Informix Product Family
Data Server Driver for JDBC and SQLJ
Version 9.7

IBM Data Server Driver for JDBC and SQLJ for Informix

SC27-3850-00

Informix Product Family
Data Server Driver for JDBC and SQLJ
Version 9.7

IBM Data Server Driver for JDBC and SQLJ for Informix

SC27-3850-00

Note
Before using this information and the product it supports, read the information in "Notices" on page B-1.

Contents

Introduction

In This Introduction

This introduction provides an overview of the information in this publication and describes the conventions it uses.

About This Publication

This publication is a guide for using the IBM® Data Server Driver for JDBC and SQLJ to connect to IBM Informix® data servers.

Types of Users

This guide is for Java programmers who use the JDBC API to connect to Informix databases using the IBM Data Server Driver for JDBC and SQLJ. To use this guide, you should know how to program in Java and, in particular, understand the classes and methods of the JDBC API.

Assumptions About Your Locale

IBM Informix products can support many languages, cultures, and code sets. All the information related to character set, collation, and representation of numeric data, currency, date, and time is brought together in a single environment, called a Global Language Support (GLS) locale.

This publication assumes that your database uses the default locale. This default is **en_us.8859-1** (ISO 8859-1) on UNIX platforms or **en_us.1252** (Microsoft 1252) in Windows environments. This locale supports U.S. English format conventions for displaying and entering date, time, number, and currency values. It also supports the ISO 8859-1 (on UNIX and Linux) or Microsoft 1252 (on Windows) code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use nondefault characters in your data or in SQL identifiers, or if you plan to use other collation rules for sorting character data, you need to specify the appropriate nondefault locale.

For instructions on how to specify a nondefault locale, and for additional syntax and other considerations related to GLS locales, see the *IBM Informix GLS User's Guide*.

What's New for IBM Data Server Driver for JDBC and SQLJ for IBM Informix

This topic lists the new features in the IBM Data Server Driver for JDBC and SQLJ for IBM Informix.

What's New in version 3.62

The following are enhancements for IBM Data Server Driver for JDBC and SQLJ, Version 3.62.

Table 1. What's New in IBM Data Server Driver for JDBC and SQLJ, Version 3.62

Overview	Reference
Diagnosis and trace enhancements The following diagnosis and trace enhancements are added: • The DB2Jcc utility tests a connection to a data server, using IBM Data Server Driver for JDBC and SQLJ type 4 connectivity. • When the tracePolling configuration property is set to enable the trace while an application is running, information about all PreparedStatement objects in the application that were prepared before the trace was enabled are written to the trace destination.	Chapter 9, "Problem diagnosis with the IBM Data Server Driver for JDBC and SQLJ," on page 9-1
New properties The following Connection and DataSource property is added: **`queryTimeoutProcessingMode`** Specifies whether the IBM Data Server Driver for JDBC and SQLJ cancels the SQL statement or closes the underlying connection when the query timeout interval for a Statement object expires.	"Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products" on page 14-8

What's New in version 3.61

The following are enhancements for IBM Data Server Driver for JDBC and SQLJ, Version 3.61.

Table 2. What's New in IBM Data Server Driver for JDBC and SQLJ, Version 3.61

Overview	Reference
New properties The following Connection and DataSource properties are added: **`stripTrailingZerosForDecimalNumbers`** Specifies whether the IBM Data Server Driver for JDBC and SQLJ removes trailing zeroes when it retrieves data from a DECFLOAT, DECIMAL, or NUMERIC column.	"Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products" on page 14-8
DB2PreparedStatement enhancements Two new DB2PreparedStatement methods are added. **getEstimateCost** Returns the estimated cost of an SQL statement after the statement is dynamically prepared. **getEstimateRowCount** Returns the estimated number of rows that can be returned by an SQL statement after the statement is dynamically prepared.	"DB2PreparedStatement interface" on page 14-162

Table 2. What's New in IBM Data Server Driver for JDBC and SQLJ, Version 3.61 (continued)

Overview	Reference
Trusted context support Trusted context support is available for Informix data servers. Trusted connections are supported for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to Informix V11.70 and later.	"IBM Data Server Driver for JDBC and SQLJ trusted context support" on page 8-6
Informix Unified Debugger support An existing method is extended to support the Informix Unified Debugger. Method DB2Connection.setDB2ClientDebugInfo can be called to notify the Informix data server that stored procedures and user-defined functions that are using the connection are running in debug mode.	"DB2Connection interface" on page 14-139
System monitoring support System monitoring support is extended to Informix data servers. You can collect core driver time, network I/O time, server time, and application time for connections to Informix servers.	Chapter 10, "System monitoring for the IBM Data Server Driver for JDBC and SQLJ," on page 10-1

What's New in version 3.59

The following are enhancements for IBM Data Server Driver for JDBC and SQLJ, Version 3.59.

Table 3. What's New in IBM Data Server Driver for JDBC and SQLJ, Version 3.59

Overview	Reference
New properties The following Connection and DataSource properties are added: **`allowNullResultSetForExecuteQuery`** Specifies whether the IBM Data Server Driver for JDBC and SQLJ returns null when Statement.executeQuery, PreparedStatement.executeQuery, or CallableStatement.executeQuery is used to execute a CALL statement for a stored procedure that does not return any result sets. **`connectionCloseWithInFlightTransaction`** Specifies whether the IBM Data Server Driver for JDBC and SQLJ throws an SQLException or rolls back a transaction without throwing an SQLException when a connection is closed in the middle of the transaction. **`interruptProcessingMode`** Specifies the behavior of the IBM Data Server Driver for JDBC and SQLJ when an application calls the Statement.cancel method.	"Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products" on page 14-8

What's New in version 3.58

The following are enhancements for IBM Data Server Driver for JDBC and SQLJ, Version 3.58.

Table 4. What's New in IBM Data Server Driver for JDBC and SQLJ, Version 3.58

Overview	Reference
Diagnostic information enhancements Diagnostic information is traced to the Java standard error output stream when an exception is thrown with an SQL error code of -805. In Java database applications, -805 often indicates that all available IBM Data Server Driver for JDBC and SQLJ packages have been used because there are too many concurrently open statements. The diagnostic information contains a list of SQL strings that contributed to the exception.	Not applicable

What's New in version 3.57

The following are enhancements for IBM Data Server Driver for JDBC and SQLJ, Version 3.57.

Table 5. What's New in IBM Data Server Driver for JDBC and SQLJ, Version 3.57

Overview	Reference
Parameter names in JDBC and SQLJ stored procedure calls In previous versions of the IBM Data Server Driver for JDBC and SQLJ, only forms of CallableStatement.RegisterOutParameter, CallableStatement.setXXX, and CallableStatement.getXXX methods that used *parameterIndex* were supported. With versions 3.57 of the driver, *parameterName* is also supported in those methods. *parameterName* is a name that is specified for a parameter in the stored procedure definition. Alternatively, for JDBC applications, new syntax allows the application to map parameter markers in the CALL statement to the parameter names in the stored procedure definition. For example, in a JDBC application, `CALL MYPROC (A=>?)` maps a parameter marker to stored procedure parameter A. For SQLJ applications, new syntax allows the application to map host variable names in the CALL statement to the parameter names in the stored procedure definition. For example, in an SQLJ application, `CALL MYPROC (A=>:INOUT x)` maps host variable x to stored procedure parameter A. With the new syntax, you do not need to specify all parameters in the CALL statement. Unspecified parameters take the default values that are specified in the stored procedure definition.	Not applicable
Savepoints The IBM Data Server Driver for JDBC and SQLJ supports setting of savepoints for connections to IBM Informix data servers.	"Savepoints in JDBC applications" on page 5-39 "Savepoints in SQLJ applications" on page 6-41
Batch insert operations The IBM Data Server Driver for JDBC and SQLJ adds the atomicMultiRowInsert Connection or DataSource property for connections to IBM Informix V11.10 and later data servers. The atomicMultiRowInsert property lets you specify whether batch insert operations that use the PreparedStatement interface have atomic or non-atomic behavior. Atomic behavior means that a batch operation succeeds only if all insert operations in the batch succeed. Non-atomic behavior, which is the default, means that insert operations succeed or fail individually.	"Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products" on page 14-8

Table 5. What's New in IBM Data Server Driver for JDBC and SQLJ, Version 3.57 (continued)

Overview	Reference
Diagnostics for binding of SQLJ applications enhancements When an SQLJ application is bound, and an SQL error or warning occurs, the following new diagnostic information is returned: • The SQL statement • The line number in the program of the SQL statement • The error or warning code and the SQLSTATE value • The error message	Not applicable
Client reroute enhancements Client reroute support is enhanced in the following ways: • Seamless failover is added to client reroute operation. During client reroute, if a connection is in a clean state, you can use the enableSeamlessFailover property to suppress the SQLException with error code -4498 that the IBM Data Server Driver for JDBC and SQLJ issues to indicate that a failed connection was re-established. • Client affinities are added to cascaded failover support. For cascaded failover, you can use the enableClientAffinitiesList property to control the order in which primary and alternate server reconnections are attempted after a connection failure.	"Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products" on page 14-8
Connections to IBM Informix enhancements For connections to IBM Informix servers, the following enhancement is added: • Support for new IBM Informix data types is added. As of IBM Informix 11.50, IBM Informix supports the BIGINT and BIGSERIAL data types. The IBM Data Server Driver for JDBC and SQLJ lets you access columns with those data types. For retrieving automatically generated keys from a BIGSERIAL column, the IBM Data Server Driver for JDBC and SQLJadds the DB2Statement.getIDSBigSerial method.	"Data types that map to database data types in Java applications" on page 14-1

Table 5. What's New in IBM Data Server Driver for JDBC and SQLJ, Version 3.57 (continued)

Overview	Reference
Automatically generated keys enhancements Batched INSERT statements can return automatically generated keys. If batch execution of a PreparedStatement object returns automatically generated keys, you can call the DB2PreparedStatement.getDBGeneratedKeys method to retrieve an array of ResultSet objects that contains the automatically generated keys. If a failure occurs during execution of a statement in a batch, you can use the DBBatchUpdateException.getDBGeneratedKeys method to retrieve any automatically generated keys that were returned.	"Making batch updates in JDBC applications" on page 5-16 "DBBatchUpdateException interface" on page 14-132 "DB2PreparedStatement interface" on page 14-162

What's New in version 3.53

The following are enhancements for IBM Data Server Driver for JDBC and SQLJ, Version 3.53.

Table 6. What's New in IBM Data Server Driver for JDBC and SQLJ, Version 3.53

Overview	Reference
Support for new IBM Informix data types is added. As of IBM Informix 11.50, IBM Informix supports the BIGINT and BIGSERIAL data types. With the IBM Data Server Driver for JDBC and SQLJ you can access columns with those data types. For retrieving automatically generated keys from a BIGSERIAL column, the DB2Statement.getIDSBigSerial method is added to the IBM Data Server Driver for JDBC and SQLJ.	"DB2Statement interface" on page 14-178 "Data types that map to database data types in Java applications" on page 14-1
The fetchSize default is configurable. The following property is added: **fetchSize** Specifies the default fetch size for newly created Statement objects. This value is overridden by the Statement.setFetchSize method.	"Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products" on page 14-8 "DB2BaseDataSource class" on page 14-132

Table 6. What's New in IBM Data Server Driver for JDBC and SQLJ, Version 3.53 (continued)

Overview	Reference
Two new SSL properties are available to provide the location and password of truststore for SSL connection. The following properties are added: **sslTrustStoreLocation** Specifies the name of the Java truststore on the client that contains the server certificate for an SSL connection. **sslTrustStorePassword** Specifies the password for the Java truststore on the client that contains the server certificate for an SSL connection.	"Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products" on page 14-8 "DB2BaseDataSource class" on page 14-132 "Configuring connections under the IBM Data Server Driver for JDBC and SQLJ to use SSL" on page 8-8 "Configuring the Java Runtime Environment to use SSL" on page 8-9
Timestamp formatting is improved for compatibility with all supported servers. The following property is added: **timestampPrecisionReporting** Specifies whether trailing zeroes in a timestamp value that is retrieved from a data source are truncated.	"Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products" on page 14-8

What's New in version 3.52

The following are enhancements for IBM Data Server Driver for JDBC and SQLJ, Version 3.52.

Table 7. What's New in IBM Data Server Driver for JDBC and SQLJ, Version 3.52

Overview	Reference
The IBM Data Server Driver for JDBC and SQLJ now supports automatic client reroute when connecting to IBM Informix. For connections to IBM Informix 11.50 and later, automatic client reroute with Connection Manager can be enabled. This enhancement allows the driver to establish a connection to an alternate member of the IBM Informix cluster if the primary member fails.	"Java client support for high availability for connections to IBM Informix servers" on page 11-18
The IBM Data Server Driver for JDBC and SQLJ now supports workload balancing when connecting to IBM Informix. For connections to IBM Informix 11.50 and later, workload balancing and the IBM Data Server Driver for JDBC and SQLJ balance the load among different high-availability servers in a cluster. The Connection Manager ensures that work is distributed efficiently among servers in the cluster and that work is transferred to another server if one server has a failure.	"Operation of workload balancing for connections to IBM Informix from Java clients" on page 11-27

Table 7. What's New in IBM Data Server Driver for JDBC and SQLJ, Version 3.52 (continued)

Overview	Reference
With the IBM Data Server Driver for JDBC and SQLJ Version 3.52 or later, preparing an SQL statement for retrieval of automatically generated keys is supported.	"DB2PreparedStatement interface" on page 14-162

What's New in version 3.51

The following are enhancements for IBM Data Server Driver for JDBC and SQLJ, Version 3.51.

Table 8. What's New in IBM Data Server Driver for JDBC and SQLJ, Version 3.51

Overview	Reference
JDBC 4.0 support has been added to the driver for connections to IBM Informix 11.50. You can now use the **db2jcc4.jar** file for JDBC 4.0 functions as well as JDBC 3.0 and earlier functions. To use JDBC 4.0, you need an SDK for Java, Version 6.	Chapter 4, "Installing the IBM Data Server Driver for JDBC and SQLJ," on page 4-1
Support for SQLJ has been added to the driver for connections to IBM Informix 11.50. You can now use the **sqlj.zip** if you plan to prepare SQLJ applications that include only JDBC 3.0 and earlier functions. Use the **sqlj4.zip** file to prepare SQLJ applications that include JDBC 4.0 functions as well as JDBC 3.0 and earlier functions. For connections to IBM Informix, SQL statements in SQLJ applications run dynamically; SQL statements cannot be run statically.	Chapter 4, "Installing the IBM Data Server Driver for JDBC and SQLJ," on page 4-1 "Program preparation for SQLJ programs" on page 7-1 Chapter 6, "SQLJ application programming," on page 6-1
Longer database names are supported. Previously, IBM Informix DRDA® connections limited database names to 18 bytes. For connections using IBM Data Server Driver for JDBC and SQLJ, Version 3.51 and later, database names can be up to 128 bytes.	Not applicable
IBM Informix ISAM error reporting is enabled. For connections to IBM Informix 11.10 and later, ISAM errors are reported as SQLException objects. You can now use SQLException methods to obtain the error code and the message description. The SQLException.printStackTrace method displays the cause of the ISAM errors.	"Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ" on page 5-53

Table 8. What's New in IBM Data Server Driver for JDBC and SQLJ, Version 3.51 (continued)

Overview	Reference
The IBM Data Server Driver for JDBC and SQLJ, Version 3.51 can access these new features of IBM Informix 11.50: • Progressive streaming for LOB data • Multi-row insert batched update capability • Secure Socket Layer (SSL) • Setting and retrieving client information properties	"Progressive streaming with the IBM Data Server Driver for JDBC and SQLJ" on page 5-32 "Making batch updates in JDBC applications" on page 5-16 "IBM Data Server Driver for JDBC and SQLJ support for SSL" on page 8-8 "Client info properties support by the IBM Data Server Driver for JDBC and SQLJ" on page 5-47
Global trace settings can be changed without shutting down the driver. You can set the db2.jcc.tracePolling property before you start the driver so that you can change global configuration trace properties while the driver is running.	Chapter 9, "Problem diagnosis with the IBM Data Server Driver for JDBC and SQLJ," on page 9-1 "IBM Data Server Driver for JDBC and SQLJ configuration properties" on page 14-52

Example code conventions

Examples of SQL code occur throughout this publication. Except as noted, the code is not specific to any single IBM Informix application development tool.

If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement. If you are using DB–Access, you must delimit multiple statements with semicolons.

Tip: Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the documentation for your product.

Additional documentation

Documentation about this release of IBM Informix products is available in various formats.

You can access or install the product documentation from the Quick Start CD that is shipped with Informix products. To get the most current information, see the Informix information centers at ibm.com®. You can access the information centers and other Informix technical information such as technotes, white papers, and IBM Redbooks® publications online at http://www.ibm.com/software/data/sw-library/.

Compliance with industry standards

IBM Informix products are compliant with various standards.

IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment (CAE) standards.

The IBM Informix Geodetic DataBlade® Module supports a subset of the data types from the *Spatial Data Transfer Standard (SDTS)—Federal Information Processing Standard 173*, as referenced by the document *Content Standard for Geospatial Metadata*, Federal Geographic Data Committee, June 8, 1994 (FGDC Metadata Standard).

Syntax diagrams

Syntax diagrams use special components to describe the syntax for statements and commands.

Table 9. Syntax Diagram Components

Component represented in PDF	Component represented in HTML	Meaning
	`>>----------`	Statement begins.
	`---------->`	Statement continues on next line.
	`>----------`	Statement continues from previous line.
	`----------><`	Statement ends.
	`--------SELECT----------`	Required item.
	`--+-----------------+---` `  '------LOCAL------'`	Optional item.

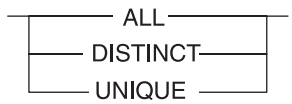
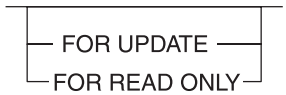
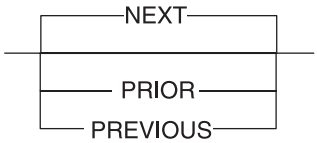
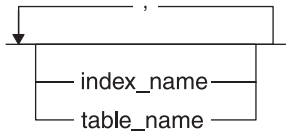
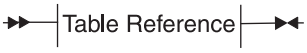
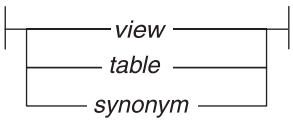
	`---+-----ALL-------+---`
	`  +--DISTINCT-----+`
	`  '---UNIQUE------'`
		Required item with choice. Only one item must be present.

Table 9. Syntax Diagram Components (continued)

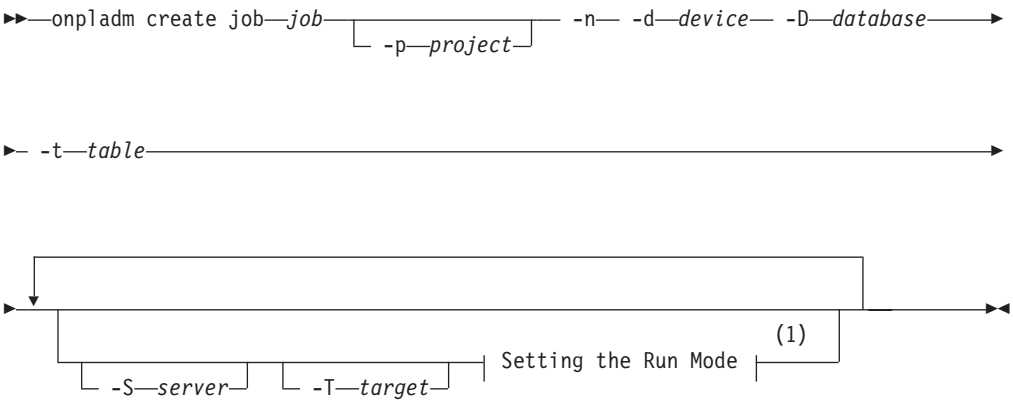
Component represented in PDF	Component represented in HTML	Meaning
	```---+------------------+--- +--FOR UPDATE-----+ '--FOR READ ONLY--'```	Optional items with choice are shown below the main line, one of which you might specify.
	```.---NEXT---------. ----+----------------+--- +---PRIOR--------+ '---PREVIOUS-----'```	The values below the main line are optional, one of which you might specify. If you do not specify an item, the value above the line is used by default.
	```.-------,-----------. V | ---+-----------------+--- +---index_name---+ '---table_name---'```	Optional items. Several items are allowed; a comma must precede each repetition.
	```>>-| Table Reference |-><```	Reference to a syntax segment.
Table Reference 	```Table Reference |--+-----view--------+--| +------table------+ '----synonym------'```	Syntax segment.

How to read a command-line syntax diagram

Command-line syntax diagrams use similar elements to those of other syntax diagrams.

Some of the elements are listed in the table in Syntax Diagrams.

Creating a no-conversion job



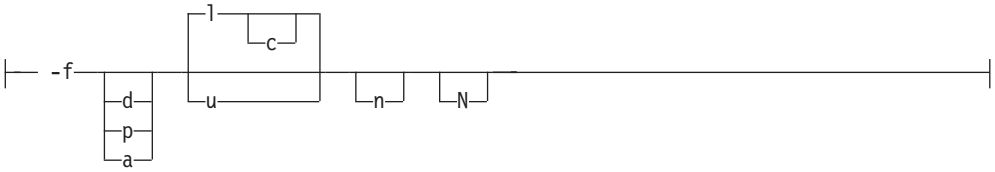
Notes:

1 See page Z-1

This diagram has a segment named “Setting the Run Mode,” which according to the diagram footnote is on page Z-1. If this was an actual cross-reference, you

would find this segment on the first page of Appendix Z. Instead, this segment is shown in the following segment diagram. Notice that the diagram uses segment start and end components.

Setting the run mode:



To see how to construct a command correctly, start at the upper left of the main diagram. Follow the diagram to the right, including the elements that you want. The elements in this diagram are case-sensitive because they illustrate utility syntax. Other types of syntax, such as SQL, are not case-sensitive.

The Creating a No-Conversion Job diagram illustrates the following steps:

1. Type **onpladm create job** and then the name of the job.
2. Optionally, type **-p** and then the name of the project.
3. Type the following required elements:
   - **-n**
   - **-d** and the name of the device
   - **-D** and the name of the database
   - **-t** and the name of the table
4. Optionally, you can choose one or more of the following elements and repeat them an arbitrary number of times:
   - **-S** and the server name
   - **-T** and the target server name
   - The run mode. To set the run mode, follow the Setting the Run Mode segment diagram to type **-f**, optionally type **d**, **p**, or **a**, and then optionally type **l** or **u**.
5. Follow the diagram to the terminator.

Keywords and punctuation

Keywords are words reserved for statements and all commands except system-level commands.

When a keyword appears in a syntax diagram, it is shown in uppercase letters. When you use a keyword in a command, you can write it in uppercase or lowercase letters, but you must spell the keyword exactly as it appears in the syntax diagram.

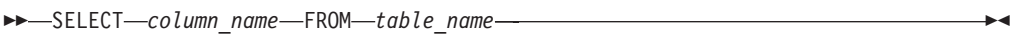
You must also use any punctuation in your statements and commands exactly as shown in the syntax diagrams.

Identifiers and names

Variables serve as placeholders for identifiers and names in the syntax diagrams and examples.

You can replace a variable with an arbitrary name, identifier, or literal, depending on the context. Variables are also used to represent complex syntax elements that are expanded in additional syntax diagrams. When a variable appears in a syntax diagram, an example, or text, it is shown in *lowercase italic*.

The following syntax diagram uses variables to illustrate the general form of a simple SELECT statement.



When you write a SELECT statement of this form, you replace the variables *column_name* and *table_name* with the name of a specific column and table.

How to provide documentation feedback

You are encouraged to send your comments about IBM Informix user documentation.

Use one of the following methods:

- Send email to docinf@us.ibm.com.
- In the Informix information center, which is available online at http://www.ibm.com/software/data/sw-library/, open the topic that you want to comment on. Click the feedback link at the bottom of the page, fill out the form, and submit your feedback.
- Add comments to topics directly in the information center and read comments that were added by other users. Share information about the product documentation, participate in discussions with other users, rate topics, and more!

Feedback from all methods is monitored by the team that maintains the user documentation. The feedback methods are reserved for reporting errors and omissions in the documentation. For immediate help with a technical problem, contact IBM Technical Support at http://www.ibm.com/planetwide/.

We appreciate your suggestions.

Chapter 1. Java application development for IBM data servers

The DB2® and IBM Informix database systems provide driver support for client applications and applets that are written in Java.

You can access data in DB2 and IBM Informix database systems using JDBC, SQL, or pureQuery.

JDBC

JDBC is an application programming interface (API) that Java applications use to access relational databases. IBM data server support for JDBC lets you write Java applications that access local DB2 or IBM Informix data or remote relational data on a server that supports DRDA.

SQLJ

SQLJ provides support for embedded static SQL in Java applications. SQLJ was initially developed by IBM, Oracle, and Tandem to complement the dynamic SQL JDBC model with a static SQL model.

For connections to DB2, in general, Java applications use JDBC for dynamic SQL and SQLJ for static SQL.

For connections to IBM Informix, SQL statements in JDBC or SQLJ applications run dynamically.

Because SQLJ can inter-operate with JDBC, an application program can use JDBC and SQLJ within the same unit of work.

pureQuery

pureQuery is a high-performance data access platform that makes it easier to develop, optimize, secure, and manage data access. It consists of:

- Application programming interfaces that are built for ease of use and for simplifying the use of best practices
- Development tools, which are delivered in IBM Optim™ Development Studio, for Java and SQL development
- A runtime, which is delivered in IBM Optim pureQuery Runtime, for optimizing and securing database access and simplifying management tasks

With pureQuery, you can write Java applications that treat relational data as objects, whether that data is in databases or JDBC DataSource objects. Your applications can also treat objects that are stored in in-memory Java collections as though those objects are relational data. To query or update your relational data or Java objects, you use SQL.

For more information on pureQuery, see the Integrated Data Management Information Center.

Chapter 2. Supported drivers for JDBC and SQLJ

The IDS product includes support for one type of JDBC driver architecture.

According to the JDBC specification, there are four types of JDBC driver architectures:

Type 1
Drivers that implement the JDBC API as a mapping to another data access API, such as Open Database Connectivity (ODBC). Drivers of this type are generally dependent on a native library, which limits their portability. The IDS database system does not provide a type 1 driver.

Type 2
Drivers that are written partly in the Java programming language and partly in native code. The drivers use a native client library specific to the data source to which they connect. Because of the native code, their portability is limited.

Type 3
Drivers that use a pure Java client and communicate with a data server using a data-server-independent protocol. The data server then communicates the client's requests to the data source. The IDS database system does not provide a type 3 driver.

Type 4
Drivers that are pure Java and implement the network protocol for a specific data source. The client connects directly to the data source.

IBM Informix supports the following drivers:

Driver name	Driver type
IBM Informix JDBC Driver	Type 4
IBM Data Server Driver for JDBC and SQLJ	Type 2[1] and Type 4

Note:

1. Although the IBM Data Server Driver for JDBC and SQLJ supports type 2 and type 4 connections, only type 4 connections can be used to connect to IBM Informix servers.

IBM Data Server Driver for JDBC and SQLJ (type 4)

The IBM Data Server Driver for JDBC and SQLJ is a single driver that includes JDBC type 2 and JDBC type 4 behavior. For connections to IBM Informix data servers, only type 4 behavior is supported. IBM Data Server Driver for JDBC and SQLJ type 4 driver behavior is referred to as *IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.*

Two versions of the IBM Data Server Driver for JDBC and SQLJ are available. IBM Data Server Driver for JDBC and SQLJ version 3.50 is JDBC 3.0-compliant. IBM Data Server Driver for JDBC and SQLJ version 4.0 is JDBC 3.0-compliant and supports some JDBC 4.0 functions.

The IBM Data Server Driver for JDBC and SQLJ supports these JDBC functions:
- All of the methods that are described in the JDBC 3.0 specifications. See "Driver support for JDBC APIs".

- SQLJ application programming interfaces, as defined by the SQLJ standards, for simplified data access from Java applications.
- Some methods that are described in the JDBC 4.0 specifications, if you install IBM Data Server Driver for JDBC and SQLJ version 4.0.
- Connections that are enabled for connection pooling. WebSphere® Application Server or another application server does the connection pooling.
- Support for distributed transaction management. This support implements the Java 2 Platform, Enterprise Edition (J2EE) Java Transaction Service (JTS) and Java Transaction API (JTA) specifications, which conform to the X/Open standard for distributed transactions (*Distributed Transaction Processing: The XA Specification*, available from http://www.opengroup.org).

IBM Informix JDBC Driver (type 4)

IBM Informix JDBC Driver is a native-protocol, pure-Java driver.

For more information about the IBM Informix JDBC Driver, refer to the *IBM Informix JDBC Driver Programmer's Guide*.

Chapter 3. IBM Data Server Driver for JDBC and SQLJ restrictions for IBM Informix

Before you install the IBM Data Server Driver for JDBC and SQLJ and use it with IBM Informix databases, make sure your environment conforms to these restrictions.

- Connections are DRDA-based; the Informix proprietary protocol is not supported.
- The IBM Data Server Driver for JDBC and SQLJ requires IBM Informix, Version 11.10, or later. However, to use features introduced with IBM Data Server Driver for JDBC and SQLJ, Version 3.51 and later, you must have IBM Informix, Version 11.50. New features are described in "What's New for IBM Data Server Driver for JDBC and SQLJ for IBM Informix" on page ix
- Type 1, 2, and 3 connections are not supported. Only type 4 connections are supported.
- Certain Informix data types are not supported. For example, INTERVAL, opaque data types, user-defined data types, and collection data types.

The IBM Data Server Driver for JDBC and SQLJ differs from the IBM Informix JDBC driver. For more information, see "Differences between the IBM Data Server Driver for JDBC and SQLJ and the IBM Informix JDBC Driver" on page 14-199.

Chapter 4. Installing the IBM Data Server Driver for JDBC and SQLJ

After you install the IBM Data Server Driver for JDBC and SQLJ, you can compile and run JDBC applications.

SDK Requirement: Before you install the IBM Data Server Driver for JDBC and SQLJ, you must have an SDK for Java installed on your computer. For JDBC 3.0 functions, you need Java SDK 1.4.2 or later. If you want to use JDBC 4.0 functions, you need an SDK for Java, 6 or later.

Follow these steps to install the IBM Data Server Driver for JDBC and SQLJ:

1. Download the zip file for the latest version of the IBM Data Server Driver for JDBC and SQLJ.
   a. Go to http://www.ibm.com/software/data/support/data-server-clients/download.html.
   b. Under Downloads and fixes, select View IBM Data Server Client Packages...
   c. In the Refine my fix list window, select Show me more options.
   d. On the Fix Central page, select Information Management in the Product Group field, IBM Data Server Client Packages in the Product field, the latest version in the Installed Version field, and All in the Platform field.
   e. On the Identify fixes page, type "Data Server Driver for JDBC" in the Text field.
   f. On the Select fixes page, select the latest version of the IBM Data Server Driver for JDBC and SQLJ.
   g. On the Download options page, select the options that are appropriate for you.
2. Extract the zip file into an empty directory.

   The zip file contains the following files:
   - **db2jcc.jar**
   - **db2jcc4.jar**
   - **sqlj.zip**
   - **sqlj4.zip**
3. Modify the CLASSPATH environment variable to include the appropriate files:

   For JDBC

   Include **db2jcc.jar** in the CLASSPATH if you plan to use the version of the IBM Data Server Driver for JDBC and SQLJ that includes only **JDBC 3.0 and earlier functions**.

   Include **db2jcc4.jar** in the CLASSPATH if you plan to use the version of the IBM Data Server Driver for JDBC and SQLJ that includes **JDBC 4.0 and earlier functions**.

   Important: Include **db2jcc.jar** or **db2jcc4.jar** in the CLASSPATH. Do not include both files.

   For SQLJ

   Include **sqlj.zip** in the CLASSPATH if you plan to prepare SQLJ applications that include only **JDBC 3.0 and earlier functions**.

Include **sqlj4.zip** in the CLASSPATH if you plan to prepare SQLJ applications that include **JDBC 4.0 and earlier functions**.

Important: Include **sqlj.zip** or **sqlj4.zip** in the CLASSPATH. Do not include both files.

- On Windows, to set the CLASSPATH for a session for the **db2jcc4.jar** file, from the command prompt enter:

  ```
  java -classpath <dir>\db2jcc4.jar
  ```

  To set the CLASSPATH for a session for the **db2jcc4.jar** file and for the **sqlj4.zip** file, separate the directory and filename combinations with a semicolon (;). For example:

  ```
  java -classpath <dir>\db2jcc4.jar;<dir>\sqlj4.zip
  ```

  Where <*dir*> is the location of the **db2jcc.jar** file.

  To permanently set the CLASSPATH environment variable use the System utility in the Control Panel.
- On UNIX, to set the CLASSPATH for a session for the **db2jcc4.jar** file, from the command prompt enter:

  ```
  java -classpath <dir>/db2jcc4.jar
  ```

  To set the CLASSPATH for a session for the **db2jcc4.jar** file and for the **sqlj4.zip** file, separate the directory and filename combinations with a colon (:). For example:

  ```
  java -classpath <dir>/db2jcc4.jar:<dir>/sqlj4.zip
  ```

  Where <*dir*> is the location of the **db2jcc4.jar** file.

  To permanently set the CLASSPATH environment variable, ask your UNIX System Administrator to update your profile.

4. Configure a new server alias in the SQLHOSTS file or Windows registry that uses either the **drtlitcp** or the **drsoctcp** connection protocol. For more information, see the topic about Configuring Informix for Connections to IBM Data Server Clients in the *IBM Informix Administrator's Guide*.
5. Customize the driver-wide configuration properties, if any of the default settings are inappropriate. For details, see the following topics:
   - "Customization of IBM Data Server Driver for JDBC and SQLJ configuration properties"
   - "IBM Data Server Driver for JDBC and SQLJ configuration properties" on page 14-52

Customization of IBM Data Server Driver for JDBC and SQLJ configuration properties

The IBM Data Server Driver for JDBC and SQLJ configuration properties let you set property values that have driver-wide scope. Those settings apply across applications and DataSource instances. You can change the settings without having to change application source code or DataSource characteristics.

Each IBM Data Server Driver for JDBC and SQLJ configuration property setting is of this form:

```
property=value
```

You can set configuration properties in the following ways:

- Set the configuration properties as Java system properties. Configuration property values that are set as Java system properties override configuration property values that are set in any other ways.

