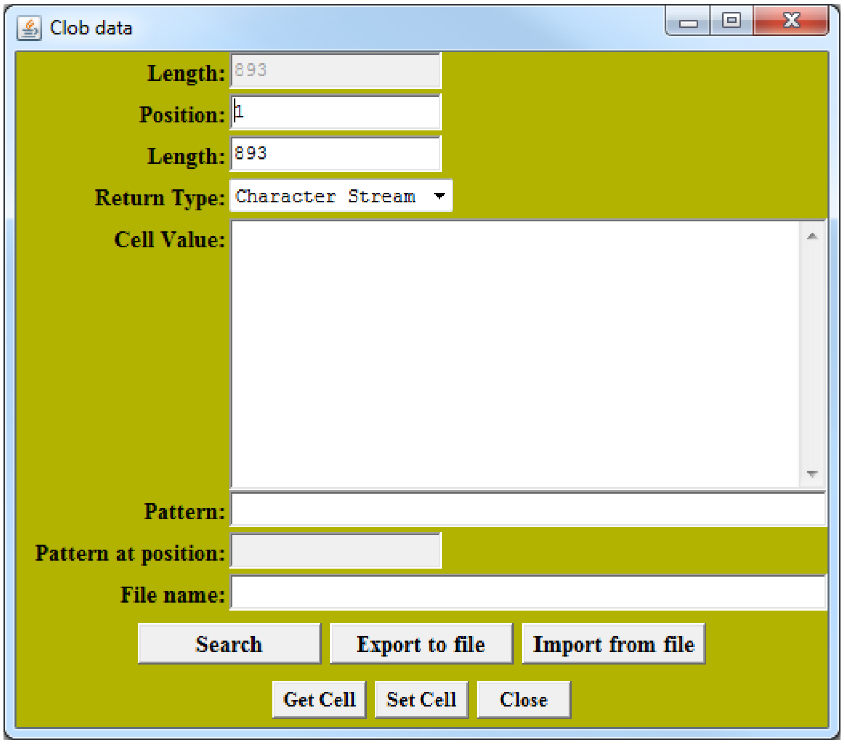


4. Click **Submit**; then, click **Close**.
5. Select **Results** / **Inspect Results**. The **Inspect Result Set** window appears.
6. Click **Next**. Current Row changes to 1.

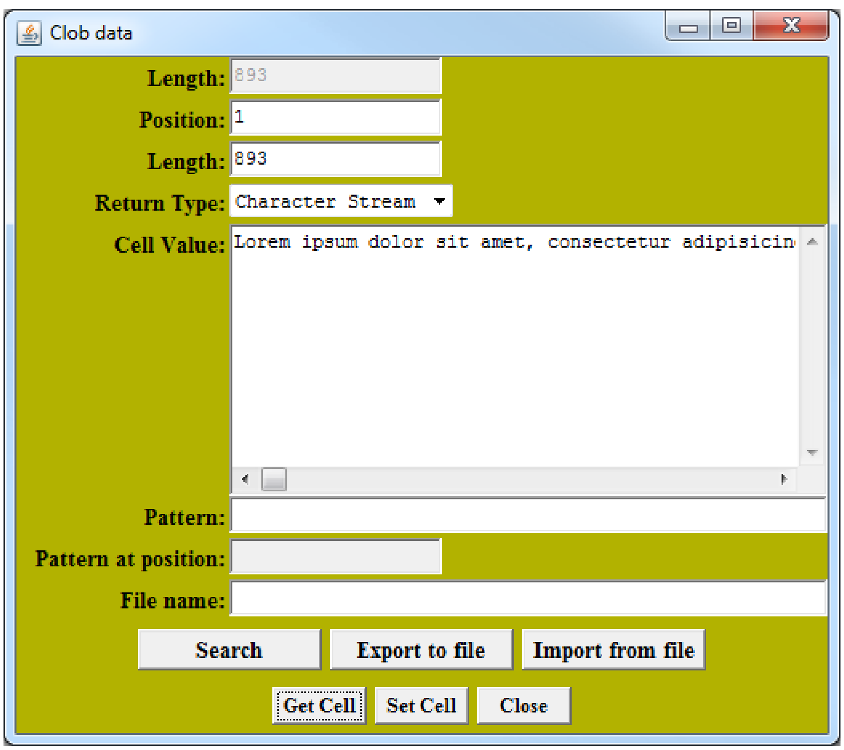
7. Deselect **Auto Traverse**. This disables automatic traversal to the next row.
8. Click **Get Cell**. Values are returned in the Get Cell Value field.

9. Change the data type to Clob.

10. Click **Get Cell**. The **Clob data** window appears.



11. Click **Get Cell**. Values are returned in the Cell Value field.



9

Tracking JDBC Calls with DataDirect Spy

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

- Logging is JDBC 4.0-compliant.
- Logging is consistent, regardless of which DataDirect Connect Series *for* JDBC driver is used.
- All parameters and function results for JDBC calls can be logged.
- Logging works with all DataDirect Connect Series *for* JDBC drivers.
- Logging can be enabled without changing the application.

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

Once logging is enabled for a connection, you can turn it on and off at runtime using the setEnableLogging() method in the com.ddtek.jdbc.extensions.ExtLogControl interface. See Troubleshooting Your Application on page 357 for information about using a DataDirect Spy log for troubleshooting.

For details, see the following topics:

- Enabling DataDirect Spy

Enabling DataDirect Spy

You can enable DataDirect Spy for a connection using either of the following methods:

- Specifying the SpyAttributes connection property for connections using the JDBC Driver Manager. See Using the JDBC Driver Manager on page 326 for instructions.
- Specifying DataDirect Spy attributes using a JDBC data source. See Using JDBC Data Sources on page 327 for instructions.

You can set one or multiple options to customize DataDirect Spy logging. See DataDirect Spy Attributes on page 328 for a complete list of supported attributes.

Using the JDBC Driver Manager

The SpyAttributes connection property allows you to specify a semi-colon separated list of DataDirect Spy attributes (see DataDirect Spy Attributes on page 328). The format for the value of the SpyAttributes property is:

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

where *spy_attribute* is any valid DataDirect Spy attribute. See DataDirect Spy Attributes on page 328 for a list of supported attributes.

Example on Windows:

The following example uses the JDBC Driver Manager to connect to Microsoft SQL Server while enabling DataDirect Spy:

```
Class.forName("com.ddtek.jdbc.sqlserver.SQLServerDriver");
Connection conn = DriverManager.getConnection
    ("jdbc:datadirect:sqlserver://Server1:1433;User=TEST;Password=secret;
    SpyAttributes=(log=(filePrefix)C:\\temp\\spy_;linelimit=80;logTName=yes;
    timestamp=yes)");
```

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

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

Example on UNIX:

The following code example uses the JDBC Driver Manager to connect to DB2 while enabling DataDirect Spy:

```
Class.forName("com.ddtek.jdbc.db2.DB2Driver");
Connection conn = DriverManager.getConnection
    ("jdbc:datadirect:db2://Server1:50000;User=TEST;Password=secret;
    SpyAttributes=(log=(filePrefix)/tmp/spy_;logTName=yes;timestamp=yes)");
```

Using this example, DataDirect Spy loads the DB2 driver and logs all JDBC activity to the spy_*x*.log file located in the `/tmp directory` (`log=(filePrefix)/tmp/spy_`), where *x* is an integer that increments by 1 for each connection on which the prefix is specified. The spy_*x*.log file includes the name of the current thread (`logTName=yes`) and a timestamp on each line in the log (`timestamp=yes`).

Using JDBC Data Sources

The drivers implement the following JDBC features:

- JNDI for Naming Databases
- Connection Pooling
- Java Transaction API (JTA)

Note: JTA is only supported by the DB2, Informix, Oracle, OpenEdge, SQL Server, and Sybase drivers.

You can use DataDirect Spy to track JDBC calls made by a running application with any of these features. The com.ddtek.jdbcx.datasource.*Driver*DataSource class, where *Driver* is the driver name, supports setting a semi-colon-separated list of DataDirect Spy attributes (see DataDirect Spy Attributes on page 328).

Refer to the *DataDirect Connect Series for JDBC User's Guide* for more information about configuring data sources.

Example on Windows:

The following example creates a JDBC data source for the DB2 driver, which enables DataDirect Spy.

```
DB2DataSource sds=new DB2DataSource():
sds.setServerName("Server1");
sds.setPortNumber(50000);
sds.setSpyAttributes("log=(file)C:\\temp\\spy.log;logIS=yes;logTName=yes");
Connection conn=sds.getConnection("TEST","secret");
...
```

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

Using this example, DataDirect Spy would load the DB2 driver and log all JDBC activity to the spy.log file located in the `C:\temp` directory (`log=(file)C:\\temp\\spy.log`). In addition to regular JDBC activity, the spy.log file also logs activity on InputStream and Reader objects (`logIS=yes`). It also includes the name of the current thread (`logTName=yes`).

Example on UNIX:

The following example creates a JDBC data source for the Oracle driver, which enables DataDirect Spy.

```
OracleDataSource mds = new OracleDataSource();
mds.setServerName("Server1");
mds.setPortNumber(1521);
mds.setSID("ORCL");...
sds.setSpyAttributes("log=(file)/tmp/spy.log;logTName=yes");
Connection conn=sds.getConnection("TEST","secret");
...
```

Using this example, DataDirect Spy would load the Oracle driver and log all JDBC activity to the spy.log file located in the `/tmp` directory (`log=(file)/tmp/spy.log`). The `spy.log` file includes the name of the current thread (`logTName=yes`).

DataDirect Spy Attributes

DataDirect Spy supports the attributes described in the following table.

Table 36: DataDirect Spy Attributes

Attribute	Description
linelimit=*numberofchars*	Sets the maximum number of characters that DataDirect Spy logs on a single line. The default is `0` (no maximum limit).
load=*classname*	Loads the driver specified by *classname*. For example, the com.ddtek.jdbc.db2.DB2Driver class name loads the DB2 driver.
log=(file)*filename*	Directs logging to the file specified by *filename*. For Windows, if coding a path to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: `log=(file)C:\\temp\\spy.log;logIS=yes;logTName=yes`.

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

log=System.out	Directs logging to the Java output standard, System.out.
logIS={`yes` | `no` | `nosingleread`}	Specifies whether DataDirect Spy logs activity on InputStream and Reader objects. When logIS=`nosingleread`, logging on InputStream and Reader objects is active; however, logging of the single-byte read InputStream.read or single-character Reader.read is suppressed to prevent generating large log files that contain single-byte or single character read messages. The default is `no`.
logLobs={`yes` | `no`}	Specifies whether DataDirect Spy logs activity on BLOB and CLOB objects.

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

10

Connection Pool Manager

Connection pooling means that connections are reused rather than created each time a connection is requested. Your application can use connection pooling through the DataDirect Connection Pool Manager.

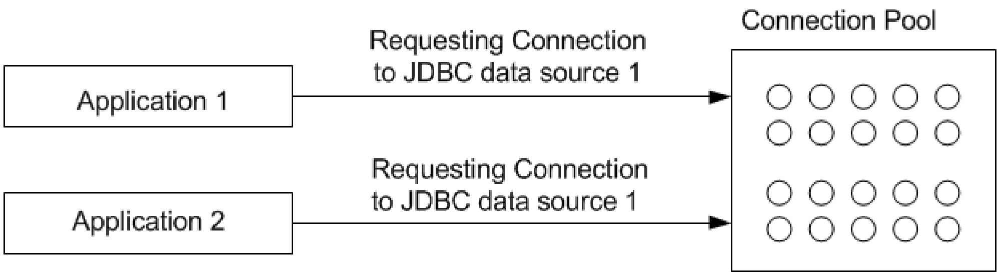
Connection pooling is performed in the background and does not affect how an application is coded; however, the application must use a DataSource object (an object implementing the DataSource interface) to obtain a connection instead of using the DriverManager class. A class implementing the DataSource interface may or may not provide connection pooling. A DataSource object registers with a JNDI naming service. Once a DataSource object is registered, the application retrieves it from the JNDI naming service in the standard way.