  For stand-alone Java applications, you can set the configuration properties as Java system properties by specifying -D*property=value* for each configuration property when you execute the java command.
- Set the configuration properties in a resource whose name you specify in the db2.jcc.propertiesFile Java system property. For example, you can specify an absolute path name for the db2.jcc.propertiesFile value.

  For stand-alone Java applications, you can set the configuration properties by specifying the -Ddb2.jcc.propertiesFile=*path* option when you execute the java command.
- Set the configuration properties in a resource named DB2JccConfiguration.properties. A standard Java resource search is used to find DB2JccConfiguration.properties. The IBM Data Server Driver for JDBC and SQLJ searches for this resource only if you have not set the db2.jcc.propertiesFile Java system property.

  DB2JccConfiguration.properties can be a stand-alone file, or it can be included in a JAR file.

  If DB2JccConfiguration.properties is in a JAR file, the JAR file must be in the CLASSPATH concatenation.

Chapter 5. JDBC application programming

Writing a JDBC application has much in common with writing an SQL application in any other language.

In general, you need to do the following things:
- Access the Java packages that contain JDBC methods.
- Declare variables for sending data to or retrieving data from IDS tables.
- Connect to a data source.
- Execute SQL statements.
- Handle SQL errors and warnings.
- Disconnect from the data source.

Although the tasks that you need to perform are similar to those in other languages, the way that you execute those tasks is somewhat different.

Example of a simple JDBC application

A simple JDBC application demonstrates the basic elements that JDBC applications need to include.

Figure 5-1. Simple JDBC application

```
import java.sql.*;                                                    1

public class EzJava
{
  public static void main(String[] args)
  {
    String urlPrefix = "jdbc:ids:";
    String url;
    String user;
    String password;
    String empNo;                                                     2
    Connection con;
    Statement stmt;
    ResultSet rs;

    System.out.println ("**** Enter class EzJava");

    // Check the that first argument has the correct form for the portion
    // of the URL that follows jdbc:ids:,
    // as described
    // in the Connecting to a data source using the DriverManager
    // interface with the IBM Data Server Driver for JDBC and SQLJ topic.

    // For example, for IBM Data Server Driver for
    // JDBC and SQLJ type 4 connectivity, args[0] might
    // be //myhost:9999/idsdb.
    if (args.length!=3)
    {
      System.err.println ("Invalid value. First argument appended to "+
       "jdbc:ids: must specify a valid URL.");
      System.err.println ("Second argument must be a valid user ID.");
      System.err.println ("Third argument must be the password for the user ID.");
      System.exit(1);
    }
```

```
url = urlPrefix + args[0];
user = args[1];
password = args[2];
try
{
  // Load the driver
  Class.forName("com.ibm.db2.jcc.DB2Driver");                              3a
  System.out.println("**** Loaded the JDBC driver");

  // Create the connection using the IBM Data Server Driver for JDBC and SQLJ
  con = DriverManager.getConnection (url, user, password);                 3b
  // Commit changes manually
  con.setAutoCommit(false);
  System.out.println("**** Created a JDBC connection to the data source");

  // Create the Statement
  stmt = con.createStatement();                                            4a
  System.out.println("**** Created JDBC Statement object");

  // Execute a query and generate a ResultSet instance
  rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE");                    4b
  System.out.println("**** Created JDBC ResultSet object");

  // Print all of the employee numbers to standard output device
  while (rs.next()) {
    empNo = rs.getString(1);
    System.out.println("Employee number = " + empNo);
  }
  System.out.println("**** Fetched all rows from JDBC ResultSet");
  // Close the ResultSet
  rs.close();
  System.out.println("**** Closed JDBC ResultSet");

  // Close the Statement
  stmt.close();
  System.out.println("**** Closed JDBC Statement");

  // Connection must be on a unit-of-work boundary to allow close
  con.commit();
  System.out.println ( "**** Transaction committed" );

  // Close the connection
  con.close();                                                             6
  System.out.println("**** Disconnected from data source");

  System.out.println("**** JDBC Exit from class EzJava - no errors");

}

catch (ClassNotFoundException e)
{
  System.err.println("Could not load JDBC driver");
  System.out.println("Exception: " + e);
  e.printStackTrace();
}

catch(SQLException ex)                                                     5
{
  System.err.println("SQLException information");
  while(ex!=null) {
    System.err.println ("Error msg: " + ex.getMessage());
    System.err.println ("SQLSTATE: " + ex.getSQLState());
    System.err.println ("Error code: " + ex.getErrorCode());
    ex.printStackTrace();
    ex = ex.getNextException(); // For drivers that support chained exceptions
```

```
      }
    }
  }  // End main
}    // End EzJava
```

Notes to Figure 5-1 on page 5-1:

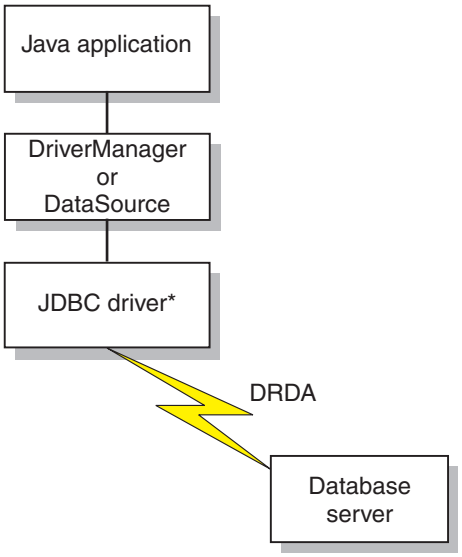
Note	Description
1	This statement imports the java.sql package, which contains the JDBC core API. For information on other Java packages that you might need to access, see "Java packages for JDBC support".
2	`String` variable empNo performs the function of a host variable. That is, it is used to hold data retrieved from an SQL query. See "Variables in JDBC applications" for more information.
3a and 3b	These two sets of statements demonstrate how to connect to a data source using one of two available interfaces. See "How JDBC applications connect to a data source" for more details. Step 3a (loading the JDBC driver) is not necessary if you use JDBC 4.0.
4a and 4b	These two sets of statements demonstrate how to perform a SELECT in JDBC. For information on how to perform other SQL operations, see "JDBC interfaces for executing SQL".
5	This `try/catch` block demonstrates the use of the SQLException class for SQL error handling. For more information on handling SQL errors, see "Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ". For information on handling SQL warnings, see "Handling an SQLWarning under the IBM Data Server Driver for JDBC and SQLJ".
6	This statement disconnects the application from the data source. See "Disconnecting from data sources in JDBC applications".

How JDBC applications connect to a data source

Before you can execute SQL statements in any SQL program, you must be connected to a data source.

The IBM Data Server Driver for JDBC and SQLJ supports type 2 and type 4 connectivity. Connections to DB2 databases can use type 2 or type 4 connectivity. Connections to IBM Informix databases can use type 4 connectivity.

The following figure shows how a Java application connects to a data source using IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.



*Java byte code executed under JVM

Figure 5-2. Java application flow for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity

Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ

A JDBC application can establish a connection to a data source using the JDBC DriverManager interface, which is part of the java.sql package.

The steps for establishing a connection are:

1. Load the JDBC driver by invoking the Class.forName method.

   If you are using JDBC 4.0, you do not need to explicitly load the JDBC driver.

   For the IBM Data Server Driver for JDBC and SQLJ, you load the driver by invoking the Class.forName method with the following argument:

   ```
   com.ibm.db2.jcc.DB2Driver
   ```

   The following code demonstrates loading the IBM Data Server Driver for JDBC and SQLJ:

   ```
   try {
     // Load the IBM Data Server Driver for JDBC and SQLJ with DriverManager
     Class.forName("com.ibm.db2.jcc.DB2Driver");
   } catch (ClassNotFoundException e) {
        e.printStackTrace();
   }
   ```

   The `catch` block is used to print an error if the driver is not found.

2. Connect to a data source by invoking the DriverManager.getConnection method.

   You can use one of the following forms of getConnection:

   ```
   getConnection(String url);
   getConnection(String url, user, password);
   getConnection(String url, java.util.Properties info);
   ```

For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, the getConnection method must specify a user ID and password, through parameters or through property values.

The *url* argument represents a data source, and indicates what type of JDBC connectivity you are using.

The *info* argument is an object of type java.util.Properties that contains a set of driver properties for the connection. Specifying the *info* argument is an alternative to specifying *property=value;* strings in the URL. See "Properties for the IBM Data Server Driver for JDBC and SQLJ" for the properties that you can specify.

There are several ways to specify a user ID and password for a connection:

- Use the form of the getConnection method that specifies *url* with `property=value`; clauses, and include the user and password properties in the URL.
- Use the form of the getConnection method that specifies *user* and *password*.
- Use the form of the getConnection method that specifies *info*, after setting the user and password properties in a java.util.Properties object.

Example: Establishing a connection and setting the user ID and password in a URL:

```
String url = "jdbc:ids://myhost:5021/mydb:" +
  "user=dbadm;password=dbadm;";

                                        // Set URL for data source
Connection con = DriverManager.getConnection(url);
                                        // Create connection
```

Example: Establishing a connection and setting the user ID and password in user and password parameters:

```
String url = "jdbc:ids://myhost:5021/mydb";
                                        // Set URL for data source
String user = "dbadm";
String password = "dbadm";
Connection con = DriverManager.getConnection(url, user, password);
                                        // Create connection
```

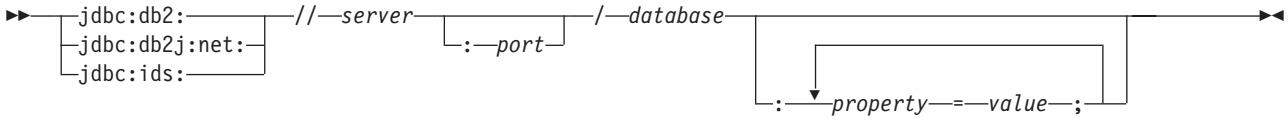
Example: Establishing a connection and setting the user ID and password in a java.util.Properties object:

```
Properties properties = new Properties(); // Create Properties object
properties.put("user", "dbadm");         // Set user ID for connection
properties.put("password", "dbadm");     // Set password for connection
String url = "jdbc:ids://myhost:5021/mydb";
                                        // Set URL for data source
Connection con = DriverManager.getConnection(url, properties);
                                        // Create connection
```

URL format for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity

If you are using type 4 connectivity in your JDBC application, and you are making a connection using the DriverManager interface, you need to specify a URL in the DriverManager.getConnection call that indicates type 4 connectivity.

IBM Data Server Driver for JDBC and SQLJ type 4 connectivity URL syntax



IBM Data Server Driver for JDBC and SQLJ type 4 connectivity URL option descriptions

The parts of the URL have the following meanings:

jdbc:db2: or jdbc:db2j:net:
The meanings of the initial portion of the URL are:

jdbc:db2:
Indicates that the connection is to a DB2 for z/OS®, DB2 Database for Linux, UNIX, and Windows.

jdbc:db2: can also be used for a connection to an IBM Informix database, for application portability.

jdbc:db2j:net:
Indicates that the connection is to a remote IBM Cloudscape server.

jdbc:ids:
Indicates that the connection is to an IBM Informix data source. jdbc:informix-sqli: also indicates that the connection is to an IBM Informix data source, but jdbc:ids: should be used.

server
The domain name or IP address of the data source.

port
The TCP/IP server port number that is assigned to the data source. This is an integer between 0 and 65535. You must specify a value for port.

database
A name for the data source.

- If the connection is to a DB2 for z/OS server, *database* is the DB2 location name that is defined during installation. All characters in the DB2 location name must be uppercase characters. The IBM Data Server Driver for JDBC and SQLJ does not convert lowercase characters in the database value to uppercase for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

  You can determine the location name by executing the following SQL statement on the server:

  ```
  SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
  ```

- If the connection is to a DB2 for z/OS server or a DB2 for i server, all characters in *database* must be uppercase characters.
- If the connection is to a DB2 Database for Linux, UNIX, and Windows server, *database* is the database name that is defined during installation.
- If the connection is to an IBM Informix server, *database* is the database name. The name is case-insensitive. The server converts the name to lowercase.

- If the connection is to an IBM Cloudscape server, the *database* is the fully-qualified name of the file that contains the database. This name must be enclosed in double quotation marks ("). For example:

  ```
  "c:/databases/testdb"
  ```

property=value;
A property and its value for the JDBC connection. You can specify one or more property and value pairs. Each property and value pair, including the last one, must end with a semicolon (;). Do not include spaces or other white space characters anywhere within the list of property and value strings.

Some properties with an int data type have predefined constant field values. You must resolve constant field values to their integer values before you can use those values in the *url* parameter. For example, you cannot use com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL in a *url* parameter. However, you can build a URL string that includes com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL, and assign the URL string to a String variable. Then you can use the String variable in the *url* parameter:

```
String url =
 "jdbc:ids://sysmvs1.stl.ibm.com:5021/STLEC1" +
 ":user=dbadm;password=dbadm;" +
 "traceLevel=" +
 (com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL) + ";";
 Connection con =
   java.sql.DriverManager.getConnection(url);
```

Connecting to a data source using the DataSource interface

If your applications need to be portable among data sources, you should use the DataSource interface.

Using DriverManager to connect to a data source reduces portability because the application must identify a specific JDBC driver class name and driver URL. The driver class name and driver URL are specific to a JDBC vendor, driver implementation, and data source.

When you connect to a data source using the DataSource interface, you use a DataSource object.

The simplest way to use a DataSource object is to create and use the object in the same application, as you do with the DriverManager interface. However, this method does not provide portability.

The best way to use a DataSource object is for your system administrator to create and manage it separately, using WebSphere Application Server or some other tool. The program that creates and manages a DataSource object also uses the Java Naming and Directory Interface (JNDI) to assign a logical name to the DataSource object. The JDBC application that uses the DataSource object can then refer to the object by its logical name, and does not need any information about the underlying data source. In addition, your system administrator can modify the data source attributes, and you do not need to change your application program.

To learn more about using WebSphere to deploy DataSource objects, go to this URL on the web:

```
http://www.ibm.com/software/webservers/appserv/
```

To learn about deploying DataSource objects yourself, see "Creating and deploying DataSource objects".

You can use the DataSource interface and the DriverManager interface in the same application, but for maximum portability, it is recommended that you use only the DataSource interface to obtain connections.

To obtain a connection using a DataSource object that the system administrator has already created and assigned a logical name to, follow these steps:

1. From your system administrator, obtain the logical name of the data source to which you need to connect.
2. Create a Context object to use in the next step. The Context interface is part of the Java Naming and Directory Interface (JNDI), not JDBC.
3. In your application program, use JNDI to get the DataSource object that is associated with the logical data source name.
4. Use the DataSource.getConnection method to obtain the connection.

   You can use one of the following forms of the getConnection method:

   ```
   getConnection();
   getConnection(String user, String password);
   ```

   Use the second form if you need to specify a user ID and password for the connection that are different from the ones that were specified when the DataSource was deployed.

Example of obtaining a connection using a DataSource object that was created by the system administrator: In this example, the logical name of the data source that you need to connect to is `jdbc/sampledb`. The numbers to the right of selected statements correspond to the previously-described steps.

```
import java.sql.*;
import javax.naming.*;
import javax.sql.*;
...
Context ctx=new InitialContext();                          2
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");     3
Connection con=ds.getConnection();                         4
```

Figure 5-3. Obtaining a connection using a DataSource object

Example of creating and using a DataSource object in the same application:

Figure 5-4. Creating and using a DataSource object in the same application

```
import java.sql.*;        // JDBC base
import javax.sql.*;       // Addtional methods for JDBC
import com.ibm.db2.jcc.*; // IBM Data Server Driver for JDBC and SQLJ   1
                          // interfaces
DB2SimpleDataSource dbds=new DB2SimpleDataSource();              2
dbds.setDatabaseName("dbloc1");                                  3
                          // Assign the location name
dbds.setDescription("Our Sample Database");
                          // Description for documentation
dbds.setUser("john");
                          // Assign the user ID
dbds.setPassword("dbadm");
                          // Assign the password
Connection con=dbds.getConnection();                             4
                          // Create a Connection object
```

Note	Description
1	Import the package that contains the implementation of the DataSource interface.

Note	Description
2	Creates a DB2SimpleDataSource object. DB2SimpleDataSource is one of the IBM Data Server Driver for JDBC and SQLJ implementations of the DataSource interface. See "Creating and deploying DataSource objects" for information on DB2's DataSource implementations.
3	The setDatabaseName, setDescription, setUser, and setPassword methods assign attributes to the DB2SimpleDataSource object. See "Properties for the IBM Data Server Driver for JDBC and SQLJ" for information about the attributes that you can set for a DB2SimpleDataSource object under the IBM Data Server Driver for JDBC and SQLJ.
4	Establishes a connection to the data source that DB2SimpleDataSource object dbds represents.

JDBC connection objects

When you connect to a data source by either connection method, you create a Connection object, which represents the connection to the data source.

You use this Connection object to do the following things:

- Create Statement, PreparedStatement, and CallableStatement objects for executing SQL statements. These are discussed in "Executing SQL statements in JDBC applications".
- Gather information about the data source to which you are connected. This process is discussed in "Learning about a data source using DatabaseMetaData methods".
- Commit or roll back transactions. You can commit transactions manually or automatically. These operations are discussed in "Commit or roll back a JDBC transaction".
- Close the connection to the data source. This operation is discussed in "Disconnecting from data sources in JDBC applications".

Creating and deploying DataSource objects

JDBC versions starting with version 2.0 provide the DataSource interface for connecting to a data source. Using the DataSource interface is the preferred way to connect to a data source.

Using the DataSource interface involves two parts:

- Creating and deploying DataSource objects. This is usually done by a system administrator, using a tool such as WebSphere Application Server.
- Using the DataSource objects to create a connection. This is done in the application program.

This topic contains information that you need if you create and deploy the DataSource objects yourself.

The IBM Data Server Driver for JDBC and SQLJ provides the following DataSource implementations:

- com.ibm.db2.jcc.DB2SimpleDataSource, which does not support connection pooling.
- com.ibm.db2.jcc.DB2ConnectionPoolDataSource, which supports connection pooling.
- com.ibm.db2.jcc.DB2XADataSource, which supports connection pooling and distributed transactions. The connection pooling is provided by WebSphere Application Server or another application server.

When you create and deploy a DataSource object, you need to perform these tasks:

1. Create an instance of the appropriate DataSource implementation.
2. Set the properties of the DataSource object.
3. Register the object with the Java Naming and Directory Interface (JNDI) naming service.

The following example shows how to perform these tasks.

```
import java.sql.*;        // JDBC base
import javax.naming.*;    // JNDI Naming Services
import javax.sql.*;       // Additional methods for JDBC
import com.ibm.db2.jcc.*; // IBM Data Server Driver for
                          // JDBC and SQLJ
                          // implementation of JDBC
                          // standard extension APIs

DB2SimpleDataSource dbds = new com.ibm.db2.jcc.DB2SimpleDataSource();   1

dbds.setDatabaseName("db2loc1");                                        2
dbds.setDescription("Our Sample Database");
dbds.setUser("john");
dbds.setPassword("mypw");
...
Context ctx=new InitialContext();                                       3
Ctx.bind("jdbc/sampledb",dbds);                                         4
```

Figure 5-5. Example of creating and deploying a DataSource object

Note	Description
1	Creates an instance of the DB2SimpleDataSource class.
2	This statement and the next three statements set values for properties of this DB2SimpleDataSource object.
3	Creates a context for use by JNDI.
4	Associates DBSimple2DataSource object dbds with the logical name jdbc/sampledb. An application that uses this object can refer to it by the name jdbc/sampledb.

Java packages for JDBC support

Before you can invoke JDBC methods, you need to be able to access all or parts of various Java packages that contain those methods.

You can do that either by importing the packages or specific classes, or by using the fully-qualified class names. You might need the following packages or classes for your JDBC program:

java.sql
Contains the core JDBC API.

javax.naming
Contains classes and interfaces for Java Naming and Directory Interface (JNDI), which is often used for implementing a DataSource.

javax.sql
Contains methods for producing server-side applications using Java

com.ibm.db2.jcc
Contains the implementation of JDBC for the IBM Data Server Driver for JDBC and SQLJ.

Learning about a data source using DatabaseMetaData methods

The DatabaseMetaData interface contains methods that retrieve information about a data source. These methods are useful when you write generic applications that can access various data sources.

In generic applications that can access various data sources, you need to test whether a data source can handle various database operations before you execute them. For example, you need to determine whether the driver at a data source is at the JDBC 3.0 level before you invoke JDBC 3.0 methods against that driver.

DatabaseMetaData methods provide the following types of information:
- Features that the data source supports, such as the ANSI SQL level
- Specific information about the JDBC driver, such as the driver level
- Limits, such as the maximum number of columns that an index can have
- Whether the data source supports data definition statements (CREATE, ALTER, DROP, GRANT, REVOKE)
- Lists of objects at the data source, such as tables, indexes, or procedures
- Whether the data source supports various JDBC functions, such as batch updates or scrollable ResultSets
- A list of scalar functions that the driver supports

For IBM Informix systems, you might also need to obtain the following information:
- Whether the database is ANSI compliant
- Whether the database supports logging

To obtain that information, you need to use IBM Data Server Driver for JDBC and SQLJ-only methods DB2DatabaseMetaData.isIDSDatabaseAnsiCompliant and DB2DatabaseMetaData.isIDSDatabaseLogging.

To invoke DatabaseMetaData methods, you need to perform these basic steps:

1. Create a DatabaseMetaData object by invoking the getMetaData method on the connection.
2. Invoke DatabaseMetaData methods to get information about the data source.
3. If the method returns a ResultSet:
   a. In a loop, position the cursor using the next method, and retrieve data from each column of the current row of the ResultSet object using getXXX methods.
   b. Invoke the close method to close the ResultSet object.

Example: The following code demonstrates how to use DatabaseMetaData methods to determine the driver version, to get a list of the stored procedures that are available at the data source, and to get a list of datetime functions that the driver supports. The numbers to the right of selected statements correspond to the previously-described steps.

Figure 5-6. Using DatabaseMetaData methods to get information about a data source

```
Connection con;
DatabaseMetaData dbmtadta;
ResultSet rs;
int mtadtaint;
String procSchema;
String procName;
String dtfnList;
```

```
...
dbmtadta = con.getMetaData();     // Create the DatabaseMetaData object  1
mtadtaint = dmtadta.getDriverVersion();                                  2
                                  // Check the driver version
System.out.println("Driver version: " + mtadtaint);
rs = dbmtadta.getProcedures(null, null, "%");
                                  // Get information for all procedures
while (rs.next()) {               // Position the cursor                 3a
 procSchema = rs.getString("PROCEDURE_SCHEM");
                                  // Get procedure schema
 procName = rs.getString("PROCEDURE_NAME");
                                  // Get procedure name
 System.out.println(procSchema + "." + procName);
                                  // Print the qualified procedure name
}
dtfnList = dbmtadta.getTimeDateFunctions();
                                  // Get list of supported datetime functions
System.out.println("Supported datetime functions:");
System.out.println(dtfnList);     // Print the list of datetime functions
rs.close();                       // Close the ResultSet                 3b
```

Example: The following code demonstrates how to use DB2DatabaseMetaData methods to determine whether an IBM Informix database is ANSI compliant and supports logging.

```
com.ibm.db2.jcc.DB2Connection db2c =
 (com.ibm.db2.jcc.DB2Connection) c;     // c is existing java.sql.Connection object
                                        // that needs to be cast to a DB2Connection
                                        // object so DB2DatabaseMetaData methods
                                        // can be used on it.
 com.ibm.db2.jcc.DB2DatabaseMetaData  dbmd =
  (com.ibm.db2.jcc.DB2DatabaseMetaData) db2c.getMetaData();
                                        // Retrieve the DB2DatabaseMetaData object.
 if (dbmd.isIDSDatabaseLogging ())      // Check whether the database supports
                                        // logging. If so, you can perform a
                                        // commit operation.
   c.createStatement.executeUpdate("commit");
 if (dbmd.isIDSDatabaseAnsiCompliant()) // Check whether the database is ANSI
                                        // compliant.
   System.out.println("Current Informix database is ANSI compliant...");
```

DatabaseMetaData methods for identifying the type of data source

You can use the DatabaseMetaData.getDatabaseProductName and DatabaseMetaData.getProductVersion methods to identify the type and level of the database manager to which you are connected, and the operating system on which the database manager is running.

DatabaseMetaData.getDatabaseProductName returns a string that identifies the database manager and the operating system. The string has one of the following formats:

database-product
database-product/operating-system

The following table shows examples of values that are returned by DatabaseMetaData.getDatabaseProductName.

Table 5-1. Examples of DatabaseMetaData.getDatabaseProductName values

getDatabaseProductName value	Database product
DB2	DB2 for z/OS

Table 5-1. Examples of DatabaseMetaData.getDatabaseProductName values (continued)

getDatabaseProductName value	Database product
DB2/LINUXX8664	DB2 Database for Linux, UNIX, and Windows on Linux on x86
IBM Informix/UNIX64	IBM Informix on UNIX

DatabaseMetaData.getDatabaseVersionName returns a string that contains the database product indicator and the version number, release number, and maintenance level of the data source.

The following table shows examples of values that are returned by DatabaseMetaData.getDatabaseProductVersion.

Table 5-2. Examples of DatabaseMetaData.getDatabaseProductVersion values

getDatabaseProductVersion value	Database product version
DSN09015	DB2 for z/OS Version 9.1 in new-function mode
SQL09010	DB2 Database for Linux, UNIX, and Windows Version 9.1
IFX11100	IBM Informix Version 11.10

Variables in JDBC applications

As in any other Java application, when you write JDBC applications, you declare variables. In Java applications, those variables are known as Java identifiers.

Some of those identifiers have the same function as host variables in other languages: they hold data that you pass to or retrieve from database tables. Identifier `empNo` in the following code holds data that you retrieve from the EMPNO table column, which has the CHAR data type.

```
String empNo;
// Execute a query and generate a ResultSet instance
rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE");
while (rs.next()) {
 String empNo = rs.getString(1);
 System.out.println("Employee number = " + empNo);
}
```

Your choice of Java data types can affect performance because IDS picks better access paths when the data types of your Java variables map closely to the IDS data types.

JDBC interfaces for executing SQL

You execute SQL statements in a traditional SQL program to update data in tables, retrieve data from the tables, or call stored procedures. To perform the same functions in a JDBC program, you invoke methods.

Those methods are defined in the following interfaces:

- The Statement interface supports all SQL statement execution. The following interfaces inherit methods from the Statement interface:

- The PreparedStatement interface supports any SQL statement containing input parameter markers. Parameter markers represent input variables. The PreparedStatement interface can also be used for SQL statements with no parameter markers.

  With the IBM Data Server Driver for JDBC and SQLJ, the PreparedStatement interface can be used to call stored procedures that have input parameters and no output parameters, and that return no result sets. However, the preferred interface is CallableStatement.
- The CallableStatement interface supports the invocation of a stored procedure.

  The CallableStatement interface can be used to call stored procedures with input parameters, output parameters, or input and output parameters, or no parameters. With the IBM Data Server Driver for JDBC and SQLJ, you can also use the Statement interface to call stored procedures, but those stored procedures must have no parameters.

- The ResultSet interface provides access to the results that a query generates. The ResultSet interface has the same purpose as the cursor that is used in SQL applications in other languages.

Creating and modifying database objects using the Statement.executeUpdate method

The Statement.executeUpdate is one of the JDBC methods that you can use to update tables and call stored procedures.

You can use the Statement.executeUpdate method to do the following things:

- Execute data definition statements, such as CREATE, ALTER, DROP, GRANT, REVOKE
- Execute INSERT, UPDATE, DELETE, and MERGE statements that do not contain parameter markers.
- With the IBM Data Server Driver for JDBC and SQLJ, execute the CALL statement to call stored procedures that have no parameters and that return no result sets.

To execute these SQL statements, you need to perform these steps:

1. Invoke the Connection.createStatement method to create a Statement object.
2. Invoke the Statement.executeUpdate method to perform the SQL operation.
3. Invoke the Statement.close method to close the Statement object.

Suppose that you want to execute this SQL statement:

```
UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'
```

The following code creates Statement object `stmt`, executes the UPDATE statement, and returns the number of rows that were updated in `numUpd`. The numbers to the right of selected statements correspond to the previously-described steps.

```
Connection con;
Statement stmt;
int numUpd;
...
stmt = con.createStatement();               // Create a Statement object  1
numUpd = stmt.executeUpdate(
  "UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'");             2
                                            // Perform the update
stmt.close();                               // Close Statement object     3
```

Figure 5-7. Using Statement.executeUpdate

Updating data in tables using the PreparedStatement.executeUpdate method

The Statement.executeUpdate method works if you update IDS tables with constant values. However, updates often need to involve passing values in variables to IDS tables. To do that, you use the PreparedStatement.executeUpdate method.

With the IBM Data Server Driver for JDBC and SQLJ, you can also use PreparedStatement.executeUpdate to call stored procedures that have input parameters and no output parameters, and that return no result sets.

For calls to stored procedures that are on IBM Informix data sources, the PreparedStatement object can be a CALL statement or an EXECUTE PROCEDURE statement.

When you execute an SQL statement many times, you can get better performance by creating the SQL statement as a PreparedStatement.

For example, the following UPDATE statement lets you update the employee table for only one phone number and one employee number:

```
UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'
```

Suppose that you want to generalize the operation to update the employee table for any set of phone numbers and employee numbers. You need to replace the constant phone number and employee number with variables:

```
UPDATE EMPLOYEE SET PHONENO=? WHERE EMPNO=?
```

Variables of this form are called parameter markers. To execute an SQL statement with parameter markers, you need to perform these steps:

1. Invoke the Connection.prepareStatement method to create a PreparedStatement object.
2. Invoke the PreparedStatement.setXXX methods to pass values to the input variables.

   This step assumes that you use standard parameter markers. Alternatively, if you use named parameter markers, you useIBM Data Server Driver for JDBC and SQLJ-only methods to pass values to the input parameters.
3. Invoke the PreparedStatement.executeUpdate method to update the table with the variable values.
4. Invoke the PreparedStatement.close method to close the PreparedStatement object when you have finished using that object.

The following code performs the previous steps to update the phone number to '4657' for the employee with employee number '000010'. The numbers to the right of selected statements correspond to the previously-described steps.

```
Connection con;
PreparedStatement pstmt;
int numUpd;
...
pstmt = con.prepareStatement(
  "UPDATE EMPLOYEE SET PHONENO=? WHERE EMPNO=?");
                                // Create a PreparedStatement object       1
pstmt.setString(1,"4657");      // Assign first value to first parameter   2
pstmt.setString(2,"000010");    // Assign first value to second parameter
numUpd = pstmt.executeUpdate(); // Perform first update                    3
pstmt.setString(1,"4658");      // Assign second value to first parameter
pstmt.setString(2,"000020");    // Assign second value to second parameter
numUpd = pstmt.executeUpdate(); // Perform second update
pstmt.close();                  // Close the PreparedStatement object      4
```

Figure 5-8. Using PreparedStatement.executeUpdate for an SQL statement with parameter markers

You can also use the PreparedStatement.executeUpdate method for statements that have no parameter markers. The steps for executing a PreparedStatement object with no parameter markers are similar to executing a PreparedStatement object with parameter markers, except you skip step 2 on page 5-15. The following example demonstrates these steps.

```
Connection con;
PreparedStatement pstmt;
int numUpd;
...
pstmt = con.prepareStatement(
  "UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'");
                                // Create a PreparedStatement object  1
numUpd = pstmt.executeUpdate(); // Perform the update                  3
pstmt.close();                  // Close the PreparedStatement object 4
```

Figure 5-9. Using PreparedStatement.executeUpdate for an SQL statement without parameter markers

Making batch updates in JDBC applications

With batch updates, instead of updating rows of a table one at a time, you can direct JDBC to execute a group of updates at the same time. Statements that can be included in the same batch of updates are known as *batchable* statements.

If a statement has input parameters or host expressions, you can include that statement only in a batch that has other instances of the same statement. This type of batch is known as a *homogeneous batch*. If a statement has no input parameters, you can include that statement in a batch only if the other statements in the batch have no input parameters or host expressions. This type of batch is known as a *heterogeneous batch*. Two statements that can be included in the same batch are known as *batch compatible*.

Use the following Statement methods for creating, executing, and removing a batch of SQL updates:

- addBatch
- executeBatch
- clearBatch

Use the following PreparedStatement and CallableStatement method for creating a batch of parameters so that a single statement can be executed multiple times in a batch, with a different set of parameters for each execution.
- addBatch

Restrictions on executing statements in a batch:
- If you try to execute a SELECT statement in a batch, a BatchUpdateException is thrown.
- A CallableStatement object that you execute in a batch can contain output parameters. However, you cannot retrieve the values of the output parameters. If you try to do so, a BatchUpdateException is thrown.
- You cannot retrieve ResultSet objects from a CallableStatement object that you execute in a batch. A BatchUpdateException is not thrown, but the getResultSet method invocation returns a null value.

To make batch updates using several statements with no input parameters, follow these basic steps:
1. For each SQL statement that you want to execute in the batch, invoke the addBatch method.
2. Invoke the executeBatch method to execute the batch of statements.
3. Check for errors. If no errors occurred:
   a. Get the number of rows that were affect by each SQL statement from the array that the executeBatch invocation returns. This number does not include rows that were affected by triggers or by referential integrity enforcement.
   b. If `AutoCommit` is disabled for the Connection object, invoke the commit method to commit the changes.

      If `AutoCommit` is enabled for the Connection object, the IBM Data Server Driver for JDBC and SQLJ adds a commit method at the end of the batch.

To make batch updates using a single statement with several sets of input parameters, follow these basic steps:
1. Invoke the prepareStatement method to create a PreparedStatement object.
2. For each set of input parameter values:
   a. Execute setXXX methods to assign values to the input parameters.
   b. Invoke the addBatch method to add the set of input parameters to the batch.
3. Invoke the executeBatch method to execute the statements with all sets of parameters.
4. If no errors occurred:
   a. Get the number of rows that were updated by each execution of the SQL statement from the array that the executeBatch invocation returns. The number of affected rows does not include rows that were affected by triggers or by referential integrity enforcement.

      If the following conditions are true, the IBM Data Server Driver for JDBC and SQLJ returns `Statement.SUCCESS_NO_INFO` (-2), instead of the number of rows that were affected by each SQL statement:
      - The application is connected to a subsystem that is in DB2 for z/OS Version 8 new-function mode, or later.
      - The application is using Version 3.1 or later of the IBM Data Server Driver for JDBC and SQLJ.

- The IBM Data Server Driver for JDBC and SQLJ uses multi-row INSERT operations to execute batch updates.

  This occurs because with multi-row INSERT, the database server executes the entire batch as a single operation, so it does not return results for individual SQL statements.

b. If `AutoCommit` is disabled for the Connection object, invoke the commit method to commit the changes.

   If `AutoCommit` is enabled for the Connection object, the IBM Data Server Driver for JDBC and SQLJ adds a commit method at the end of the batch.

c. If the PreparedStatement object returns automatically generated keys, call DB2PreparedStatement.getDBGeneratedKeys to retrieve an array of ResultSet objects that contains the automatically generated keys.

   Check the length of the returned array. If the length of the returned array is 0, an error occurred during retrieval of the automatically generated keys.

5. If errors occurred, process the BatchUpdateException.

In the following code fragment, two sets of parameters are batched. An UPDATE statement that takes two input parameters is then executed twice, once with each set of parameters. The numbers to the right of selected statements correspond to the previously-described steps.

```
try {
...
  PreparedStatement preps = conn.prepareStatement(
    "UPDATE DEPT SET MGRNO=? WHERE DEPTNO=?");          1
  ps.setString(1,mgrnum1);                              2a
  ps.setString(2,deptnum1);
  ps.addBatch();                                        2b

  ps.setString(1,mgrnum2);
  ps.setString(2,deptnum2);
  ps.addBatch();
  int [] numUpdates=ps.executeBatch();                  3
  for (int i=0; i < numUpdates.length; i++) {           4a
    if (numUpdates[i] == SUCCESS_NO_INFO)
      System.out.println("Execution " + i +
        ": unknown number of rows updated");
    else
      System.out.println("Execution " + i +
        "successful: " numUpdates[i] + " rows updated");
  }
  conn.commit();                                        4b
} catch(BatchUpdateException b) {                       5
  // process BatchUpdateException
}
```

In the following code fragment, a batched INSERT statement returns automatically generated keys.

```
import java.sql.*;
import com.ibm.db2.jcc.*;
...
Connection conn;
...
try {
...
  PreparedStatement ps = conn.prepareStatement(         1
    "INSERT INTO DEPT (DEPTNO, DEPTNAME, ADMRDEPT) " +
    "VALUES (?,?,?)",
    Statement.RETURN_GENERATED_KEYS);
  ps.setString(1,"X01");                                2a
  ps.setString(2,"Finance");
```

```
  ps.setString(3,"A00");
  ps.addBatch();                                              2b
  ps.setString(1,"Y01");
  ps.setString(2,"Accounting");
  ps.setString(3,"A00");
  ps.addBatch();

  int [] numUpdates=preps.executeBatch();                     3

  for (int i=0; i < numUpdates.length; i++) {                 4a
    if (numUpdates[i] == SUCCESS_NO_INFO)
      System.out.println("Execution " + i +
        ": unknown number of rows updated");
    else
      System.out.println("Execution " + i +
        "successful: " numUpdates[i] + " rows updated");
  }
  conn.commit();                                              4b
  ResultSet[] resultList =
    ((DB2PreparedStatement)ps).getDBGeneratedKeys();          4c
  if (resultList.length != 0) {
    for (i = 0; i < resultList.length; i++) {
        while (resultList[i].next()) {
          java.math.BigDecimal idColVar = rs.getBigDecimal(1);
                              // Get automatically generated key
                              // value
          System.out.println("Automatically generated key value = "
            + idColVar);
        }
    }
  }
  else {
    System.out.println("Error retrieving automatically generated keys");
  }
} catch(BatchUpdateException b) {                             5
  // process BatchUpdateException
}
```

Learning about parameters in a PreparedStatement using ParameterMetaData methods

The IBM Data Server Driver for JDBC and SQLJ includes support for the ParameterMetaData interface. The ParameterMetaData interface contains methods that retrieve information about the parameter markers in a PreparedStatement object.

ParameterMetaData methods provide the following types of information:

- The data types of parameters, including the precision and scale of decimal parameters.
- The parameters' database-specific type names. For parameters that correspond to table columns that are defined with distinct types, these names are the distinct type names.
- Whether parameters are nullable.
- Whether parameters are input or output parameters.
- Whether the values of a numeric parameter can be signed.
- The fully-qualified Java class name that PreparedStatement.setObject uses when it sets a parameter value.

To invoke ParameterMetaData methods, you need to perform these basic steps:

1. Invoke the Connection.prepareStatement method to create a PreparedStatement object.

2. Invoke the PreparedStatement.getParameterMetaData method to retrieve a ParameterMetaData object.
3. Invoke ParameterMetaData.getParameterCount to determine the number of parameters in the PreparedStatement.
4. Invoke ParameterMetaData methods on individual parameters.

The following code demonstrates how to use ParameterMetaData methods to determine the number and data types of parameters in an SQL UPDATE statement. The numbers to the right of selected statements correspond to the previously-described steps.

```
Connection con;
ParameterMetaData pmtadta;
int mtadtacnt;
String sqlType;
...
pstmt = con.prepareStatement(
  "UPDATE EMPLOYEE SET PHONENO=? WHERE EMPNO=?");
                                  // Create a PreparedStatement object  1
pmtadta = pstmt.getParameterMetaData();                                 2
                                  // Create a ParameterMetaData object
mtadtacnt = pmtadta.getParameterCount();                                3
                                  // Determine the number of parameters
System.out.println("Number of statement parameters: " + mtadtacnt);
for (int i = 1; i <= mtadtacnt; i++) {
  sqlType = pmtadta.getParameterTypeName(i);                            4
                                  // Get SQL type for each parameter
 System.out.println("SQL type of parameter " + i " is " + sqlType);
}
...
pstmt.close();                       // Close the PreparedStatement
```

Figure 5-10. Using ParameterMetaData methods to get information about a PreparedStatement

Data retrieval in JDBC applications

In JDBC applications, you retrieve data using ResultSet objects. A ResultSet represents the result set of a query.

Retrieving data from tables using the Statement.executeQuery method

To retrieve data from a table using a SELECT statement with no parameter markers, you can use the Statement.executeQuery method.

This method returns a result table in a ResultSet object. After you obtain the result table, you need to use ResultSet methods to move through the result table and obtain the individual column values from each row.

With the IBM Data Server Driver for JDBC and SQLJ, you can also use the Statement.executeQuery method to retrieve a result set from a stored procedure call, if that stored procedure returns only one result set. If the stored procedure returns multiple result sets, you need to use the Statement.execute method.

This topic discusses the simplest kind of ResultSet, which is a read-only ResultSet in which you can only move forward, one row at a time. The IBM Data Server Driver for JDBC and SQLJ also supports updatable and scrollable ResultSets.

To retrieve rows from a table using a SELECT statement with no parameter markers, you need to perform these steps:

1. Invoke the Connection.createStatement method to create a Statement object.
2. Invoke the Statement.executeQuery method to obtain the result table from the SELECT statement in a ResultSet object.
3. In a loop, position the cursor using the next method, and retrieve data from each column of the current row of the ResultSet object using getXXX methods. XXX represents a data type.
4. Invoke the ResultSet.close method to close the ResultSet object.
5. Invoke the Statement.close method to close the Statement object when you have finished using that object.

The following code demonstrates how to retrieve all rows from the employee table. The numbers to the right of selected statements correspond to the previously-described steps.

```
String empNo;
Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement();     // Create a Statement object          1
rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE");                   2
                                  // Get the result table from the query
while (rs.next()) {               // Position the cursor               3
 empNo = rs.getString(1);             // Retrieve only the first column value
 System.out.println("Employee number = " + empNo);
                                  // Print the column value
}
rs.close();                       // Close the ResultSet               4
stmt.close();                     // Close the Statement               5
```

Figure 5-11. Using Statement.executeQuery

Retrieving data from tables using the PreparedStatement.executeQuery method

To retrieve data from a table using a SELECT statement with parameter markers, you use the PreparedStatement.executeQuery method.

This method returns a result table in a ResultSet object. After you obtain the result table, you need to use ResultSet methods to move through the result table and obtain the individual column values from each row.

With the IBM Data Server Driver for JDBC and SQLJ, you can also use the PreparedStatement.executeQuery method to retrieve a result set from a stored procedure call, if that stored procedure returns only one result set and has only input parameters. If the stored procedure returns multiple result sets, you need to use the PreparedStatement.execute method.

You can also use the PreparedStatement.executeQuery method for statements that have no parameter markers. When you execute a query many times, you can get better performance by creating the SQL statement as a PreparedStatement.

To retrieve rows from a table using a SELECT statement with parameter markers, you need to perform these steps:

1. Invoke the Connection.prepareStatement method to create a PreparedStatement object.
2. Invoke PreparedStatement.setXXX methods to pass values to the input parameters.

3. Invoke the PreparedStatement.executeQuery method to obtain the result table from the SELECT statement in a ResultSet object.
4. In a loop, position the cursor using the ResultSet.next method, and retrieve data from each column of the current row of the ResultSet object using getXXX methods.
5. Invoke the ResultSet.close method to close the ResultSet object.
6. Invoke the PreparedStatement.close method to close the PreparedStatement object when you have finished using that object.

The following code demonstrates how to retrieve rows from the employee table for a specific employee. The numbers to the right of selected statements correspond to the previously-described steps.

```
String empnum, phonenum;
Connection con;
PreparedStatement pstmt;
ResultSet rs;
...
pstmt = con.prepareStatement(
  "SELECT EMPNO, PHONENO FROM EMPLOYEE WHERE EMPNO=?");
                                  // Create a PreparedStatement object    1
pstmt.setString(1,"000010");      // Assign value to input parameter      2

rs = pstmt.executeQuery();        // Get the result table from the query  3
while (rs.next()) {               // Position the cursor                  4
 empnum = rs.getString(1);        // Retrieve the first column value
 phonenum = rs.getString(2);      // Retrieve the first column value
 System.out.println("Employee number = " + empnum +
   "Phone number = " + phonenum);
                                  // Print the column values
}
rs.close();                       // Close the ResultSet                  5
pstmt.close();                    // Close the PreparedStatement          6
```

Figure 5-12. Example of using PreparedStatement.executeQuery

Learning about a ResultSet using ResultSetMetaData methods

You cannot always know the number of columns and data types of the columns in a table or result set. This is true especially when you are retrieving data from a remote data source.

When you write programs that retrieve unknown ResultSets, you need to use ResultSetMetaData methods to determine the characteristics of the ResultSets before you can retrieve data from them.

ResultSetMetaData methods provide the following types of information:
- The number of columns in a ResultSet
- The qualifier for the underlying table of the ResultSet
- Information about a column, such as the data type, length, precision, scale, and nullability
- Whether a column is read-only

After you invoke the executeQuery method to generate a ResultSet for a query on a table, follow these basic steps to determine the contents of the ResultSet:

1. Invoke the getMetaData method on the ResultSet object to create a ResultSetMetaData object.
2. Invoke the getColumnCount method to determine how many columns are in the ResultSet.

3. For each column in the ResultSet, execute ResultSetMetaData methods to determine column characteristics.

   The results of ResultSetMetaData.getColumnName call reflects the column name information that is stored in the IDS catalog for that data source.

The following code demonstrates how to determine the data types of all the columns in the employee table. The numbers to the right of selected statements correspond to the previously-described steps.

```
String s;
Connection con;
Statement stmt;
ResultSet rs;
ResultSetMetaData rsmtadta;
int colCount
int mtadtaint;
int i;
String colName;
String colType;
...
stmt = con.createStatement();     // Create a Statement object
rs = stmt.executeQuery("SELECT * FROM EMPLOYEE");
                                  // Get the ResultSet from the query
rsmtadta = rs.getMetaData();      // Create a ResultSetMetaData object  1
colCount = rsmtadta.getColumnCount();                                   2
                                  // Find number of columns in EMP
for (i=1; i<= colCount; i++) {                                          3
 colName = rsmtadta.getColumnName();    // Get column name
 colType = rsmtadta.getColumnTypeName();
                                        // Get column data type
 System.out.println("Column = " + colName +
  " is data type " + colType);
                                        // Print the column value
}
```

Figure 5-13. Using ResultSetMetaData methods to get information about a ResultSet

Characteristics of a JDBC ResultSet under the IBM Data Server Driver for JDBC and SQLJ

The IBM Data Server Driver for JDBC and SQLJ provides support for scrollable, updatable, and holdable cursors.

In addition to moving forward, one row at a time, through a ResultSet, you might want to do the following things:

- Move backward or go directly to a specific row
- Update, delete, or insert rows in a ResultSet
- Leave the ResultSet open after a COMMIT

The following terms describe characteristics of a ResultSet:

scrollability
Whether the cursor for the ResultSet can move forward only, or forward one or more rows, backward one or more rows, or to a specific row.

If a cursor for a ResultSet is scrollable, it also has a sensitivity attribute, which describes whether the cursor is sensitive to changes to the underlying table.

updatability
Whether the cursor can be used to update or delete rows. This characteristic does not apply to a ResultSet that is returned from a stored procedure, because a stored procedure ResultSet cannot be updated.

holdability
Whether the cursor stays open after a COMMIT.

You set the updatability, scrollability, and holdability characteristics of a ResultSet through parameters in the Connection.prepareStatement or `Connection.createStatement` methods. The ResultSet settings map to attributes of a cursor in the database. The following table lists the JDBC scrollability, updatability, and holdability settings, and the corresponding cursor attributes.

Table 5-3. JDBC ResultSet characteristics and SQL cursor attributes

JDBC setting	DB2 cursor setting	IBM Informix cursor setting
CONCUR_READ_ONLY	FOR READ ONLY	FOR READ ONLY
CONCUR_UPDATABLE	FOR UPDATE	FOR UPDATE
HOLD_CURSORS_OVER_COMMIT	WITH HOLD	WITH HOLD
TYPE_FORWARD_ONLY	SCROLL not specified	SCROLL not specified
TYPE_SCROLL_INSENSITIVE	INSENSITIVE SCROLL	SCROLL
TYPE_SCROLL_SENSITIVE	SENSITIVE STATIC, SENSITIVE DYNAMIC, or ASENSITIVE, depending on the cursorSensitvity Connection and DataSource property	Not supported

If a JDBC ResultSet is static, the size of the result table and the order of the rows in the result table do not change after the cursor is opened. This means that if you insert rows into the underlying table, the result table for a static ResultSet does not change. If you delete a row of a result table, a delete hole occurs. You cannot update or delete a delete hole.

Specifying updatability, scrollability, and holdability for ResultSets in JDBC applications:

You use special parameters in the Connection.prepareStatement or Connection.createStatement methods to specify the updatability, scrollability, and holdability of a ResultSet.

By default, ResultSet objects are not scrollable and not updatable. The default holdability depends on the data source, and can be determined from the DatabaseMetaData.getResultSetHoldability method. To change the scrollability, updatability, and holdability attributes for a ResultSet, follow these steps:

1. If the SELECT statement that defines the ResultSet has no input parameters, invoke the createStatement method to create a Statement object. Otherwise, invoke the prepareStatement method to create a PreparedStatement object. You need to specify forms of the createStatement or prepareStatement methods that include the *resultSetType*, *resultSetConcurrency*, or *resultSetHoldability* parameters.

   The form of the createStatement method that supports scrollability and updatability is:

   ```
   createStatement(int resultSetType, int resultSetConcurrency);
   ```

   The form of the createStatement method that supports scrollability, updatability, and holdability is:

   ```
   createStatement(int resultSetType, int resultSetConcurrency,
    int resultSetHoldability);
   ```

The form of the prepareStatement method that supports scrollability and updatability is:

```
prepareStatement(String sql, int resultSetType,
 int resultSetConcurrency);
```

The form of the prepareStatement method that supports scrollability, updatability, and holdability is:

```
prepareStatement(String sql, int resultSetType,
 int resultSetConcurrency, int resultSetHoldability);
```

The following table contains a list of valid values for *resultSetType* and *resultSetConcurrency*.

Table 5-4. Valid combinations of resultSetType and resultSetConcurrency for ResultSets

resultSetType **value**	*resultSetConcurrency* **value**
TYPE_FORWARD_ONLY	CONCUR_READ_ONLY
TYPE_FORWARD_ONLY	CONCUR_UPDATABLE
TYPE_SCROLL_INSENSITIVE	CONCUR_READ_ONLY
TYPE_SCROLL_SENSITIVE[1]	CONCUR_READ_ONLY
TYPE_SCROLL_SENSITIVE[1]	CONCUR_UPDATABLE

Note:

1. This value does not apply to connections to IBM Informix.

resultSetHoldability has two possible values: HOLD_CURSORS_OVER_COMMIT and CLOSE_CURSORS_AT_COMMIT. Either of these values can be specified with any valid combination of *resultSetConcurrency* and *resultSetHoldability*. The value that you set overrides the default holdability for the connection.

Restriction: If the ResultSet is scrollable, and the ResultSet is used to select columns from a table on a DB2 Database for Linux, UNIX, and Windows server, the SELECT list of the SELECT statement that defines the ResultSet cannot include columns with the following data types:

- LONG VARCHAR
- LONG VARGRAPHIC
- BLOB
- CLOB
- XML
- A distinct type that is based on any of the previous data types in this list
- A structured type

2. If the SELECT statement has input parameters, invoke setXXX methods to pass values to the input parameters.
3. Invoke the executeQuery method to obtain the result table from the SELECT statement in a ResultSet object.
4. For each row that you want to access:
   a. Position the cursor using one of the methods that are listed in the following table.

Table 5-5. ResultSet methods for positioning a scrollable cursor

Method	**Positions the cursor**
first[1]	On the first row of the ResultSet
last[1]	On the last row of the ResultSet
next[2]	On the next row of the ResultSet
previous[1,3]	On the previous row of the ResultSet

Table 5-5. ResultSet methods for positioning a scrollable cursor (continued)

Method	Positions the cursor
`absolute(int n)`[1,4]	If *n*>0, on row *n* of the ResultSet. If *n*<0, and *m* is the number of rows in the ResultSet, on row *m*+*n*+*1* of the ResultSet.
`relative(int n)`[1,5,6,]	If *n*>0, on the row that is *n* rows after the current row. If *n*<0, on the row that is *n* rows before the current row. If *n*=0, on the current row.
afterLast[1]	After the last row in the ResultSet
beforeFirst[1]	Before the first row in the ResultSet

Notes:

1. This method does not apply to connections to IBM Informix.
2. If the cursor is before the first row of the ResultSet, this method positions the cursor on the first row.
3. If the cursor is after the last row of the ResultSet, this method positions the cursor on the last row.
4. If the absolute value of *n* is greater than the number of rows in the result set, this method positions the cursor after the last row if *n* is positive, or before the first row if *n* is negative.
5. The cursor must be on a valid row of the ResultSet before you can use this method. If the cursor is before the first row or after the last row, the method throws an SQLException.
6. Suppose that *m* is the number of rows in the ResultSet and *x* is the current row number in the ResultSet. If *n*>0 and *x*+*n*>*m*, the driver positions the cursor after the last row. If *n*<0 and *x*+*n*<1, the driver positions the cursor before the first row.

   b. If you need to know the current cursor position, use the getRow, isFirst, isLast, `isBeforeFirst`, or isAfterLast method to obtain this information.

   c. If you specified a *resultSetType* value of `TYPE_SCROLL_SENSITIVE` in step 1 on page 5-24, and you need to see the latest values of the current row, invoke the refreshRow method.

      Recommendation: Because refreshing the rows of a ResultSet can have a detrimental effect on the performance of your applications, you should invoke refreshRow *only* when you need to see the latest data.

   d. Perform one or more of the following operations:

      - To retrieve data from each column of the current row of the ResultSet object, use getXXX methods.
      - To update the current row from the underlying table, use updateXXX methods to assign column values to the current row of the ResultSet. Then use updateRow to update the corresponding row of the underlying table. If you decide that you do not want to update the underlying table, invoke the cancelRowUpdates method instead of the updateRow method.

        The *resultSetConcurrency* value for the ResultSet must be `CONCUR_UPDATABLE` for you to use these methods.
      - To delete the current row from the underlying table, use the deleteRow method. Invoking deleteRow causes the driver to replace the current row of the ResultSet with a hole.

        The *resultSetConcurrency* value for the ResultSet must be `CONCUR_UPDATABLE` for you to use this method.

5. Invoke the close method to close the ResultSet object.
6. Invoke the close method to close the Statement or PreparedStatement object.

The following code demonstrates how to retrieve all employee numbers from the employee table. The numbers to the right of selected statements correspond to the previously-described steps.

```
String s;
Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                           ResultSet.CONCUR_UPDATABLE);                1
                                        // Create a Statement object
                                        // for a scrollable, updatable
                                        // ResultSet
rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE");
                                       // Create the ResultSet          3
while (rs.next()) {                    // Position the cursor
 s = rs.getString("EMPNO");            // Retrieve the employee number  4d
                                       // (column 1 in the result
                                       // table)
 System.out.println("Employee number = " + s);
                                       // Print the column value
}
rs.close();                            // Close the ResultSet           5
stmt.close();                          // Close the Statement           6
```

Figure 5-14. Using a scrollable cursor

Multi-row SQL operations in JDBC applications:

The IBM Data Server Driver for JDBC and SQLJ supports multi-row INSERT, UPDATE, and FETCH for connections to data sources that support these operations.

Multi-row INSERT

In JDBC applications, when you execute INSERT or MERGE statements that use parameter markers in a batch, if the data server supports multi-row INSERT, the IBM Data Server Driver for JDBC and SQLJ can transform the batch INSERT or MERGE operations into multi-row INSERT statements. Multi-row INSERT operations can provide better performance in the following ways:

- For local applications, multi-row INSERTs result in fewer accesses of the data server.
- For distributed applications, multi-row INSERTs result in fewer network operations.

You cannot execute a multi-row INSERT operation by including a multi-row INSERT statement in a statement string in your JDBC application.

Multi-row INSERT is used by default. You can use the Connection or DataSource property enableMultiRowInsertSupport to control whether multi-row INSERT is used. Multi-row INSERT cannot be used for INSERT FROM SELECT statements that are executed in a batch.

Multi-row FETCH

Multi-row FETCH can provide better performance than retrieving one row with each FETCH statement. For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, multi-row FETCH can be used for forward-only

cursors and scrollable cursors. For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, multi-row FETCH can be used only for scrollable cursors.

When you retrieve data in your applications, the IBM Data Server Driver for JDBC and SQLJ determines whether to use multi-row FETCH, depending on several factors:

- The settings of the enableRowsetSupport and useRowsetCursor properties
- The type of IBM Data Server Driver for JDBC and SQLJ connectivity that is being used
- The version of the IBM Data Server Driver for JDBC and SQLJ

For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS, one of the following sets of conditions must be true for multi-row FETCH to be used.

- First set of conditions:
  - The IBM Data Server Driver for JDBC and SQLJ version is 3.51 or later.
  - The enableRowsetSupport property value is com.ibm.db2.jcc.DB2BaseDataSource.YES (1), **or** the enableRowsetSupport property value is com.ibm.db2.jcc.DB2BaseDataSource.NOT_SET (0) and the useRowsetCursor property value is com.ibm.db2.jcc.DB2BaseDataSource.YES (1).
  - The FETCH operation uses a scrollable cursor.

    For forward-only cursors, fetching of multiple rows might occur through DRDA block FETCH. However, this behavior does not utilize the data source's multi-row FETCH capability.
- Second set of conditions:
  - The IBM Data Server Driver for JDBC and SQLJ version is 3.1.
  - The useRowsetCursor property value is com.ibm.db2.jcc.DB2BaseDataSource.YES (1).
  - The FETCH operation uses a scrollable cursor.

    For forward-only cursors, fetching of multiple rows might occur through DRDA block FETCH. However, this behavior does not utilize the data source's multi-row FETCH capability.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS the following conditions must be true for multi-row FETCH to be used.

- The IBM Data Server Driver for JDBC and SQLJ version is 3.51 or later.
- The enableRowsetSupport property value is com.ibm.db2.jcc.DB2BaseDataSource.YES (1).
- The FETCH operation uses a scrollable cursor or a forward-only cursor.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, you can control the maximum size of a rowset for each statement by setting the maxRowsetSize property.

Multi-row positioned UPDATE or DELETE

The IBM Data Server Driver for JDBC and SQLJ supports a technique for performing positioned update or delete operations that follows the JDBC 1 standard. That technique involves using the ResultSet.getCursorName method to obtain the name of the cursor for the ResultSet, and defining a positioned UPDATE or positioned DELETE statement of the following form:

```
UPDATE table SET col1=value1,...coln=valueN WHERE CURRENT OF cursorname
DELETE FROM table WHERE CURRENT OF cursorname
```

Multi-row UPDATE or DELETE when useRowsetCursor is set to `true`: If you use the JDBC 1 technique to update or delete data on a database server that supports multi-row FETCH, and multi-row FETCH is enabled through the useRowsetCursor property, the positioned UPDATE or DELETE statement might update or delete multiple rows, when you expect it to update or delete a single row. To avoid unexpected updates or deletes, you can take one of the following actions:

- Use an updatable `ResultSet` to retrieve and update one row at a time, as shown in the previous example.
- Set useRowsetCursor to `false`.

Multi-row UPDATE or DELETE when enableRowsetSupport is set to com.ibm.db2.jcc.DB2BaseDataSource.YES (1): The JDBC 1 technique for updating or deleting data is incompatible with multi-row FETCH that is enabled through the enableRowsetSupport property.

Recommendation: If your applications use the JDBC 1 technique, update them to use the JDBC 2.0 ResultSet.updateRow or ResultSet.deleteRow methods for positioned update or delete activity.

Inserting a row into a ResultSet in a JDBC application:

If a ResultSet has a *resultSetConcurrency* attribute of CONCUR_UPDATABLE, you can insert rows into the ResultSet.

To insert a row into a ResultSet, follow these steps:

1. Perform the following steps for each row that you want to insert.
   a. Call the ResultSet.moveToInsertRow method to create the row that you want to insert. The row is created in a buffer outside the ResultSet.

      If an insert buffer already exists, all old values are cleared from the buffer.
   b. Call ResultSet.updateXXX methods to assign values to the row that you want to insert.

      You need to assign a value to at least one column in the ResultSet. If you do not do so, an SQLException is thrown when the row is inserted into the ResultSet.

      If you do not assign a value to a column of the ResultSet, when the underlying table is updated, the data source inserts the default value for the associated table column.

      If you assign a null value to a column that is defined as NOT NULL, the JDBC driver throws and SQLException.
   c. Call ResultSet.insertRow to insert the row into the ResultSet.

      After you call ResultSet.insertRow, all values are always cleared from the insert buffer, even if ResultSet.insertRow fails.
2. Reposition the cursor within the ResultSet.

   To move the cursor from the insert row to the ResultSet, you can invoke any of the methods that position the cursor at a specific row, such as ResultSet.first, ResultSet.absolute, or ResultSet.relative. Alternatively, you can call ResultSet.moveToCurrentRow to move the cursor to the row in the ResultSet that was the current row before the insert operation occurred.

   After you call ResultSet.moveToCurrentRow, all values are cleared from the insert buffer.

Example: The following code illustrates inserting a row into a ResultSet that consists of all rows in the sample DEPARTMENT table. After the row is inserted, the code places the cursor where it was located in the ResultSet before the insert operation. The numbers to the right of selected statements correspond to the previously-described steps.

```
stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
             ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT * FROM DEPARTMENT");
rs.moveToInsertRow();                                        1a
rs.updateString("DEPT_NO", "M13");                           1b
rs.updateString("DEPTNAME", "TECHNICAL SUPPORT");
rs.updateString("MGRNO", "000010");
rs.updateString("ADMRDEPT", "A00");
rs.insertRow();                                              1c
rs.moveToCurrentRow();                                       2
```

Testing whether the current row was inserted into a ResultSet in a JDBC application:

If a ResultSet is dynamic, you can insert rows into it. After you insert rows into a ResultSet you might need to know which rows were inserted.

To test whether the current row in a ResultSet was inserted, follow these steps:

1. Call the DatabaseMetaData.ownInsertsAreVisible and DatabaseMetaData.othersInsertsAreVisible methods to determine whether inserts can be visible to the given type of ResultSet.
2. If inserts can be visible to the ResultSet, call the DatabaseMetaData.insertsAreDetected method to determine whether the given type of ResultSet can detect inserts.
3. If the ResultSet can detect inserts, call the ResultSet.rowInserted method to determine whether the current row was inserted.

Calling stored procedures in JDBC applications

To call stored procedures, you invoke methods in the CallableStatement class.

The basic steps for calling a stored procedures using standard CallableStatement methods are:

1. Invoke the Connection.prepareCall method with the CALL statement as its argument to create a CallableStatement object.

   You can represent parameters with standard parameter markers (?) or named parameter markers. You cannot mix named parameter markers with standard parameter markers in the same CALL statement.
2. Invoke the CallableStatement.setXXX methods to pass values to the input parameters (parameters that are defined as IN or INOUT in the CREATE PROCEDURE statement).

   This step assumes that you use standard parameter markers or named parameters. Alternatively, if you use named parameter markers, you use IBM Data Server Driver for JDBC and SQLJ-only methods to pass values to the input parameters.

   Restriction: If the data source does not support dynamic execution of the CALL statement, you must specify the data types for CALL statement input parameters **exactly** as they are specified in the stored procedure definition.

3. Invoke the CallableStatement.registerOutParameter method to register parameters that are defined as OUT in the CREATE PROCEDURE statement with specific data types.

   This step assumes that you use standard parameter markers. Alternatively, if you use named parameter markers, you use IBM Data Server Driver for JDBC and SQLJ-only methods to register OUT parameters with specific data types.

   Restriction: If the data source does not support dynamic execution of the CALL statement, you must specify the data types for CALL statement OUT, IN, or INOUT parameters **exactly** as they are specified in the stored procedure definition.

4. Invoke one of the following methods to call the stored procedure:

   `CallableStatement.executeUpdate`
   Invoke this method if the stored procedure does not return result sets.

   `CallableStatement.executeQuery`
   Invoke this method if the stored procedure returns one result set.

   You can invoke CallableStatement.executeQuery for a stored procedure that returns no result sets if you set property allowNullResultSetForExecuteQuery to DB2BaseDataSource.YES (1). In that case, CallableStatement.executeQuery returns null. This behavior does not conform to the JDBC standard.

   `CallableStatement.execute`
   Invoke this method if the stored procedure returns multiple result sets, or an unknown number of result sets.

   Restriction: IBM Informix data sources do not support multiple result sets.

5. If the stored procedure returns multiple result sets, retrieve the result sets.

   Restriction: IBM Informix data sources do not support multiple result sets.

6. Invoke the CallableStatement.getXXX methods to retrieve values from the OUT parameters or INOUT parameters.
7. Invoke the CallableStatement.close method to close the CallableStatement object when you have finished using that object.

Example: The following code illustrates calling a stored procedure that has one input parameter, four output parameters, and no returned ResultSets. The numbers to the right of selected statements correspond to the previously-described steps.

```
int ifcaret;
int ifcareas;
int xsbytes;
String errbuff;
Connection con;
CallableStatement cstmt;
ResultSet rs;
...
cstmt = con.prepareCall("CALL DSN8.DSN8ED2(?,?,?,?,?)");              1
                                  // Create a CallableStatement object
cstmt.setString (1, "DISPLAY THREAD(*)");                             2
                                  // Set input parameter (DB2 command)
cstmt.registerOutParameter (2, Types.INTEGER);                        3
                                  // Register output parameters
cstmt.registerOutParameter (3, Types.INTEGER);
cstmt.registerOutParameter (4, Types.INTEGER);
cstmt.registerOutParameter (5, Types.VARCHAR);
cstmt.executeUpdate();            // Call the stored procedure        4
ifcaret = cstmt.getInt(2);        // Get the output parameter values  6
```

```
ifcareas = cstmt.getInt(3);
xsbytes = cstmt.getInt(4);
errbuff = cstmt.getString(5);
cstmt.close();                                                    7
```

LOBs in JDBC applications with the IBM Data Server Driver for JDBC and SQLJ

The IBM Data Server Driver for JDBC and SQLJ supports methods for updating and retrieving data from BLOB, CLOB, and DBCLOB columns in a table, and for calling stored procedures or user-defined functions with BLOB or CLOB parameters.

Progressive streaming with the IBM Data Server Driver for JDBC and SQLJ

If the data source supports progressive streaming, also known as dynamic data format, the IBM Data Server Driver for JDBC and SQLJ can use progressive streaming to retrieve data in LOB or XML columns.

DB2 for z/OS Version 9.1 and later supports progressive streaming for LOBs and XML objects. DB2 Database for Linux, UNIX, and Windows Version 9.5 and later, IBM Informix Version 11.50 and later, and DB2 for i V6R1 and later support progressive streaming for LOBs.

With progressive streaming, the data source dynamically determines the most efficient mode in which to return LOB or XML data, based on the size of the LOBs or XML objects.

Progressive streaming is the default behavior in the following environments:

MinimumIBM Data Server Driver for JDBC and SQLJ version	Minimum data server version	Types of objects
3.53	DB2 for i V6R1	LOB, XML
3.50	DB2 Database for Linux, UNIX, and Windows Version 9.5	LOB
3.50	IBM Informix Version 11.50	LOB
3.2	DB2 for z/OS Version 9	LOB, XML

You set the progressive streaming behavior on new connections using the IBM Data Server Driver for JDBC and SQLJ `progressiveStreaming` property.

When progressive streaming is enabled, you can control when the JDBC driver materializes LOBs with the `streamBufferSize` property. If a LOB or XML object is less than or equal to the `streamBufferSize` value, the object is materialized.