For details, see the following topics:

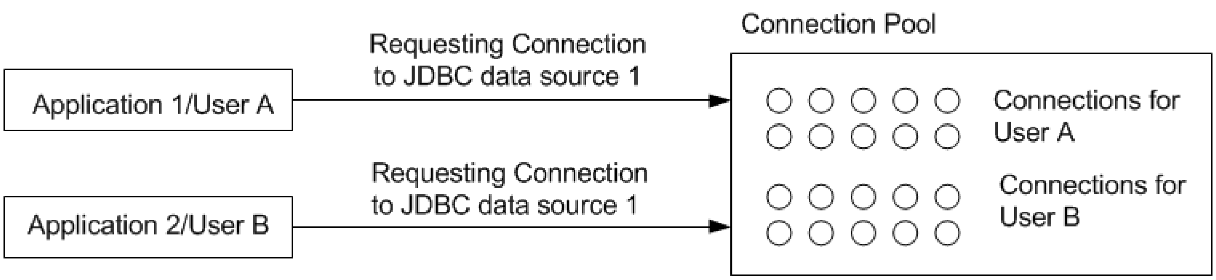
- About JDBC Connection Pools
- Configuring the Connection Pool
- Checking the Pool Manager Version
- Enabling Pool Manager Tracing
- Using a DataDirect Connection Pool
- Connecting Using a Connection Pool
- Closing the Connection Pool
- DataDirect Connection Pool Manager Interfaces

About JDBC Connection Pools

There is a one-to-one relationship between a JDBC connection pool and a data source, so the number of connection pools used by an application depends on the number of data sources configured to use connection pooling. If multiple applications are configured to use the same data source, those applications share the same connection pool as shown in the following figure.



An application may use only one data source, but allow multiple users, each with their own set of login credentials. The connection pool contains connections for all unique users using the same data source as shown in the following figure.



Connections are one of the following types:

- *Active connection* is a connection that is in use by the application.
- *Idle connection* is a connection in the connection pool that is available for use.

Configuring the Connection Pool

You can configure attributes of a connection pool for optimal performance and scalability using the methods provided by the DataDirect Connection Pool Manager classes (see DataDirect Connection Pool Manager Interfaces on page 339).

Some commonly set connection pool attributes include:

- Minimum pool size, which is the minimum number of connections that will be kept in the pool for each user
- Maximum pool size, which is the maximum number of connections in the pool for each user
- Initial pool size, which is the number of connections created for each user when the connection pool is initialized
- Maximum idle time, which is the amount of time a pooled connection remains idle before it is removed from the connection pool

See Understanding the Maximum Pool Size on page 333 for more information about how the Pool Manager implements the maximum pool size. In addition, the Pool Manager implements minimum pool size, maximum pool size, and initial pool size differently depending on whether reauthentication is enabled. See Using Reauthentication with the Pool Manager on page 333 for details.

Understanding the Maximum Pool Size

You set the maximum pool size using the PooledConnectionDataSource.setMaxPoolSize() method. For example, the following code sets the maximum pool size to 10:

```
ds.setMaxPoolSize(10);
```

You can control how the Pool Manager implements the maximum pool size by setting the PooledConnectionDataSource.setMaxPoolSizeBehavior() method:

- If `setMaxPoolSizeBehavior(softCap)`, the number of active connections can exceed the maximum pool size, but the number of idle connections for each user in the pool cannot exceed this limit. If a user requests a connection and an idle connection is unavailable, the Pool Manager creates a new connection for that user. When the connection is no longer needed, it is returned to the pool. If the number of idle connections exceeds the maximum pool size, the Pool Manager closes idle connections to enforce the pool size limit. This is the default behavior.
- If `setMaxPoolSizeBehavior(hardCap)`, the total number of active and idle connections cannot exceed the maximum pool size. Instead of creating a new connection for a connection request if an idle connection is unavailable, the Pool Manager queues the connection request until a connection is available or the request times out. This behavior is useful if your client or application server has memory limitations or if your database server is licensed for only a certain number of connections.

See PooledConnectionDataSource on page 340 for more information about these methods.

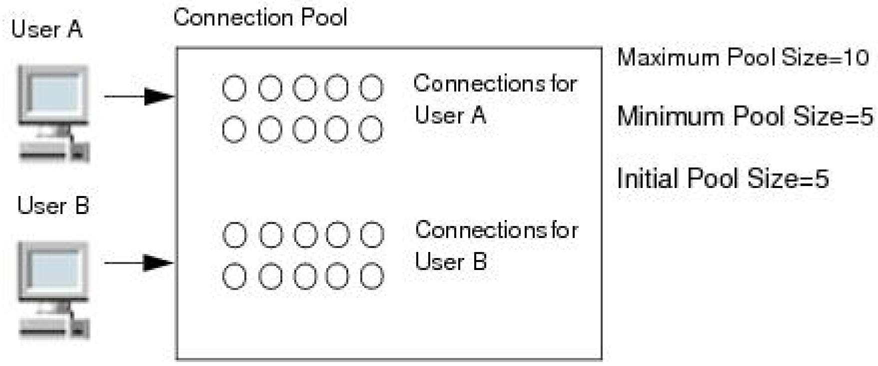
Using Reauthentication with the Pool Manager

Reauthentication, or the ability to switch a user on a connection, is a useful strategy for minimizing the number of connections that are required in a connection pool. Refer to the *DataDirect Connect Series for JDBC User's Guide* for an introduction to reauthentication.

If you are using the DataDirect Connection Pool Manager for connection pooling, you can enable reauthentication in the Pool Manager. By default, reauthentication is disabled. To enable reauthentication, call `setReauthentication(enable)` on the PooledConnectionDataSource. To disable reauthentication, call `setReauthentication(disable)`.

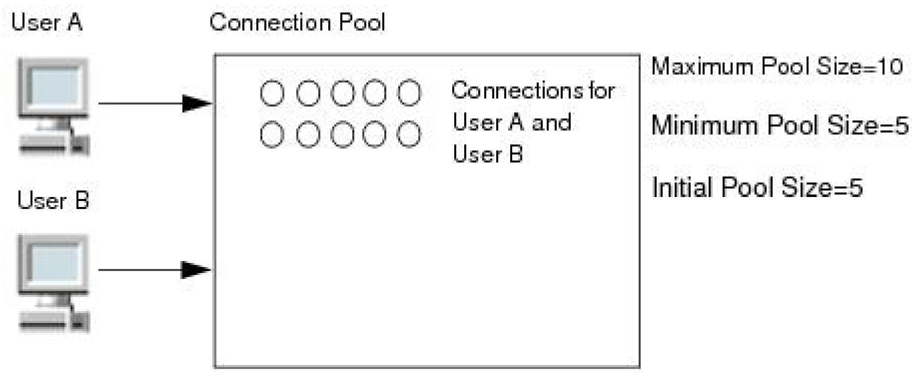
The Pool Manager implements the maximum pool size, minimum pool size, and initial pool size attributes differently depending on whether reauthentication is enabled. For example, in both of the following figures, the maximum pool size is set to a value of `10`, the minimum pool size is set to `5`, and the initial pool size is set to `5`.

The following figure shows a connection pool that is configured to work without reauthentication while using the default behavior for maximum pool size. When User A requests a connection, the Pool Manager assigns an available connection associated with User A. Similarly, if User B requests a connection, the Pool Manager assigns an available connection associated with User B. If a connection is unavailable for a particular user, the Pool Manager creates a new connection for that user. Because the maximum pool size is set to `10`, a maximum of 10 idle connections can exist for each user. In this case, the total number of idle connections is 20, or 10 idle connections for each user.



The Pool Manager implements the minimum pool size and initial pool size in a similar way. The Pool Manager initially populates five connections for User A and five connections for User B, and ensures that, at a minimum, five idle connections are maintained in the pool for each user.

In contrast, the following figure shows a connection pool that is configured to work with reauthentication while using the default behavior for maximum pool size. The Pool Manager treats all connections as one group of connections. When User A requests a connection, the Pool Manager assigns an available connection associated with User A. Similarly, when User B requests a connection, the Pool Manager assigns an available connection associated with User B. If a connection is unavailable for a particular user, the Pool Manager assigns any available connection to that user, switching the user associated with the connection to the new user. In this case, the maximum number of idle connections in the pool is 10, regardless of how many users are using the connection pool.



The Pool Manager initially populates the pool with five connections and ensures that, at a minimum, five idle connections are maintained in the pool for all users.

Checking the Pool Manager Version

To check the version of your DataDirect Connection Pool Manager, navigate to the directory containing the DataDirect Connection Pool Manager (*install_dir*/`pool manager` where *install_dir* is your product installation directory). At a command prompt, enter the command:

On Windows:

```
java -classpath poolmgr_dir\pool.jar com.ddtek.pool.PoolManagerInfo
```

On UNIX:

```
java -classpath poolmgr_dir/pool.jar com.ddtek.pool.PoolManagerInfo
```

where:

poolmgr_dir

is the directory containing the DataDirect Connection Pool Manager.

Alternatively, you can obtain the name and version of the DataDirect Connection Pool Manager programmatically by invoking the following static methods: com.ddtek.pool.PoolManagerInfo.getPoolManagerName() and com.ddtek.pool.PoolManagerInfo.getPoolManagerVersion().

Enabling Pool Manager Tracing

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

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

See Troubleshooting Connection Pooling on page 360 for information about using a Pool Manager trace file for troubleshooting.

Using a DataDirect Connection Pool

1. Create and register with JNDI a DataDirect Connect Series *for* JDBC driver DataSource object. Once created, this DataSource object can be used by a connection pool (PooledConnectionDataSource object created in Step 2 on page 335) to create connections for one or multiple connection pools.
2. To create a connection pool, you must create and register with JNDI a PooledConnectionDataSource object. A PooledConnectionDataSource creates and manages one or multiple connection pools. The PooledConnectionDataSource uses the driver DataSource object created in Step 1 on page 335 to create the connections for the connection pool.

Creating a Driver DataSource Object

The following Java code example creates a DataDirect Connect Series *for* JDBC driver DataSource object and registers it with a JNDI naming service.

The DataSource class is provided by DataDirect Connect Series *for* JDBC and is database-dependent. In this example we use Oracle, so the DataSource class is OracleDataSource. Refer to the appropriate driver chapter in the *DataDirect Connect Series for JDBC User's Guide* for the name of the DataSource class for your driver.

Note: The DataSource class implements the ConnectionPoolDataSource interface for pooling in addition to the DataSource interface for non-pooling.

```
//********************************************************************
// This code creates a DataDirect Connect Series for JDBC data source and
// registers it to a JNDI naming service. This JDBC data source uses the
// DataSource implementation provided by DataDirect Connect Series
// for JDBC Drivers.
//
// This data source registers its name as <jdbc/ConnectSparkyOracle>.
//// NOTE: To connect using a data source, the driver needs to access a JNDI data
// store to persist the data source information. To download the JNDI File
```

```
// System Service Provider, go to:
//
// http://www.oracle.com/technetwork/java/javasebusiness/downloads/
// java-archive-downloads-java-plat-419418.html#7110-jndi-1.2.1-oth-JPR
////
// Make sure that the fscontext.jar and providerutil.jar files from the
// download are on your classpath.
//************************************************************************
// From DataDirect Connect Series for JDBC:
import com.ddtek.jdbcx.oracle.OracleDataSource;
import javax.sql.*;
import java.sql.*;
import javax.naming.*;
import javax.naming.directory.*;
import java.util.Hashtable;
public class OracleDataSourceRegisterJNDI
{   public static void main(String argv[])
   {
      try {
      // Set up data source reference data for naming context:
      // ----------------------------------------------------
      // Create a class instance that implements the interface
      // ConnectionPoolDataSource
      OracleDataSource ds = new OracleDataSource();
      ds.setDescription("Oracle on Sparky - Oracle Data Source");
      ds.setServerName("sparky");
      ds.setPortNumber(1521);
      ds.setUser("scott");
      ds.setPassword("test");
      // Set up environment for creating initial context
      Hashtable env = new Hashtable();
      env.put(Context.INITIAL_CONTEXT_FACTORY,
         "com.sun.jndi.fscontext.RefFSContextFactory");
      env.put(Context.PROVIDER_URL, "file:c:\\JDBCDataSource");
      Context ctx = new InitialContext(env);
      // Register the data source to JNDI naming service
      ctx.bind("jdbc/ConnectSparkyOracle", ds);
      } catch (Exception e) {
      System.out.println(e);
      return;
      }
   } // Main
   // class OracleDataSourceRegisterJNDI
```

Creating the Connection Pool

To create a connection pool, you must create and register with JNDI a PooledConnectionDataSource object. The following Java code creates a PooledConnectionDataSource object and registers it with a JNDI naming service.

To specify the driver DataSource object to be used by the connection pool to create pooled connections, set the parameter of the DataSourceName method to the JNDI name of a registered driver DataSource object. For example, the following code sets the parameter of the DataSourceName method to the JNDI name of the driver DataSource object created in Creating a Driver DataSource Object on page 335.

The PooledConnectionDataSource class is provided by the DataDirect com.ddtek.pool package. See PooledConnectionDataSource on page 340 for a description of the methods supported by the PooledConnectionDataSource class.

```
//************************************************************************
// This code creates a data source and registers it to a JNDI naming service.
// This data source uses the PooledConnectionDataSource
// implementation provided by the DataDirect com.ddtek.pool package.
//
// This data source refers to a registered
// DataDirect Connect Series for JDBC driver DataSource object.
//
```

```
// This data source registers its name as <jdbc/SparkyOracle>.
//
// NOTE: To connect using a data source, the driver needs to access a JNDI data
// store to persist the data source information. To download the JNDI File
// System Service Provider, go to:
//
// http://www.oracle.com/technetwork/java/javasebusiness/downloads/
// java-archive-downloads-java-plat-419418.html#7110-jndi-1.2.1-oth-JPR
//
// Make sure that the fscontext.jar and providerutil.jar files from the
// download are on your classpath.
//*********************************************************************
// From the DataDirect connection pooling package:
import com.ddtek.pool.PooledConnectionDataSource;

import javax.sql.*;
import java.sql.*;
import javax.naming.*;
import javax.naming.directory.*;
import java.util.Hashtable;

public class PoolMgrDataSourceRegisterJNDI
{
   public static void main(String argv[])
   {
      try {
         // Set up data source reference data for naming context:
         // ----------------------------------------------------
         // Create a pooling manager's class instance that implements
         // the interface DataSource
         PooledConnectionDataSource ds = new PooledConnectionDataSource();

         ds.setDescription("Sparky Oracle - Oracle Data Source");

         // Specify a registered driver DataSource object to be used
         // by this data source to create pooled connections
         ds.setDataSourceName("jdbc/ConnectSparkyOracle");

         // The pool manager will be initiated with 5 physical connections
         ds.setInitialPoolSize(5);

         // The pool maintenance thread will make sure that there are 5
         // physical connections available
         ds.setMinPoolSize(5);

         // The pool maintenance thread will check that there are no more
         // than 10 physical connections available
         ds.setMaxPoolSize(10);

         // The pool maintenance thread will wake up and check the pool
         // every 20 seconds
         ds.setPropertyCycle(20);

         // The pool maintenance thread will remove physical connections
         // that are inactive for more than 300 seconds
         ds.setMaxIdleTime(300);

         // Set tracing off because we choose not to see an output listing
         // of activities on a connection
         ds.setTracing(false);

         // Set up environment for creating initial context
         Hashtable env = new Hashtable();
         env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
         env.put(Context.PROVIDER_URL, "file:c:\\JDBCDataSource");
         Context ctx = new InitialContext(env);

         // Register this data source to the JNDI naming service
         ctx.bind("jdbc/SparkyOracle", ds);
```

```
            catch (Exception e) {
            System.out.println(e);
            return;
         }
      }
   }
```

Connecting Using a Connection Pool

Because an application uses connection pooling by referencing the JNDI name of a registered PooledConnectionDataSource object, code changes are not required for an application to use connection pooling.

The following example shows Java code that looks up and uses the JNDI-registered PooledConnectionDataSource object created in Creating the Connection Pool on page 336.

```
//********************************************************************
// Test program to look up and use a JNDI-registered data source.
//
// To run the program, specify the JNDI lookup name for the
// command-line argument, for example:
//
//     java  TestDataSourceApp  <jdbc/SparkyOracle>
//********************************************************************
import javax.sql.*;
import java.sql.*;
import javax.naming.*;
import java.util.Hashtable;
public class TestDataSourceApp
{   public static void main(String argv[])
   {
      String strJNDILookupName = "";
      // Get the JNDI lookup name for a data source
      int nArgv = argv.length;
      if (nArgv != 1) {
         // User does not specify a JNDI lookup name for a data source,
         System.out.println(
            "Please specify a JNDI name for your data source");
         System.exit(0);
         else {
         strJNDILookupName = argv[0];
      }
      DataSource ds = null;
      Connection con = null;
      Context ctx = null;
      Hashtable env = null;
      long nStartTime, nStopTime, nElapsedTime;
      // Set up environment for creating InitialContext object
      env = new Hashtable();
      env.put(Context.INITIAL_CONTEXT_FACTORY,
         "com.sun.jndi.fscontext.RefFSContextFactory");
      env.put(Context.PROVIDER_URL, "file:c:\\JDBCDataSource");
      try {
         // Retrieve the DataSource object that is bound to the logical
         // lookup JNDI name
         ctx = new InitialContext(env);
         ds = (DataSource) ctx.lookup(strJNDILookupName);
         catch (NamingException eName) {
         System.out.println("Error looking up " +
            strJNDILookupName + ": " +eName);
         System.exit(0);
      }
      int numOfTest = 4;
      int [] nCount = {100, 100, 1000, 3000};
      for (int i = 0; i < numOfTest; i ++) {
         // Log the start time
```

```
         nStartTime = System.currentTimeMillis();
         for (int j = 1; j <= nCount[i]; j++) {
            // Get Database Connection
            try {
                con = ds.getConnection("scott", "tiger");
                // Do something with the connection
                // ...
                // Close Database Connection
                if (con != null) con.close();
                } catch (SQLException eCon) {
                System.out.println("Error getting a connection: " + eCon);
                System.exit(0);
                } // try getConnection
             } // for j loop
         // Log the end time
         nStopTime = System.currentTimeMillis();
         // Compute elapsed time
         nElapsedTime = nStopTime - nStartTime;
         System.out.println("Test number " + i + ": looping " +
            nCount[i] + " times");
         System.out.println("Elapsed Time: " + nElapsedTime + "\n");
      } // for i loop
      // All done
      System.exit(0);
      // Main
} // TestDataSourceApp
```

Note: To use non-pooled connections, specify the JNDI name of a registered driver DataSource object as the command-line argument when you run the preceding application. For example, the following command specifies the driver DataSource object created in Creating a Driver DataSource Object on page 335: `java TestDataSourceApp jdbc/ConnectSparkyOracle`

Closing the Connection Pool

To ensure that the connection pool is closed correctly when an application stops running, the application must notify the DataDirect Connection Pool Manager when it stops. For applications running on J2SE 5 and higher, notification occurs automatically when the application stops running.

The PooledConnectionDataSource.close() method also can be used to explicitly close the connection pool while the application is running. For example, if changes are made to the pool configuration using a pool management tool, the PooledConnectionDataSource.close() method can be used to force the connection pool to close and re-create the pool using the new configuration values.

DataDirect Connection Pool Manager Interfaces

This section describes the methods used by the DataDirect Connection Pool Manager interfaces: PooledConnectionDataSourceFactory, PooledConnectionDataSource, and ConnectionPoolMonitor.

PooledConnectionDataSourceFactory

The PooledConnectionDataSourceFactory interface is used to create a PooledConnectionDataSource object from a Reference object that is stored in a naming or directory service. These methods are typically invoked by a JNDI service provider; they are not usually invoked by a user application.

PooledConnectionDataSourceFactory Methods	Description
static Object getObjectInstance(Object *refObj*, Name *name*, Context *nameCtx*, Hashtable *env*)	Creates a PooledConnectionDataSource object from a Reference object that is stored in a naming or directory service. This is an implementation of the method of the same name defined in the javax.naming.spi.ObjectFactory interface. Refer to the Javadoc for this interface for a description.

PooledConnectionDataSource

The PooledConnectionDataSource interface is used to create a PooledConnectionDataSource object for use with the DataDirect Connection Pool Manager.

PooledConnectionDataSource Methods	Description
void close()	Closes the connection pool. All physical connections in the pool are closed. Any subsequent connection request re-initializes the connection pool.
Connection getConnection()	Obtains a physical connection from the connection pool.
Connection getConnection(String *user*, String *password*)	Obtains a physical connection from the connection pool, where *user* is the user requesting the connection and *password* is the password for the connection.
String getDataSourceName()	Returns the JNDI name that is used to look up the DataDirect DataSource object referenced by this PooledConnectionDataSource.
String getDescription()	Returns the description of this PooledConnectionDataSource.
String getReauthentication()	Returns whether the Pool Manager is enabled for reauthentication. See Using Reauthentication with the Pool Manager on page 333 for more information.
int getInitialPoolSize()	Returns the value of the initial pool size, which is the number of physical connections created when the connection pool is initialized.
int getLoginTimeout()	Returns the value of the login timeout, which is the time allowed for the database login to be validated.
PrintWriter getLogWriter()	Returns the writer to which the Pool Manager sends trace information about its activities.

PooledConnectionDataSource Methods	Description
int getMaxIdleTime()	Returns the value of the maximum idle time, which is the time a physical connection can remain idle in the connection pool before it is removed from the connection pool.
int getMaxPoolSize()	Returns the value of the maximum pool size. See Understanding the Maximum Pool Size on page 333 for more information about how the Pool Manager implements the maximum pool size.
int getMaxPoolSizeBehavior()	Returns the value of the maximum pool size behavior. See Understanding the Maximum Pool Size on page 333 for more information about how the Pool Manager implements the maximum pool size.
int getMinPoolSize()	Returns the value of the minimum pool size, which is the minimum number of idle connections to be kept in the pool.
int getPropertyCycle()	Returns the value of the property cycle, which specifies how often the pool maintenance thread wakes up and checks the connection pool.
Reference getReference()	Obtains a javax.naming.Reference object for this PooledConnectionDataSource. The Reference object contains all the state information needed to recreate an instance of this data source using the PooledConnectionDataSourceFactory object. This method is typically called by a JNDI service provider when this PooledConnectionDataSource is bound to a JNDI naming service.
public static ConnectionPoolMonitor[ ] getMonitor()	Returns an array of Connection Pool Monitors, one for each connection pool managed by the Pool Manager.

<table>
<tr><th>PooledConnectionDataSource Methods</th><th>Description</th></tr>
<tr><td>public static ConnectionPoolMonitor getMonitor(String name)</td><td>Returns the name of the Connection Pool Monitor for the connection pool specified by name. If a pool with the specified name cannot be found, this method returns null. The connection pool name has the form:
<code>jndi_name-user_id</code>
where:
<code>jndi_name</code>
is the name used for the JNDI lookup of the driver DataSource object from which the pooled connection was obtained and
<code>user_id</code>
is the user ID used to establish the connections contained in the pool.The following example shows how to return the Connection Pool Monitor for the connection pool that is bound to the JNDI lookup name jdbc/SQLServerPool and connections established by user test04.
<pre>DataSource ds = (DataSource)
ctx.lookup("jdbc/SQLServerPool");
Connection con = ds.getConnection;
   ("test04", "test04");
ConnectionPoolMonitor monitor =
PooledConnectionDataSource.getMonitor
   ("jdbc/SQLServerPool-test04");</pre></td></tr>
<tr><td>boolean isTracing()</td><td>Determines whether tracing is enabled. If enabled, tracing information is sent to the PrintWriter that is passed to the setLogWriter() method or the standard output System.out if the setLogWriter() method is not called.</td></tr>
<tr><td>void setDataSourceName(String dataSourceName)</td><td>Sets the JNDI name, which is used to look up the driver DataSource object referenced by this PooledConnectionDataSource. The driver DataSource object bound to this PooleConnectionDataSource, specified by dataSourceName, is not persisted. Any changes made to the PooledConnectionDataSource bound to the specified driver DataSource object affect this PooledConnectionDataSource.</td></tr>
<tr><td>void setDataSourceName(String dataSourceName, ConnectionPoolDataSource dataSource)</td><td>Sets the JNDI name associated with this PooledConnectionDataSource, specified by dataSourceName, and the driver DataSource object, specified by dataSource, referenced by this PooledConnectionDataSource.
The driver DataSource object, specified by dataSource, is persisted with this PooledConnectionDataSource. Changes made to the specified driver DataSource object after this PooledConnectionDataSource is persisted do not affect this PooledConnectionDataSource.</td></tr>
</table>

PooledConnectionDataSource Methods	Description
void setDataSourceName(String *dataSourceName*, Context *ctx*)	Sets the JNDI name, specified by *dataSourceName*, and context, specified by *ctx*, to be used to look up the driver DataSource referenced by this PooledConnectionDataSource. The JNDI name, specified by *dataSourceName*, and context, specified by *ctx*, are used to look up a driver DataSource object. The driver DataSource object is persisted with this PooledConnectionDataSource. Changes made to the driver DataSource after this PooledConnectionDataSource is persisted do not affect this PooledConnectionDataSource.
void setDescription(String *description*)	Sets the description of the PooledConnectionDataSource, where *description* is the description.
void setInitialPoolSize(int *initialPoolSize*)	Sets the value of the initial pool size, which is the number of connections created when the connection pool is initialized.
void setLoginTimeout(int *i*)	Sets the value of the login timeout, where *i* is the login timeout, which is the time allowed for the database login to be validated.
void setLogTimestamp(boolean *value*)	If set to `true`, the timestamp is logged when DataDirect Spy logging is enabled. If set to `false`, the timestamp is not logged.
void setLogTname(boolean *value*)	If set to `true`, the thread name is logged when DataDirect Spy logging is enabled. If set to `false`, the thread name is not logged.
void setLogWriter(PrintWriter *printWriter*)	Sets the writer, where *printWriter* is the writer to which the stream will be printed.
void setMaxIdleTime(int *maxIdleTime*)	Sets the value in seconds of the maximum idle time, which is the time a connection can remain unused in the connection pool before it is closed and removed from the pool. Zero (0) indicates no limit.
void setMaxPoolSize(int *maxPoolSize*)	Sets the value of the maximum pool size, which is the maximum number of connections for each user allowed in the pool. See Understanding the Maximum Pool Size on page 333 for more information about how the Pool Manager implements the maximum pool size. In addition, see Using Reauthentication with the Pool Manager on page 333 .