Important: With progressive streaming, when you retrieve a LOB or XML value from a ResultSet into an application variable, you can manipulate the contents of that application variable until you move the cursor or close the cursor on the ResultSet. After that, the contents of the application variable are no longer available to you. If you perform any actions on the LOB in the application variable, you receive an SQLException. For example, suppose that progressive streaming is enabled, and you execute statements like this:

```
...
ResultSet rs = stmt.executeQuery("SELECT CLOBCOL FROM MY_TABLE");
rs.next();                // Retrieve the first row of the ResultSet
Clob clobFromRow1  = rs.getClob(1);
                          // Put the CLOB from the first column of
                          // the first row in an application variable
String substr1Clob = clobFromRow1.getSubString(1,50);
                          // Retrieve the first 50 bytes of the CLOB
rs.next();                // Move the cursor to the next row.
                          // clobFromRow1 is no longer available.
// String substr2Clob = clobFromRow1.getSubString(51,100);
                          // This statement would yield an SQLException
Clob clobFromRow2  = rs.getClob(1);
                          // Put the CLOB from the first column of
                          // the second row in an application variable
rs.close();               // Close the ResultSet.
                          // clobFromRow2 is also no longer available.
```

After you execute rs.next() to position the cursor at the second row of the ResultSet, the CLOB value in clobFromRow1 is no longer available to you. Similarly, after you execute rs.close() to close the ResultSet, the values in clobFromRow1 and clobFromRow2 are no longer available.

If you disable progressive streaming, the way in which the IBM Data Server Driver for JDBC and SQLJ handles LOBs depends on the value of the `fullyMaterializeLobData` property.

Use of progressive streaming is the preferred method of LOB or XML data retrieval.

LOB locators with the IBM Data Server Driver for JDBC and SQLJ

The IBM Data Server Driver for JDBC and SQLJ can use LOB locators to retrieve data in LOB columns.

To cause JDBC to use LOB locators to retrieve data from LOB columns, you need to set the `fullyMaterializeLobData` property to `false` and set the `progressiveStreaming` property to NO (`DB2BaseDataSource.NO` in an application program).

The effect of fullyMaterializeLobData depends on whether the data source supports progressive streaming and the value of the progressiveStreaming property:

- If the data source does not support progressive locators:

  If the value of fullyMaterializeLobData is `true`, LOB data is fully materialized within the JDBC driver when a row is fetched. If the value is `false`, LOB data is streamed. The driver uses locators internally to retrieve LOB data in chunks on an as-needed basis It is highly recommended that you set this value to `false` when you retrieve LOBs that contain large amounts of data. The default is `true`.
- If the data source supports progressive streaming, also known as dynamic data format:

  The JDBC driver ignores the value of fullyMaterializeLobData if the progressiveStreaming property is set to YES (`DB2BaseDataSource.YES` in an application program) or is not set.

`fullyMaterializeLobData` has no effect on stored procedure parameters.

As in any other language, a LOB locator in a Java application is associated with only one data source. You cannot use a single LOB locator to move data between two different data sources. To move LOB data between two data sources, you need to materialize the LOB data when you retrieve it from a table in the first data source and then insert that data into the table in the second data source.

LOB operations with the IBM Data Server Driver for JDBC and SQLJ

The IBM Data Server Driver for JDBC and SQLJ supports methods for updating and retrieving data from BLOB, CLOB, and DBCLOB columns in a table, and for calling stored procedures or user-defined functions with BLOB or CLOB parameters.

Among the operations that you can perform on LOB data under the IBM Data Server Driver for JDBC and SQLJ are:

- Specify a BLOB or column as an argument of the following ResultSet methods to retrieve data from a BLOB or CLOB column:

  For BLOB columns:
  - getBinaryStream
  - getBlob
  - getBytes

  For CLOB columns:
  - getAsciiStream
  - getCharacterStream
  - getClob
  - getString
- Call the following ResultSet methods to update a BLOB or CLOB column in an updatable ResultSet:

  For BLOB columns:
  - updateBinaryStream
  - updateBlob

  For CLOB columns:
  - updateAsciiStream
  - updateCharacterStream
  - updateClob

  If you specify -1 for the *length* parameter in any of the previously listed methods, the IBM Data Server Driver for JDBC and SQLJ reads the input data until it is exhausted.
- Use the following PreparedStatement methods to set the values for parameters that correspond to BLOB or CLOB columns:

  For BLOB columns:
  - setBytes
  - setBlob
  - setBinaryStream
  - setObject, where the *Object* parameter value is an InputStream.

  For CLOB columns:
  - setString
  - setAsciiStream
  - setClob
  - setCharacterStream
  - setObject, where the *Object* parameter value is a Reader.

  If you specify -1 for *length*, the IBM Data Server Driver for JDBC and SQLJ reads the input data until it is exhausted.

- Retrieve the value of a JDBC CLOB parameter using the CallableStatement.getString method.

Restriction: With IBM Data Server Driver for JDBC and SQLJ type 2 connectivity, you cannot call a stored procedure that has DBCLOB OUT or INOUT parameters.

If you are using the IBM Data Server Driver for JDBC and SQLJ version 4.0 or later, you can perform the following additional operations:

- Use ResultSet.updateXXX or PreparedStatement.setXXX methods to update a BLOB or CLOB with a *length* value of up to 2GB for a BLOB or CLOB. For example, these methods are defined for BLOBs:

```
ResultSet.updateBlob(int columnIndex, InputStream x, long length)
ResultSet.updateBlob(String columnLabel, InputStream x, long length)
ResultSet.updateBinaryStream(int columnIndex, InputStream x, long length)
ResultSet.updateBinaryStream(String columnLabel, InputStream x, long length)
PreparedStatement.setBlob(int columnIndex, InputStream x, long length)
PreparedStatement.setBlob(String columnLabel, InputStream x, long length)
PreparedStatement.setBinaryStream(int columnIndex, InputStream x, long length)
PreparedStatement.setBinaryStream(String columnLabel, InputStream x, long length)
```

- Use ResultSet.updateXXX or PreparedStatement.setXXX methods without the *length* parameter when you update a BLOB or CLOB, to cause the IBM Data Server Driver for JDBC and SQLJ to read the input data until it is exhausted. For example:

```
ResultSet.updateBlob(int columnIndex, InputStream x)
ResultSet.updateBlob(String columnLabel, InputStream x)
ResultSet.updateBinaryStream(int columnIndex, InputStream x)
ResultSet.updateBinaryStream(String columnLabel, InputStream x)
PreparedStatement.setBlob(int columnIndex, InputStream x)
PreparedStatement.setBlob(String columnLabel, InputStream x)
PreparedStatement.setBinaryStream(int columnIndex, InputStream x)
PreparedStatement.setBinaryStream(String columnLabel, InputStream x)
```

- Create a Blob or Clob object that contains no data, using the Connection.createBlob or Connection.createClob method.
- Materialize a Blob or Clob object on the client, when progressive streaming or locators are in use, using the Blob.getBinaryStream or Clob.getCharacterStream method.
- Free the resources that a Blob or Clob object holds, using the Blob.free or Clob.free method.

Java data types for retrieving or updating LOB column data in JDBC applications

When the JDBC driver cannot immediately determine the data type of a parameter that is used with a LOB column, you need to choose a parameter data type that is compatible with the LOB data type.

When the deferPrepares property is set to true, and the IBM Data Server Driver for JDBC and SQLJ processes a PreparedStatement.setXXX call, the driver might need to do extra processing to determine data types. This extra processing can impact performance.

Input parameters for BLOB columns

For IN parameters for BLOB columns, or INOUT parameters that are used for input to BLOB columns, you can use one of the following techniques:

- Use a java.sql.Blob input variable, which is an exact match for a BLOB column:

```
cstmt.setBlob(parmIndex, blobData);
```

- Use a CallableStatement.setObject call that specifies that the target data type is BLOB:

```
byte[] byteData = {(byte)0x1a, (byte)0x2b, (byte)0x3c};
cstmt.setObject(parmInd, byteData, java.sql.Types.BLOB);
```

- Use an input parameter of type of java.io.ByteArrayInputStream with a CallableStatement.setBinaryStream call. A java.io.ByteArrayInputStream object is compatible with a BLOB data type. For this call, you need to specify the exact length of the input data:

```
java.io.ByteArrayInputStream byteStream =
  new java.io.ByteArrayInputStream(byteData);
int numBytes = byteData.length;
cstmt.setBinaryStream(parmIndex, byteStream, numBytes);
```

Output parameters for BLOB columns

For OUT parameters for BLOB columns, or INOUT parameters that are used for output from BLOB columns, you can use the following technique:

- Use the CallableStatement.registerOutParameter call to specify that an output parameter is of type BLOB. Then you can retrieve the parameter value into any variable that has a data type that is compatible with a BLOB data type. For example, the following code lets you retrieve a BLOB value into a byte[] variable:

```
cstmt.registerOutParameter(parmIndex, java.sql.Types.BLOB);
cstmt.execute();
byte[] byteData = cstmt.getBytes(parmIndex);
```

Input parameters for CLOB columns

For IN parameters for CLOB columns, or INOUT parameters that are used for input to CLOB columns, you can use one of the following techniques:

- Use a java.sql.Clob input variable, which is an exact match for a CLOB column:

```
cstmt.setClob(parmIndex, clobData);
```

- Use a CallableStatement.setObject call that specifies that the target data type is CLOB:

```
String charData = "CharacterString";
cstmt.setObject(parmInd, charData, java.sql.Types.CLOB);
```

- Use one of the following types of stream input parameters:
  - A java.io.StringReader input parameter with a cstmt.setCharacterStream call:

```
java.io.StringReader reader = new java.io.StringReader(charData);
cstmt.setCharacterStream(parmIndex, reader, charData.length);
```

  - A java.io.ByteArrayInputStream parameter with a cstmt.setAsciiStream call, for ASCII data:

```
byte[] charDataBytes = charData.getBytes("US-ASCII");
java.io.ByteArrayInputStream byteStream =
  new java.io.ByteArrayInputStream (charDataBytes);
cstmt.setAsciiStream(parmIndex, byteStream, charDataBytes.length);
```

  For these calls, you need to specify the exact length of the input data.

- Use a String input parameter with a cstmt.setString call:

```
cstmt.setString(parmIndex, charData);
```

  If the length of the data is greater than 32KB, and the JDBC driver has no DESCRIBE information about the parameter data type, the JDBC driver assigns the CLOB data type to the input data.

- Use a String input parameter with a cstmt.setObject call, and specify the target data type as VARCHAR or LONGVARCHAR:

  ```
  cstmt.setObject(parmIndex, charData, java.sql.Types.VARCHAR);
  ```

  If the length of the data is greater than 32KB, and the JDBC driver has no DESCRIBE information about the parameter data type, the JDBC driver assigns the CLOB data type to the input data.

Output parameters for CLOB columns

For OUT parameters for CLOB columns, or INOUT parameters that are used for output from CLOB columns, you can use one of the following techniques:

- Use the CallableStatement.registerOutParameter call to specify that an output parameter is of type CLOB. Then you can retrieve the parameter value into a Clob variable. For example:

  ```
  cstmt.registerOutParameter(parmIndex, java.sql.Types.CLOB);
  cstmt.execute();
  Clob clobData = cstmt.getClob(parmIndex);
  ```

- Use the CallableStatement.registerOutParameter call to specify that an output parameter is of type VARCHAR or LONGVARCHAR:

  ```
  cstmt.registerOutParameter(parmIndex, java.sql.Types.VARCHAR);
  cstmt.execute();
  String charData = cstmt.getString(parmIndex);
  ```

  This technique should be used only if you know that the length of the retrieved data is less than or equal to 32KB. Otherwise, the data is truncated.

ROWIDs in JDBC with the IBM Data Server Driver for JDBC and SQLJ

DB2 for z/OS and DB2 for i support the ROWID data type for a column in a database table. A ROWID is a value that uniquely identifies a row in a table.

Although IBM Informix also supports rowids, those rowids have the INTEGER data type. You can select an IBM Informix rowid column into a variable with a four-byte integer data type.

You can use the following ResultSet methods to retrieve data from a ROWID column:

- getRowId (JDBC 4.0 and later)
- getBytes
- getObject

You can use the following ResultSet method to update a ROWID column of an updatable ResultSet:

- updateRowId (JDBC 4.0 and later)

  updateRowId is valid only if the target database system supports updating of ROWID columns.

If you are using JDBC 3.0, for getObject, the IBM Data Server Driver for JDBC and SQLJ returns an instance of the IBM Data Server Driver for JDBC and SQLJ-only class com.ibm.db2.jcc.DB2RowID.

If you are using JDBC 4.0, for getObject, the IBM Data Server Driver for JDBC and SQLJ returns an instance of the class java.sql.RowId.

You can use the following PreparedStatement methods to set a value for a parameter that is associated with a ROWID column:
- setRowId (JDBC 4.0 and later)
- setBytes
- setObject

If you are using JDBC 3.0, for setObject, use the IBM Data Server Driver for JDBC and SQLJ-only type `com.ibm.db2.jcc.Types.ROWID` or an instance of the `com.ibm.db2.jcc.DB2RowID` class as the target type for the parameter.

If you are using JDBC 4.0, for setObject, use the type `java.sql.Types.ROWID` or an instance of the `java.sql.RowId` class as the target type for the parameter.

You can use the following CallableStatement methods to retrieve a ROWID column as an output parameter from a stored procedure call:
- getRowId (JDBC 4.0 and later)
- getObject

To call a stored procedure that is defined with a ROWID output parameter, register that parameter to be of the `java.sql.Types.ROWID` type.

ROWID values are valid for different periods of time, depending on the data source on which those ROWID values are defined. Use the DatabaseMetaData.getRowIdLifetime method to determine the time period for which a ROWID value is valid. The values that are returned for the data sources are listed in the following table.

Table 5-6. DatabaseMetaData.getRowIdLifetime values for supported data sources

Database server	**DatabaseMetaData.getRowIdLifetime**
DB2 for z/OS	ROWID_VALID_TRANSACTION
DB2 Database for Linux, UNIX, and Windows	ROWID_UNSUPPORTED
DB2 for i	ROWID_VALID_FOREVER
IBM Informix	ROWID_VALID_FOREVER

Example: Using PreparedStatement.setRowId with a java.sql.RowId target type: Suppose that rwid is a RowId object. To set parameter 1, use this form of the setRowId method:

```
ps.setRowId(1, rid);
```

Example: Using ResultSet.getRowId to retrieve a ROWID value from a data source: To retrieve a ROWID value from the first column of a result set into RowId object rwid, use this form of the ResultSet.getRowId method:

```
java.sql.RowId rwid = rs.getRowId(1);
```

Example: Using CallableStatement.registerOutParameter with a java.sql.Types.ROWID parameter type: To register parameter 1 of a CALL statement as a `java.sql.Types.ROWID` data type, use this form of the registerOutParameter method:

```
cs.registerOutParameter(1, java.sql.Types.ROWID)
```

Savepoints in JDBC applications

An SQL savepoint represents the state of data and schemas at a particular point in time within a unit of work. You can use SQL statements to set a savepoint, release a savepoint, and restore data and schemas to the state that the savepoint represents.

The IBM Data Server Driver for JDBC and SQLJ supports the following methods for using savepoints:

Connection.setSavepoint() or Connection.setSavepoint(String *name*)
Sets a savepoint. These methods return a Savepoint object that is used in later releaseSavepoint or rollback operations.

When you execute either of these methods, IDS executes the form of the SAVEPOINT statement that includes ON ROLLBACK RETAIN CURSORS.

Connection.releaseSavepoint(Savepoint *savepoint*)
Releases the specified savepoint, and all subsequently established savepoints.

Connection.rollback(Savepoint *savepoint*)
Rolls back work to the specified savepoint.

DatabaseMetaData.supportsSavepoints()
Indicates whether a data source supports savepoints.

You can indicate whether savepoints are unique by calling the method DB2Connection.setSavePointUniqueOption. If you call this method with a value of `true`, the application cannot set more than one savepoint with the same name within the same unit of recovery. If you call this method with a value of `false` (the default), multiple savepoints with the same name can be created within the same unit of recovery, but creation of a savepoint destroys a previously created savepoint with the same name.

The following example demonstrates how to set a savepoint, roll back to the savepoint, and release the savepoint.

```
Connection con;
Statement stmt;
ResultSet rs;
String empNumVar;
int shoeSizeVar;
...
con.setAutoCommit(false);                    // set autocommit OFF
stmt = con.createStatement();                // Create a Statement object
...                                            // Perform some SQL
con.commit();                                // Commit the transaction
stmt.executeUpdate("INSERT INTO EMP_SHOE " +
  "VALUES ('000010', 6)");                   // Insert a row
((com.ibm.db2.jcc.DB2Connection)con).setSavePointUniqueOption(true);
                                             // Indicate that savepoints
                                             // are unique within a unit
                                             // of recovery
Savepoint savept = con.setSavepoint("savepoint1");
                                             // Create a savepoint
...
stmt.executeUpdate("INSERT INTO EMP_SHOE " +
  "VALUES ('000020', 10)");                  // Insert another row
conn.rollback(savept);                       // Roll back work to the point
                                             //  after the first insert
...
con.releaseSavepoint(savept);                // Release the savepoint
stmt.close();                                // Close the Statement
conn.commit();                               // Commit the transaction
```

Figure 5-15. Setting, rolling back to, and releasing a savepoint in a JDBC application

Retrieval of automatically generated keys in JDBC applications

With the IBM Data Server Driver for JDBC and SQLJ, you can retrieve automatically generated keys (also called auto-generated keys) from a table using JDBC 3.0 methods. Alternatively, you can use IBM Data Server Driver for JDBC and SQLJ-only methods to retrieve the automatically generated keys.

An *automatically generated key* is any value that is generated by the data server, instead of being specified by the user. One type of automatically generated key is the contents of a SERIAL, SERIAL8, or BIGSERIAL column. A table column of one of those types provides a way for the data source to automatically generate a numeric value for each row. A table cannot have more than one column of each type, but it can have a one column of each type.

For connections to IBM Informix, the IBM Data Server Driver for JDBC and SQLJ supports the return of automatically generated keys for INSERT statements.

Restriction: If the Connection or DataSource property atomicMultiRowInsert is set to `DB2BaseDataSource.YES` (1), you cannot prepare an SQL statement for retrieval of automatically generated keys and use the PreparedStatement object for batch updates. The IBM Data Server Driver for JDBC and SQLJ version 3.50 or later throws an SQLException when you call the addBatch or executeBatch method on a PreparedStatement object that is prepared to return automatically generated keys.

Retrieving auto-generated keys for an INSERT statement

With the IBM Data Server Driver for JDBC and SQLJ, you can use JDBC 3.0 methods to retrieve the keys that are automatically generated when you execute an INSERT statement.

To retrieve automatically generated keys that are generated by an INSERT statement, you need to perform these steps.

1. Use one of the following methods to indicate that you want to return automatically generated keys:
   - If you plan to use the PreparedStatement.executeUpdate method to insert rows, invoke one of these forms of the Connection.prepareStatement method to create a PreparedStatement object:

     ```
     Connection.prepareStatement(sql-statement,
       Statement.RETURN_GENERATED_KEYS);
     Connection.prepareStatement(sql-statement, String [] columnNames);
     Connection.prepareStatement(sql-statement, int [] columnIndexes);
     ```

     With the first form, you specify whether all automatically generated keys should be returned. With the second form, you specify the names of the columns for which you want automatically generated keys. With the third form, you specify the positions in the table of the columns for which you want automatically generated keys.
   - If you use the Statement.executeUpdate method to insert rows, invoke one of these forms of the Statement.executeUpdate method:

     ```
     Statement.executeUpdate(sql-statement,
       Statement.RETURN_GENERATED_KEYS);
     Statement.executeUpdate(sql-statement, String [] columnNames);
     Statement.executeUpdate(sql-statement, int [] columnIndexes);
     ```

     With the first form, you specify whether all automatically generated keys should be returned. With the second form, you specify the names of the columns for which you want automatically generated keys. With the third form, you specify the positions in the table of the columns for which you want automatically generated keys.
2. Invoke the PreparedStatement.getGeneratedKeys method or the Statement.getGeneratedKeys method to retrieve a ResultSet object that contains the automatically generated key values.

   If the column data type is SERIAL, the automatically generated keys in the ResultSet have a data type of INT. Use ResultSet.getInt to retrieve the values. If the column data type is SERIAL8, the automatically generated keys in the ResultSet have a data type of BIGINT. Use ResultSet.getLong to retrieve the values.

The following code creates a table with a SERIAL column, inserts a row into the table, and retrieves the automatically generated key value for the SERIAL column. The numbers to the right of selected statements correspond to the previously described steps.

```
import java.sql.*;
import java.math.*;
import com.ibm.db2.jcc.*;

Connection con;
Statement stmt;
ResultSet rs;
java.math.BigDecimal serColVar;
...
stmt = con.createStatement();         // Create a Statement object

stmt.executeUpdate(
  "CREATE TABLE EMP_PHONE (EMPNO CHAR(6), PHONENO CHAR(4), " +
    "SERIALCOL SERIAL)");
                                      // Create table with identity column
stmt.executeUpdate("INSERT INTO EMP_PHONE (EMPNO, PHONENO) " +          1
  "VALUES ('000010', '5555')",        // Insert a row
  Statement.RETURN_GENERATED_KEYS);   // Indicate you want automatically
```

```
                                    // generated keys
rs = stmt.getGeneratedKeys();       // Retrieve the automatically      2
                                    // generated key value in a ResultSet.
                                    // Create ResultSet for query
while (rs.next()) {
  java.math.BigDecimal serColVar = rs.getBigDecimal(1);
                                    // Get automatically generated key
                                    // value
  System.out.println("Automatically generated key value = " + serColVar);
}
rs.close();                         // Close ResultSet
stmt.close();                       // Close Statement
```

Retrieving automatically generated keys using IBM Data Server Driver for JDBC and SQLJ-only methods

The IBM Data Server Driver for JDBC and SQLJ provides a set of methods that you can use to retrieve automatically generated keys (auto-generated keys) from IBM Informix databases. Use of these methods is an alternative to using JDBC 3.0 methods.

Follow these steps to use IBM Data Server Driver for JDBC and SQLJ-only methods to retrieve automatically generated keys from IBM Informix data sources.

1. Execute an INSERT statement on a table that contains SERIAL, SERIAL8, or BIGSERIAL columns.
2. Execute the DB2Statement.getIDSSerial or DB2Statement.getIDSSerial8 method to retrieve the automatically generated keys for the inserted row.

   The returned value for DB2Statement.getIDSSerial has the int data type. The returned value for DB2Statement.getIDSSerial8 has the long data type.

The following code creates a table with a SERIAL column, inserts a row into the table, and retrieves the automatically generated key value for the identity column. The numbers to the right of selected statements correspond to the previously described steps.

Figure 5-16. Example of retrieving automatically generated keys from an IBM Informix table using IBM Data Server Driver for JDBC and SQLJ-only methods

```
import java.sql.*;
import java.math.*;
import com.ibm.db2.jcc.*;

Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement();       // Create a Statement object

stmt.executeUpdate(
  "CREATE TABLE EMP_PHONE (EMPNO CHAR(6), PHONENO CHAR(4), " +
    "SERIALCOL SERIAL)");
                                    // Create table with identity column
stmt.executeUpdate("INSERT INTO EMP_PHONE (EMPNO, PHONENO) " +          1
  "VALUES ('000010', '5555')");      // Insert a row
int serColVar = ((com.ibm.db2.jcc.DB2Statement)stmt).getIDSSerial();    2
                                    // Retrieve the automatically
                                    // generated key value
System.out.println("Automatically generated key value = " + serColVar);
rs.close();                         // Close ResultSet
stmt.close();                       // Close Statement
```

Retrieving automatically generated keys using IBM Data Server Driver for JDBC and SQLJ-only methods

The IBM Data Server Driver for JDBC and SQLJ provides a set of methods that you can use to retrieve automatically generated keys (auto-generated keys) from IBM Informix databases. Use of these methods is an alternative to using JDBC 3.0 methods.

Follow these steps to use IBM Data Server Driver for JDBC and SQLJ-only methods to retrieve automatically generated keys from IBM Informix data sources.

1. Execute an INSERT statement on a table that contains SERIAL, SERIAL8, or BIGSERIAL columns.
2. Execute the DB2Statement.getIDSSerial or DB2Statement.getIDSSerial8 method to retrieve the automatically generated keys for the inserted row.

   The returned value for DB2Statement.getIDSSerial has the int data type. The returned value for DB2Statement.getIDSSerial8 has the long data type.

The following code creates a table with a SERIAL column, inserts a row into the table, and retrieves the automatically generated key value for the identity column. The numbers to the right of selected statements correspond to the previously described steps.

Figure 5-17. Example of retrieving automatically generated keys from an IBM Informix table using IBM Data Server Driver for JDBC and SQLJ-only methods

```
import java.sql.*;
import java.math.*;
import com.ibm.db2.jcc.*;

Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement();         // Create a Statement object

stmt.executeUpdate(
  "CREATE TABLE EMP_PHONE (EMPNO CHAR(6), PHONENO CHAR(4), " +
    "SERIALCOL SERIAL)");
                                      // Create table with identity column
stmt.executeUpdate("INSERT INTO EMP_PHONE (EMPNO, PHONENO) " +          1
  "VALUES ('000010', '5555')");        // Insert a row
int serColVar = ((com.ibm.db2.jcc.DB2Statement)stmt).getIDSSerial();    2
                                      // Retrieve the automatically
                                      // generated key value
System.out.println("Automatically generated key value = " + serColVar);
rs.close();                           // Close ResultSet
stmt.close();                         // Close Statement
```

Using named parameter markers in JDBC applications

You can use named parameter markers instead of standard parameter markers in PreparedStatement and CallableStatement objects to assign values to the input parameter markers. You can also use named parameter markers instead of standard parameter markers in CallableStatement objects to register OUT parameters that have named parameter markers.

SQL strings that contain the following SQL elements can include named parameter markers:

- CALL
- DELETE

- INSERT
- MERGE
- PL/SQL block
- SELECT
- SET
- UPDATE

Named parameter markers make your JDBC applications more readable. If you have named parameter markers in an application, set the IBM Data Server Driver for JDBC and SQLJ Connection or DataSource property enableNamedParameterMarkers to `DB2BaseDataSource.YES` (1) to direct the driver to accept named parameter markers and send them to the data source as standard parameter markers.

Using named parameter markers with PreparedStatement objects

You can use named parameter markers instead of standard parameter markers in PreparedStatement objects to assign values to the parameter markers.

To ensure that applications with named parameters work correctly, regardless of the data server type and version, before you use named parameter markers in your applications, set the Connection or DataSource property enableNamedParameterMarkers to `DB2BaseDataSource.YES`.

To use named parameter markers with PreparedStatement objects, follow these steps.

1. Execute the Connection.prepareStatement method on an SQL statement string that contains named parameter markers. The named parameter markers must follow the rules for SQL host variable names.

   You cannot mix named parameter markers and standard parameter markers in the same SQL statement string.

   Named parameter markers are case-insensitive.
2. For each named parameter marker, use a DB2PreparedStatement.setJccXXXAtName method to assign a value to each named input parameter.

   If you use the same named parameter marker more than once in the same SQL statement string, you need to call a setJccXXXAtName method for that parameter marker only once.

   Recommendation: Do not use the same named parameter marker more than once in the same SQL statement string if the input to that parameter marker is a stream. Doing so can cause unexpected results.

   Restriction: You cannot use standard JDBC PreparedStatement.setXXX methods with named parameter markers. Doing so causes an exception to be thrown.
3. Execute the PreparedStatement.

The following code uses named parameter markers to update the phone number to '4657' for the employee with employee number '000010'. The numbers to the right of selected statements correspond to the previously described steps.

```
Connection con;
PreparedStatement pstmt;
int numUpd;
...
pstmt = con.prepareStatement(
  "UPDATE EMPLOYEE SET PHONENO=:phonenum WHERE EMPNO=:empnum");
```

```
                                    // Create a PreparedStatement object     1
((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).setJccStringAtName
  ("phonenum", "4567");
                                    // Assign a value to phonenum parameter  2
((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).setJccStringAtName
  ("empnum", "000010");
                                    // Assign a value to empnum parameter
numUpd = pstmt.executeUpdate();     // Perform the update                    3
pstmt.close();                      // Close the PreparedStatement object
```

The following code uses named parameter markers to update values in a PL/SQL block. The numbers to the right of selected statements correspond to the previously described steps.

```
Connection con;
PreparedStatement pstmt;
int numUpd;
...
String sql =
  "BEGIN " +
  "  UPDATE EMPLOYEE SET PHONENO = :phonenum WHERE EMPNO = :empnum; " +
  "END;";
pstmt = con.prepareStatement(sql); // Create a PreparedStatement object     1
((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).setJccStringAtName
  ("phonenum", "4567");
                                    // Assign a value to phonenum parameter  2
((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).setJccStringAtName
  ("empnum", "000010");
                                    // Assign a value to empnum parameter
numUpd = pstmt.executeUpdate();     // Perform the update                    3
pstmt.close();                      // Close the PreparedStatement object
```

Using named parameter markers with CallableStatement objects

You can use named parameter markers instead of standard parameter markers in CallableStatement objects to assign values to IN or INOUT parameters and to register OUT parameters.

To ensure that applications with named parameters work correctly, regardless of the data server type and version, before you use named parameter markers in your applications, set the Connection or DataSource property enableNamedParameterMarkers to `DB2BaseDataSource.YES`.

To use named parameter markers with CallableStatement objects, follow these steps.

1. Execute the Connection.prepareCall method on an SQL statement string that contains named parameter markers.

   The named parameter markers must follow the rules for SQL host variable names.

   You cannot mix named parameter markers and standard parameter markers in the same SQL statement string.

   Named parameter markers are case-insensitive.
2. If you do not know the names of the named parameter markers in the CALL statement, or the mode of the parameters (IN, OUT, or INOUT):
   a. Call the CallableStatement.getParameterMetaData method to obtain a ParameterMetaData object with information about the parameters.
   b. Call the ParameterMetaData.getParameterMode method to retrieve the parameter mode.
   c. Cast the ParameterMetaData object to a DB2ParameterMetaData object.

d. Call the DB2ParameterMetaData.getParameterMarkerNames method to retrieve the parameter names.

3. For each named parameter marker that represents an OUT parameter, use a DB2CallableStatement.registerJccOutParameterAtName method to register the OUT parameter with a data type.

   If you use the same named parameter marker more than once in the same SQL statement string, you need to call a registerJccOutParameterAtName method for that parameter marker only once. All parameters with the same name are registered as the same data type.

   Restriction: You cannot use standard JDBC CallableStatement.registerOutParameter methods with named parameter markers. Doing so causes an exception to be thrown.

4. For each named parameter marker for an input parameter, use a DB2CallableStatement.setJccXXXAtName method to assign a value to each named input parameter.

   setJccXXXAtName methods are inherited from DB2PreparedStatement.

   If you use the same named parameter marker more than once in the same SQL statement string, you need to call a setJccXXXAtName method for that parameter marker only once.

   Recommendation: Do not use the same named parameter marker more than once in the same SQL statement string if the input to that parameter marker is a stream. Doing so can cause unexpected results.

5. Execute the CallableStatement.

6. Call CallableStatement.getXXX methods or DB2CallableStatement.getJccXXXAtName methods to retrieve output parameter values.

The following code illustrates calling a stored procedure that has one input VARCHAR parameter and one output INTEGER parameter, which are represented by named parameter markers. The numbers to the right of selected statements correspond to the previously described steps.

```
...
CallableStatement cstmt =
  con.prepareCall("CALL MYSP(:inparm,:outparm)");
                                    // Create a CallableStatement object  1
((com.ibm.db2.jcc.DB2CallableStatement)cstmt).
   registerJccOutParameterAtName("outparm", java.sql.Types.INTEGER);
                                    // Register OUT parameter data type   3
((com.ibm.db2.jcc.DB2CallableStatement)cstmt).setJccStringAtName("inparm", "4567");
                                 // Assign a value to inparm parameter    4

cstmt.executeUpdate();           // Call the stored procedure             5
int outssid = cstmt.getInt(2);   // Get the output parameter value        6
cstmt.close();
```

Providing extended client information to the data source with client info properties

The IBM Data Server Driver for JDBC and SQLJ version 4.0 supports JDBC 4.0 client info properties, which you can use to provide extra information about the client to the server. This information can be used for accounting, workload management, or debugging.

Extended client information is sent to the database server when the application performs an action that accesses the server, such as executing SQL.

The application can also use the Connection.getClientInfo method to retrieve client information from the database server, or execute the DatabaseMetaData.getClientInfoProperties method to determine which client information the driver supports.

The JDBC 4.0 client info properties should be used instead IBM Data Server Driver for JDBC and SQLJ-only methods, which are deprecated.

To set client info properties, follow these steps:

1. Create a Connection.
2. Call the java.sql.Connection.setClientInfo method to set any of the client info properties that the database server supports.
3. Execute an SQL statement to cause the information to be sent to the database server.

The following code performs the previous steps to pass a client's user name and host name to theIDS server. The numbers to the right of selected statements correspond to the previously-described steps.

```
public class ClientInfoTest {
  public static void main(String[] args) {
   String url = "jdbc:ids://sysmvs1.stl.ibm.com:5021/san_jose";
    try {
      Class.forName("com.ibm.db2.jcc.DB2Driver");
      String user = "db2adm";
      String password = "db2adm";
      Connection conn = DriverManager.getConnection(url,       1
        user, password);
      conn.setClientInfo("ClientUser", "Michael L Thompson");  2
      conn.setClientInfo("ClientHostname, "sjwkstn1");
      // Execute SQL to force extended client information to be sent
      // to the server
      conn.prepareStatement("SELECT * FROM SYSIBM.SYSDUMMY1"
         + "WHERE 0 = 1").executeQuery();                      3
    } catch (Throwable e) {
      e.printStackTrace();
    }
  }
}
```

Figure 5-18. Example of passing extended client information to aIDS server

Client info properties support by the IBM Data Server Driver for JDBC and SQLJ

JDBC 4.0 includes client info properties, which contain information about a connection to a data source. The DatabaseMetaData.getClientInfoProperties method returns a list of client info properties that the IBM Data Server Driver for JDBC and SQLJ supports.

When you call DatabaseMetaData.getClientInfoProperties, a result set is returned that contains the following columns:

- NAME
- MAX_LEN
- DEFAULT_VALUE
- DESCRIPTION

The following table lists the client info property values that the IBM Data Server Driver for JDBC and SQLJ returns for DB2 Database for Linux, UNIX, and Windows and for DB2 for i.

Table 5-7. Client info property values for DB2 Database for Linux, UNIX, and Windows and for DB2 for i

NAME	MAX_LEN (bytes)	DEFAULT_VALUE	DESCRIPTION
ApplicationName	255	Empty string	The name of the application that is currently using the connection. This value is stored in DB2 special register CURRENT CLIENT_APPLNAME.
ClientAccountingInformation	255	Empty string	The value of the accounting string from the client information that is specified for the connection. This value is stored in DB2 special register CURRENT CLIENT_ACCTNG.
ClientHostname	255	The value that is set by DB2Connection.setDB2ClientWorkstation. If the value is not set, the default is the host name of the local host.	The host name of the computer on which the application that is using the connection is running. This value is stored in DB2 special register CURRENT CLIENT_WRKSTNNAME.
ClientUser	255	Empty string	The name of the user on whose behalf the application that is using the connection is running. This value is stored in DB2 special register CURRENT CLIENT_USERID.