PooledConnectionDataSource Methods	Description
void setMaxPoolSizeBehavior(String *value*)	Sets the value of the maximum pool size behavior, which is either softCap or hardCap. If `setMaxPoolSizeBehavior(softCap)`, the number of active connections may exceed the maximum pool size, but the number of idle connections in the connection pool for each user cannot exceed this limit. If a user requests a connection and an idle connection is unavailable, the Pool Manager creates a new connection for that user. When the connection is no longer needed, it is returned to the pool. If the number of idle connections exceeds the maximum pool size, the Pool Manager closes idle connections to enforce the maximum pool size limit. This is the default behavior. If `setMaxPoolSizeBehavior(hardCap)`, the total number of active and idle connections cannot exceed the maximum pool size. Instead of creating a new connection for a connection request if an idle connection is unavailable, the Pool Manager queues the connection request until a connection is available or the request times out. This behavior is useful if your database server has memory limitations or is licensed for only a specific number of connections.The timeout is set using the LoginTimeout connection property. If the connection request times out, the driver throws an exception. See Understanding the Maximum Pool Size on page 333 for more information about how the Pool Manager implements the maximum pool size.
void setMinPoolSize(int *minPoolSize*)	Sets the value of the minimum pool size, which is the minimum number of idle connections to be kept in the connection pool.
void setPropertyCycle(int *propertyCycle*)	Sets the value in seconds of the property cycle, which specifies how often the pool maintenance thread wakes up and checks the connection pool.
void setReauthentication(String *value*)	Enables and disables reauthentication for the Pool Manager. To enable reauthentication, use `setReauthentication(enable)`. To disable reauthentication, use `setReauthentication(disable)`. See Using Reauthentication with the Pool Manager on page 333 for more information.
void setTracing(boolean *value*)	Enables or disables tracing. If set to `true`, tracing is enabled; if `false`, it is disabled. If enabled, tracing information is sent to the PrintWriter that is passed to the setLogWriter() method or the standard output System.out if the setLogWriter() method is not called.

ConnectionPoolMonitor

The ConnectionPoolMonitor interface is used to return information that is useful for monitoring the status of your connection pools.

ConnectionPoolMonitor Methods	Description

String getName()	Returns the name of the connection pool associated with the monitor. The connection pool name has the form:
	jndi_name-user_id
	where:
	`jndi_name`
	is the name used for the JNDI lookup of the PooledConnectionDataSource object from which the pooled connection was obtained
	`user_id`
	is the user ID used to establish the connections contained in the pool.

int getNumActive()	Returns the number of connections that have been checked out of the pool and are currently in use.
int getNumAvailable()	Returns the number of connections that are idle in the pool (available connections).
int getInitialPoolSize()	Returns the initial size of the connection pool (the number of available connections in the pool when the pool was first created).
int getMaxPoolSize()	Returns the maximum number of available connection in the connection pool. If the number of available connections exceeds this value, the Pool Manager removes one or multiple available connections from the pool.
int getMinPoolSize()	Returns the minimum number of available connections in the connection pool. When the number of available connections is lower than this value, the Pool Manager creates additional connections and makes them available.
int getPoolSize()	Returns the current size of the connection pool, which is the total of active connections and available connections.

11

Statement Pool Monitor

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

For details, see the following topics:

- Using DataDirect-Specific Methods to Access the Statement Pool Monitor
- Using JMX to Access the Statement Pool Monitor
- Importing Statements into a Statement Pool
- Clearing All Statements in a Statement Pool
- Freezing and Unfreezing the Statement Pool
- Generating a Statement Pool Export File
- DataDirect Statement Pool Monitor Interfaces and Classes

Using DataDirect-Specific Methods to Access the Statement Pool Monitor

To access the Statement Pool Monitor using DataDirect-specific methods, you should first enable statement pooling. You can enable statement pooling by setting the MaxPooledStatements connection property to a value greater than zero (0). For more information, refer to "MaxPooledStatements" in the *Progress DataDirect Connect Series for JDBC User's Guide*.

The ExtConnection.getStatementPoolMonitor() method returns an ExtStatementPoolMonitor object for the statement pool associated with the connection. This method is provided by the ExtConnection interface in the com.ddtek.jdbc.extensions package. If the connection does not have a statement pool, the method returns null.

Once you have an ExtStatementPoolMonitor object, you can use the poolEntries() method of the ExtStatementPoolMonitorMBean interface implemented by the ExtStatementPoolMonitor to return a list of statements in the statement pool as an array.

Using the poolEntries Method

Using the poolEntries() method, your application can return all statements in the pool or filter the list based on the following criteria:

- Statement type (prepared statement or callable statement)
- Result set type (forward only, scroll insensitive, or scroll sensitive)
- Concurrency type of the result set (read only and updateable)

The following table lists the parameters and the valid values supported by the poolEntries() method.

Table 37: poolEntries() Parameters

Parameter	Value	Description
statementType	ExtStatementPoolMonitor.TYPE_PREPARED_STATEMENT	Returns only prepared statements
	ExtStatementPoolMonitor.TYPE_CALLABLE_STATEMENT	Returns only callable statements
	-1	Returns all statements regardless of statement type
resultSetType	ResultSet.TYPE_FORWARD_ONLY	Returns only statements with forward-only result sets
	ResultSet.TYPE_SCROLL_INSENSITIVE	Returns only statements with scroll insensitive result sets
	ResultSet.TYPE_SCROLL_SENSITIVE	Returns only statements with scroll sensitive result sets
	-1	Returns statements regardless of result set type

Parameter	Value	Description
resultSetConcurrency	ResultSet.CONCUR_READ_ONLY	Returns only statements with a read-only result set concurrency
	ResultSet.CONCUR_UPDATABLE	Returns only statements with an updateable result set concurrency
	-1	Returns statements regardless of result set concurrency type

The result of the poolEntries() method is an array that contains a String entry for each statement in the statement pool using the format:

```
SQL_TEXT=[SQL_text];STATEMENT_TYPE=TYPE_PREPARED_STATEMENT|
TYPE_CALLABLE_STATEMENT;RESULTSET_TYPE=TYPE_FORWARD_ONLY|
TYPE_SCROLL_INSENSITIVE|TYPE_SCROLL_SENSITIVE;
RESULTSET_CONCURRENCY=CONCUR_READ_ONLY|CONCUR_UPDATABLE;
AUTOGENERATEDKEYSREQUESTED=true|false;
REQUESTEDKEYCOLUMNS=comma-separated_list
```

where *SQL_text* is the SQL text of the statement and *comma-separated_list* is a list of column names that will be returned as generated keys.

For example:

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

Generating a List of Statements in the Statement Pool

The following code shows how to return an ExtStatementPoolMonitor object using a connection and how to generate a list of statements in the statement pool associated with the connection.

Note: The following example is drawn from a Microsoft SQL Server use case but applies to most Progress DataDirect drivers.

```
private void run(String[] args) {
   Connection con = null;
   PreparedStatement prepStmt = null;
   String sql = null;
   try {
      // Create the connection and enable statement pooling
      Class.forName("com.ddtek.jdbc.sqlserver.SQLServerDriver");
      con = DriverManager.getConnection(
         "jdbc:datadirect:sqlserver://SMITH:1440;" +
         "RegisterStatementPoolMonitorMBean=true",
         "maxPooledStatements=10",
         "test", "test");
      // Prepare a couple of statements
      sql = "INSERT INTO employees (id, name) VALUES(?, ?)";
      prepStmt = con.prepareStatement(sql);
      prepStmt.close();
      sql = "SELECT name FROM employees WHERE id = ?";
      prepStmt = con.prepareStatement(sql);
      prepStmt.close();
```

```
      ExtStatementPoolMonitor monitor =
         ((ExtConnection) con).getStatementPoolMonitor();
      System.out.println("Statement Pool - " + monitor.getName());
      System.out.println("Max Size:     " + monitor.getMaxSize());
      System.out.println("Current Size: " + monitor.getCurrentSize());
      System.out.println("Hit Count:    " + monitor.getHitCount());
      System.out.println("Miss Count:   " + monitor.getMissCount());
      System.out.println("Statements:");
      ArrayList statements = monitor.poolEntries(-1, -1, -1);
      Iterator itr = statements.iterator();
      while (itr.hasNext()) {
         String entry = (String)itr.next();
         System.out.println(entry);
      }
}
catch (Throwable except) {
   System.out.println("ERROR: " + except);
   }
   finally {
      if (con != null) {
         try {
            con.close();
         }
         catch (SQLException except) {}
         }
      }
   }
```

In the previous code example, the PoolEntries() method returns all statements in the statement pool regardless of statement type, result set cursor type, and concurrency type by specifying the value `-1` for each parameter as shown in the following code:

```
ArrayList statements = monitor.poolEntries(-1, -1, -1);
```

We could have easily filtered the list of statements to return only prepared statements that have a forward-only result set with a concurrency type of updateable using the following code:

```
ArrayList statements = monitor.poolEntries(
   ExtStatementPoolMonitor.TYPE_PREPARED_STATEMENT,
   ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);
```

Using JMX to Access the Statement Pool Monitor

Your application cannot access the Statement Pool Monitor using JMX unless the driver registers 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. For more information, refer to "MaxPooledStatements" and "RegisterStatementPoolMonitorMBean" in the *Progress DataDirect Connect Series for JDBC User's Guide*.

When the Statement Pool Monitor is enabled, the drivers register a single MBean for each statement pool. The registered MBean name has the following form, where *monitor_name* is the string returned by the ExtStatementPoolMonitor.getName() method:

`com.ddtek.jdbc.type=StatementPoolMonitor,name=`*`monitor_name`*

Note: 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.

To return information about the statement pool, retrieve the names of all MBeans that are registered with the com.ddtek.jdbc domain and search through the list for the StatementPoolMonitor type attribute. The following code shows how to use the standard JMX API calls to return the state of all active statement pools in the JVM:

```
private void run(String[] args) {
   if (args.length < 2) {
      System.out.println("Not enough arguments supplied");
      System.out.println("Usage: " + "ShowStatementPoolInfo hostname port");
   }
   String hostname = args[0];
   String port = args[1];
   JMXServiceURL           url = null;
   JMXConnector            connector = null;
   MBeanServerConnection   server = null;
   try {
      url = new JMXServiceURL("service:jmx:rmi:///jndi/rmi://" +
                              hostname +":" + port +  "/jmxrmi");
      connector = JMXConnectorFactory.connect(url);
      server = connector.getMBeanServerConnection();
      System.out.println("Connected to JMX MBean Server at " +
                        args[0] + ":" + args[1]);
      // Get the MBeans that have been registered with the
      // com.ddtek.jdbc domain.
      ObjectName ddMBeans = new ObjectName("com.ddtek.jdbc:*");
      Set<ObjectName> mbeans = server.queryNames(ddMBeans, null);
      // For each statement pool monitor MBean, display statistics and
      // contents of the statement pool monitored by that MBean
      for (ObjectName name: mbeans) {
         if (name.getDomain().equals("com.ddtek.jdbc") &&
             name.getKeyProperty("type")
                 .equals("StatementPoolMonitor")) {
            System.out.println("Statement Pool - " +
               server.getAttribute(name, "Name"));
            System.out.println("Max Size:     " +
               server.getAttribute(name, "MaxSize"));
            System.out.println("Current Size: " +
               server.getAttribute(name, "CurrentSize"));
            System.out.println("Hit Count:    " +
               server.getAttribute(name, "HitCount"));
            System.out.println("Miss Count:   " +
               server.getAttribute(name, "MissCount"));
            System.out.println("Statements:");
            Object[] params = new Object[3];
            params[0] = new Integer(-1);
            params[1] = new Integer(-1);
            params[2] = new Integer(-1);
            String[] types = new String[3];
            types[0] = "int";
            types[1] = "int";
            types[2] = "int";
            ArrayList<String>statements = (ArrayList<String>)
               server.invoke(name,
                             "poolEntries",
                              params,
                              types);
            for (String stmt : statements) {
               int index = stmt.indexOf(";");
               System.out.println("   " + stmt.substring(0, index));
            }
         }
      }
   }
   catch (Throwable except) {
      System.out.println("ERROR: " + except);
   }
}
```

Importing Statements into a Statement Pool

When importing statements into a statement pool, a statement is added to the statement pool for each statement entry in the export file as long as a statement with the same attributes and SQL text does not already exist in the statement pool. Existing statements that correspond to a statement entry are kept in the pool unless the addition of new statements causes the number of statements to exceed the maximum pool size. In this case, the driver closes and discards some statements until the pool size is shrunk to the maximum pool size.

For example, if the maximum number of statements allowed for a statement pool is 10 and the number of statements to be imported is 20, only the last 10 imported statements are placed in the statement pool. The other statements are created, closed, and discarded. Importing more statements than the maximum number of statements allowed in the statement pool can negatively affect performance because the driver unnecessarily creates some statements that are never placed in the pool.

To import statements into a statement pool:

1. Create a statement pool export file. See Statement Pool Export File Example on page 364 for an example of a statement pool export file.

   Note: The easiest way to create a statement pool export file is to generate an export file from the statement pool associated with the connection as described in Generating a Statement Pool Export File on page 353
   .

   ---

2. Edit the export file to contain statements to be added to the statement pool.
3. Import the contents of the export file to the statement pool using either of the following methods to specify the path and file name of the export file:
   - Use the ImportStatementPool property. For example:

     ```
     jdbc:datadirect:db2://server1:50000;DatabaseName=jdbc;
     User=test;Password=secret;ImportStatementPool=
     C:\\statement_pooling\\stmt_export.txt
     ```

   - Use the importStatements() method of the ExtStatementPoolMonitorMBean interface. For example:

     ```
     ExtStatementPoolMonitor monitor =
        ((ExtConnection)
     con).getStatementPoolMonitor().importStatements
        ("C:\\statement_pooling\\stmt_export.txt");
     ```

Clearing All Statements in a Statement Pool

To close and discard all statements in a statement pool, use the emptyPool() method of the ExtStatementPoolMonitorMBean interface. For example:

```
ExtStatementPoolMonitor monitor =
   ((ExtConnection) con).getStatementPoolMonitor().emptyPool();
```

Freezing and Unfreezing the Statement Pool

Freezing the statement pool restricts the statements in the pool to those that were in the pool at the time the pool was frozen. For example, perhaps you have a core set of statements that you do not want replaced by new statements when your core statements are closed. You can freeze the pool using the setFrozen() method:

```
ExtStatementPoolMonitor monitor =
    ((ExtConnection) con).getStatementPoolMonitor().setFrozen(true);
```

Similarly, you can use the same method to unfreeze the statement pool:

```
ExtStatementPoolMonitor monitor =
    ((ExtConnection) con).getStatementPoolMonitor().setFrozen(false);
```

When the statement pool is frozen, your application can still clear the pool and import statements into the pool. In addition, if your application is using Java SE 6 or higher, you can use the Statement.setPoolable() method to add or remove single statements from the pool regardless of the pool's frozen state, assuming the pool is not full. If the pool is frozen and the number of statements in the pool is the maximum, no statements can be added to the pool.

To determine if a pool is frozen, use the isFrozen() method.

Generating a Statement Pool Export File

You may want to generate an export file in the following circumstances:

- To import statements to the statement pool, you can create an export file, edit its contents, and import the file into the statement pool to import statements to the pool.
- To examine the characteristics of the statements in the statement pool to help you troubleshoot statement pool performance.

To generate a statement pool export file, use the exportStatements() method of the ExtStatementPoolMonitorMBean interface. For example, the following code exports the contents of the statement pool associated with the connection to a file named stmt_export:

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

See the "Statement Pool Export File Example" topic for information on interpreting the contents of an export file.

See also

Statement Pool Export File Example on page 364

DataDirect Statement Pool Monitor Interfaces and Classes

This section describes the methods used by the DataDirect Statement Pool Monitor interfaces and classes.

ExtStatementPoolMonitor Class

This class is used to control and monitor a single statement pool. This class implements the ExtStatementPoolMonitorMBean interface.

ExtStatementPoolMonitorMBean Interface

ExtStatementPoolMonitorMBean Methods	Description
String getName()	Returns the name of a Statement Pool Monitor instance associated with the connection. The name is comprised of the name of the driver that established the connection, and the name and port of the server to which the Statement Pool Monitor is connected, and the MBean ID of the connection.
int getCurrentSize()	Returns the total number of statements cached in the statement pool.
long getHitCount()	Returns the hit count for the statement pool. The hit count is the number of times a lookup is performed for a statement that results in a cache hit. A cache hit occurs when the Statement Pool Monitor successfully finds a statement in the pool with the same SQL text, statement type, result set type, result set concurrency, and requested generated key information. This method is useful to determine if your workload is using the statement pool effectively. For example, if the hit count is low, the statement pool is probably not being used to its best advantage.
long getMissCount()	Returns the miss count for the statement pool. The miss count is the number of times a lookup is performed for a statement that fails to result in a cache hit. A cache hit occurs when the Statement Pool Monitor successfully finds a statement in the pool with the same SQL text, statement type, result set type, result set concurrency, and requested generated key information. This method is useful to determine if your workload is using the statement pool effectively. For example, if the miss count is high, the statement pool is probably not being used to its best advantage.
int getMaxSize()	Returns the maximum number of statements that can be stored in the statement pool.
int setMaxSize(int *value*)	Changes the maximum number of statements that can be stored in the statement pool to the specified value.

ExtStatementPoolMonitorMBean Methods	Description
void emptyPool()	Closes and discards all the statements in the statement pool.
void resetCounts()	Resets the hit and miss counts to zero (0). See long getHitCount() and long getMissCount() for more information.
ArrayList poolEntries(int *statementType*, int *resultSetType*, int *resultSetConcurrency*)	Returns a list of statements in the pool. The list is an array that contains a String entry for each statement in the statement pool.
void exportStatements(File *file_object*)	Exports statements from the statement pool into the specified file. The file format contains an entry for each statement in the statement pool.
void exportStatements(String *file_name*)	Exports statements from statement pool into the specified file. The file format contains an entry for each statement in the statement pool.
void importStatements(File *file_object*)	Imports statements from the specified File object into the statement pool.
void importStatements(String *file_name*)	Imports statements from the specified file into the statement pool.
boolean isFrozen()	Returns whether the state of the statement pool is frozen. When the statement pool is frozen, the statements that can be stored in the pool are restricted to those that were in the pool at the time the pool was frozen. Freezing a pool is useful if you have a core set of statements that you do not want replaced by other statements when the core statements are closed.
void setFrozen(boolean)	`setFrozen(true)` freezes the statement pool. `setFrozen(false)` unfreezes the statement pool. When the statement pool is frozen, the statements that can be stored in the pool are restricted to those that were in the pool at the time the pool was frozen. Freezing a pool is useful if you have a core set of statements that you do not want replaced by other statements when the core statements are closed.

12

Troubleshooting

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

For details, see the following topics:

- Troubleshooting Your Application
- Troubleshooting Connection Pooling
- Troubleshooting Statement Pooling
- Using Java Logging (Salesforce)
- Configuring Logging

Troubleshooting Your Application

To help you troubleshoot any problems that occur with your application, you can use DataDirect Spy to log detailed information about calls issued by the drivers on behalf of your application. When you enable DataDirect Spy for a connection, you can customize DataDirect Spy logging by setting one or multiple options. See Tracking JDBC Calls with DataDirect Spy on page 325 for information about using DataDirect Spy and instructions on enabling and customizing logging.

Turning On and Off DataDirect Spy Logging

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

For example, the following code turns off logging using setEnableLogging(false):

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

// Get Database Connection
Connection con = DriverManager.getConnection
   ("jdbc:datadirect:sqlserver://server1:1433;User=TEST;Password=secret;
      SpyAttributes=(log=(file)/tmp/spy.log");

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

The setEnableLogging() method only turns on and off logging if DataDirect Spy logging has already been enabled for a connection; it does not set or change DataDirect Spy attributes. See Enabling DataDirect Spy on page 325 for information about enabling and customizing DataDirect Spy logging.

DataDirect Spy Log Example

This section provides information to help you understand the content of your own DataDirect Spy logs. For example, suppose your application executes the following code and performs some operations:

```
Class.forName("com.ddtek.jdbc.sqlserver.SQLServerDriver");
DriverManager.getConnection("jdbc:datadirect:sqlserver:// nc-myserver\
\sqlserver2005;useServerSideUpdatableCursors=true;resultsetMetaDataOptions=1;
sendStringParametersAsUnicode=true;alwaysReportTriggerResults=false;
spyAttributes=(log=(file)c:\\temp\\spy.log)","test04", "test04");
```

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

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

spy>> DatabaseMetaData[1].getURL()
spy>> OK
(jdbc:datadirect:sqlserver://nc-myserver\sqlserver2005:1433;CONNECTIONRETRYCOUNT=5;
RECEIVESTRINGPARAMETERTYPE=nvarchar;ALTERNATESERVERS=;DATABASENAME=;PACKETSIZE=16;INITIALIZATIONSTRING=;
ENABLECANCELTIMEOUT=false;BATCHPERFORMANCEWORKAROUND=false;AUTHENTICATIONMETHOD=auto;
SENDSTRINGPARAMETERSASUNICODE=true;LOGINTIMEOUT=0;WSID=;SPYATTRIBUTES=(log=(file)c:\temp\spy.log);
RESULTSETMETADATAOPTIONS=1;ALWAYSREPORTTRIGGERRESULTS=false;TRANSACTIONMODE=implicit;
USESERVERSIDEUPDATABLECURSORS=true;SNAPSHOTSERIALIZABLE=false;JAVADOUBLETOSTRING=false;
SELECTMETHOD=direct;LOADLIBRARYPATH=;CONNECTIONRETRYDELAY=1;INSENSITIVERESULTSETBUFFERSIZE=2048;
MAXPOOLEDSTATEMENTS=0;DESCRIBEPARAMETERS=noDescribe;CODEPAGEOVERRIDE=;NETADDRESS=000000000000;
PROGRAMNAME=;LOADBALANCING=false;HOSTPROCESS=0) 69
spy>> DatabaseMetaData[1].getDriverName()
spy>> OK (SQLServer)

spy>> DatabaseMetaData[1].getDriverVersion()
spy>> OK (3.60.0 (000000.000000.000000))

spy>> DatabaseMetaData[1].getDatabaseProductName()
spy>> OK (Microsoft SQL Server)

spy>> DatabaseMetaData[1].getDatabaseProductVersion()
spy>> OK (Microsoft SQL Server Yukon - 9.00.1399)
```

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