The following table lists the client info property values that the IBM Data Server Driver for JDBC and SQLJ returns for DB2 for z/OS when the connection uses type 4 connectivity.

Table 5-8. Client info property values for type 4 connectivity to DB2 for z/OS

NAME	MAX_LEN (bytes)	DEFAULT_VALUE	DESCRIPTION
ApplicationName	32	clientProgramName property value, if set. "db2jcc_application" otherwise.	The name of the application that is currently using the connection. This value is stored in DB2 special register CURRENT CLIENT_APPLNAME.
ClientAccountingInformation	200	A string that is the concatenation of the following values: • "JCC*nnnnn*", where *nnnnn* is the driver level, such as 04000. • The value that is set by DB2Connection.setDB2ClientWorkstation. If the value is not set, the default is the host name of the local host. • applicationName property value, if set. 20 blanks otherwise. • clientUser property value, if set. Eight blanks otherwise.	The value of the accounting string from the client information that is specified for the connection. This value is stored in DB2 special register CURRENT CLIENT_ACCTNG.
ClientHostname	18	The value that is set by DB2Connection.setDB2ClientWorkstation. If the value is not set, the default is the host name of the local host.	The host name of the computer on which the application that is using the connection is running. This value is stored in DB2 special register CURRENT CLIENT_WRKSTNNAME.

Table 5-8. Client info property values for type 4 connectivity to DB2 for z/OS (continued)

NAME	MAX_LEN (bytes)	DEFAULT_VALUE	DESCRIPTION
ClientUser	16	The value that is set by DB2Connection.setDB2ClientUser. If the value is not set, the default is the current user ID that is used to connect to the database.	The name of the user on whose behalf the application that is using the connection is running. This value is stored in DB2 special register CURRENT CLIENT_USERID.

The following table lists the client info property values that the IBM Data Server Driver for JDBC and SQLJ returns for DB2 for z/OS when the connection uses type 2 connectivity.

Table 5-9. Client info property values for type 2 connectivity to DB2 for z/OS

NAME	MAX_LEN (bytes)	DEFAULT_VALUE	DESCRIPTION
ApplicationName	32	Empty string	The name of the application that is currently using the connection. This value is stored in DB2 special register CURRENT CLIENT_APPLNAME.
ClientAccountingInformation	200	Empty string	The value of the accounting string from the client information that is specified for the connection. This value is stored in DB2 special register CURRENT CLIENT_ACCTNG.
ClientHostname	18	Empty string	The host name of the computer on which the application that is using the connection is running. This value is stored in DB2 special register CURRENT CLIENT_WRKSTNNAME.
ClientUser	16	Empty string	The name of the user on whose behalf the application that is using the connection is running. This value is stored in DB2 special register CURRENT CLIENT_USERID.

The following table lists the client info property values that the IBM Data Server Driver for JDBC and SQLJ returns for IBM Informix

Table 5-10. Client info property values for IBM Informix

NAME	MAX_LEN (bytes)	DEFAULT_VALUE	DESCRIPTION
ApplicationName	20	Empty string	The name of the application that is currently using the connection.
ClientAccountingInformation	199	Empty string	The value of the accounting string from the client information that is specified for the connection.
ClientHostname	20	The value that is set by DB2Connection.setDB2ClientWorkstation. If the value is not set, the default is the host name of the local host.	The host name of the computer on which the application that is using the connection is running.
ClientUser	1024	Empty string	The name of the user on whose behalf the application that is using the connection is running.

Transaction control in JDBC applications

In JDBC applications, as in other types of SQL applications, transaction control involves explicitly or implicitly committing and rolling back transactions, and setting the isolation level for transactions.

IBM Data Server Driver for JDBC and SQLJ isolation levels

The IBM Data Server Driver for JDBC and SQLJ supports a number of isolation levels, which correspond to database server isolation levels.

JDBC isolation levels can be set for a unit of work within a JDBC program, using the Connection.setTransactionIsolation method. The default isolation level can be set with the defaultIsolationLevel property.

The following table shows the values of *level* that you can specify in the Connection.setTransactionIsolation method and their DB2 database server equivalents.

Table 5-11. Equivalent JDBC and DB2 isolation levels

JDBC value	DB2 isolation level
java.sql.Connection.TRANSACTION_SERIALIZABLE	Repeatable read
java.sql.Connection.TRANSACTION_REPEATABLE_READ	Read stability
java.sql.Connection.TRANSACTION_READ_COMMITTED	Cursor stability
java.sql.Connection.TRANSACTION_READ_UNCOMMITTED	Uncommitted read

The following table shows the values of *level* that you can specify in the Connection.setTransactionIsolation method and their IBM Informix equivalents.

Table 5-12. Equivalent JDBC and IBM Informix isolation levels

JDBC value	IBM Informix isolation level
java.sql.Connection.TRANSACTION_SERIALIZABLE	Repeatable read
java.sql.Connection.TRANSACTION_REPEATABLE_READ	Repeatable read
java.sql.Connection.TRANSACTION_READ_COMMITTED	Committed read
java.sql.Connection.TRANSACTION_READ_UNCOMMITTED	Dirty read
com.ibm.db2.jcc.DB2Connection.TRANSACTION_IDS_CURSOR_STABILITY	IBM Informix cursor stability
com.ibm.db2.jcc.DB2Connection.TRANSACTION_IDS_LAST_COMMITTED	Committed read, last committed

Committing or rolling back JDBC transactions

In JDBC, to commit or roll back transactions explicitly, use the commit or rollback methods.

For example:

```
 Connection con;
 ...
con.commit();
```

If autocommit mode is on, the database manager performs a commit operation after every SQL statement completes. To set autocommit mode on, invoke the Connection.setAutoCommit(true) method. To set autocommit mode off, invoke the

Connection.setAutoCommit(false) method. To determine whether autocommit mode is on, invoke the Connection.getAutoCommit method.

Connections that participate in distributed transactions cannot invoke the setAutoCommit(true) method.

When you change the autocommit state, the database manager executes a commit operation, if the application is not already on a transaction boundary.

While a connection is participating in a distributed transaction, the associated application cannot issue the commit or rollback methods.

Default JDBC autocommit modes

The default autocommit mode depends on the data source to which the JDBC application connects.

Autocommit default for DB2 data sources

For connections to DB2 data sources, the default autocommit mode is `true`.

Autocommit default for IBM Informix data sources

For connections to IBM Informix data sources, the default autocommit mode depends on the type of data source. The following table shows the defaults.

Table 5-13. Default autocommit modes for IBM Informix data sources

Type of data source	Default autocommit mode for local transactions	Default autocommit mode for global transactions
ANSI-compliant database	true	false
Non-ANSI-compliant database without logging	false	not applicable
Non-ANSI-compliant database with logging	true	false

Exceptions and warnings under the IBM Data Server Driver for JDBC and SQLJ

In JDBC applications, SQL errors throw exceptions, which you handle using `try/catch` blocks. SQL warnings do not throw exceptions, so you need to invoke methods to check whether warnings occurred after you execute SQL statements.

The IBM Data Server Driver for JDBC and SQLJ provides the following classes and interfaces, which provide information about errors and warnings.

SQLException

The SQLException class for handling errors. All JDBC methods throw an instance of SQLException when an error occurs during their execution. According to the JDBC specification, an SQLException object contains the following information:

- An `int` value that contains an error code. SQLException.getErrorCode retrieves this value.
- A String object that contains the SQLSTATE, or null. SQLException.getSQLState retrieves this value.

- A String object that contains a description of the error, or null. SQLException.getMessage retrieves this value.
- A pointer to the next SQLException, or null. SQLException.getNextException retrieves this value.

When a JDBC method throws a single SQLException, that SQLException might be caused by an underlying Java exception that occurred when the IBM Data Server Driver for JDBC and SQLJ processed the method. In this case, the SQLException wraps the underlying exception, and you can use the SQLException.getCause method to retrieve information about the error.

DB2Diagnosable

The IBM Data Server Driver for JDBC and SQLJ-only interface com.ibm.db2.jcc.DB2Diagnosable extends the SQLException class. The DB2Diagnosable interface gives you more information about errors that occur when the data source is accessed. If the JDBC driver detects an error, DB2Diagnosable gives you the same information as the standard SQLException class. However, if the database server detects the error, DB2Diagnosable adds the following methods, which give you additional information about the error:

getSqlca
Returns an DB2Sqlca object with the following information:
- An SQL error code
- The SQLERRMC values
- The SQLERRP value
- The SQLERRD values
- The SQLWARN values
- The SQLSTATE

getThrowable
Returns a java.lang.Throwable object that caused the SQLException, or null, if no such object exists.

printTrace
Prints diagnostic information.

SQLException subclasses

If you are using JDBC 4.0 or later, you can obtain more specific information than an SQLException provides by catching the following exception classes:

- SQLNonTransientException

  An SQLNonTransientException is thrown when an SQL operation that failed previously cannot succeed when the operation is retried, unless some corrective action is taken. The SQLNonTransientException class has these subclasses:
  - SQLFeatureNotSupportedException
  - SQLNonTransientConnectionException
  - SQLDataException
  - SQLIntegrityConstraintViolationException
  - SQLInvalidAuthorizationSpecException
  - SQLSyntaxException
- SQLTransientException

  An SQLTransientException is thrown when an SQL operation that failed previously might succeed when the operation is retried, without intervention from the application. A connection is still valid after an SQLTransientException is thrown. The SQLTransientException class has these subclasses:

  - SQLTransientConnectionException
  - SQLTransientRollbackException
  - SQLTimeoutException
- SQLRecoverableException

  An SQLRecoverableException is thrown when an operation that failed previously might succeed if the application performs some recovery steps, and retries the transaction. A connection is no longer valid after an SQLRecoverableException is thrown.
- SQLClientInfoException

  A SQLClientInfoException is thrown by the Connection.setClientInfo method when one or more client properties cannot be set. The SQLClientInfoException indicates which properties cannot be set.

SQLWarning

The IBM Data Server Driver for JDBC and SQLJ accumulates warnings when SQL statements return positive SQLCODEs, and when SQL statements return 0 SQLCODEs with non-zero SQLSTATEs.

Calling getWarnings retrieves an SQLWarning object.

Important: When a call to Statement.executeUpdate or PreparedStatement.executeUpdate affects no rows, the IBM Data Server Driver for JDBC and SQLJ generates an SQLWarning with error code +100.

When a call to ResultSet.next returns no rows, the IBM Data Server Driver for JDBC and SQLJ does not generate an SQLWarning.

A generic SQLWarning object contains the following information:
- A String object that contains a description of the warning, or null
- A String object that contains the SQLSTATE, or null
- An `int` value that contains an error code
- A pointer to the next SQLWarning, or null

Under the IBM Data Server Driver for JDBC and SQLJ, like an SQLException object, an SQLWarning object can also contain IDS-specific information. The IDS-specific information for an SQLWarning object is the same as the IDS-specific information for an SQLException object.

Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ

As in all Java programs, error handling for JDBC applications is done using `try/catch` blocks. Methods throw exceptions when an error occurs, and the code in the `catch` block handles those exceptions.

The basic steps for handling an SQLException in a JDBC program that runs under the IBM Data Server Driver for JDBC and SQLJ are:

1. Give the program access to the com.ibm.db2.jcc.DB2Diagnosable interface and the com.ibm.db2.jcc.DB2Sqlca class. You can fully qualify all references to them, or you can import them:

   ```
   import com.ibm.db2.jcc.DB2Diagnosable;
   import com.ibm.db2.jcc.DB2Sqlca;
   ```

2. Optional: During a connection to a data server, set the retrieveMessagesFromServerOnGetMessage property to `true` if you want full message text from an SQLException.getMessage call.
3. Optional: During a IBM Data Server Driver for JDBC and SQLJ type 2 connectivity connection to a DB2 for z/OS data source, set the extendedDiagnosticLevel property to `EXTENDED_DIAG_MESSAGE_TEXT` (241) if you want extended diagnostic information similar to the information that is provided by the SQL GET DIAGNOSTICS statement from an SQLException.getMessage call.
4. Put code that can generate an SQLException in a `try` block.
5. In the `catch` block, perform the following steps in a loop:
   a. Test whether you have retrieved the last SQLException. If not, continue to the next step.
   b. Optional: For an SQL statement that executes on an IBM Informix data source, execute the com.ibm.db2.jcc.DB2Statement.getIDSSQLStatementOffSet method to determine which columns have syntax errors.

      DB2Statement.getIDSSQLStatementOffSet returns the offset into the SQL statement of the first syntax error.
   c. Optional: For an SQL statement that executes on an IBM Informix data source, execute the SQLException.getCause method to retrieve any ISAM error messages.
      1) If the Throwable that is returned by SQLException.getCause is not null, perform one of the following sets of steps:
         - Issue SQLException.printStackTrace to print an error message that includes the ISAM error message text. The ISAM error message text is preceded by the string "Caused by:".
         - Retrieve the error code and message text for the ISAM message:
           a) Test whether the Throwable is an instance of an SQLException. If so, retrieve the SQL error code from that SQLException.
           b) Execute the Throwable.getMessage method to retrieve the text of the ISAM message.
   d. Check whether any IBM Data Server Driver for JDBC and SQLJ-only information exists by testing whether the SQLException is an instance of DB2Diagnosable. If so:
      1) Cast the object to a DB2Diagnosable object.
      2) Optional: Invoke the DB2Diagnosable.printTrace method to write all SQLException information to a java.io.PrintWriter object.
      3) Invoke the DB2Diagnosable.getThrowable method to determine whether an underlying java.lang.Throwable caused the SQLException.
      4) Invoke the DB2Diagnosable.getSqlca method to retrieve the DB2Sqlca object.
      5) Invoke the DB2Sqlca.getSqlCode method to retrieve an SQL error code value.
      6) Invoke the DB2Sqlca.getSqlErrmc method to retrieve a string that contains all SQLERRMC values, or invoke the DB2Sqlca.getSqlErrmcTokens method to retrieve the SQLERRMC values in an array.
      7) Invoke the DB2Sqlca.getSqlErrp method to retrieve the SQLERRP value.

8) Invoke the DB2Sqlca.getSqlErrd method to retrieve the SQLERRD values in an array.
9) Invoke the DB2Sqlca.getSqlWarn method to retrieve the SQLWARN values in an array.
10) Invoke the DB2Sqlca.getSqlState method to retrieve the SQLSTATE value.
11) Invoke the DB2Sqlca.getMessage method to retrieve error message text from the data source.

e. Invoke the SQLException.getNextException method to retrieve the next SQLException.

The following code demonstrates how to obtain IBM Data Server Driver for JDBC and SQLJ-specific information from an SQLException that is provided with the IBM Data Server Driver for JDBC and SQLJ. The numbers to the right of selected statements correspond to the previously-described steps.

Figure 5-19. Processing an SQLException under the IBM Data Server Driver for JDBC and SQLJ

```
import java.sql.*;                      // Import JDBC API package
import com.ibm.db2.jcc.DB2Diagnosable;  // Import packages for DB2       1
import com.ibm.db2.jcc.DB2Sqlca;        // SQLException support
java.io.PrintWriter printWriter;        // For dumping all SQLException
                                        // information
String url = "jdbc:ids://myhost:9999/myDB:" +                           2
  "retrieveMessagesFromServerOnGetMessage=true;";
                                  // Set properties to retrieve full message
                                  // text
String user = "db2adm";
String password = "db2adm";
java.sql.Connection con =
  java.sql.DriverManager.getConnection (url, user, password)
                                  // Connect to a DB2 for z/OS data source


...
try {                                                                   4
  // Code that could generate SQLExceptions
 ...
} catch(SQLException sqle) {
    while(sqle != null) {             // Check whether there are more    5a
                                      // SQLExceptions to process
 //=====> Optional IBM Data Server Driver for JDBC and SQLJ-only
 // error processing
      if (sqle instanceof DB2Diagnosable) {                             5d
      // Check if IBM Data Server Driver for JDBC and SQLJ-only
      // information exists
          com.ibm.db2.jcc.DB2Diagnosable diagnosable =
            (com.ibm.db2.jcc.DB2Diagnosable)sqle;                       5d1
          diagnosable.printTrace (printWriter, "");                     5d2
          java.lang.Throwable throwable =
            diagnosable.getThrowable();                                 5d3
          if (throwable != null) {
            // Extract java.lang.Throwable information
            // such as message or stack trace.
            ...
          }
          DB2Sqlca sqlca = diagnosable.getSqlca();                      5d4
                                          // Get DB2Sqlca object
          if (sqlca != null) {            // Check that DB2Sqlca is not null
            int sqlCode = sqlca.getSqlCode(); // Get the SQL error code 5d5
            String sqlErrmc = sqlca.getSqlErrmc();                      5d6
```

```
                                        // Get the entire SQLERRMC
            String[] sqlErrmcTokens = sqlca.getSqlErrmcTokens();
                                        // You can also retrieve the
                                        // individual SQLERRMC tokens
            String sqlErrp = sqlca.getSqlErrp();                          5d7
                                        // Get the SQLERRP
            int[] sqlErrd = sqlca.getSqlErrd();                           5d8
                                        // Get SQLERRD fields
            char[] sqlWarn = sqlca.getSqlWarn();                          5d9
                                        // Get SQLWARN fields
            String sqlState = sqlca.getSqlState();                        5d10
                                        // Get SQLSTATE
            String errMessage = sqlca.getMessage();                       5d11
                                        // Get error message

            System.err.println ("Server error message: " + errMessage);

            System.err.println ("--------------- SQLCA ---------------");
            System.err.println ("Error code: " + sqlCode);
            System.err.println ("SQLERRMC: " + sqlErrmc);
            If (sqlErrmcTokens != null) {
              for (int i=0; i< sqlErrmcTokens.length; i++) {
                System.err.println ("  token " + i + ": " + sqlErrmcTokens[i]);
              }
            }
            System.err.println ( "SQLERRP: " + sqlErrp );
            System.err.println (
              "SQLERRD(1): " + sqlErrd[0] + "\n" +
              "SQLERRD(2): " + sqlErrd[1] + "\n" +
              "SQLERRD(3): " + sqlErrd[2] + "\n" +
              "SQLERRD(4): " + sqlErrd[3] + "\n" +
              "SQLERRD(5): " + sqlErrd[4] + "\n" +
              "SQLERRD(6): " + sqlErrd[5] );
            System.err.println (
              "SQLWARN1: " + sqlWarn[0] + "\n" +
              "SQLWARN2: " + sqlWarn[1] + "\n" +
              "SQLWARN3: " + sqlWarn[2] + "\n" +
              "SQLWARN4: " + sqlWarn[3] + "\n" +
              "SQLWARN5: " + sqlWarn[4] + "\n" +
              "SQLWARN6: " + sqlWarn[5] + "\n" +
              "SQLWARN7: " + sqlWarn[6] + "\n" +
              "SQLWARN8: " + sqlWarn[7] + "\n" +
              "SQLWARN9: " + sqlWarn[8] + "\n" +
              "SQLWARNA: " + sqlWarn[9] );
            System.err.println ("SQLSTATE: " + sqlState);
                                        // portion of SQLException
          }
        sqle=sqle.getNextException();     // Retrieve next SQLException      5e
      }
}
```

The following code demonstrates how to obtain the location of a syntax error in an SQL statement from an SQLException.

```
...
Statement stmt=null;
try {
 stmt = con.createStatement();
 stmt.execute("select * fro tab1");
                                        // This statement has a syntax error
                                        // at offset 10
}
catch(SQLException e)
{
```

```
  System.out.println ("Error offset :"+
    ((DB2Statement) stmt).getIDSSQLStatementOffSet());
                                        // This code prints Error offset : 10
}
```

The following code demonstrates how to obtain the ISAM error text from an SQLException.

```
...
try
{
 // Execute an SQL statement
}
catch (SQLException e)
{
  SQLException eNext = e;
  while (eNext != null) {
    System.out.println("SQLCODE: "
      + eNext.getErrorCode()
      + "  " + eNext.getMessage());     // Get the error code and message
                                        // text from the SQLException
    Throwable cause = eNext.getCause(); // Get Throwable with ISAM text
    if (cause != null) {
      if (cause instanceof SQLException)
          System.out.print("SQLCODE: "
          + ((SQLException) cause).getErrorCode() + " ");
                                        // If the Throwable is an SQLException,
                                        // get the error code
      System.out.println(cause.getMessage());
                                        // Get the ISAM message text
    }
  }
  eNext = eNext.getNextException();
}
```

Handling an SQLWarning under the IBM Data Server Driver for JDBC and SQLJ

Unlike SQL errors, SQL warnings do not cause JDBC methods to throw exceptions. Instead, the Connection, Statement, PreparedStatement, `CallableStatement`, and ResultSet classes contain getWarnings methods, which you need to invoke after you execute SQL statements to determine whether any SQL warnings were generated.

The basic steps for retrieving SQL warning information are:

1. Optional: During connection to the database server, set properties that affect SQLWarning objects.

   If you want full message text from a data server when you execute SQLWarning.getMessage calls, set the retrieveMessagesFromServerOnGetMessage property to `true`.

   If you are using IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to a DB2 for z/OS data source, and you want extended diagnostic information that is similar to the information that is provided by the SQL GET DIAGNOSTICS statement when you execute SQLWarning.getMessage calls, set the extendedDiagnosticLevel property to `EXTENDED_DIAG_MESSAGE_TEXT` (241).
2. Immediately after invoking a method that connects to a database server or executes an SQL statement, invoke the getWarnings method to retrieve an SQLWarning object.
3. Perform the following steps in a loop:
   a. Test whether the SQLWarning object is null. If not, continue to the next step.

b. Invoke the SQLWarning.getMessage method to retrieve the warning description.
c. Invoke the SQLWarning.getSQLState method to retrieve the SQLSTATE value.
d. Invoke the SQLWarning.getErrorCode method to retrieve the error code value.
e. If you want IDS-specific warning information, perform the same steps that you perform to get IDS-specific information for an SQLException.
f. Invoke the SQLWarning.getNextWarning method to retrieve the next SQLWarning.

The following code illustrates how to obtain generic SQLWarning information. The numbers to the right of selected statements correspond to the previously-described steps.

```
String url = "jdbc:ids://myhost:9999/informixdb:" +                    1
  "retrieveMessagesFromServerOnGetMessage=true;";
                                  // Set properties to retrieve full message
                                  // text
String user = "idsadm";
String password = "idsadm";
java.sql.Connection con =
  java.sql.DriverManager.getConnection (url, user, password)
                                  // Connect to an Informix data source
SQLWarning warn = con.getWarnings();
  while (warn != null) {
       System.out.println(" SQLMESSAGE : " + warn.getMessage ());
        warn = warn.getNextWarning();

Statement stmt;
ResultSet rs;
SQLWarning sqlwarn;
...
stmt = con.createStatement();     // Create a Statement object
rs = stmt.executeQuery("SELECT * FROM EMPLOYEE");
                                  // Get the result table from the query
sqlwarn = stmt.getWarnings();     // Get any warnings generated          2
while (sqlwarn != null) {         // While there are warnings, get and   3a
                                  // print warning information
  System.out.println ("Warning description: " + sqlwarn.getMessage());   3b
  System.out.println ("SQLSTATE: " + sqlwarn.getSQLState());             3c
  System.out.println ("Error code: " + sqlwarn.getErrorCode());         3d
  sqlwarn=sqlwarn.getNextWarning();     // Get next SQLWarning           3f
}
```

Figure 5-20. Example of processing an SQLWarning

Retrieving information from a BatchUpdateException

When an error occurs during execution of a statement in a batch, processing continues. However, executeBatch throws a BatchUpdateException.

To retrieve information from the BatchUpdateException, follow these steps:

1. Use the BatchUpdateException.getUpdateCounts method to determine the number of rows that each SQL statement in the batch updated before the exception was thrown.

   getUpdateCount returns an array with an element for each statement in the batch. An element has one of the following values:

   n The number of rows that the statement updated.

Statement.SUCCESS_NO_INFO

This value is returned if the number of updated rows cannot be determined. The number of updated rows cannot be determined if the following conditions are true:

- The application is connected to a subsystem that is in DB2 for z/OS Version 8 new-function mode, or later.
- The application is using Version 3.1 or later of the IBM Data Server Driver for JDBC and SQLJ.
- The IBM Data Server Driver for JDBC and SQLJ uses multi-row INSERT operations to execute batch updates.

Statement.EXECUTE_FAILED

This value is returned if the statement did not execute successfully.

2. If the batched statement can return automatically generated keys:
   a. Cast the BatchUpdateException to a com.ibm.db2.jcc.DBBatchUpdateException.
   b. Call the DBBatchUpdateException.getDBGeneratedKeys method to retrieve an array of ResultSet objects that contains the automatically generated keys for each execution of the batched SQL statement.
   c. Test whether each ResultSet in the array is null.

      Each ResultSet contains:

      - If the ResultSet is not null, it contains the automatically generated keys for an execution of the batched SQL statement.
      - If the ResultSet is null, execution of the batched statement failed.
3. Use SQLException methods getMessage, getSQLState, and getErrorCode to retrieve the description of the error, the SQLSTATE, and the error code for the first error.
4. Use the BatchUpdateException.getNextException method to get a chained SQLException.
5. In a loop, execute the getMessage, getSQLState, getErrorCode, and getNextException method calls to obtain information about an SQLException and get the next SQLException.

The following code fragment demonstrates how to obtain the fields of a BatchUpdateException and the chained SQLException objects for a batched statement that returns automatically generated keys. The example assumes that there is only one column in the automatically generated key, and that there is always exactly one key value, whose data type is numeric. The numbers to the right of selected statements correspond to the previously-described steps.

```
try {
  // Batch updates
} catch(BatchUpdateException buex) {
    System.err.println("Contents of BatchUpdateException:");
    System.err.println(" Update counts: ");
    int [] updateCounts = buex.getUpdateCounts();             1
    for (int i = 0; i < updateCounts.length; i++) {
      System.err.println("  Statement " + i + ":" + updateCounts[i]);
    }
    ResultSet[] resultList =
      ((DBBatchUpdateException)buex).getDBGeneratedKeys();    2
    for (i = 0; i < resultList.length; i++)
    {
      if (resultList[i] == null)
         continue; // Skip the ResultSet for which there was a failure
      else {
        rs.next();
```

```
        java.math.BigDecimal idColVar = rs.getBigDecimal(1);
                                        // Get automatically generated key
                                        // value
        System.out.println("Automatically generated key value = " + idColVar);
      }
    }
    System.err.println(" Message: " + buex.getMessage());     3
    System.err.println(" SQLSTATE: " + buex.getSQLState());
    System.err.println(" Error code: " + buex.getErrorCode());
    SQLException ex = buex.getNextException();                4
    while (ex != null) {                                      5
      System.err.println("SQL exception:");
      System.err.println(" Message: " + ex.getMessage());
      System.err.println(" SQLSTATE: " + ex.getSQLState());
      System.err.println(" Error code: " + ex.getErrorCode());
      ex = ex.getNextException();
    }
}
```

Disconnecting from data sources in JDBC applications

When you have finished with a connection to a data source, it is *essential* that you close the connection to the data source. Doing this releases the Connection object's database and JDBC resources immediately.

To close the connection to the data source, use the close method. For example:

```
Connection con;
...
con.close();
```

For a connection to a DB2 data source, if autocommit mode is not on, the connection needs to be on a unit-of-work boundary before you close the connection.

For a connection to an IBM Informix database, if the database supports logging, and autocommit mode is not on, the connection needs to be on a unit-of-work boundary before you close the connection.

Chapter 6. SQLJ application programming

Writing an SQLJ application has much in common with writing an SQL application in any other language.

In general, you need to do the following things:
- Import the Java packages that contain SQLJ and JDBC methods.
- Declare variables for sending data to or retrieving data from IDS tables.
- Connect to a data source.
- Execute SQL statements.
- Handle SQL errors and warnings.
- Disconnect from the data source.

Although the tasks that you need to perform are similar to those in other languages, the way that you execute those tasks, and the order in which you execute those tasks, is somewhat different.

Example of a simple SQLJ application

A simple SQLJ application demonstrates the basic elements that JDBC applications need to include.

Figure 6-1. Simple SQLJ application

```
import sqlj.runtime.*;                                              1
import java.sql.*;

#sql context EzSqljCtx;                                             3a
#sql iterator EzSqljNameIter (String LASTNAME);                     4a

public class EzSqlj {
  public static void main(String args[])
    throws SQLException
  {
    EzSqljCtx ctx = null;
    String URLprefix = "jdbc:ids:";
    String url;
    url = new String(URLprefix + args[0]);
                                          // Location name is an input parameter
    String hvmgr="000010";                                          2
    String hvdeptno="A00";
    try {                                                           3b
      Class.forName("com.ibm.db2.jcc.DB2Driver");
    } catch (Exception e)
    {
       throw new SQLException("Error in EzSqlj: Could not load the driver");
    }
    try
    {
       System.out.println("About to connect using url: " + url);
       Connection con0 = DriverManager.getConnection(url);          3c
                                          // Create a JDBC Connection
       con0.setAutoCommit(false);         // set autocommit OFF
       ctx = new EzSqljCtx(con0);                                   3d

       try
       {
```

```
        EzSqljNameIter iter;
        int count=0;

        #sql [ctx] iter =
          {SELECT LASTNAME FROM EMPLOYEE};                          4b
                                          // Create result table of the SELECT
        while (iter.next()) {                                       4c
           System.out.println(iter.LASTNAME());
                                          // Retrieve rows from result table
           count++;
        }
        System.out.println("Retrieved " + count + " rows of data");
        iter.close();                     // Close the iterator
      }
      catch( SQLException e )                                       5
      {
         System.out.println ("**** SELECT SQLException...");
         while(e!=null) {
           System.out.println ("Error msg: " + e.getMessage());
           System.out.println ("SQLSTATE: " + e.getSQLState());
           System.out.println ("Error code: " + e.getErrorCode());
           e = e.getNextException(); // Check for chained exceptions
         }
      }
      catch( Exception e )
      {
         System.out.println("**** NON-SQL exception   = " + e);
         e.printStackTrace();
      }
      try
      {
        #sql [ctx]                                                  4d
          {UPDATE DEPARTMENT SET MGRNO=:hvmgr
             WHERE DEPTNO=:hvdeptno};   // Update data for one department
                                                                    6
        #sql [ctx] {COMMIT};            // Commit the update
      }
      catch( SQLException e )
      {
         System.out.println ("**** UPDATE SQLException...");
         System.out.println ("Error msg: " + e.getMessage() + ". SQLSTATE=" +
           e.getSQLState() + " Error code=" + e.getErrorCode());
         e.printStackTrace();
      }
      catch( Exception e )
      {
         System.out.println("**** NON-SQL exception   = " + e);
         e.printStackTrace();
      }
      ctx.close();                                                  7
  }
  catch(SQLException e)
  {
     System.out.println ("**** SQLException ...");
     System.out.println ("Error msg: " + e.getMessage() + ". SQLSTATE=" +
       e.getSQLState() + " Error code=" + e.getErrorCode());
     e.printStackTrace();
  }
  catch(Exception e)
  {
    System.out.println ("**** NON-SQL exception = " + e);
    e.printStackTrace();
  }

}
```

Notes to Figure 6-1 on page 6-1:

Note	Description
1	These statements import the java.sql package, which contains the JDBC core API, and the sqlj.runtime package, which contains the SQLJ API. For information on other packages or classes that you might need to access, see "Java packages for SQLJ support".
2	`String` variables hvmgr and hvdeptno are *host identifiers*, which are equivalent to IDS host variables. See "Variables in SQLJ applications" for more information.
3a, 3b, 3c, and 3d	These statements demonstrate how to connect to a data source using one of the three available techniques. See "Connecting to a data source using SQLJ" for more details. Step 3b (loading the JDBC driver) is not necessary if you use JDBC 4.0.
4a , 4b, 4c, and 4d	These statements demonstrate how to execute SQL statements in SQLJ. Statement 4a demonstrates the SQLJ equivalent of declaring an SQL cursor. Statements 4b and 4c show one way of doing the SQLJ equivalent of executing an SQL OPEN CURSOR and SQL FETCHes. Statement 4d shows how to do the SQLJ equivalent of performing an SQL UPDATE. For more information, see "SQL statements in an SQLJ application".
5	This `try/catch` block demonstrates the use of the SQLException class for SQL error handling. For more information on handling SQL errors, see "Handling SQL errors in an SQLJ application". For more information on handling SQL warnings, see "Handling SQL warnings in an SQLJ application".
6	This is an example of a comment. For rules on including comments in SQLJ programs, see "Comments in an SQLJ application".
7	This statement closes the connection to the data source. See "Closing the connection to the data source in an SQLJ application".

Connecting to a data source using SQLJ

In an SQLJ application, as in any other IDS application, you must be connected to a data source before you can execute SQL statements.

You can use one five techniques to connect to a data source in an SQLJ program. Two use the JDBC DriverManager interface, two use the JDBC DataSource interface, and one uses a previously created connection context. Connections to IBM Informix must use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

SQLJ connection technique 1: JDBC DriverManager interface

SQLJ connection technique 1 uses the JDBC DriverManager interface as the underlying means for creating the connection.

To use SQLJ connection technique 1, follow these steps:

1. Execute an SQLJ *connection declaration clause*.

   Doing this generates a *connection context class*. The simplest form of the connection declaration clause is:

   ```
   #sql context context-class-name;
   ```

   The name of the generated connection context class is *context-class-name*.
2. Load a JDBC driver by invoking the Class.forName method.
   - Class.forName this way:

     ```
     Class.forName("com.ibm.db2.jcc.DB2Driver");
     ```

     This step is unnecessary if you use the JDBC 4.0 driver.