```
spy>> Connection Options : 70
spy>>    CONNECTIONRETRYCOUNT=5
spy>>    RECEIVESTRINGPARAMETERTYPE=nvarchar
spy>>    ALTERNATESERVERS=
spy>>    DATABASENAME=
spy>>    PACKETSIZE=16
spy>>    INITIALIZATIONSTRING=
spy>>    ENABLECANCELTIMEOUT=false
spy>>    BATCHPERFORMANCEWORKAROUND=false
spy>>    AUTHENTICATIONMETHOD=auto
spy>>    SENDSTRINGPARAMETERSASUNICODE=true
spy>>    LOGINTIMEOUT=0
spy>>    WSID=
spy>>    SPYATTRIBUTES=(log=(file)c:\temp\spy.log)
spy>>    RESULTSETMETADATAOPTIONS=1
spy>>    ALWAYSREPORTTRIGGERRESULTS=false
spy>>    TRANSACTIONMODE=implicit
spy>>    USESERVERSIDEUPDATABLECURSORS=true
spy>>    SNAPSHOTSERIALIZABLE=false
spy>>    JAVADOUBLETOSTRING=false
spy>>    SELECTMETHOD=direct
spy>>    LOADLIBRARYPATH=
spy>>    CONNECTIONRETRYDELAY=1
spy>>    INSENSITIVERESULTSETBUFFERSIZE=2048
spy>>    MAXPOOLEDSTATEMENTS=0
spy>>    DESCRIBEPARAMETERS=noDescribe
spy>>    CODEPAGEOVERRIDE=
spy>>    NETADDRESS=000000000000
spy>>    PROGRAMNAME=
spy>>    LOADBALANCING=false
spy>>    HOSTPROCESS=0
spy>> Driver Name = SQLServer 71
spy>> Driver Version = 3.60.0 (000000.000000.000000) 72
spy>> Database Name = Microsoft SQL Server 73
spy>> Database Version = Microsoft SQL Server Yukon - 9.00.1399 74
spy>> Connection[1].getWarnings()
spy>> OK 75spy>> Connection[1].createStatement
spy>> OK (Statement[1])

spy>> Statement[1].executeQuery(String sql)
spy>> sql = select empno,ename,job from emp where empno=7369
spy>> OK (ResultSet[1]) 76
spy>> ResultSet[1].getMetaData()
spy>> OK (ResultSetMetaData[1]) 77
spy>> ResultSetMetaData[1].getColumnCount()
spy>> OK (3)78
spy>> ResultSetMetaData[1].getColumnLabel(int column)
spy>> column = 1
spy>> OK (EMPNO)79spy>> ResultSetMetaData[1].getColumnLabel(int column)
spy>> column = 2
spy>> OK (ENAME)80
spy>> ResultSetMetaData[1].getColumnLabel(int column)
spy>> column = 3
spy>> OK (JOB)81spy>> ResultSet[1].next()
spy>> OK (true)82
```

[70] The combination of the connection properties specified by the application and the default values of all connection properties not specified.
[71] The name of the driver.
[72] The version of the driver.
[73] The name of the database server to which the driver connects.
[74] The version of the database to which the driver connects.
[75] The application checks to see if there are any warnings. In this example, no warnings are present.
[76] The statement *select empno,ename,job from emp where empno=7369* is created.
[77] Some metadata is requested.
[78] Some metadata is requested.
[79] Some metadata is requested.
[80] Some metadata is requested.
[81] Some metadata is requested.
[82] The first row is retrieved and the application retrieves the result values.

```
spy>> ResultSet[1].getString(int columnIndex)
spy>> columnIndex = 1
spy>> OK (7369) 83
spy>> ResultSet[1].getString(int columnIndex)
spy>> columnIndex = 2
spy>> OK (SMITH) 84
spy>> ResultSet[1].getString(int columnIndex)
spy>> columnIndex = 3
spy>> OK (CLERK) 85
spy>> ResultSet[1].next()
spy>> OK (false) 86spy>> ResultSet[1].close()
spy>> OK 87
spy>> Connection[1].close()
spy>> OK 88
```

Troubleshooting Connection Pooling

Connection pooling allows connections to be reused rather than created each time a connection is requested. If your application is using connection pooling through the DataDirect Connection Pool Manager, you can generate a trace file that shows all the actions taken by the Pool Manager. See Connection Pool Manager on page 331 for information about using the Pool Manager.

Enabling Pool Manager Tracing

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

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

Pool Manager Trace File Example

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

```
jdbc/SQLServerNCMarkBPool: *** ConnectionPool Created
   (jdbc/SQLServerNCMarkBPool,
   com.ddtek.jdbcx.sqlserver.SQLServerDataSource@1835282, 5, 5, 10, scott) 89
```

[83] The first row is retrieved and the application retrieves the result values.
[84] The first row is retrieved and the application retrieves the result values.
[85] The first row is retrieved and the application retrieves the result values.
[86] The application attempts to retrieve the next row, but only one row was returned for this query.
[87] After the application has completed retrieving result values, the result set is closed.
[88] The application finishes and disconnects.
[89] The Pool Manager creates a connection pool. In this example, the characteristics of the connection pool are shown using the following format:

(*JNDI_name*,*DataSource_class*,*initial_pool_size*,*min_pool_size*,*max_pool_size*,
user)

where:

JNDI_name is the JNDI name used to look up the connection pool (for example, jdbc/SQLServerNCMarkBPool).

DataSource_class is the DataSource class associated with the connection pool (for example com.ddtek.jdbcx.sqlserver.SQLServerDataSource).

initial_pool_size is the number of physical connections created when the connection pool is initialized (for example, 5).

min_pool_size is the minimum number of physical connections be kept open in the connection pool (for example, 5).

```
jdbc/SQLServerNCMarkBPool: Number pooled connections = 0.
jdbc/SQLServerNCMarkBPool: Number free connections = 0.

jdbc/SQLServerNCMarkBPool: Enforced minimum![90]
NrFreeConnections was: 0
jdbc/SQLServerNCMarkBPool: Number pooled connections = 5.
jdbc/SQLServerNCMarkBPool: Number free connections = 5.

jdbc/SQLServerNCMarkBPool: Reused free connection.[91]
jdbc/SQLServerNCMarkBPool: Number pooled connections = 5.
jdbc/SQLServerNCMarkBPool: Number free connections = 4.

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

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

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

jdbc/SQLServerNCMarkBPool: Reused free connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 5.
jdbc/SQLServerNCMarkBPool: Number free connections = 0.

jdbc/SQLServerNCMarkBPool: Created new connection.[92]
jdbc/SQLServerNCMarkBPool: Number pooled connections = 6.
jdbc/SQLServerNCMarkBPool: Number free connections = 0.

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

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

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

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

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

jdbc/SQLServerNCMarkBPool: Connection was closed and added to the cache.[93]
jdbc/SQLServerNCMarkBPool: Number pooled connections = 11.
jdbc/SQLServerNCMarkBPool: Number free connections = 1.

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

max_pool_size is the maximum number of physical connections allowed within a single pool at any one time. When this number is reached, additional connections that would normally be placed in a connection pool are closed (for example, 10).

user is the name of the user establishing the connection (for example, scott).

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

91 The driver requests a connection from the connection pool. The driver retrieves an available connection.

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

93 A connection is closed by the application and returned to the connection pool.

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

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

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

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

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

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

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

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

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

jdbc/SQLServerNCMarkBPool: Enforced minimum![94]
NrFreeConnections was: 11
jdbc/SQLServerNCMarkBPool: Number pooled connections = 11.
jdbc/SQLServerNCMarkBPool: Number free connections = 11.

jdbc/SQLServerNCMarkBPool: Enforced maximum![95]
NrFreeConnections was: 11
jdbc/SQLServerNCMarkBPool: Number pooled connections = 10.
jdbc/SQLServerNCMarkBPool: Number free connections = 10.

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

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

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

jdbc/SQLServerNCMarkBPool: Enforced maximum!
NrFreeConnections was: 10
```

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

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

```
jdbc/SQLServerNCMarkBPool: Number pooled connections = 10.
jdbc/SQLServerNCMarkBPool: Number free connections = 10.

jdbc/SQLServerNCMarkBPool: Dumped free connection.[96]
jdbc/SQLServerNCMarkBPool: Number pooled connections = 9.
jdbc/SQLServerNCMarkBPool: Number free connections = 9.

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

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

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

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

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

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

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

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

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

jdbc/SQLServerNCMarkBPool: Enforced minimum![97]
NrFreeConnections was: 0
jdbc/SQLServerNCMarkBPool: Number pooled connections = 5.
jdbc/SQLServerNCMarkBPool: Number free connections = 5.

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

jdbc/SQLServerNCMarkBPool: Closing a pool of the group
      jdbc/SQLServerNCMarkBPool[98]
jdbc/SQLServerNCMarkBPool: Number pooled connections = 5.
jdbc/SQLServerNCMarkBPool: Number free connections = 5.

jdbc/SQLServerNCMarkBPool: Pool closed[99]
```

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

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

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

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

```
jdbc/SQLServerNCMarkBPool: Number pooled connections = 0.
jdbc/SQLServerNCMarkBPool: Number free connections = 0.
```

Troubleshooting Statement Pooling

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

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

See Statement Pool Monitor on page 347 for more information about using the Statement Pool Monitor.

Generating a Statement Pool Export File

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

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

Statement Pool Export File Example

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

```
[DDTEK_STMT_POOL][100]
VERSION=1[101]

[STMT_ENTRY][102]
SQL_TEXT=[
INSERT INTO emp(id, name) VALUES(?,?)
]
STATEMENT_TYPE=Prepared Statement
RESULTSET_TYPE=Forward Only
```

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