3. Invoke the constructor for the connection context class that you created in step 1 on page 6-3.

   Doing this creates a connection context object that you specify in each SQL statement that you execute at the associated data source. The constructor invocation statement needs to be in one of the following forms:

```
connection-context-class connection-context-object=
  new connection-context-class(String url, boolean autocommit);

connection-context-class connection-context-object=
  new connection-context-class(String url, String user,
    String password, boolean autocommit);
connection-context-class connection-context-object=
  new connection-context-class(String url, Properties info,
    boolean autocommit);
```

   The meanings of the parameters are:

   url
   : A string that specifies the location name that is associated with the data source. That argument has one of the forms that are specified in "Connect to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ". The form depends on which JDBC driver you are using.

   user **and** *password*
   : Specify a user ID and password for connection to the data source, if the data source to which you are connecting requires them.

   info
   : Specifies an object of type java.util.Properties that contains a set of driver properties for the connection. For the IBM Data Server Driver for JDBC and SQLJ, you can specify any of the properties listed in "Properties for the IBM Data Server Driver for JDBC and SQLJ".

   autocommit
   : Specifies whether you want the database manager to issue a COMMIT after every statement. Possible values are `true` or `false`. If you specify `false`, you need to do explicit commit operations.

The following code uses connection technique 1 to create a connection to location NEWYORK. The connection requires a user ID and password, and does not require autocommit. The numbers to the right of selected statements correspond to the previously-described steps.

```
#sql context Ctx;            // Create connection context class Ctx        1
String userid="dbadm";       // Declare variables for user ID and password
String password="dbadm";
String empname;              // Declare a host variable
...
try {                        // Load the JDBC driver
  Class.forName("com.ibm.db2.jcc.DB2Driver");                              2
}
catch (ClassNotFoundException e) {
   e.printStackTrace();
}
Ctx myConnCtx=                                                             3
  new Ctx("jdbc:ids://sysmvs1.stl.ibm.com:5021/NEWYORK",
  userid,password,false);    // Create connection context object myConnCtx
                             // for the connection to NEWYORK
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
   WHERE EMPNO='000010'};
                             // Use myConnCtx for executing an SQL statement
```

Figure 6-2. Using connection technique 1 to connect to a data source

SQLJ connection technique 2: JDBC DriverManager interface

SQLJ connection technique 2 uses the JDBC DriverManager interface as the underlying means for creating the connection.

To use SQLJ connection technique 2, follow these steps:

1. Execute an SQLJ *connection declaration clause*.

   Doing this generates a *connection context class*. The simplest form of the connection declaration clause is:

   ```
   #sql context context-class-name;
   ```

   The name of the generated connection context class is *context-class-name*.

2. Load a JDBC driver by invoking the Class.forName method.

   - Class.forName this way:

     ```
     Class.forName("com.ibm.db2.jcc.DB2Driver");
     ```

     This step is unnecessary if you use the JDBC 4.0 driver.

3. Invoke the JDBC DriverManager.getConnection method.

   Doing this creates a JDBC connection object for the connection to the data source. You can use any of the forms of getConnection that are specified in "Connect to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ".

   The meanings of the *url*, *user*, and *password* parameters are:

   url
   : A string that specifies the location name that is associated with the data source. That argument has one of the forms that are specified in "Connect to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ". The form depends on which JDBC driver you are using.

   user **and** *password*
   : Specify a user ID and password for connection to the data source, if the data source to which you are connecting requires them.

4. Invoke the constructor for the connection context class that you created in step 1

Doing this creates a connection context object that you specify in each SQL statement that you execute at the associated data source. The constructor invocation statement needs to be in the following form:

```
connection-context-class connection-context-object=
  new connection-context-class(Connection JDBC-connection-object);
```

The *JDBC-connection-object* parameter is the Connection object that you created in step 3 on page 6-5.

The following code uses connection technique 2 to create a connection to location NEWYORK. The connection requires a user ID and password, and does not require autocommit. The numbers to the right of selected statements correspond to the previously-described steps.

```
#sql context Ctx;           // Create connection context class Ctx        1
String userid="dbadm";      // Declare variables for user ID and password
String password="dbadm";
String empname;             // Declare a host variable
...
try {                       // Load the JDBC driver
  Class.forName("com.ibm.db2.jcc.DB2Driver");                            2
}
catch (ClassNotFoundException e) {
   e.printStackTrace();
}
Connection jdbccon=                                                      3
  DriverManager.getConnection("jdbc:ids://sysmvs1.stl.ibm.com:5021/NEWYORK",
    userid,password);
                            // Create JDBC connection object jdbccon
jdbccon.setAutoCommit(false); // Do not autocommit
Ctx myConnCtx=new Ctx(jdbccon);                                          4
                            // Create connection context object myConnCtx
                            // for the connection to NEWYORK
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
   WHERE EMPNO='000010'};
                            // Use myConnCtx for executing an SQL statement
```

Figure 6-3. Using connection technique 2 to connect to a data source

SQLJ connection technique 3: JDBC DataSource interface

SQLJ connection technique 3 uses the JDBC DataSource as the underlying means for creating the connection.

To use SQLJ connection technique 3, follow these steps:

1. Execute an SQLJ *connection declaration clause*.

   Doing this generates a *connection context class*. The simplest form of the connection declaration clause is:

   ```
   #sql context context-class-name;
   ```

   The name of the generated connection context class is *context-class-name*.

2. If your system administrator created a DataSource object in a different program, follow these steps. Otherwise, create a DataSource object and assign properties to it.
   a. Obtain the logical name of the data source to which you need to connect.
   b. Create a context to use in the next step.
   c. In your application program, use the Java Naming and Directory Interface (JNDI) to get the DataSource object that is associated with the logical data source name.

3. Invoke the JDBC DataSource.getConnection method.

   Doing this creates a JDBC connection object for the connection to the data source. You can use one of the following forms of getConnection:

   ```
   getConnection();
   getConnection(user, password);
   ```

   The meanings of the *user* and *password* parameters are:

   user **and** *password*
   Specify a user ID and password for connection to the data source, if the data source to which you are connecting requires them.

4. If the default autocommit mode is not appropriate, invoke the JDBC Connection.setAutoCommit method.

   Doing this indicates whether you want the database manager to issue a COMMIT after every statement. The form of this method is:

   ```
   setAutoCommit(boolean autocommit);
   ```

5. Invoke the constructor for the connection context class that you created in step 1 on page 6-6.

   Doing this creates a connection context object that you specify in each SQL statement that you execute at the associated data source. The constructor invocation statement needs to be in the following form:

   ```
   connection-context-class connection-context-object=
     new connection-context-class(Connection JDBC-connection-object);
   ```

   The *JDBC-connection-object* parameter is the Connection object that you created in step 3.

The following code uses connection technique 3 to create a connection to a location with logical name jdbc/sampledb. This example assumes that the system administrator created and deployed a DataSource object that is available through JNDI lookup. The numbers to the right of selected statements correspond to the previously-described steps.

```
import java.sql.*;
import javax.naming.*;
import javax.sql.*;
...
#sql context CtxSqlj;       // Create connection context class CtxSqlj   1
Context ctx=new InitialContext();                                         2b
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");                    2c
Connection con=ds.getConnection();                                        3
String empname;             // Declare a host variable
...
con.setAutoCommit(false);   // Do not autocommit                          4
CtxSqlj myConnCtx=new CtxSqlj(con);                                       5
                            // Create connection context object myConnCtx
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
   WHERE EMPNO='000010'};
                            // Use myConnCtx for executing an SQL statement
```

Figure 6-4. Using connection technique 3 to connect to a data source

SQLJ connection technique 4: JDBC DataSource interface

SQLJ connection technique 4 uses the JDBC DataSource as the underlying means for creating the connection. This technique **requires** that the DataSource is registered with JNDI.

To use SQLJ connection technique 4, follow these steps:

1. From your system administrator, obtain the logical name of the data source to which you need to connect.
2. Execute an SQLJ connection declaration clause.

   For this type of connection, the connection declaration clause needs to be of this form:

   ```
   #sql public static context context-class-name
    with (dataSource="logical-name");
   ```

   The connection context must be declared as public and static. *logical-name* is the data source name that you obtained in step 1.
3. Invoke the constructor for the connection context class that you created in step 2.

   Doing this creates a connection context object that you specify in each SQL statement that you execute at the associated data source. The constructor invocation statement needs to be in one of the following forms:

   ```
   connection-context-class connection-context-object=
     new connection-context-class();

   connection-context-class connection-context-object=
     new connection-context-class (String user,
       String password);
   ```

   The meanings of the *user* and *password* parameters are:

   user **and** *password*
   : Specify a user ID and password for connection to the data source, if the data source to which you are connecting requires them.

The following code uses connection technique 4 to create a connection to a location with logical name jdbc/sampledb. The connection requires a user ID and password.

```
#sql public static context Ctx
  with (dataSource="jdbc/sampledb");                                    2
                               // Create connection context class Ctx
String userid="dbadm";         // Declare variables for user ID and password
String password="dbadm";

String empname;                // Declare a host variable
...
Ctx myConnCtx=new Ctx(userid, password);                                3
                               // Create connection context object myConnCtx
                               // for the connection to jdbc/sampledb
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
   WHERE EMPNO='000010'};
                               // Use myConnCtx for executing an SQL statement
```

Figure 6-5. Using connection technique 4 to connect to a data source

SQLJ connection technique 5: Use a previously created connection context

SQLJ connection technique 5 uses a previously created connection context to connect to the data source.

In general, one program declares a connection context class, creates connection contexts, and passes them as parameters to other programs. A program that uses the connection context invokes a constructor with the passed connection context object as its argument.

Program CtxGen.sqlj declares connection context Ctx and creates instance oldCtx:

```
#sql context Ctx;
...
// Create connection context object oldCtx
```

Program test.sqlj receives oldCtx as a parameter and uses oldCtx as the argument of its connection context constructor:

```
void useContext(sqlj.runtime.ConnectionContext oldCtx)
                                    // oldCtx was created in CtxGen.sqlj
{
  Ctx myConnCtx=
    new Ctx(oldCtx);                // Create connection context object myConnCtx
                                    // from oldCtx
  #sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
     WHERE EMPNO='000010'};
                                    // Use myConnCtx for executing an SQL statement
...
}
```

Java packages for SQLJ support

Before you can execute SQLJ statements or invoke JDBC methods in your SQLJ program, you need to be able to access all or parts of various Java packages that contain support for those statements.

You can do that either by importing the packages or specific classes, or by using fully-qualified class names. You might need the following packages or classes for your SQLJ program:

sqlj.runtime
Contains the SQLJ run-time API.

java.sql
Contains the core JDBC API.

com.ibm.db2.jcc
Contains the driver-specific implementation of JDBC and SQLJ.

javax.naming
Contains methods for performing Java Naming and Directory Interface (JNDI) lookup.

javax.sql
Contains methods for creating DataSource objects.

Variables in SQLJ applications

In IDS programs in other languages, you use host variables to pass data between the application program and IDS. In SQLJ programs, In SQLJ programs, you can use host variables or *host expressions*.

A host expression begins with a colon (:). The colon is followed by an optional parameter mode identifier (IN, OUT, or INOUT), which is followed by a parenthesized expression clause.

Host variables and host expressions are case sensitive.

A complex expression is an array element or Java expression that evaluates to a single value. A complex expression in an SQLJ clause must be surrounded by parentheses.

The following examples demonstrate how to use host expressions.

Example: Declaring a Java identifier and using it in a SELECT statement:

In this example, the statement that begins with `#sql` has the same function as a SELECT statement in other languages. This statement assigns the last name of the employee with employee number 000010 to Java identifier `empname`.

```
String empname;
...
#sql [ctxt]
  {SELECT LASTNAME INTO :empname FROM EMPLOYEE WHERE EMPNO='000010'};
```

Example: Declaring a Java identifier and using it in a stored procedure call:

In this example, the statement that begins with `#sql` has the same function as an SQL CALL statement in other languages. This statement uses Java identifier `empno` as an input parameter to stored procedure `A`. The keyword `IN`, which precedes `empno`, specifies that `empno` is an input parameter. For a parameter in a CALL statement, `IN` is the default. The explicit or default qualifier that indicates how the parameter is used (IN, OUT, or INOUT) must match the corresponding value in the parameter definition that you specified in the CREATE PROCEDURE statement for the stored procedure.

```
String empno = "0000010";
...
#sql [ctxt] {CALL A (:IN empno)};
```

Example: Using a complex expression as a host identifier:

This example uses complex expression (((int)yearsEmployed++/5)*500) as a host expression.

```
#sql [ctxt] {UPDATE EMPLOYEE
        SET BONUS=:(((int)yearsEmployed++/5)*500) WHERE EMPNO=:empID};
```

SQLJ performs the following actions when it processes a complex host expression:

- Evaluates each of the host expressions in the statement, from left to right, before assigning their respective values to the database.
- Evaluates side effects, such as operations with postfix operators, according to normal Java rules. All host expressions are fully evaluated before any of their values are passed to IDS.
- Uses Java rules for rounding and truncation.

Therefore, if the value of yearsEmployed is 6 before the UPDATE statement is executed, the value that is assigned to column BONUS by the UPDATE statement is ((int)6/5)*500, or 500. After 500 is assigned to BONUS, the value of yearsEmployed is incremented.

Restrictions on variable names: Two strings have special meanings in SQLJ programs. Observe the following restrictions when you use these strings in your SQLJ programs:

- The string __sJT_ is a reserved prefix for variable names that are generated by SQLJ. Do not begin the following types of names with __sJT_:
  - Host expression names
  - Java variable names that are declared in blocks that include executable SQL statements
  - Names of parameters for methods that contain executable SQL statements

  - Names of fields in classes that contain executable SQL statements, or in classes with subclasses or enclosed classes that contain executable SQL statements
- The string _SJ is a reserved suffix for resource files and classes that are generated by SQLJ. Avoid using the string _SJ in class names and input source file names.

Indicator variables in SQLJ applications

In SQLJ programs, you can use indicator variables to pass the NULL value to or from a data server, to pass the default value for a column to the data server, or to indicate that a host variable value is unassigned.

A host variable or host expression can be followed by an indicator variable. An indicator variable begins with a colon (:) and has the data type short. For input, an indicator variable indicates whether the corresponding host variable or host expression has the default value, a non-null value, the null value, or is unassigned. An unassigned variable in an SQL statement yields the same results as if the variable and its target column were not in the SQL statement. For output, the indicator variable indicates where the corresponding host variable or host expression has a non-null value or a null value.

In SQLJ programs, indicator variables that indicate a null value perform the same function as assigning the Java null value to a table column. However, you need to use an indicator variable to retrieve the SQL NULL value from a table into a host variable.

You can use indicator variables that assign the default value or the unassigned value to columns to simplify the coding in your applications. For example, if a table has four columns, and you might need to update any combination of those columns, without the use of default indicator variables or unassigned indicator variables, you need 15 UPDATE statements to perform all possible combinations of updates. With default indicator variables and unassigned indicator variables, you can use one UPDATE statement with all four columns in the SET statement to perform all possible updates. You use the indicator variables to indicate which columns you want to set to their default values, and which columns you do not want to update.

For input, SQLJ supports the use of indicator variables for INSERT, UPDATE, or MERGE statements.

If you customize your SQLJ application, you can assign one of the following values to an indicator variable in an SQLJ application to specify the type of the corresponding input host variable.

Indicator value	Equivalent constant	Meaning of value
-1	sqlj.runtime.ExecutionContext.DBNull	Null
-2, -3, -4, -6		
-5	sqlj.runtime.ExecutionContext.DBDefault	Default
-7	sqlj.runtime.ExecutionContext.DBUnassigned	Unassigned
short-value >=0	sqlj.runtime.ExecutionContext.DBNonNull	Non-null

If you do not customize the application, you can assign one of the following values to an indicator variable to specify the type of the corresponding input host variable.

Indicator value	Equivalent constant	Meaning of value
-1	sqlj.runtime.ExecutionContext.DBNull	Null
-7 <= *short-value* < -1		
0	sqlj.runtime.ExecutionContext.DBNonNull	Non-null
short-value >0		

For output, SQLJ supports the use of indicator variables for the following statements:

- CALL with OUT or INOUT parameters
- FETCH *iterator* INTO *host-variable*
- SELECT ... INTO *host-variable-1,...host-variable-n*

SQLJ assigns one of the following values to an indicator variable to indicate whether an SQL NULL value was retrieved into the corresponding host variable.

Indicator value	Equivalent constant	Meaning of value
-1	sqlj.runtime.ExecutionContext.DBNull	Retrieved value is SQL NULL
0		Retrieved value is not SQL NULL

You cannot use indicator variables to update result sets. To assign null values or default values to result sets, or to indicate that columns are unassigned, call ResultSet.updateObject on the underlying JDBC ResultSet objects of the SQLJ iterators.

The following examples demonstrate how to use indicator variables.

All examples require that the data server supports extended indicators.

Example of using indicators to assign the default value to columns during an INSERT:

In this example, the MGRNO and LOCATION columns need to be set to their default values. To do this, the code performs these steps:

1. Assigns the value ExecutionContext.DBNonNull to the indicator variables (deptInd, dNameInd, rptDeptInd) for the input host variables (dept, dName, rptDept) that send non-default values to the target columns.
2. Assigns the value ExecutionContext.DBDefault to the indicator variables (mgrInd, locnInd) for the input host variables (mgr, locn) that send default values to the target columns.
3. Executes an INSERT statement with the host variable and indicator variable pairs as input.

The numbers to the right of selected statements correspond to the previously described steps.

```
import sqlj.runtime.*;
...
String dept = "F01";
String dName = "SHIPPING";
String rptDept = "A00";
```

```
String mgr, locn = null;
short deptInd, dNameInd, mgrInd, rptDeptInd, locnInd;
// Set indicator variables for dept, dName, rptDept to non-null
deptInd = dNameInd = rptDeptInd = ExecutionContext.DBNonNull;    1
mgrInd = ExecutionContext.DBDefault;                             2
locnInd = ExecutionContext.DBDefault;
#sql [ctxt]                                                      3
  {INSERT INTO DEPARTMENT
    (DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION)
    VALUES (:dept :deptInd, :dName :dNameInd,:mgr :mgrInd,
    :rptDept :rptDeptInd, :locn :locnInd)};
```

Example of using indicators to assign the default value to leave column values unassigned during an UPDATE:

In this example, in rows for department F01, the MGRNO column needs to be set to its default value, the DEPTNAME column value needs to be changed to RECEIVING, and the DEPTNO, DEPTNAME, ADMRDEPT, and LOCATION columns need to remain unchanged. To do this, the code performs these steps:

1. Assigns the new value for the DEPTNAME column to the dName input host variable.
2. Assigns the value ExecutionContext.DBDefault to the indicator variable (mgrInd) for the input host variable (mgr) that sends the default value to the target column.
3. Assigns the value ExecutionContext.DBUnassigned to the indicator variables (deptInd, dNameInd, rptDeptInd, and locnInd) for the input host variables (dept, dName, rptDept, and locn) that need to remain unchanged by the UPDATE operation.
4. Executes an UPDATE statement with the host variable and indicator variable pairs as input.

The numbers to the right of selected statements correspond to the previously described steps.

```
import sqlj.runtime.*;
...
String dept = null;
String dName = "RECEIVING";                                      1
String rptDept = null;
String mgr, locn = null;
short deptInd, dNameInd, mgrInd, rptDeptInd, locnInd;
dNameInd = ExecutionContext.DBNonNull;
mgrInd = ExecutionContext.DBDefault;                             2
deptInd = rptDeptInd = locnInd = ExecutionContext.DBUnassigned;  3
#sql [ctxt]                                                      4
  {UPDATE DEPARTMENT
    SET DEPTNO = :dept :deptInd,
        DEPTNAME = :dName :dNameInd,
        MGRNO = :mgr :mgrInd,
        ADMRDEPT = :rptDept :rptDeptInd,
        LOCATION = :locn :locnInd
    WHERE DEPTNO = "F01"
  };
```

Example of using indicators to retrieve NULL values from columns:

In this example, the HIREDATE column can return the NULL value. To handle this case, the code performs these steps:

1. Defines an indicator variable to indicate when the NULL value is returned from HIREDATE.

2. Executes FETCH statements with the host variable and indicator variable pairs as output.
3. Checks the indicator variable to determine whether a NULL value was returned.

The numbers to the right of selected statements correspond to the previously described steps.

```
import sqlj.runtime.*;
...
#sql iterator ByPos(String, Date);  // Declare positioned iterator ByPos
{
  ...
  ByPos positer;                    // Declare object of ByPos class
  String name = null;               // Declare host variables
  Date hrdate = null;
  short indhrdate = null;           // Declare indicator variable      1
  #sql [ctxt] positer =
    {SELECT LASTNAME, HIREDATE FROM EMPLOYEE};
                                    // Assign the result table of the SELECT
                                    // to iterator object positer
  #sql {FETCH :positer INTO :name, :hrdate :indhrdate };               2
                                    // Retrieve the first row
  while (!positer.endFetch())       // Check whether the FETCH returned a row
  { if(indhrdate == ExecutionContext.DBNonNull {                       3
     System.out.println(name + " was hired in " +
      hrdate); }
    else {
     System.out.println(name + " has no hire date "); }
    #sql {FETCH :positer INTO :name, :hrdate };
                                    // Fetch the next row
  }
  positer.close();                  // Close the iterator              5
}
```

Example of assigning default values to result set columns:

In this example, the HIREDATE column in a result set needs to be set to its default value. To do this, the code performs these steps:

1. Retrieves the underlying ResultSet from the iterator that holds the retrieved data.
2. Executes the ResultSet.updateObject method with the DB2PreparedStatement.DB_PARAMETER_DEFAULT constant to assign the default value to the result set column.

The numbers to the right of selected statements correspond to the previously described steps.

```
#sql public iterator sensitiveUpdateIter
 implements sqlj.runtime.Scrollable, sqlj.runtime.ForUpdate
 with (sensitivity=sqlj.runtime.ResultSetIterator.SENSITIVE,
 updateColumns="LASTNAME, HIREDATE") (String, Date);

  String name;                    // Declare host variables
  Date hrdate;

sensitiveUpdateIter iter = null;
#sql [ctx] iter = { SELECT LASTNAME, HIREDATE FROM EMPLOYEE};

iter.next();

java.sql.ResultSet rs =  iter.getResultSet();                          1
rs.updateString("LASTNAME", "FORREST");
```

```
rs.updateObject
 (2, com.ibm.db2.jcc.DB2PreparedStatement.DB_PARAMETER_DEFAULT);); 2,3
rs.updateRow();
iter.close();
```

Comments in an SQLJ application

To document your SQLJ program, you need to include comments. To do that, use Java comments. Java comments are denoted by /* */ or //.

You can include Java comments outside SQLJ clauses, wherever the Java language permits them. Within an SQLJ clause, you can use Java comments in the following places:

- Within a host expression (/* */ or //).
- Within an SQL statement in an executable clause, if the data source supports a comment within the SQL statement (/* */ or --).

  /* and */ pairs in an SQL statement can be nested.

SQL statement execution in SQLJ applications

You execute SQL statements in a traditional SQL program to create tables, update data in tables, retrieve data from the tables, call stored procedures, or commit or roll back transactions. In an SQLJ program, you also execute these statements, within SQLJ *executable clauses*.

An executable clause can have one of the following general forms:

```
#sql [connection-context] {sql-statement};
#sql [connection-context,execution-context] {sql-statement};
#sql [execution-context] {sql-statement};
```

execution-context specification
In an executable clause, you should **always** specify an explicit connection context, with one exception: you do not specify an explicit connection context for a FETCH statement. You include an execution context only for specific cases. See "Control the execution of SQL statements in SQLJ" for information about when you need an execution context.

connection-context specification
In an executable clause, if you do not explicitly specify a connection context, the executable clause uses the default connection context.

Creating and modifying database objects in an SQLJ application

Use SQLJ executable clauses to execute data definition statements (CREATE, ALTER, DROP, GRANT, REVOKE) or to execute INSERT, searched or positioned UPDATE, and searched or positioned DELETE statements.

The following executable statements demonstrate an INSERT, a searched UPDATE, and a searched DELETE:

```
#sql [myConnCtx] {INSERT INTO DEPARTMENT VALUES
  ("X00","Operations 2","000030","E01",NULL)};
#sql [myConnCtx] {UPDATE DEPARTMENT
  SET MGRNO="000090" WHERE MGRNO="000030"};
#sql [myConnCtx] {DELETE FROM DEPARTMENT
  WHERE DEPTNO="X00"};
```

Performing positioned UPDATE and DELETE operations in an SQLJ application

As in IDS applications in other languages, performing positioned UPDATEs and DELETEs with SQLJ is an extension of retrieving rows from a result table.

The basic steps are:

1. Declare the iterator.

   The iterator can be positioned or named. For positioned UPDATE or DELETE operations, declare the iterator as updatable, using one or both of the following clauses:

   implements sqlj.runtime.ForUpdate
   This clause causes the generated iterator class to include methods for using updatable iterators. This clause is required for programs with positioned UPDATE or DELETE operations.

   with (updateColumns="*column-list*")
   This clause specifies a comma-separated list of the columns of the result table that the iterator will update. This clause is optional.

   You need to declare the iterator as `public`, so you need to follow the rules for declaring and using `public` iterators in the same file or different files.

   If you declare the iterator in a file by itself, any SQLJ source file that has addressability to the iterator and imports the generated class can retrieve data and execute positioned UPDATE or DELETE statements using the iterator.
2. Disable autocommit mode for the connection.

   If autocommit mode is enabled, a COMMIT operation occurs every time the positioned UPDATE statement executes, which causes the iterator to be destroyed unless the iterator has the `with (holdability=true)` attribute. Therefore, you need to turn autocommit off to prevent COMMIT operations until you have finished using the iterator. If you want a COMMIT to occur after every update operation, an alternative way to keep the iterator from being destroyed after each COMMIT operation is to declare the iterator `with (holdability=true)`.
3. Create an instance of the iterator class.

   This is the same step as for a non-updatable iterator.
4. Assign the result table of a SELECT to an instance of the iterator.

   This is the same step as for a non-updatable iterator. The SELECT statement must not include a FOR UPDATE clause.
5. Retrieve and update rows.

   For a positioned iterator, do this by performing the following actions in a loop:

   a. Execute a FETCH statement in an executable clause to obtain the current row.
   b. Test whether the iterator is pointing to a row of the result table by invoking the PositionedIterator.endFetch method.
   c. If the iterator is pointing to a row of the result table, execute an SQL UPDATE... WHERE CURRENT OF *:iterator-object* statement in an executable clause to update the columns in the current row. Execute an SQL DELETE... WHERE CURRENT OF *:iterator-object* statement in an executable clause to delete the current row.

   For a named iterator, do this by performing the following actions in a loop:

   a. Invoke the next method to move the iterator forward.

b. Test whether the iterator is pointing to a row of the result table by checking whether next returns `true`.

c. Execute an SQL UPDATE... WHERE CURRENT OF *iterator-object* statement in an executable clause to update the columns in the current row. Execute an SQL DELETE... WHERE CURRENT OF *iterator-object* statement in an executable clause to delete the current row.

6. Close the iterator.

   Use the close method to do this.

The following code shows how to declare a positioned iterator and use it for positioned UPDATEs. The numbers to the right of selected statements correspond to the previously described steps.

First, in one file, declare positioned iterator `UpdByPos`, specifying that you want to use the iterator to update column `SALARY`:

```
import java.math.*;    // Import this class for BigDecimal data type
#sql public iterator UpdByPos implements sqlj.runtime.ForUpdate       1
  with(updateColumns="SALARY") (String, BigDecimal);
```

Figure 6-6. Example of declaring a positioned iterator for a positioned UPDATE

Then, in another file, use `UpdByPos` for a positioned UPDATE, as shown in the following code fragment:

```
import sqlj.runtime.*;       // Import files for SQLJ and JDBC APIs
import java.sql.*;
import java.math.*;          // Import this class for BigDecimal data type
import UpdByPos;             // Import the generated iterator class that
                             // was created by the iterator declaration clause
                             // for UpdByName in another file
#sql context HSCtx;          // Create a connnection context class HSCtx
public static void main (String args[])
{
  try {
    Class.forName("com.ibm.db2.jcc.DB2Driver");
  }
  catch (ClassNotFoundException e) {
    e.printStackTrace();
          }
  Connection HSjdbccon=
  DriverManager.getConnection("jdbc:ids:SANJOSE");
                     // Create a JDBC connection object
  HSjdbccon.setAutoCommit(false);
                     // Set autocommit off so automatic commits       2
                     // do not destroy the cursor between updates
  HSCtx myConnCtx=new HSCtx(HSjdbccon);
                     // Create a connection context object
  UpdByPos upditer;  // Declare iterator object of UpdByPos class     3
  String empnum;     // Declares host variable to receive EMPNO
  BigDecimal sal;    // and SALARY column values
  #sql [myConnCtx]
    upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE                     4
      WHERE WORKDEPT='D11'};
                     // Assign result table to iterator object
  #sql {FETCH :upditer INTO :empnum,:sal};                            5a
                     // Move cursor to next row
  while (!upditer.endFetch())                                         5b
                     // Check if on a row
  {
    #sql [myConnCtx] {UPDATE EMPLOYEE SET SALARY=SALARY*1.05
      WHERE CURRENT OF :upditer};                                     5c
                     // Perform positioned update
    System.out.println("Updating row for " + empnum);
    #sql {FETCH :upditer INTO :empnum,:sal};
                     // Move cursor to next row
  }
  upditer.close();   // Close the iterator                           6
  #sql [myConnCtx] {COMMIT};
                     // Commit the changes
  myConnCtx.close(); // Close the connection context
}
```

Figure 6-7. Example of performing a positioned UPDATE with a positioned iterator

The following code shows how to declare a named iterator and use it for positioned UPDATEs. The numbers to the right of selected statements correspond to the previously described steps.

First, in one file, declare named iterator `UpdByName`, specifying that you want to use the iterator to update column `SALARY`:

```
import java.math.*;          // Import this class for BigDecimal data type
#sql public iterator UpdByName implements sqlj.runtime.ForUpdate     1
  with(updateColumns="SALARY") (String EmpNo, BigDecimal Salary);
```

Figure 6-8. Example of declaring a named iterator for a positioned UPDATE

Then, in another file, use UpdByName for a positioned UPDATE, as shown in the following code fragment:

```
import sqlj.runtime.*;       // Import files for SQLJ and JDBC APIs
import java.sql.*;
import java.math.*;          // Import this class for BigDecimal data type
import UpdByName;            // Import the generated iterator class that
                             // was created by the iterator declaration clause
                             // for UpdByName in another file
#sql context HSCtx;          // Create a connnection context class HSCtx
public static void main (String args[])
{
  try {
    Class.forName("com.ibm.db2.jcc.DB2Driver");
  }
  catch (ClassNotFoundException e) {
    e.printStackTrace();
          }
  Connection HSjdbccon=
  DriverManager.getConnection("jdbc:ids:SANJOSE");
                         // Create a JDBC connection object
  HSjdbccon.setAutoCommit(false);
                         // Set autocommit off so automatic commits  2
                         // do not destroy the cursor between updates
  HSCtx myConnCtx=new HSCtx(HSjdbccon);
                         // Create a connection context object
  UpdByName upditer;                                                 3
                         // Declare iterator object of UpdByName class
  String empnum;               // Declare host variable to receive EmpNo
                            // column values
  #sql [myConnCtx]
    upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE                    4
      WHERE WORKDEPT='D11'};
                            // Assign result table to iterator object
  while (upditer.next())                                             5a,5b
                            // Move cursor to next row and
                            // check ifon a row
  {
    empnum = upditer.EmpNo();  // Get employee number from current row
    #sql [myConnCtx]
      {UPDATE EMPLOYEE SET SALARY=SALARY*1.05
        WHERE CURRENT OF :upditer};                                  5c
                            // Perform positioned update
    System.out.println("Updating row for " + empnum);
  }
  upditer.close();           // Close the iterator                  6
  #sql [myConnCtx] {COMMIT};
                            // Commit the changes
  myConnCtx.close();         // Close the connection context
}
```

Figure 6-9. Example of performing a positioned UPDATE with a named iterator

Making batch updates in SQLJ applications

The IBM Data Server Driver for JDBC and SQLJ supports batch updates in SQLJ. With batch updates, instead of updating rows of a table one at a time, you can direct SQLJ to execute a group of updates at the same time.

You can include the following types of statements in a batch update:

- Searched INSERT, UPDATE, or DELETE, or MERGE statements
- CREATE, ALTER, DROP, GRANT, or REVOKE statements
- CALL statements with input parameters only

Unlike JDBC, SQLJ allows heterogeneous batches that contain statements with input parameters or host expressions. You can therefore combine any of the following items in an SQLJ batch:

- Instances of the same statement
- Different statements
- Statements with different numbers of input parameters or host expressions
- Statements with different data types for input parameters or host expressions
- Statements with no input parameters or host expressions

For all cases except homogeneous batches of INSERT statements, when an error occurs during execution of a statement in a batch, the remaining statements are executed, and a BatchUpdateException is thrown after all the statements in the batch have executed.

For homogeneous batches of INSERT statements, the behavior is as follows:

- If you set atomicMultiRowInsert to DB2BaseDataSource.YES (1) when you run db2sqljcustomize, and the target data server is DB2 for z/OS, when an error occurs during execution of an INSERT statement in a batch, the remaining statements are not executed, and a BatchUpdateException is thrown.
- If you do not set atomicMultiRowInsert to DB2BaseDataSource.YES (1) when you run db2sqljcustomize, or the target data server is not DB2 for z/OS, when an error occurs during execution of an INSERT statement in a batch, the remaining statements are executed, and a BatchUpdateException is thrown after all the statements in the batch have executed.

To obtain information about warnings, use the ExecutionContext.getWarnings method on the ExecutionContext that you used to submit statements to be batched. You can then retrieve an error description, SQLSTATE, and error code for each SQLWarning object.

When a batch is executed implicitly because the program contains a statement that cannot be added to the batch, the batch is executed before the new statement is processed. If an error occurs during execution of the batch, the statement that caused the batch to execute does not execute.

The basic steps for creating, executing, and deleting a batch of statements are:

1. Disable `AutoCommit` for the connection.

   Do this so that you can control whether to commit changes to already-executed statements when an error occurs during batch execution.
2. Acquire an execution context.

   All statements that execute in a batch must use this execution context.
3. Invoke the ExecutionContext.setBatching(true) method to create a batch.

   Subsequent batchable statements that are associated with the execution context that you created in step 2 are added to the batch for later execution.

   If you want to batch sets of statements that are not batch compatible in parallel, you need to create an execution context for each set of batch compatible statements.
4. Include SQLJ executable clauses for SQL statements that you want to batch.

   These clauses must include the execution context that you created in step 2.

   If an SQLJ executable clause has input parameters or host expressions, you can include the statement in the batch multiple times with different values for the input parameters or host expressions.