```
RESULTSET_CONCURRENCY=Read Only
AUTOGENERATEDKEYSREQUESTED=false
REQUESTEDKEYCOLUMNS=

[STMT_ENTRY][103]
SQL_TEXT=[
INSERT INTO emp(id, name) VALUES(99,?)
]
STATEMENT_TYPE=Prepared Statement
RESULTSET_TYPE=Forward Only
RESULTSET_CONCURRENCY=Read Only
AUTOGENERATEDKEYSREQUESTED=false
REQUESTEDKEYCOLUMNS=id,name
```

Using Java Logging (Salesforce)

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

Logging Components

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

The Java Logging API allows applications or components to define one or more named loggers. Messages written to the loggers can be given different levels of importance. For example, errors that occur in the driver can be written to a logger at the CONFIG level, while progress or flow information may be written to a logger at the FINE or FINER level. Each logger used by the driver can be configured independently. The configuration for a logger includes what level of log messages are written, the location to which they are written, and the format of the log message.

The Java Logging API defines the following levels:

- SEVERE
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST

Note: Log messages logged by the driver only use the CONFIG, FINE, FINER, and FINEST logging levels.

---

[103] The next statement pool entry.

Setting the log threshold of a logger to a particular level causes the logger to write log messages of that level and higher to the log. For example, if the threshold is set to `FINE`, the logger writes messages of levels FINE. CONFIG, INFO, WARNING, and SEVERE to its log. Messages of level FINER or FINEST are not written to the log.

The driver exposes loggers for the following functional areas:

- JDBC API
- SQL Engine
- Web service adapter

JDBC API Logger

Name

com.ddtek.jdbc.cloud.level

Purpose

Logs the JDBC calls made by the application to the driver and the responses from the driver back to the application. DataDirect Spy is used to log the JDBC calls.

Message Levels

`FINER` - Calls to the JDBC methods are logged at the FINER level. The value of all input parameters passed to these methods and the return values passed from them are also logged, except that input parameter or result data contained in InputStream, Reader, Blob, or Clob objects are not written at this level.

`FINEST` - In addition to the same information logged by the FINER level, input parameter values and return values contained in InputStream, Reader, Blob and Clob objects are written at this level.

`OFF` - Calls to the JDBC methods are not logged.

SQL Engine Logger

Name

com.ddtek.cloud.sql.level

Purpose

Logs the operations that the SQL engine performs while executing a query. Operations include preparing a statement to be executed, executing the statement, and fetching the data, if needed. These are internal operations that do not necessarily directly correlate with Web service calls made to the remote data source.

Message Levels

`CONFIG` - Any errors or warnings detected by the SQL engine are written at this level.

`FINE` - In addition to the same information logged by the CONFIG level, SQL engine operations are logged at this level. In particular, the SQL statement that is being executed is written at this level.

`FINER` - In addition to the same information logged by the CONFIG and FINE levels, data sent or received in the process of performing an operation is written at this level.

Web Service Adapter Logger

Name

com.ddtek.cloud.adapter.level

Purpose

Logs the Web service calls the driver makes to the remote data source and the responses it receives from the remote data source.

Message Levels

`CONFIG` - Any errors or warnings detected by the Web service adapter are written at this level.

`FINE` - In addition to the same information logged by the CONFIG level, information about Web service calls made by the Web service adapter and responses received by the Web service adapter are written at this level. In particular, the Web service calls made to execute the query and the calls to fetch or send the data are logged. The log entries for the calls to execute the query include the Salesforce-specific query being executed. The actual data sent or fetched is not written at this level.

`FINER` - In addition to the same information logged by the CONFIG and FINE levels, this level provides additional information.

`FINEST` - In addition to the same information logged by the CONFIG, FINE, and FINER levels, data associated with the Web service calls made by the Web service adapter is written.

Configuring Logging

You can configure logging using a standard Java properties file in either of the following ways:

- Using the properties file that is shipped with your JVM. See Using the JVM on page 367 for details.
- Using the driver. See Using the Driver on page 368 for details.

Using the JVM

If you want to configure logging using the properties file that is shipped with your JVM, use a text editor to modify the properties file in your JVM. Typically, this file is named logging.properties and is located in the `JRE/lib` subdirectory of your JVM. The JRE looks for this file when it is loading.

You can also specify which properties file to use by setting the java.util.logging.config.file system property. At a command prompt, enter:

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

where:

`properties_file`

is the name of the properties file you want to load.

Using the Driver

If you want to configure logging using the driver, you can use either of the following approaches:

- Use a single properties file for all Salesforce connections.
- Use a different properties file for each embedded database. For example, if you have two embedded databases (johnsmith.xxx and pattijohnson.xxx, for example), you can load one properties file for the johnsmith.xxx database and load another properties file for the pattijohnson.xxx database.

Note: By default, the name of the embedded database is the user ID specified for the connection. You can specify the name of the embedded database using the DatabaseName connection property. Refer to the "Salesforce Driver" section of the *DataDirect Connect Series for JDBC User's Guide* for details about using the DatabaseName connection property.

By default, the driver looks for the file named ddlogging.properties in the current working directory to load for all Salesforce connections.

If a property's file is specified for the LogConfigFile connection property, the driver uses the following process to determine which file to load:

1. The driver looks for the file specified by the LogConfigFile property.
2. If the driver cannot find the file in Step 1 on page 368, it looks for a properties file named `database_name`.logging.properties in the directory containing the embedded database for the connection, where *database_name* is the name of the embedded database.
3. If the driver cannot find the file in Step 2 on page 368, it looks for a properties file named ddlogging.properties in the current working directory.
4. If the driver cannot find the file in Step 3 on page 368 , it abandons its attempt to load a properties file.

If any of these files exist, but the logging initialization fails for some reason while using that file, the driver writes a warning to the standard output (System.out), specifying the name of the properties file being used.

Refer to "Salesforce Driver" in the *DataDirect Connect Series for JDBC User's Guide* for details about using the LogConfigFile connection property.

A sample properties file is installed in the `install_dir/testforjdbc` and `installdir/Examples/SforceSamples` directories, where:

`installdir`

is your product installation directory.

The file is named `ddlogging.properties`.You can copy this file from either location to the current working directory of your application or embedded database directory, and modify it using a text editor for your needs.

Glossary

authentication
The process of identifying a user, typically based on a user ID and password. Authentication ensures that the user is who they claim to be. See also *client authentication*, *NTLM authentication*, *OS authentication*, and *user ID/password authentication*.

bulk load
The process of sending large numbers of rows of data to the database in a continuous stream instead of in numerous smaller database protocol packets. This process also is referred to as bulk copy.

client authentication
Client authentication uses the user ID and password of the user logged onto the system on which the driver is running to authenticate the user to the database. The database server depends on the client to authenticate the user and does not provide additional authentication. See also *authentication*.

client load balancing
Client load balancing distributes new connections in a computing environment so that no one server is overwhelmed with connection requests.

connection pooling
Connection pooling allows you to reuse connections rather than create a new one every time a driver needs to establish a connection to the database. Connection pooling manages connection sharing across different user requests to maintain performance and reduce the number of new connections that must be created. See also *DataDirect Connection Pool Manager*.

connection retry
Connection retry defines the number of times the driver attempts to connect to the primary and, if configured, alternate database servers after an initial unsuccessful connection attempt. Connection retry can be an important strategy for system recovery.

connection URL
A connection URL is a string passed by an application to the Driver Manager that contains information required to establish a connection. See also *Driver Manager*.

DataDirect Connection Pool Manager
The DataDirect Connection Pool Manager is a component shipped with Progress DataDirect drivers that allows applications to use connection pooling.

DataDirect DB2 Package Manager
A Java graphical tool shipped with DataDirect Connect Series *for* JDBC for creating, dropping, and replacing DB2 packages for DB2.

DataDirect Spy
DataDirect Spy allows you to track and log detailed information about JDBC calls made by the drivers at runtime. This functionality is built into the drivers.

DataDirect Test
DataDirect Test is a menu-driven component shipped with Progress DataDirect drivers that helps you debug your applications and learn how to use the drivers. DataDirect Test displays the results of all JDBC function calls in one window, while displaying fully commented, Java JDBC code in an alternate window.

data source
A data source is a DataSource object that provides the connection information needed to connect to a database. The main advantage of using a data source is that it works with the Java Naming Directory Interface (JNDI) naming service, and it is created and managed apart from the applications that use it.

Driver Manager
The main purpose of the Driver Manager is to load drivers for the application. The Driver Manager also processes JDBC initialization calls and maps data sources to a specific driver.

failover
Failover allows an application to connect to an alternate, or backup, database server. Progress DataDirect drivers provide different levels of failover: connection failover, extended connection failover, and select failover.

index
A database structure used to improve the performance of database activity. A database table can have one or more indexes associated with it.

isolation level
An isolation level represents a particular locking strategy employed in the database system to improve data consistency. The higher the isolation level number, the more complex the locking strategy behind it. The isolation level provided by the database determines how a transaction handles data consistency.

The American National Standards Institute (ANSI) defines four isolation levels:

- Read uncommitted (0)
- Read committed (1)
- Repeatable read (2)
- Serializable (3)

J2EE
J2EE (Java 2 Platform, Enterprise Edition) technology and its component-based model simplify enterprise development and deployment. The J2EE platform manages the infrastructure and supports the Web services to enable development of secure, robust and interoperable business applications. Also known as Java EE (Java Platform, Enterprise Edition).

JDBC data source
See *data source*.

JNDI
The Java Naming and Directory Interface (JNDI) is a standard extension to the Java platform, providing Java technology-enabled applications with a unified interface to multiple naming and directory services in the enterprise. As part of the Java Enterprise API set, JNDI enables seamless connectivity to heterogeneous enterprise naming and directory services. Developers can now build powerful and portable directory-enabled applications using this industry standard.

JTA

JTA (Java Transaction API) specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the resource manager, the application server, and the transactional applications.

Kerberos

Kerberos is an OS authentication protocol that provides authentication using secret key cryptography. See also *authentication* and *OS authentication*.

load balancing

See *client load balancing*.

locking level

Locking is a database operation that restricts a user from accessing a table or record. Locking is used in situations where more than one user might try to use the same table at the same time. By locking the table or record, the system ensures that only one user at a time can affect the data.

NTLM authentication

NTLM (NT LAN Manager) is an authentication protocol that provides security for connections between Windows clients and servers. See also *authentication* and *OS authentication*.

OS authentication

OS authentication can take advantage of the user name and password maintained by the operating system to authenticate users to the database or use another set of user credentials specified by the application. 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. See also *authentication*, *Kerberos authentication*, and *NTLM authentication*.

reauthentication

The process of switching the user associated with a connection to another user to help minimize the number of connections required in a connection pool.

resource adapter

A resource adapter is a system-level software driver used by an application server to connect to an Enterprise Information Service (EIS). The resource adapter communicates with the server to provide the underlying transaction, security, and connection pooling mechanisms.

Secure Socket Layer

Secure Socket Layer (SSL) is an industry-standard protocol for sending encrypted data over database connections. SSL secures the integrity of your data by encrypting information and providing SSL client/SSL server authentication. See also *SSL client/server authentication*.

SSL client and server authentication

SSL (Secure Socket Layer) 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.

See also *Secure Socket Layer*.

Unicode

A standard for representing characters as integers. Unlike ASCII, which uses 7 bits for each character, Unicode uses 16 bits, which means that it can represent more than 65,000 unique characters. This is necessary for many languages, such as Greek, Chinese, and Japanese.

user ID and password authentication

User ID and password authentication authenticates the user to the database using a database user name and password. See also *authentication*.

Index

A

accessing the DataDirect Statement Pool Monitor *350*
Add clause (for local tables) *105*
Add clause (for remote tables) *103*
aggregate functions *138*
Alter Cache (EXT) *98*
Alter Index *100*
Alter Sequence *100*
Alter Session (EXT) *101*
Alter Table *102*
Apache Hive, driver for
    between clause *89*
    column name qualification *89*
    From clause *89*
    Group By clause *90*
    Having clause *90*
    Insert statement *88*
    Order By clause *90*
    restrictions *95*
    SQL statement support *87*
    subqueries *91*
Apache Hive, getTypeInfo() *244*
application
    connection pooling *12*