To determine whether a statement was added to an existing batch, was the first statement in a new batch, or was executed inside or outside a batch, invoke the ExecutionContext.getUpdateCount method. This method returns one of the following values:

ExecutionContext.ADD_BATCH_COUNT
This is a constant that is returned if the statement was added to an existing batch.

ExecutionContext.NEW_BATCH_COUNT
This is a constant that is returned if the statement was the first statement in a new batch.

ExecutionContext.EXEC_BATCH_COUNT
This is a constant that is returned if the statement was part of a batch, and the batch was executed.

Other integer
This value is the number of rows that were updated by the statement. This value is returned if the statement was executed rather than added to a batch.

5. Execute the batch explicitly or implicitly.
   - Invoke the ExecutionContext.executeBatch method to execute the batch explicitly.

     executeBatch returns an integer array that contains the number of rows that were updated by each statement in the batch. The order of the elements in the array corresponds to the order in which you added statements to the batch.
   - Alternatively, a batch executes implicitly under the following circumstances:
     - You include a batchable statement in your program that is not compatible with statements that are already in the batch. In this case, SQLJ executes the statements that are already in the batch and creates a new batch that includes the incompatible statement.
     - You include a statement in your program that is not batchable. In this case, SQLJ executes the statements that are already in the batch. SQLJ also executes the statement that is not batchable.
     - After you invoke the `ExecutionContext.setBatchLimit(n)` method, you add a statement to the batch that brings the number of statements in the batch to *n* or greater. *n* can have one of the following values:

       ExecutionContext.UNLIMITED_BATCH
       This constant indicates that implicit execution occurs only when SQLJ encounters a statement that is batchable but incompatible, or not batchable. Setting this value is the same as not invoking setBatchLimit.

       ExecutionContext.AUTO_BATCH
       This constant indicates that implicit execution occurs when the number of statements in the batch reaches a number that is set by SQLJ.

       Positive integer
       When this number of statements have been added to the batch, SQLJ executes the batch implicitly. However, the batch might be executed before this many statements have been added if SQLJ encounters a statement that is batchable but incompatible, or not batchable.

     To determine the number of rows that were updated by a batch that was executed implicitly, invoke the ExecutionContext.getBatchUpdateCounts

method. getBatchUpdateCounts returns an integer array that contains the number of rows that were updated by each statement in the batch. The order of the elements in the array corresponds to the order in which you added statements to the batch. Each array element can be one of the following values:

-2 This value indicates that the SQL statement executed successfully, but the number of rows that were updated could not be determined.

-3 This value indicates that the SQL statement failed.

Other integer
This value is the number of rows that were updated by the statement.

6. Optionally, when all statements have been added to the batch, disable batching.

   Do this by invoking the ExecutionContext.setBatching(false) method. When you disable batching, you can still execute the batch implicitly or explicitly, but no more statements are added to the batch. Disabling batching is useful when a batch already exists, and you want to execute a batch compatible statement, rather than adding it to the batch.

   If you want to clear a batch without executing it, invoke the ExecutionContext.cancel method.

7. If batch execution was implicit, perform a final, explicit executeBatch to ensure that all statements have been executed.

Example

The following example demonstrates batching of UPDATE statements. The numbers to the right of selected statements correspond to the previously described steps.

```
#sql iterator GetMgr(String);       // Declare positioned iterator
...
{
  GetMgr deptiter;                  // Declare object of GetMgr class
  String mgrnum = null;             // Declare host variable for manager number
  int raise = 400;                  // Declare raise amount
  int currentSalary;                // Declare current salary
  String url, username, password;   // Declare url, user ID, password
  ...
  TestContext c1 = new TestContext (url, username, password, false);  1
  ExecutionContext ec = new ExecutionContext();                       2
  ec.setBatching(true);                                               3

  #sql [c1] deptiter =
    {SELECT MGRNO FROM DEPARTMENT};
                                    // Assign the result table of the SELECT
                                    // to iterator object deptiter
  #sql {FETCH :deptiter INTO :mgrnum};
                                    // Retrieve the first manager number
  while (!deptiter.endFetch()) {    // Check whether the FETCH returned a row
    #sql [c1]
      {SELECT SALARY INTO :currentSalary FROM EMPLOYEE
        WHERE EMPNO=:mgrnum};
    #sql [c1, ec]                                                     4
      {UPDATE EMPLOYEE SET SALARY=:(currentSalary+raise)
        WHERE EMPNO=:mgrnum};
    #sql {FETCH :deptiter INTO :mgrnum };
                                    // Fetch the next row
  }
  ec.executeBatch();                                                  5
  ec.setBatching(false);                                              6
```

```
  #sql [c1] {COMMIT};
  deptiter.close();                 // Close the iterator
  c1.close();                       // Close the connection
}
```

The following example demonstrates batching of INSERT statements. Suppose that ATOMICTBL is defined like this:

```
CREATE TABLE ATOMICTBL(
 INTCOL INTEGER NOT NULL UNIQUE,
 CHARCOL VARCHAR(10))
```

Also suppose that the table already has a row with the values 2 and "val2". Because of the uniqueness constraint on INTCOL, when the following code is executed, the second INSERT statement in the batch fails.

If the target data server is DB2 for z/OS, and this application is customized without atomicMultiRowInsert set to DB2BaseDataSource.YES, the batch INSERT is non-atomic, so the first set of values is inserted in the table. However, if the application is customized with atomicMultiRowInsert set to DB2BaseDataSource.YES, the batch INSERT is atomic, so the first set of values is not inserted.

The numbers to the right of selected statements correspond to the previously described steps.

```
 ...
TestContext ctx = new TestContext (url, username, password, false);  1
ctx.getExecutionContext().setBatching(true);                         2,3
try {
  for (int i = 1; i<= 2; ++i) {
   if (i == 1) {
    intVar = 3;
    strVar = "val1";
   {
   if (i == 2) {
    intVar = 1;
    strVar = "val2";
   }
   #sql [ctx] {INSERT INTO ATOMICTBL values(:intVar, :strVar)};       4
  }
  int[] counts = ctx.getExecutionContext().executeBatch();           5
  for (int i = 0; i<counts.length; ++i) {
    System.out.println(" count[" + i + "]:" + counts[i]);
  }
}
catch (SQLException e) {
  System.out.println(" Exception Caught: " + e.getMessage());
   SQLException excp = null;
  if (e instanceof SQLException)
  {
    System.out.println(" SQLCode: " + ((SQLException)e).getErrorCode() + "
      Message: " + e.getMessage() );
    excp = ((SQLException)e).getNextException();
    while (  excp != null ) {
      System.out.println(" SQLCode: " + ((SQLException)excp).getErrorCode() +
        " Message: " + excp.getMessage() );
      excp = excp.getNextException();
    }
  }
}
```

Data retrieval in SQLJ applications

SQLJ applications use a *result set iterator* to retrieve result sets. Like a cursor, a result set iterator can be non-scrollable or scrollable.

Just as in IDS applications in other languages, if you want to retrieve a single row from a table in an SQLJ application, you can write a SELECT INTO statement with a WHERE clause that defines a result table that contains only that row:

```
#sql [myConnCtx] {SELECT DEPTNO INTO :hvdeptno
  FROM DEPARTMENT WHERE DEPTNAME="OPERATIONS"};
```

However, most SELECT statements that you use create result tables that contain many rows. In IDS applications in other languages, you use a cursor to select the individual rows from the result table. That cursor can be non-scrollable, which means that when you use it to fetch rows, you move the cursor serially, from the beginning of the result table to the end. Alternatively, the cursor can be scrollable, which means that when you use it to fetch rows, you can move the cursor forward, backward, or to any row in the result table.

This topic discusses how to use non-scrollable iterators. For information on using scrollable iterators, see "Use scrollable iterators in an SQLJ application".

A result set iterator is a Java object that you use to retrieve rows from a result table. Unlike a cursor, a result set iterator can be passed as a parameter to a method.

The basic steps in using a result set iterator are:
1. Declare the iterator, which results in an iterator class
2. Define an instance of the iterator class.
3. Assign the result table of a SELECT to an instance of the iterator.
4. Retrieve rows.
5. Close the iterator.

There are two types of iterators: *positioned iterators* and *named iterators*. Postitioned iterators extend the interface sqlj.runtime.PositionedIterator. Positioned iterators identify the columns of a result table by their position in the result table. Named iterators extend the interface sqlj.runtime.NamedIterator. Named iterators identify the columns of the result table by result table column names.

Using a named iterator in an SQLJ application

Use a named iterator to refer to each of the columns in a result table by name.

The steps in using a named iterator are:

1. Declare the iterator.

   You declare any result set iterator using an *iterator declaration clause*. This causes an iterator class to be created that has the same name as the iterator. For a named iterator, the iterator declaration clause specifies the following information:

   - The name of the iterator
   - A list of column names and Java data types
   - Information for a Java class declaration, such as whether the iterator is `public` or `static`
   - A set of attributes, such as whether the iterator is holdable, or whether its columns can be updated

When you declare a named iterator for a query, you specify names for each of the iterator columns. Those names must match the names of columns in the result table for the query. An iterator column name and a result table column name that differ only in case are considered to be matching names. The named iterator class that results from the iterator declaration clause contains *accessor methods*. There is one accessor method for each column of the iterator. Each accessor method name is the same as the corresponding iterator column name. You use the accessor methods to retrieve data from columns of the result table.

You need to specify Java data types in the iterators that closely match the corresponding IDS column data types. See "Java, JDBC, and SQL data types" for a list of the best mappings between Java data types and IDS data types.

You can declare an iterator in a number of ways. However, because a Java class underlies each iterator, you need to ensure that when you declare an iterator, the underlying class obeys Java rules. For example, iterators that contain a *with-clause* must be declared as `public`. Therefore, if an iterator needs to be `public`, it can be declared only where a `public` class is allowed. The following list describes some alternative methods of declaring an iterator:

- As `public`, in a source file by itself

  This method lets you use the iterator declaration in other code modules, and provides an iterator that works for all SQLJ applications. In addition, there are no concerns about having other top-level classes or `public` classes in the same source file.
- As a top-level class in a source file that contains other top-level class definitions

  Java allows only one public, top-level class in a code module. Therefore, if you need to declare the iterator as public, such as when the iterator includes a with-clause, no other classes in the code module can be declared as public.
- As a nested static class within another class

  Using this alternative lets you combine the iterator declaration with other class declarations in the same source file, declare the iterator and other classes as public, and make the iterator class visible to other code modules or packages. However, when you reference the iterator from outside the nesting class, you must fully-qualify the iterator name with the name of the nesting class.
- As an inner class within another class

  When you declare an iterator in this way, you can instantiate it only within an instance of the nesting class. However, you can declare the iterator and other classes in the file as public.

  You cannot cast a JDBC ResultSet to an iterator if the iterator is declared as an inner class. This restriction does not apply to an iterator that is declared as a static nested class. See "Use SQLJ and JDBC in the same application" for more information on casting a ResultSet to a iterator.

2. Create an instance of the iterator class.

   You declare an object of the named iterator class to retrieve rows from a result table.
3. Assign the result table of a SELECT to an instance of the iterator.

   To assign the result table of a SELECT to an iterator, you use an SQLJ *assignment clause*. The format of the assignment clause for a named iterator is:

   ```
   #sql context-clause iterator-object={select-statement};
   ```

   See "SQLJ assignment-clause" and "SQLJ context-clause" for more information.
4. Retrieve rows.

Do this by invoking accessor methods in a loop. Accessor methods have the same names as the corresponding columns in the iterator, and have no parameters. An accessor method returns the value from the corresponding column of the current row in the result table. Use the NamedIterator.next() method to move the cursor forward through the result table.

To test whether you have retrieved all rows, check the value that is returned when you invoke the next method. next returns a `boolean` with a value of `false` if there is no next row.

5. Close the iterator.

   Use the NamedIterator.close method to do this.

The following code demonstrates how to declare and use a named iterator. The numbers to the right of selected statements correspond to the previously-described steps.

```
#sql  iterator ByName(String LastName, Date HireDate);                    1
                                // Declare named iterator ByName
{
  ...
  ByName nameiter;              // Declare object of ByName class          2
  #sql [ctxt]
 nameiter={SELECT LASTNAME, HIREDATE FROM EMPLOYEE};                       3
                                // Assign the result table of the SELECT
                                // to iterator object nameiter
  while (nameiter.next())       // Move the iterator through the result    4
                                // table and test whether all rows retrieved
  {
    System.out.println( nameiter.LastName() + " was hired on "
      + nameiter.HireDate());   // Use accessor methods LastName and
                                // HireDate to retrieve column values
  }
  nameiter.close();             // Close the iterator                      5
}
```

Figure 6-10. Example of using a named iterator

Using a positioned iterator in an SQLJ application

Use a positioned iterator to refer to columns in a result table by their position in the result set.

The steps in using a positioned iterator are:

1. Declare the iterator.

   You declare any result set iterator using an *iterator declaration clause*. This causes an iterator class to be created that has the same name and attributes as the iterator. For a positioned iterator, the iterator declaration clause specifies the following information:

   - The name of the iterator
   - A list of Java data types
   - Information for a Java class declaration, such as whether the iterator is `public` or `static`
   - A set of attributes, such as whether the iterator is holdable, or whether its columns can be updated

   The data type declarations represent columns in the result table and are referred to as columns of the result set iterator. The columns of the result set iterator correspond to the columns of the result table, in left-to-right order. For example, if an iterator declaration clause has two data type declarations, the

first data type declaration corresponds to the first column in the result table, and the second data type declaration corresponds to the second column in the result table.

You need to specify Java data types in the iterators that closely match the corresponding IDS column data types. See "Java, JDBC, and SQL data types" for a list of the best mappings between Java data types and IDS data types.

You can declare an iterator in a number of ways. However, because a Java class underlies each iterator, you need to ensure that when you declare an iterator, the underlying class obeys Java rules. For example, iterators that contain a *with-clause* must be declared as `public`. Therefore, if an iterator needs to be `public`, it can be declared only where a `public` class is allowed. The following list describes some alternative methods of declaring an iterator:

- As `public`, in a source file by itself

  This is the most versatile method of declaring an iterator. This method lets you use the iterator declaration in other code modules, and provides an iterator that works for all SQLJ applications. In addition, there are no concerns about having other top-level classes or `public` classes in the same source file.
- As a top-level class in a source file that contains other top-level class definitions

  Java allows only one public, top-level class in a code module. Therefore, if you need to declare the iterator as public, such as when the iterator includes a with-clause, no other classes in the code module can be declared as public.
- As a nested static class within another class

  Using this alternative lets you combine the iterator declaration with other class declarations in the same source file, declare the iterator and other classes as public, and make the iterator class visible from other code modules or packages. However, when you reference the iterator from outside the nesting class, you must fully-qualify the iterator name with the name of the nesting class.
- As an inner class within another class

  When you declare an iterator in this way, you can instantiate it only within an instance of the nesting class. However, you can declare the iterator and other classes in the file as public.

  You cannot cast a JDBC ResultSet to an iterator if the iterator is declared as an inner class. This restriction does not apply to an iterator that is declared as a static nested class. See "Use SQLJ and JDBC in the same application" for more information on casting a ResultSet to a iterator.

2. Create an instance of the iterator class.

   You declare an object of the positioned iterator class to retrieve rows from a result table.
3. Assign the result table of a SELECT to an instance of the iterator.

   To assign the result table of a SELECT to an iterator, you use an SQLJ *assignment clause*. The format of the assignment clause for a positioned iterator is:

   ```
   #sql context-clause iterator-object={select-statement};
   ```
4. Retrieve rows.

   Do this by executing FETCH statements in executable clauses in a loop. The FETCH statements looks the same as a FETCH statements in other languages.

To test whether you have retrieved all rows, invoke the PositionedIterator.endFetch method after each FETCH. endFetch returns a `boolean` with the value `true` if the FETCH failed because there are no rows to retrieve.

5. Close the iterator.

   Use the PositionedIterator.close method to do this.

The following code demonstrates how to declare and use a positioned iterator. The numbers to the right of selected statements correspond to the previously-described steps.

```
#sql iterator ByPos(String,Date); // Declare positioned iterator ByPos  1
{
  ...
  ByPos positer;                  // Declare object of ByPos class    2
  String name = null;             // Declare host variables
  Date hrdate;
  #sql [ctxt] positer =
    {SELECT LASTNAME, HIREDATE FROM EMPLOYEE};                         3
                                  // Assign the result table of the SELECT
                                  // to iterator object positer
  #sql {FETCH :positer INTO :name, :hrdate };                          4
                                  // Retrieve the first row
  while (!positer.endFetch())     // Check whether the FETCH returned a row
  { System.out.println(name + " was hired in " +
      hrdate);
    #sql {FETCH :positer INTO :name, :hrdate };
                                  // Fetch the next row
  }
  positer.close();                // Close the iterator               5
}
```

Figure 6-11. Example of using a positioned iterator

Multiple open iterators for the same SQL statement in an SQLJ application

With the IBM Data Server Driver for JDBC and SQLJ, your application can have multiple concurrently open iterators for a single SQL statement in an SQLJ application. With this capability, you can perform one operation on a table using one iterator while you perform a different operation on the same table using another iterator.

When you use concurrently open iterators in an application, you should close iterators when you no longer need them to prevent excessive storage consumption in the Java heap.

The following examples demonstrate how to perform the same operations on a table without concurrently open iterators on a single SQL statement and with concurrently open iterators on a single SQL statement. These examples use the following iterator declaration:

```
import java.math.*;
#sql public iterator  MultiIter(String EmpNo, BigDecimal Salary);
```

Without the capability for multiple, concurrently open iterators for a single SQL statement, if you want to select employee and salary values for a specific employee number, you need to define a different SQL statement for each employee number, as shown in Figure 6-12 on page 6-29.

```
MultiIter iter1 = null;            // Iterator instance for retrieving
                                   // data for first employee
String EmpNo1 = "000100";          // Employee number for first employee
#sql [ctx] iter1 =
  {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo1};
                                   // Assign result table to first iterator
MultiIter iter2 = null;            // Iterator instance for retrieving
                                   // data for second employee
String EmpNo2 = "000200";          // Employee number for second employee
#sql [ctx] iter2 =
  {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo2};
                                   // Assign result table to second iterator
// Process with iter1
// Process with iter2
iter1.close();                     // Close the iterators
iter2.close();
```

Figure 6-12. Example of concurrent table operations using iterators with different SQL statements

Figure 6-13 demonstrates how you can perform the same operations when you have the capability for multiple, concurrently open iterators for a single SQL statement.

```
...
MultiIter iter1 = openIter("000100"); // Invoke openIter to assign the result table
                                      // (for employee 100) to the first iterator
MultiIter iter2 = openIter("000200"); // Invoke openIter to assign the result
                                      // table to the second iterator
                                      // iter1 stays open when iter2 is opened
// Process with iter1
// Process with iter2
...
iter1.close();                        // Close the iterators
iter2.close();
...
public MultiIter openIter(String EmpNo)
                                      // Method to assign a result table
                                      // to an iterator instance
{
  MultiIter iter;
  #sql [ctxt] iter =
    {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo};
  return iter;                        // Method returns an iterator instance
}
```

Figure 6-13. Example of concurrent table operations using iterators with the same SQL statement

Multiple open instances of an iterator in an SQLJ application

Multiple instances of an iterator can be open concurrently in a single SQLJ application. One application for this ability is to open several instances of an iterator that uses host expressions. Each instance can use a different set of host expression values.

The following example shows an application with two concurrently open instances of an iterator.

```
...
ResultSet myFunc(String empid)  // Method to open an iterator and get a resultSet
{
  MyIter iter;
  #sql iter = {SELECT * FROM EMPLOYEE WHERE EMPNO = :empid};
  return iter.getResultSet();
}

// An application can call this method to get a resultSet for each
// employee ID. The application can process each resultSet separately.
...
ResultSet rs1 = myFunc("000100");  // Get employee record for employee ID 000100
...
ResultSet rs2 = myFunc("000200");  // Get employee record for employee ID 000200
```

Figure 6-14. Example of opening more than one instance of an iterator in a single application

As with any other iterator, you need to remember to close this iterator after the last time you use it to prevent excessive storage consumption.

Using scrollable iterators in an SQLJ application

In addition to moving forward, one row at a time, through a result table, you might want to move backward or go directly to a specific row. The IBM Data Server Driver for JDBC and SQLJ provides this capability.

An iterator in which you can move forward, backward, or to a specific row is called a *scrollable iterator*. A scrollable iterator in SQLJ is equivalent to the result table of a database cursor that is declared as SCROLL.

Like a scrollable cursor, a scrollable iterator for a connection to IBM Informix is *insensitive*. Insensitive means that changes to the underlying table after the iterator is opened are not visible to the iterator. Insensitive iterators are read-only.

Important:

To create and use a scrollable iterator, you need to follow these steps:

1. Specify an iterator declaration clause that includes the following clauses:
   - `implements sqlj.runtime.Scrollable`

     This indicates that the iterator is scrollable.
   - `with (sensitivity=INSENSITIVE)`

   The iterator can be a named or positioned iterator.

   Example: The following iterator declaration clause declares a positioned, insensitive, scrollable iterator:

   ```
   #sql public iterator ByPos
     implements sqlj.runtime.Scrollable
     with (sensitivity=INSENSITIVE)  (String);
   ```

   Example: The following iterator declaration clause declares a named, insensitive, scrollable iterator:

   ```
   #sql public iterator ByName
     implements sqlj.runtime.Scrollable
     with (sensitivity=INSENSITIVE) (String EmpNo);
   ```

   Restriction: You cannot use a scrollable iterator to select columns with the following data types from a table on a DB2 Database for Linux, UNIX, and Windows server:
   - LONG VARCHAR
   - LONG VARGRAPHIC

- BLOB
- CLOB
- XML
- A distinct type that is based on any of the previous data types in this list
- A structured type

2. Create an iterator object, which is an instance of your iterator class.
3. For each row that you want to access:

   For a named iterator, perform the following steps:

   a. Position the cursor using one of the methods listed in the following table.

Table 6-1. sqlj.runtime.Scrollable methods for positioning a scrollable cursor

Method	Positions the cursor
first[1]	On the first row of the result table
last[1]	On the last row of the result table
previous[1,2]	On the previous row of the result table
next	On the next row of the result table
`absolute(int n)`[1,3]	If *n*>0, on row *n* of the result table. If *n*<0, and *m* is the number of rows in the result table, on row *m+n+1* of the result table.
`relative(int n)`[1,4]	If *n*>0, on the row that is *n* rows after the current row. If *n*<0, on the row that is *n* rows before the current row. If *n*=0, on the current row.
afterLast[1]	After the last row in the result table
beforeFirst[1]	Before the first row in the result table

Notes:

1. This method does not apply to connections to IBM Informix.
2. If the cursor is after the last row of the result table, this method positions the cursor on the last row.
3. If the absolute value of *n* is greater than the number of rows in the result table, this method positions the cursor after the last row if *n* is positive, or before the first row if *n* is negative.
4. Suppose that *m* is the number of rows in the result table and *x* is the current row number in the result table. If *n*>0 and *x+n*>*m*, the iterator is positioned after the last row. If *n*<0 and *x+n*<1, the iterator is positioned before the first row.

   b. If you need to know the current cursor position, use the getRow, isFirst, isLast, `isBeforeFirst`, or isAfterLast method to obtain this information.

      If you need to know the current fetch direction, invoke the getFetchDirection method.

   c. Use accessor methods to retrieve the current row of the result table.

   d. If update or delete operations by the iterator or by other means are visible in the result table, invoke the getWarnings method to check whether the current row is a hole.

   For a positioned iterator, perform the following steps:

   a. Use a FETCH statement with a fetch orientation clause to position the iterator and retrieve the current row of the result table. Table 6-2 on page 6-32 lists the clauses that you can use to position the cursor.

Table 6-2. FETCH clauses for positioning a scrollable cursor

Method	Positions the cursor
FIRST[1]	On the first row of the result table
LAST[1]	On the last row of the result table
PRIOR[1,2]	On the previous row of the result table
NEXT	On the next row of the result table
ABSOLUTE(*n*)[1,3]	If *n*>0, on row *n* of the result table. If *n*<0, and *m* is the number of rows in the result table, on row *m+n+1* of the result table.
RELATIVE(*n*)[1,4]	If *n*>0, on the row that is *n* rows after the current row. If *n*<0, on the row that is *n* rows before the current row. If *n*=0, on the current row.
AFTER[1,5]	After the last row in the result table
BEFORE[1,5]	Before the first row in the result table

Notes:

1. This value is not supported for connections to IBM Informix
2. If the cursor is after the last row of the result table, this method positions the cursor on the last row.
3. If the absolute value of *n* is greater than the number of rows in the result table, this method positions the cursor after the last row if *n* is positive, or before the first row if *n* is negative.
4. Suppose that *m* is the number of rows in the result table and *x* is the current row number in the result table. If *n*>0 and *x+n*>*m*, the iterator is positioned after the last row. If *n*<0 and *x+n*<1, the iterator is positioned before the first row.
5. Values are not assigned to host expressions.

   b. If update or delete operations by the iterator or by other means are visible in the result table, invoke the getWarnings method to check whether the current row is a hole.

4. Invoke the close method to close the iterator.

The following code demonstrates how to use a named iterator to retrieve the employee number and last name from all rows from the employee table. The numbers to the right of selected statements correspond to the previously-described steps.

```
#sql context Ctx;            // Create connection context class Ctx
#sql  iterator ScrollIter implements sqlj.runtime.Scrollable        1
  (String EmpNo, String LastName);
{
  ...
  Ctx ctxt =
   new Ctx("jdbc:db2://sysmvs1.stl.ibm.com:5021/NEWYORK",
   userid,password,false);   // Create connection context object ctxt
                             // for the connection to NEWYORK
  ScrollIter scrliter;                                              2
  #sql [ctxt]
  scrliter={SELECT EMPNO, LASTNAME FROM EMPLOYEE};
  while (scrliter.next())                                           4a
  {
    System.out.println(scrliter.EmpNo() + " "                       4c
      + scrliter.LastName());
  }
  scrliter.close();                                                 5
}
```

Calling stored procedures in SQLJ applications

To call a stored procedure, you use an executable clause that contains an SQL CALL statement.

You can execute the CALL statement with host identifier parameters. You can execute the CALL statement with literal parameters only if the IDS server on which the CALL statement runs supports execution of the CALL statement dynamically.

The basic steps in calling a stored procedure are:

1. Assign values to input (IN or INOUT) parameters.
2. Call the stored procedure.
3. Process output (OUT or INOUT) parameters.
4. If the stored procedure returns multiple result sets, retrieve those result sets.

The following code illustrates calling a stored procedure that has three input parameters and three output parameters. The numbers to the right of selected statements correspond to the previously-described steps.

```
String FirstName="TOM";            // Input parameters                  1
String LastName="NARISINST";
String Address="IBM";
int CustNo;                        // Output parameters
String Mark;
String MarkErrorText;
...
#sql [myConnCtx] {CALL ADD_CUSTOMER(:IN FirstName,                       2
                            :IN LastName,
                            :IN Address,
                            :OUT CustNo,
                            :OUT Mark,
                            :OUT MarkErrorText)};
                                  // Call the stored procedure
System.out.println("Output parameters from ADD_CUSTOMER call: ");
System.out.println("Customer number for " + LastName + ": " + CustNo);  3
System.out.println(Mark);
If (MarkErrorText != null)
  System.out.println(" Error messages:" + MarkErrorText);
```

Figure 6-15. Example of calling a stored procedure in an SQLJ application

LOBs in SQLJ applications with the IBM Data Server Driver for JDBC and SQLJ

With the IBM Data Server Driver for JDBC and SQLJ, you can retrieve LOB data into `Clob` or `Blob` host expressions or update CLOB, BLOB, or DBCLOB columns from `Clob` or `Blob` host expressions. You can also declare iterators with `Clob` or `Blob` data types to retrieve data from CLOB, BLOB, or DBCLOB columns.

Retrieving or updating LOB data: To retrieve data from a BLOB column, declare an iterator that includes a data type of `Blob` or `byte[]`. To retrieve data from a CLOB or DBCLOB column, declare an iterator in which the corresponding column has a `Clob` data type.

To update data in a BLOB column, use a host expression with data type `Blob`. To update data in a CLOB or DBCLOB column, use a host expression with data type `Clob`.

Progressive streaming or LOB locators: In SQLJ applications, you can use progressive streaming, also known as dynamic data format, or LOB locators in the same way that you use them in JDBC applications.

Java data types for retrieving or updating LOB column data in SQLJ applications

When the deferPrepares property is set to true, and the IBM Data Server Driver for JDBC and SQLJ processes an SQLJ statement that includes host expressions, the driver might need to do extra processing to determine data types. This extra processing can impact performance.

When the JDBC driver cannot immediately determine the data type of a parameter that is used with a LOB column, you need to choose a parameter data type that is compatible with the LOB data type.

Input parameters for BLOB columns

For input parameters for BLOB columns, you can use either of the following techniques:

- Use a java.sql.Blob input variable, which is an exact match for a BLOB column:

  ```
  java.sql.Blob blobData;
  #sql {CALL STORPROC(:IN blobData)};
  ```

  Before you can use a java.sql.Blob input variable, you need to create a java.sql.Blob object, and then populate that object.
- Use an input parameter of type of sqlj.runtime.BinaryStream. A sqlj.runtime.BinaryStream object is compatible with a BLOB data type. For example:

  ```
  java.io.ByteArrayInputStream byteStream =
    new java.io.ByteArrayInputStream(byteData);
  int numBytes = byteData.length;
  sqlj.runtime.BinaryStream binStream =
    new sqlj.runtime.BinaryStream(byteStream, numBytes);
  #sql {CALL STORPROC(:IN binStream)};
  ```

  You cannot use this technique for INOUT parameters.

Output parameters for BLOB columns

For output or INOUT parameters for BLOB columns, you can use the following technique:

- Declare the output parameter or INOUT variable with a java.sql.Blob data type:

  ```
  java.sql.Blob blobData = null;
  #sql CALL STORPROC (:OUT blobData)};
  ```

  ```
  java.sql.Blob blobData = null;
  #sql CALL STORPROC (:INOUT blobData)};
  ```

Input parameters for CLOB columns

For input parameters for CLOB columns, you can use one of the following techniques:

- Use a java.sql.Clob input variable, which is an exact match for a CLOB column:

  ```
  #sql CALL STORPROC(:IN clobData)};
  ```

  Before you can use a java.sql.Clob input variable, you need to create a java.sql.Clob object, and then populate that object.

- Use one of the following types of stream IN parameters:
  - A sqlj.runtime.CharacterStream input parameter:

```
java.lang.String charData;
java.io.StringReader reader = new java.io.StringReader(charData);
sqlj.runtime.CharacterStream charStream =
  new sqlj.runtime.CharacterStream (reader, charData.length);
#sql {CALL STORPROC(:IN charStream)};
```

  - A sqlj.runtime.UnicodeStream parameter, for Unicode UTF-16 data:

```
byte[] charDataBytes = charData.getBytes("UnicodeBigUnmarked");
java.io.ByteArrayInputStream byteStream =
  new java.io.ByteArrayInputStream(charDataBytes);
sqlj.runtime.UnicodeStream uniStream =
  new sqlj.runtime.UnicodeStream(byteStream, charDataBytes.length );
#sql {CALL STORPROC(:IN uniStream)};
```

  - A sqlj.runtime.AsciiStream parameter, for ASCII data:

```
byte[] charDataBytes = charData.getBytes("US-ASCII");
java.io.ByteArrayInputStream byteStream =
  new java.io.ByteArrayInputStream (charDataBytes);
sqlj.runtime.AsciiStream asciiStream =
  new sqlj.runtime.AsciiStream (byteStream, charDataBytes.length);
#sql {CALL STORPROC(:IN asciiStream)};
```

  For these calls, you need to specify the exact length of the input data. You cannot use this technique for INOUT parameters.
- Use a java.lang.String input parameter:

```
java.lang.String charData;
#sql {CALL STORPROC(:IN charData)};
```

Output parameters for CLOB columns

For output or INOUT parameters for CLOB columns, you can use one of the following techniques:

- Use a java.sql.Clob output variable, which is an exact match for a CLOB column:

```
java.sql.Clob clobData = null;
#sql CALL STORPROC(:OUT clobData)};
```

- Use a java.lang.String output variable:

```
java.lang.String charData = null;
#sql CALL STORPROC(:OUT charData)};
```

  This technique should be used only if you know that the length of the retrieved data is less than or equal to 32KB. Otherwise, the data is truncated.

Output parameters for DBCLOB columns

DBCLOB output or INOUT parameters for stored procedures are not supported.

SQLJ and JDBC in the same application

You can combine SQLJ clauses and JDBC calls in a single program.

To do this effectively, you need to be able to do the following things:

- Use a JDBC Connection to build an SQLJ ConnectionContext, or obtain a JDBC Connection from an SQLJ ConnectionContext.
- Use an SQLJ iterator to retrieve data from a JDBC ResultSet or generate a JDBC ResultSet from an SQLJ iterator.

Building an SQLJ ConnectionContext from a JDBC Connection: To do that:

1. Execute an SQLJ connection declaration clause to create a ConnectionContext class.
2. Load the driver or obtain a DataSource instance.
3. Invoke the SQLJ DriverManager.getConnection or DataSource.getConnection method to obtain a JDBC Connection.
4. Invoke the ConnectionContext constructor with the Connection as its argument to create the ConnectionContext object.

Obtaining a JDBC Connection from an SQLJ ConnectionContext: To do this,

1. Execute an SQLJ connection declaration clause to create a ConnectionContext class.
2. Load the driver or obtain a DataSource instance.
3. Invoke the ConnectionContext constructor with the URL of the driver and any other necessary parameters as its arguments to create the ConnectionContext object.
4. Invoke the JDBC ConnectionContext.getConnection method to create the JDBC Connection object.

See "Connect to a data source using SQLJ" for more information on SQLJ connections.

Retrieving JDBC result sets using SQLJ iterators: Use the *iterator conversion statement* to manipulate a JDBC result set as an SQLJ iterator. The general form of an iterator conversion statement is:

```
#sql iterator={CAST :result-set};
```

Before you can successfully cast a result set to an iterator, the iterator must conform to the following rules:

- The iterator must be declared as public.
- If the iterator is a positioned iterator, the number of columns in the result set must match the number of columns in the iterator. In addition, the data type of each column in the result set must match the data type of the corresponding column in the iterator.
- If the iterator is a named iterator, the name of each accessor method must match the name of a column in the result set. In addition, the data type of the object that an accessor method returns must match the data type of the corresponding column in the result set.