    DataDirect Spy, using to troubleshoot *357*
    troubleshooting problems *357*
arithmetic operators *92*, *151*
Array interface, methods *18*
attributes, DataDirect Spy *328*
auto generated keys
    example, retrieving *273*
    performance optimization *273*
Autocommit mode *274*

B

batch inserts and updates
    batch execution on a prepared statement with DataDirect Test *304*
    using instead of prepared statements *271*
binary
    literals *150*
    operators *150*
Blob interface, methods *18*
Blobs
    retrieving with DataDirect Test *319*

C

Call Limit clause *113*
CallableStatement interface, methods *19*
changing the maximum pool size behavior *333*
character operators *92*
character string literals *149*
checking version of DataDirect Connection Pool Manager *334*
Checkpoint *108*
clearing statements in the statement pool *352*
Clob interface, methods *30*
Clobs
    retrieving with DataDirect Test *319*
closing the connection pool *339*
column
    definition *122*
    names *148*
com.ddtek.jdbc.extensions package *12*, *71*
committing transactions *274*
comparison operators *151*
components, Java logging (Salesforce) *365*
concatenation operator *92*, *151*
conditions *159*
connecting
    DataDirect Test
        using a data source *293*
        using driver/database selection *294*
    example
        Driver Manager and DataDirect Spy *326*
    using connection pool *338*
    using DataDirect Test *293*
Connection interface, methods *31*
connection management *274*
Connection object
    ExtLogControl interface *358*
    managing connections *274*
    transaction model *275*
connection pool, See connection pooling
Connection Pool Monitor *342*
connection pooling
    connection pool
        closing *339*
        configuring *332*
        connecting using *338*
        creating *335–336*
        PooledConnectionDataSource object, creating *336*
    Connection Pool Monitor *342*
    driver DataSource object for connection pooling, creating *335*
    example
        DataDirect Connection Pool Manager trace file *360*

connection pooling *(continued)*
- example *(continued)*
  - pooled data source *335*
- performance optimization *274*
- PooledConnectionDataSource object, creating *336*

ConnectionEventListener interface, methods *36*
ConnectionPoolDataSource interface, methods *36*
ConnectionPoolMonitor interface, methods *344*
constants
- Apache Hive, driver for *91*

constraint definition *119*, *123*
conventions, typographical *13*
copyright *3*
correlated subqueries *161*
Create Cache (EXT) *108*
Create Index *115*
Create Sequence *115*
Create Table *116*
Create View *127*
creating
- connection pool *336*
- driver DataSource object for connection pooling *335*
- PooledConnectionDataSource object *336*

cursors
- choosing *272*

D

data sources
- creating
  - driver DataSource object for connection pooling *335*
  - PooledConnectionDataSource object *336*

data types, choosing for performance *269*
database
- table characteristics, using dummy query to determine *267*

database metadata
- retrieving with DataDirect Test *300*

DatabaseMetaData interface (Salesforce) *73*
DatabaseMetaData interface, methods *36*
DataDirect Connection Pool Manager
- checking the version *334*
- trace file
  - example *360*
  - using *360*
- tracing, enabling *335*, *360*
- using reauthentication with *333*

DataDirect Spy
- attributes *328*
- enabling
  - using JDBC data sources *327*
  - using JDBC Driver Manager *326*
- example
  - JDBC data source connection *327*
  - JDBC Driver Manager connection *326*
- logging
  - generating a log *357*
  - JDBC calls to System.out *328*
  - log example *358*
  - turning on and off *358*
- setEnableLogging() method *358*
- SpyAttributes connection property, using *326*

DataDirect Statement Pool Monitor
- accessing
  - using the JMX API *350*
- classes and interfaces *353*
- using *12*

DataDirect Test
- batch execution *304*
- configuring *292*
- connecting with *293*
- database metadata, retrieving *300*
- deleting rows *311*
- executing
  - Select statement *296*
- executing prepared statement *297*
- inserting rows *311*, *313*
- LOB support *319*
- ParameterMetaData, returning *307*
- result set, scrolling through *302*
- savepoints, establishing *308*
- starting *292*
- tutorial *291*
- updatable result sets *311*
- updating rows *311*, *316*
- using *291*

DataSource interface, methods *45*
DataSource object
- connection pooling
  - creating DataSource object for *335*
  - obtaining a connection *331*
  - referencing for *336*, *340*

date, time, timestamp escape sequences *278*
date/time literals *150*
DB2
- getTypeInfo() method *164*

DDBulkLoad interface, methods *73*
Delete *128*
deleting rows
- with DataDirect Test *311*

documentation library *13*
Driver interface, methods *45*
driver, DataSource object for connection pooling *335*
Drop Cache (EXT) *129*
Drop clause *104*, *106*
Drop Index *129*
Drop Sequence *130*
Drop Table *131*
Drop View *131*

dummy query, using to determine table characteristics 267

E

Enabled clause 112
escape sequences
  date, time, and timestamp 278
  LIKE escape character for wildcards 289
  outer join 287
  procedure call 289
example
  data source for connection pooling 335
  DataDirect Spy
    JDBC data source connection 327
    JDBC Driver Manager connection 326
    log 358
  date, time, and timestamp escape sequence 278
  Driver Manager, using to connect 326
  outer join escape sequence 287
  scalar functions 278
  trace file, DataDirect Connection Pool Manager 360
Except operator 143
EXISTS predicate 160
Explain Plan 132
export file for statement pool
  example 364
  generating 353
ExtConnection interface, methods 80
ExtDatabaseMetaData interface, methods 85
extensions for metadata (Salesforce) 73
external ID column, specifying 133
ExtLogControl class, methods 85
ExtStatementPoolMonitor class 354
ExtStatementPoolMonitorMBean interface 354

F

Filter clause 114
Foreign Key clause 123
forward-only cursor
  performance implications 272
freezing the statement pool 353
From clause 139
functions
  Apache Hive 93
  Salesforce 154

G

generating statement pool export file 364
getBestRowIdentifier() method 275
getBlob() method 268
getClob() method 268
getObject() method 272
getting network timeout (SQL Server) 81
getTypeInfo() method
  Apache Hive 244
  DB2 164
  Greenplum 249
  Informix 175
  MySQL 184
  Oracle 195
  PostgreSQL 203
  Progress OpenEdge 210
  Salesforce 256
  SQL Server 218
  Sybase 232
  Windows Azure 218
Greenplum, getTypeInfo() 249
Group By clause 141

H

Having clause 141
help, online 14

I

importing statements into the statement pool 352
IN predicate 160
Informix, getTypeInfo() method 175
Initial Check clause 111
initial pool size 332
InputStream object (DataDirect Spy) 327–328
insensitive cursors
  performance implications 272
Insert 132
Insert statement
  Apache Hive, driver for 88
inserting rows
  with DataDirect Test 313
integer literals 149
Interfaces, JDBC 17
Intersect operator 143

J

Java logging (Salesforce)
  components 365
  JDBC API logger 366
  SQL engine logger 366
  using 365
  Web service adapter logger 367
JDBC
  functionality supported 17
  interfaces 17
  JVM compatibility 17
  methods supported 17
  versions supported 17

JDBC API logger (Salesforce) 366
JDBC Driver Manager, specifying SpyAttributes connection property with 325
JDBC extensions 12, 71
join in a From clause 139
JTA support
  transaction
    model, choosing for performance 275
JTA transaction support
  managing commits 274
JVM
  JDBC compatibility 17
  properties file 367

K

keyset-driven cursors, performance implications 272

L

library, documentation 13
LIKE escape character for wildcards escape sequence 289
Limit clause 145
literals
  about 148
  arguments, using parameter markers 270
  binary 150
  character string 149
  date/time 150
  escape sequences 278
  integer 149
  numeric 150
LOBs support
  executing a query with DataDirect Test 319
log for DataDirect Spy, using 357
logging
  JVM properties file 367
logging, Java (Salesforce)
  components 365
  JDBC API logger 366
  SQL engine logger 366
  using 365
  Web service adapter logger 367
logical operators 93, 153
long data, retrieving and performance 268

M

maximum idle time 332
maximum pool size 332
maximum pool size behavior 333
MBean name, registering 350
metadata methods, minimizing use of 266
methods
  Array interface 18
  Blob interface 18
  CallableStatement interface 19
  Clob interface 30
  Connection interface 31
  ConnectionEventListener interface 36
  ConnectionPoolDataSource interface 36
  DatabaseMetaData interface 36
  DataSource interface 45
  Driver interface 45
  ExtConnection class 80
  ExtDatabaseMetaData class 85
  ExtLogControl class 85
  ParameterMetaData interface 46
  PooledConnection interface 47
  PooledConnectionDataSource interface 340
  PooledConnectionDataSourceFactory interface 340
  PreparedStatement interface 47
  Ref interface 52
  ResultSet interface 52
  ResultSetMetaData interface 62
  RowSet interface 63
  SavePoint interface 64
  Statement interface 64
  StatementEventListener interface 68
  Struct interface 68
  XAConnection interface 68
  XADataSource interface 69
  XAResource interface 69
minimum pool size 332
Minus operator 143
MySQL, getTypeInfo() 184

N

network timeout (SQL Server)
  getting 81
  setting 84
Next Value For clause 116
numeric literals 150

O

object
  Connection
    ExtLogControl interface 358
    managing connections 274
    transaction model 275
  DataSource
    connection pooling 331, 335–336
  DDBulkLoad, methods 73
  ExtConnection, methods 84
  ExtControl, methods 85
  ExtDatabaseMetaData, methods 85
  InputStream (DataDirect Spy) 327–328

object *(continued)*
Long (DB2) 55
pooled data source 335
PooledConnectionDataSource
connecting with 338
connection pooling 335–336, 340
ConnectionPoolMonitor 345
creating 335–336, 340
PreparedStatement
using Statement object instead of 270
Reader (DataDirect Spy) 327–328
Reference, connection pooling 340–341
ResultSet
database metadata 266
generating 266
updating data 275
Statement
using instead of PreparedStatement object 270
using multiple 274
using addBatch() instead of PreparedStatement 271
OpenEdge, See Progress OpenEdge
operators
arithmetic 92, 151
comparison 151
concatenation 92, 151
logical 93, 153
precedence 153
relational 92
Oracle, getTypeInfo() method 195
Order By clause 144
outer join escape sequence, example 287

P

parameter markers, using as arguments to stored procedures 270
parameter metadata
returning with DataDirect Test 307
ParameterMetaData interface, methods 46
performance optimization
auto generated keys, retrieving 273
batches, using instead of prepared statements 271
commits, managing 274
connection
management 274
pooling 274
database metadata methods 266
designing JDBC applications 273
get methods, using effectively 272
getBestRowIdentifier() 275
result sets, retrieving 269
retrieving long data 268
scrollable cursors 269
selecting JDBC objects and methods 270
transaction model, choosing 275
update methods of the ResultSet object 275
performance optimization *(continued)*
updating data 275
Persist clause 111
PooledConnection interface, methods 47
PooledConnectionDataSource interface, methods 340
PooledConnectionDataSource object
connecting with 338
ConnectionPoolMonitor 345
creating 335–336, 340
PooledConnectionDataSourceFactory interface, methods 340
poolEntries() method 348
PostgreSQL, getTypeInfo() 203
predicate
EXISTS 160
IN 160
UNIQUE 160
prepared statement pooling, performance optimization 274
prepared statement, executing with DataDirect Test 297
prepared statements
using batches instead of 271
PreparedStatement interface, methods 47
PreparedStatement object
performance
implications of using Statement object instead 270
of prepared statement pool 274
prepared statement pooling 274
using Statement object instead of 270
Primary Key clause 123
procedure call 289
product documentation 13
Progress OpenEdge, getTypeInfo() 210
properties file for Java logging (Salesforce) driver 368

R

Reader object (DataDirect Spy) 327–328
reauthentication
enabling in DataDirect Connection Pool Manager 333
using with the DataDirect Connection Pool Manager 333
Ref interface, methods 52
Reference object
using to create a PooledConnectionDataSource object 340
Referencing clause 110
Refresh Cache (EXT) 134
Refresh Interval clause 110
Refresh Schema (EXT) 135
registering MBean name 350
relational operators 92
Rename clause 107

result sets
deleting rows with DataDirect Test 311
inserting rows with DataDirect Test 311
scrolling through a result set with DataDirect Test 302
updating rows with DataDirect Test 311
result sets, scrollable
performance optimization 269
ResultSet interface, methods 52
ResultSet object
database metadata 266
generating 266
updating data 275
ResultSetMetaData interface, methods 62
RowSet interface, methods 63

S

Salesforce
getTypeInfo() 256
Java logging 365
Java logging components 365
JDBC API logger 366
Remote Object Mapping 135
SQL engine logger 366
Web Service adapter logger 367
SavePoint interface, methods 64
savepoints
establishing with DataDirect Test 308
scalar functions 278
search patterns, avoiding 267
Select clause 136
Select statement
executing with DataDirect Test 296
sensitive cursors
performance implications 272
server-side RPCs 270
Set Checkpoint Defrag 145
Set Logsize 146
setEnableLogging(), using to turn on and off DataDirect Spy logging 358
setting the network timeout (SQL Server) 84
SQL
expressions
driver for Apache Hive 91
SQL engine logger (Salesforce) 366
SQL escape sequences
date, time, timestamp 278
LIKE escape character for wildcards 289
outer join 287
procedure call 289
scalar functions 278
SQL expressions 147
SQL Server
getTypeInfo() method 218
SQL Server *(continued)*
network timeout
getting 81
setting 84
SQL statements
Alter Cache 98
Alter Index 100
Alter Sequence 100
Alter Session 101
Alter Table 102
Checkpoint 108
Create Cache (EXT) 108
Create Index 115
Create Sequence 115
Create Table 116
Create View 127
Delete 128
Drop Cache 129
Drop Index 129
Drop Sequence 130
Drop Table 131
Drop View 131
Explain Plan 132
Insert 132
Refresh Cache 134
Refresh Schema 135
Select 135
Set Checkpoint Defrag 145
Set Logsize 146
SQL expressions 147
Update 146
Statement interface, methods 64
Statement object
Connection object association 274
using instead of PreparedStatement object 270
using multiple 274
when to use 270
statement pool
clearing statements 352
export file, generating 353
freezing and unfreezing 353
importing statements to 352
statement pool export file
example 364
generating 364
Statement Pool Monitor
accessing with DataDirect-specific methods 348
classes and interfaces 353
poolEntries() method 348
statement pooling
statement pool export file 364
troubleshooting problems 364
StatementEventListener interface, methods 68
stored procedures
escape sequences 289
parameter markers as arguments, using 270

Struct interface, methods 68
subqueries
Apache Hive, driver for 91
correlated 161
overview 159
subquery in a From clause 140
support
online help 14
technical support 14
Sybase, getTypeInfo() method 232
System.out, logging JDBC calls to using DataDirect Spy 328

T

Technical Support 14
time literal escape sequence 278
timestamp literal escape sequence 278
tracing, enabling for Pool Manager 335
transactions , See JTA support
troubleshooting
application problems 357
connection pooling problems 360
statement pooling problems 364
turning on and off DataDirect Spy logging 358
Type 4 11
Type 5 11

U

unary operator 150
understanding the maximum pool size 333
unfreezing the statement pool 353
Union operator 142
Unique clause 123
UNIQUE predicate 160
updatable result sets, DataDirect Test 311
Update 146
updating rows
with DataDirect Test 316
using
DataDirect Spy log 357
DataDirect Statement Pool Monitor 12
JMX API 350
Wrapper methods 72

W

Web service adapter logger (Salesforce) 367
Where clause 140
Windows Azure
getTypeInfo() method 218
Wrapper methods 72

X

XAConnection interface, methods 68
XADataSource interface, methods 69
XAResource interface, methods 69