The code in Figure 6-16 on page 6-37 builds and executes a query using a JDBC call, executes an iterator conversion statement to convert the JDBC result set to an SQLJ iterator, and retrieves rows from the result table using the iterator.

```
#sql public iterator ByName(String LastName, Date HireDate);      1
public void HireDates(ConnectionContext connCtx, String whereClause)
{

  ByName nameiter;           // Declare object of ByName class
  Connection conn=connCtx.getConnection();
                             // Create JDBC connection
  Statement stmt = conn.createStatement();                         2
  String query = "SELECT LASTNAME, HIREDATE FROM EMPLOYEE";
  query+=whereClause;   // Build the query
  ResultSet rs = stmt.executeQuery(query);                         3
  #sql [connCtx] nameiter = {CAST :rs};                            4
  while (nameiter.next())
  {
    System.out.println( nameiter.LastName() + " was hired on "
      + nameiter.HireDate());

  }
  nameiter.close();                                                5
  stmt.close();
}
```

Figure 6-16. Converting a JDBC result set to an SQLJ iterator

Notes to Figure 6-16:

Note	Description
1	This SQLJ clause creates the named iterator class ByName, which has accessor methods LastName() and HireDate() that return the data from result table columns LASTNAME and HIREDATE.
2	This statement and the following two statements build and prepare a query for dynamic execution using JDBC.
3	This JDBC statement executes the SELECT statement and assigns the result table to result set rs.
4	This iterator conversion clause converts the JDBC ResultSet rs to SQLJ iterator nameiter, and the following statements use nameiter to retrieve values from the result table.
5	The nameiter.close() method closes the SQLJ iterator and JDBC ResultSet rs.

Generating JDBC ResultSets from SQLJ iterators: Use the getResultSet method to generate a JDBC ResultSet from an SQLJ iterator. Every SQLJ iterator has a getResultSet method. After you access the ResultSet that underlies an iterator, you need to fetch rows using only the ResultSet.

The code in Figure 6-17 generates a positioned iterator for a query, converts the iterator to a result set, and uses JDBC methods to fetch rows from the table.

```
#sql iterator EmpIter(String, java.sql.Date);
{
...
  EmpIter iter=null;
  #sql [connCtx] iter=
    {SELECT LASTNAME, HIREDATE FROM EMPLOYEE};                     1
  ResultSet rs=iter.getResultSet();                                2
  while (rs.next())                                                3
  { System.out.println(rs.getString(1) + " was hired in " +
      rs.getDate(2));
  }
  rs.close();                                                      4
}
```

Figure 6-17. Converting an SQLJ iterator to a JDBC ResultSet

Notes to Figure 6-17 on page 6-37:

Note	Description
1	This SQLJ clause executes the SELECT statement, constructs an iterator object that contains the result table for the SELECT statement, and assigns the iterator object to variable iter.
2	The getResultSet() method accesses the ResultSet that underlies iterator iter.
3	The JDBC getString() and getDate() methods retrieve values from the ResultSet. The next() method moves the cursor to the next row in the ResultSet.
4	The rs.close() method closes the SQLJ iterator as well as the ResultSet.

Rules and restrictions for using JDBC ResultSets in SQLJ applications: When you write SQLJ applications that include JDBC result sets, observe the following rules and restrictions:

- You cannot cast a ResultSet to an SQLJ iterator if the ResultSet and the iterator have different holdability attributes.

  A JDBC ResultSet or an SQLJ iterator can remain open after a COMMIT operation. For a JDBC ResultSet, this characteristic is controlled by the IBM Data Server Driver for JDBC and SQLJ property `resultSetHoldability`. For an SQLJ iterator, this characteristic is controlled by the `with holdability` parameter of the iterator declaration. Casting a ResultSet that has holdability to an SQLJ iterator that does not, or casting a ResultSet that does not have holdability to an SQLJ iterator that does, is not supported.
- Close the iterator or the underlying ResultSet object as soon as the program no longer uses the iterator or ResultSet, and before the end of the program.

  Closing the iterator also closes the ResultSet object. Closing the ResultSet object also closes the iterator object. In general, it is best to close the object that is used last.
- For the IBM Data Server Driver for JDBC and SQLJ, which supports scrollable iterators and scrollable and updatable ResultSet objects, the following restrictions apply:
  - Scrollable iterators have the same restrictions as their underlying JDBC ResultSet objects.
  - You cannot cast a JDBC ResultSet that is not updatable to an SQLJ iterator that is updatable.

Controlling the execution of SQL statements in SQLJ

You can use selected methods of the SQLJ ExecutionContext class to control or monitor the execution of SQL statements.

To use ExecutionContext methods, follow these steps:

1. Acquire the default execution context from the connection context.

   There are two ways to acquire an execution context:

   - Acquire the default execution context from the connection context. For example:

     ```
     ExecutionContext execCtx = connCtx.getExecutionContext();
     ```
   - Create a new execution context by invoking the constructor for ExecutionContext. For example:

     ```
     ExecutionContext execCtx=new ExecutionContext();
     ```
2. Associate the execution context with an SQL statement.

   To do that, specify an execution context after the connection context in the execution clause that contains the SQL statement.

3. Invoke ExecutionContext methods.

   Some ExecutionContext methods are applicable before the associated SQL statement is executed, and some are applicable only after their associated SQL statement is executed.

   For example, you can use method getUpdateCount to count the number of rows that are deleted by a DELETE statement after you execute the DELETE statement.

The following code demonstrates how to acquire an execution context, and then use the getUpdateCount method on that execution context to determine the number of rows that were deleted by a DELETE statement. The numbers to the right of selected statements correspond to the previously-described steps.

```
ExecutionContext execCtx=new ExecutionContext();                          1
#sql [connCtx, execCtx] {DELETE FROM EMPLOYEE WHERE SALARY > 10000};      2
System.out.println("Deleted " + execCtx.getUpdateCount() + " rows");      3
```

ROWIDs in SQLJ with the IBM Data Server Driver for JDBC and SQLJ

DB2 for z/OS and DB2 for i support the ROWID data type for a column in a table. A ROWID is a value that uniquely identifies a row in a table.

Although IBM Informix also supports rowids, those rowids have the INTEGER data type. You can select an IBM Informix rowid column into a variable with a four-byte integer data type.

If you use columns with the ROWID data type in SQLJ programs, you need to customize those programs.

JDBC 4.0 includes interface java.sql.RowId that you can use in iterators and in CALL statement parameters. If you do not have JDBC 4.0, you can use the IBM Data Server Driver for JDBC and SQLJ-only class com.ibm.db2.jcc.DB2RowID. For an iterator, you can also use the byte[] object type to retrieve ROWID values.

The following code shows an example of an iterator that is used to select values from a ROWID column:

```
#sql iterator PosIter(int,String,java.sql.RowId);
                                // Declare positioned iterator
                                // for retrieving ITEM_ID (INTEGER),
                                // ITEM_FORMAT (VARCHAR), and ITEM_ROWID (ROWID)
                                // values from table ROWIDTAB
{
  PosIter positrowid;           // Declare object of PosIter class
  java.sql.RowId rowid = null;
  int id = 0;
  String i_fmt = null;
                                // Declare host expressions
  #sql [ctxt] positrowid =
    {SELECT ITEM_ID, ITEM_FORMAT, ITEM_ROWID FROM ROWIDTAB
       WHERE ITEM_ID=3};
                                // Assign the result table of the SELECT
                                // to iterator object positrowid
  #sql {FETCH :positrowid INTO :id, :i_fmt, :rowid};
                                // Retrieve the first row
  while (!positrowid.endFetch())
                                // Check whether the FETCH returned a row
    {System.out.println("Item ID " + id + " Item format " +
       i_fmt + " Item ROWID ");
       MyUtilities.printBytes(rowid.getBytes());
                           // Use the getBytes method to
                           // convert the value to bytes for printing.
                           // Call a user-defined method called
                           // printBytes (not shown) to print
                           // the value.
       #sql {FETCH :positrowid INTO :id, :i_fmt, :rowid};
                                // Retrieve the next row
    }
  positrowid.close();           // Close the iterator
}
```

Figure 6-18. Example of using an iterator to retrieve ROWID values

The following code shows an example of calling a stored procedure that takes three ROWID parameters: an IN parameter, an OUT parameter, and an INOUT parameter.

```
java.sql.RowId in_rowid = rowid;
java.sqlRowId out_rowid = null;
java.sql.RowId inout_rowid = rowid;
                                   // Declare an IN, OUT, and
                                   // INOUT ROWID parameter
...
#sql [myConnCtx] {CALL SP_ROWID(:IN in_rowid,
                             :OUT out_rowid,
                             :INOUT inout_rowid)};
                                   // Call the stored procedure
System.out.println("Parameter values from SP_ROWID call: ");
System.out.println("OUT parameter value ");
MyUtilities.printBytes(out_rowid.getBytes());
                           // Use the getBytes method to
                           // convert the value to bytes for printing
                           // Call a user-defined method called
                           // printBytes (not shown) to print
                           // the value.
System.out.println("INOUT parameter value ");
MyUtilities.printBytes(inout_rowid.getBytes());
```

Figure 6-19. Example of calling a stored procedure with a ROWID parameter

Savepoints in SQLJ applications

Under the IBM Data Server Driver for JDBC and SQLJ, you can include any form of the SQL SAVEPOINT statement in your SQLJ program.

An SQL savepoint represents the state of data and schemas at a particular point in time within a unit of work. SQL statements exist to set a savepoint, release a savepoint, and restore data and schemas to the state that the savepoint represents.

The following example demonstrates how to set a savepoint, roll back to the savepoint, and release the savepoint.

Figure 6-20. Setting, rolling back to, and releasing a savepoint in an SQLJ application

```
#sql context Ctx;             // Create connection context class Ctx
String empNumVar;
int shoeSizeVar;
...
try {                         // Load the JDBC driver
  Class.forName("com.ibm.db2.jcc.DB2Driver");
}
catch (ClassNotFoundException e) {
   e.printStackTrace();
}
Connection jdbccon=
  DriverManager.getConnection("jdbc:ids://sysmvs1.stl.ibm.com:5021/NEWYORK",
    userid,password);
                              // Create JDBC connection object jdbccon
jdbccon.setAutoCommit(false); // Do not autocommit
Ctx ctxt=new Ctx(jdbccon);
                              // Create connection context object myConnCtx
                              // for the connection to NEWYORK
...                             // Perform some SQL
#sql [ctxt] {COMMIT};         // Commit the transaction
                              // Commit the create
#sql [ctxt]
  {INSERT INTO EMP_SHOE VALUES ('000010', 6)};
                              // Insert a row
#sql [ctxt]
  {SAVEPOINT SVPT1 ON ROLLBACK RETAIN CURSORS};
                              // Create a savepoint
...
#sql [ctxt]
  {INSERT INTO EMP_SHOE VALUES ('000020', 10)};
                               // Insert another row
#sql [ctxt] {ROLLBACK TO SAVEPOINT SVPT1};
                               // Roll back work to the point
                               //  after the first insert
...
#sql [ctxt] {RELEASE SAVEPOINT SVPT1};
                               // Release the savepoint
ctx.close();                   // Close the connection context
```

SQLJ utilization of SDK for Java Version 5 function

Your SQLJ applications can use a number of functions that were introduced with the SDK for Java Version 5.

Static import

The static import construct lets you access static members without qualifying those members with the name of the class to which they belong. For SQLJ applications, this means that you can use static members in host expressions without qualifying them.

Example: Suppose that you want to declare a host expression of this form:

```
double r = cos(PI * E);
```

cos, PI, and E are members of the java.lang.Math class. To declare r without explicitly qualifying cos, PI, and E, include the following static import statement in your program:

```
import static java.lang.Math.*;
```

Annotations

Java annotations are a means for adding metadata to Java programs that can also affect the way that those programs are treated by tools and libraries. Annotations are declared with annotation type declarations, which are similar to interface declarations. Java annotations can appear in the following types of classes or interfaces:

- Class declaration
- Interface declaration
- Nested class declaration
- Nested interface declaration

You cannot include Java annotations directly in SQLJ programs, but you can include annotations in Java source code, and then include that source code in your SQLJ programs.

Example: Suppose that you declare the following marker annotation in a program called MyAnnot.java:

```
public @interface MyAnot { }
```

You also declare the following marker annotation in a program called MyAnnot2.java:

```
public @interface MyAnot2 { }
```

You can then use those annotations in an SQLJ program:

```
// Class annotations
@MyAnot2 public @MyAnot class  TestAnnotation
{
  // Field annotation
  @MyAnot
  private static final int field1 = 0;
  // Constructor annotation
  @MyAnot2 public @MyAnot TestAnnotation  () { }
   // Method annotation
   @MyAnot
   public static void main (String a[])
   {
      TestAnnotation TestAnnotation_o = new TestAnnotation();
      TestAnnotation_o.runThis();
   }
  // Inner class annotation
```

```
  public static @MyAnot class TestAnotherInnerClass { }
  // Inner interface annotation
  public static @MyAnot interface TestAnotInnerInterface { }
}
```

Enumerated types

An enumerated type is a data type that consists of a set of ordered values. The SDK for Java version 5 introduces the `enum` type for enumerated types.

You can include enums in the following places:

- In Java source files (.java files) that you include in an SQLJ program
- In SQLJ class declarations

Example: The TestEnum.sqlj class declaration includes an `enum` type:

```
public class TestEnum2
{
  public enum Color {
   RED,ORANGE,YELLOW,GREEN,BLUE,INDIGO,VIOLET}
  Color color;
  ...                          // Get the value of color
  switch (color) {
 case RED:
    System.out.println("Red is at one end of the spectrum.");
    #sql[ctx] { INSERT INTO MYTABLE VALUES (:color) };
   break;
 case VIOLET:
    System.out.println("Violet is on the other end of the spectrum.");
   break;
 case ORANGE:
 case YELLOW:
 case GREEN:
  case BLUE:
  case INDIGO:
    System.out.println("Everything else is in the middle.");
   break;
}
```

Generics

You can use generics in your Java programs to assign a type to a Java collection. The SQLJ translator tolerates Java generic syntax. Examples of generics that you can use in SQLJ programs are:

- A List of List objects:

  ```
  List <List<String>> strList2 = new ArrayList<List<String>>();
  ```

- A HashMap in which the key/value pair has the String type:

  ```
  Map <String,String> map = new HashMap<String,String>();
  ```

- A method that takes a List with elements of any type:

  ```
  public void mthd(List <?> obj) {
  ...
  }
  ```

Although you can use generics in SQLJ host variables, the value of doing so is limited because the SQLJ translator cannot determine the types of those host variables.

Enhanced for loop

The enhanced for lets you specify that a set of operations is performed on each member of a collection or array. You can use the iterator in the enhanced for loop in host expressions.

Example: INSERT each of the items in array names into table TAB.

```
String[] names = {"ABC","DEF","GHI"};
for (String n : names)
{
  #sql {INSERT INTO TAB (VARCHARCOL) VALUES(:n) };
}
```

Varargs

Varargs make it easier to pass an arbitrary number of values to a method. A Vararg in the last argument position of a method declaration indicates that the last arguments are an array or a sequence of arguments. An SQLJ program can use the passed arguments in host expressions.

Example: Pass an arbitrary number of parameters of type Object, to a method that inserts each parameter value into table TAB.

```
public void runThis(Object...  objects) throws SQLException
{
  for (Object obj : objects)
  {
    #sql { INSERT INTO TAB (VARCHARCOL) VALUES(:obj) };
  }
}
```

Transaction control in SQLJ applications

In SQLJ applications, as in other types of SQL applications, transaction control involves explicitly or implicitly committing and rolling back transactions, and setting the isolation level for transactions.

Setting the isolation level for an SQLJ transaction

To set the isolation level for a unit of work within an SQLJ program, use the SET TRANSACTION ISOLATION LEVEL clause.

The following table shows the values that you can specify in the SET TRANSACTION ISOLATION LEVEL clause and their IDS equivalents.

Table 6-3. Equivalent SQLJ and IDS isolation levels

SET TRANSACTION value	IDS isolation level
SERIALIZABLE	Repeatable read
REPEATABLE READ	Read stability
READ COMMITTED	Cursor stability
READ UNCOMMITTED	Uncommitted read

The isolation level affects the underlying JDBC connection as well as the SQLJ connection.

Committing or rolling back SQLJ transactions

If you disable autocommit for an SQLJ connection, you need to perform explicit commit or rollback operations.

You do this using execution clauses that contain the SQL COMMIT or ROLLBACK statements.

To commit a transaction in an SQLJ program, use a statement like this:

```
#sql [myConnCtx] {COMMIT};
```

To roll back a transaction in an SQLJ program, use a statement like this:

```
#sql [myConnCtx] {ROLLBACK};
```

Handling SQL errors and warnings in SQLJ applications

SQLJ clauses throw SQLExceptions when SQL errors occur, but not when most SQL warnings occur.

SQLJ generates an SQLException under the following circumstances:

- When any SQL statement returns a negative SQL error code
- When a SELECT INTO SQL statement returns a +100 SQL error code

You need to explicitly check for other SQL warnings.

- For SQL error handling, include try/catch blocks around SQLJ statements.
- For SQL warning handling, invoke the getWarnings method after every SQLJ statement.

Handling SQL errors in an SQLJ application

SQLJ clauses use the JDBC class java.sql.SQLException for error handling.

To handle SQL errors in SQLJ applications, following these steps:

1. Import the java.sql.SQLException class.
2. Use the Java error handling `try/catch` blocks to modify program flow when an SQL error occurs.
3. Obtain error information from the SQLException.

   You can use the getErrorCode method to retrieve SQL error codes and the getSQLState method to retrieve SQLSTATEs.

   If you are using the IBM Data Server Driver for JDBC and SQLJ, obtain additional information from the SQLException by casting it to a DB2Diagnosable object, in the same way that you obtain this information in a JDBC application.

The following code prints out the SQL error that occurred if a SELECT statement fails.

```
try {
  #sql [ctxt] {SELECT LASTNAME INTO :empname
    FROM EMPLOYEE WHERE EMPNO='000010'};
}
catch(SQLException e) {
  System.out.println("Error code returned: " + e.getErrorCode());
}
```

Handling SQL warnings in an SQLJ application

Other than a +100 SQL error code on a SELECT INTO statement, warnings from the data server do not throw SQLExceptions. To handle warnings from the data server, you need to give the program access to the java.sql.SQLWarning class.

If you want to retrieve data-server-specific information about a warning, you also need to give the program access to the com.ibm.db2.jcc.DB2Diagnosable interface and the com.ibm.db2.jcc.DB2Sqlca class. Then follow these steps:

1. Set up an execution context for that SQL clause. See "Control the execution of SQL statements in SQLJ" for information on how to set up an execution context.
2. To check for a warning from the data server, invoke the getWarnings method after you execute an SQLJ clause.

   getWarnings returns the first SQLWarning object that an SQL statement generates. Subsequent SQLWarning objects are chained to the first one.
3. To retrieve data-server-specific information from the SQLWarning object with the IBM Data Server Driver for JDBC and SQLJ, follow the instructions in "Handle an SQLException under the IBM Data Server Driver for JDBC and SQLJ".

The following example demonstrates how to retrieve an SQLWarning object for an SQL clause with execution context execCtx. The numbers to the right of selected statements correspond to the previously-described steps.

```
ExecutionContext execCtx=myConnCtx.getExecutionContext();    1
                                    // Get default execution context from
                                    // connection context
SQLWarning sqlWarn;
...
#sql [myConnCtx,execCtx] {SELECT LASTNAME INTO :empname
  FROM EMPLOYEE WHERE EMPNO='000010'};
if ((sqlWarn = execCtx.getWarnings()) != null)              2
System.out.println("SQLWarning " + sqlWarn);
```

Closing the connection to a data source in an SQLJ application

When you have finished with a connection to a data source, you need to close the connection to the data source. Doing so releases the connection context object's IDS and SQLJ resources immediately.

To close the connection to the data source, use one of the ConnectionContext.close methods.

- If you execute ConnectionContext.close() or ConnectionContext.close(ConnectionContext.CLOSE_CONNECTION), the connection context, as well as the connection to the data source, are closed.
- If you execute ConnectionContext.close(ConnectionContext.KEEP_CONNECTION) the connection context is closed, but the connection to the data source is not.

The following code closes the connection context, but does not close the connection to the data source.

```
...
ctx = new EzSqljctx(con0);          // Create a connection context object
                                    // from JDBC connection con0
...                                   // Perform various SQL operations
 EzSqljctx.close(ConnectionContext.KEEP_CONNECTION);
                                    // Close the connection context but keep
                                    // the connection to the data source open
```

Chapter 7. Preparing and running JDBC and SQLJ programs

The following topics contain information about preparing and running Java and SQLJ programs with the IBM Data Server Driver for JDBC and SQLJ.

Program preparation for JDBC programs

Preparing a Java program that contains only JDBC methods is the same as preparing any other Java program. You compile the program using the javac command. No precompile steps are required.

To prepare a program named sample.java, execute this command from the directory that contains the source file:

```
javac sample.java
```

Program preparation for SQLJ programs

Program preparation for SQLJ programs involves translating and compiling.

SQLJ on IBM Informix does not support static SQL; You must use dynamic SQL with the IBM Data Server Driver for JDBC and SQLJ on IBM Informix.

To prepare an SQLJ program, run the sqlj command from the command line to translate and compile the source code. For complete syntax for the sqlj command, see "sqlj - SQLJ translator" on page 14-215.

The SQLJ command generates a Java source program, optionally compiles the Java source program, and produces zero or more serialized profiles. You can compile the Java program separately, but the default behavior of the sqlj command is to compile the program. The SQLJ command runs without connecting to the database server.

Running JDBC and SQLJ programs

You run a JDBC or SQLJ program using the java command. Before you run the program, you need to ensure that the JVM can find all of the files that it needs.

To run a JDBC or SQLJ program, follow these steps:

1. Ensure that the program files can be found.
   - For an SQLJ program, put the serialized profiles for the program in the same directory as the class files for the program.
   - Include directories for the class files that are used by the program in the CLASSPATH.
2. Run the java command from the command line, with the top-level file name in the program as the argument.

   To run a program that is in the EzJava class, add the directory that contains EzJava to the CLASSPATH. Then run this command:

   ```
   java EzJava
   ```

Chapter 8. Security under the IBM Data Server Driver for JDBC and SQLJ

When you use the IBM Data Server Driver for JDBC and SQLJ, you choose a security mechanism by specifying a value for the `securityMechanism` property.

You can set this property in one of the following ways:

- If you use the DriverManager interface, set `securityMechanism` in a java.util.Properties object before you invoke the form of the getConnection method that includes the java.util.Properties parameter.
- If you use the DataSource interface, and you are creating and deploying your own DataSource objects, invoke the DataSource.setSecurityMechanism method after you create a DataSource object.

You can determine the security mechanism that is in effect for a connection by calling the DB2Connection.getDB2SecurityMechanism method.

The following table lists the security mechanisms that the IBM Data Server Driver for JDBC and SQLJ supports, and the data sources that support those security mechanisms.

Table 8-1. Database server support for IBM Data Server Driver for JDBC and SQLJ security mechanisms

Security mechanism	**Supported by**			
	DB2 Database for Linux, UNIX, and Windows	DB2 for z/OS	IBM Informix	DB2 for i
User ID and password	Yes	Yes	Yes	Yes
User ID only	Yes	Yes	Yes	Yes
User ID and encrypted password	Yes	Yes	Yes	Yes[2]
Encrypted user ID	Yes	Yes	No	No
Encrypted user ID and encrypted password	Yes	Yes	Yes	Yes[2]
Encrypted user ID and encrypted security-sensitive data	No	Yes	No	No
Encrypted user ID, encrypted password, and encrypted security-sensitive data	Yes	Yes	No	No
Kerberos[1]	Yes	Yes	No	Yes
Plugin[1]	Yes	No	No	No

Note:

1. Available for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity only.
2. The version of the data source must be DB2 for i V6R1 or later.

The following table lists the security mechanisms that the IBM Data Server Driver for JDBC and SQLJ supports, and the value that you need to specify for the `securityMechanism` property to specify each security mechanism.

The default security mechanism is CLEAR_TEXT_PASSWORD_SECURITY. If the server does not support CLEAR_TEXT_PASSWORD_SECURITY but supports ENCRYPTED_USER_AND_PASSWORD_SECURITY, the IBM Data Server Driver for JDBC and SQLJ driver updates the security mechanism to ENCRYPTED_USER_AND_PASSWORD_SECURITY and attempts to connect to the server. Any other mismatch in security mechanism support between the requester and the server results in an error.

Table 8-2. Security mechanisms supported by the IBM Data Server Driver for JDBC and SQLJ

Security mechanism	**securityMechanism property value**
User ID and password	DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY
User ID only	DB2BaseDataSource.USER_ONLY_SECURITY
User ID and encrypted password	DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY
Encrypted user ID	DB2BaseDataSource.ENCRYPTED_USER_ONLY_SECURITY
Encrypted user ID and encrypted password	DB2BaseDataSource.ENCRYPTED_USER_AND_PASSWORD_SECURITY
Encrypted user ID and encrypted security-sensitive data	DB2BaseDataSource.ENCRYPTED_USER_AND_DATA_SECURITY
Encrypted user ID, encrypted password, and encrypted security-sensitive data	DB2BaseDataSource.ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY
Kerberos	DB2BaseDataSource.KERBEROS_SECURITY
Plugin	DB2BaseDataSource.PLUGIN_SECURITY

User ID and password security under the IBM Data Server Driver for JDBC and SQLJ

With the IBM Data Server Driver for JDBC and SQLJ, one of the available security methods is user ID and password security.

To specify user ID and password security for a JDBC connection, use one of the following techniques.

For the DriverManager interface: You can specify the user ID and password directly in the DriverManager.getConnection invocation. For example:

```
import java.sql.*;          // JDBC base
...
String id = "dbadm";        // Set user ID
String pw = "dbadm";        // Set password
String url = "jdbc:ids://mvs1.sj.ibm.com:5021/san_jose";
                            // Set URL for the data source

Connection con = DriverManager.getConnection(url, id, pw);
                            // Create connection
```

Another method is to set the user ID and password directly in the URL string. For example:

```
import java.sql.*;         // JDBC base
...
String url =
  "jdbc:ids://mvs1.sj.ibm.com:5021/san_jose:user=dbadm;password=dbadm;";

                           // Set URL for the data source
Connection con = DriverManager.getConnection(url);
                           // Create connection
```

Alternatively, you can set the user ID and password by setting the `user` and `password` properties in a Properties object, and then invoking the form of the getConnection method that includes the Properties object as a parameter. Optionally, you can set the `securityMechanism` property to indicate that you are using user ID and password security. For example:

```
import java.sql.*;                         // JDBC base
import com.ibm.db2.jcc.*;                  // IBM Data Server Driver for JDBC
                                           // and SQLJ implementation of JDBC
...
Properties properties = new java.util.Properties();
                                           // Create Properties object
properties.put("user", "dbadm");           // Set user ID for the connection
properties.put("password", "dbadm");       // Set password for the connection
properties.put("securityMechanism",
  new String("" + com.ibm.db2.jcc.DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY +
  ""));
                                           // Set security mechanism to
                                           // user ID and password
String url = "jdbc:ids://mvs1.sj.ibm.com:5021/san_jose";
                                           // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
                                           // Create connection
```

For the DataSource interface: you can specify the user ID and password directly in the DataSource.getConnection invocation. For example:

```
import java.sql.*;                         // JDBC base
import com.ibm.db2.jcc.*;                  // IBM Data Server Driver for JDBC
                                           // and SQLJ implementation of JDBC
...
Context ctx=new InitialContext();          // Create context for JNDI
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");
                                           // Get DataSource object
String id = "dbadm";                       // Set user ID
String pw = "dbadm";                       // Set password
Connection con = ds.getConnection(id, pw);
                                           // Create connection
```

Alternatively, if you create and deploy the DataSource object, you can set the user ID and password by invoking the DataSource.setUser and DataSource.setPassword methods after you create the DataSource object. Optionally, you can invoke the DataSource.setSecurityMechanism method property to indicate that you are using user ID and password security. For example:

```
...
com.ibm.db2.jcc.DB2SimpleDataSource ds =       // Create DB2SimpleDataSource object
  new com.ibm.db2.jcc.DB2SimpleDataSource();
ds.setDriverType(4);                           // Set driver type
ds.setDatabaseName("san_jose");                // Set location
ds.setServerName("mvs1.sj.ibm.com");           // Set server name
ds.setPortNumber(5021);                        // Set port number
ds.setUser("dbadm");                           // Set user ID
ds.setPassword("dbadm");                       // Set password
ds.setSecurityMechanism(
```

```
  com.ibm.db2.jcc.DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY);
                                                  // Set security mechanism to
                                                  // user ID and password
```

User ID-only security under the IBM Data Server Driver for JDBC and SQLJ

With the IBM Data Server Driver for JDBC and SQLJ, one of the available security methods is user-ID only security.

To specify user ID security for a JDBC connection, use one of the following techniques.

For the DriverManager interface: Set the user ID and security mechanism by setting the `user` and `securityMechanism` properties in a Properties object, and then invoking the form of the getConnection method that includes the Properties object as a parameter. For example:

```
import java.sql.*;                      // JDBC base
import com.ibm.db2.jcc.*;               // IBM Data Server Driver
                                        // for JDBC and SQLJ
                                        // implementation of JDBC
...
Properties properties = new Properties();
                                        // Create a Properties object
properties.put("user", "db2adm");       // Set user ID for the connection
properties.put("securityMechanism",
  new String("" + com.ibm.db2.jcc.DB2BaseDataSource.USER_ONLY_SECURITY + ""));
                                        // Set security mechanism to
                                        // user ID only
String url = "jdbc:ids://mvs1.sj.ibm.com:5021/san_jose";
                                        // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
                                        // Create the connection
```

For the DataSource interface: If you create and deploy the DataSource object, you can set the user ID and security mechanism by invoking the DataSource.setUser and DataSource.setSecurityMechanism methods after you create the DataSource object. For example:

```
import java.sql.*;                      // JDBC base
import com.ibm.db2.jcc.*;               // IBM Data Server Driver
                                        // for JDBC and SQLJ
                                        // implementation of JDBC
...
com.ibm.db2.jcc.DB2SimpleDataSource db2ds =
  new com.ibm.db2.jcc.DB2SimpleDataSource();
                                        // Create DB2SimpleDataSource object
db2ds.setDriverType(4);                 // Set the driver type
db2ds.setDatabaseName("san_jose");      // Set the location
db2ds.setServerName("mvs1.sj.ibm.com");
                                        // Set the server name
db2ds.setPortNumber(5021);              // Set the port number
db2ds.setUser("db2adm");                // Set the user ID
db2ds.setSecurityMechanism(
  com.ibm.db2.jcc.DB2BaseDataSource.USER_ONLY_SECURITY);
                                        // Set security mechanism to
                                        // user ID only
```

Encrypted password, user ID, or user ID and password security under the IBM Data Server Driver for JDBC and SQLJ

IBM Data Server Driver for JDBC and SQLJ supports encrypted password security, encrypted user ID security, or encrypted user ID and encrypted password security for accessing data sources.

Connections to IBM Informix servers can use encrypted password security or encrypted user ID and encrypted password security. For encrypted password security or encrypted user ID and encrypted password security, the IBM Java Cryptography Extension (ibmjceprovidere.jar) must be installed on your client. The IBM JCE is part of the IBM SDK for Java, Version 1.4.2 or later.

Restriction: Because the IBM SDK for Java is not available on Mac OS X, IBM Data Server Driver for JDBC and SQLJ encrypted password security or encrypted user ID and encrypted password security is not available for Mac OS X clients.

Connections to DB2 for i V6R1 or later servers can use encrypted password security or encrypted user ID and encrypted password security. For encrypted password security or encrypted user ID and encrypted password security, the IBM Java Cryptography Extension (ibmjceprovidere.jar) must be installed on your client. The IBM JCE is part of the IBM SDK for Java, Version 1.4.2 or later.

To specify encrypted user ID or encrypted password security for a JDBC connection, use one of the following techniques.

For the DriverManager interface: Set the user ID, password, and security mechanism by setting the `user`, `password`, and `securityMechanism` properties in a Properties object, and then invoking the form of the getConnection method that includes the Properties object as a parameter. For example, use code like this to set the user ID and encrypted password security mechanism, with AES encryption:

```
import java.sql.*;                          // JDBC base
import com.ibm.db2.jcc.*;                   // IBM Data Server Driver for JDBC
                                            // and SQLJ implementation of JDBC
...
Properties properties = new Properties(); // Create a Properties object
properties.put("user", "dbadm");          // Set user ID for the connection
properties.put("password", "dbadm");      // Set password for the connection
properties.put("securityMechanism", "2");
  new String("" + com.ibm.db2.jcc.DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY +
  ""));
                                            // Set security mechanism to
                                            // user ID and encrypted password
properties.put("encryptionAlgorithm", "2");
                                            // Request AES security
String url = "jdbc:ids://mvs1.sj.ibm.com:5021/san_jose";
                                            // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
                                            // Create the connection
```

For the DataSource interface: If you create and deploy the DataSource object, you can set the user ID, password, and security mechanism by invoking the DataSource.setUser, DataSource.setPassword, and DataSource.setSecurityMechanism methods after you create the DataSource object. For example, use code like this to set the encrypted user ID and encrypted password security mechanism, with AES encryption:

```
import java.sql.*;                          // JDBC base
import com.ibm.db2.jcc.*;                   // IBM Data Server Driver for JDBC
                                            // and SQLJ implementation of JDBC
...
com.ibm.db2.jcc.DB2SimpleDataSource ds =
  new com.ibm.db2.jcc.DB2SimpleDataSource();
                                            // Create the DataSource object
ds.setDriverType(4);                        // Set the driver type
ds.setDatabaseName("san_jose");             // Set the location
ds.setServerName("mvs1.sj.ibm.com");
                                            // Set the server name
ds.setPortNumber(5021);                     // Set the port number
ds.setUser("db2adm");                       // Set the user ID
ds.setPassword("db2adm");                   // Set the password
ds.setSecurityMechanism(
  com.ibm.db2.jcc.DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY);
                                            // Set security mechanism to
                                            // User ID and encrypted password
ds.setEncryptionAlgorithm(2);               // Request AES encryption
```

IBM Data Server Driver for JDBC and SQLJ trusted context support

The IBM Data Server Driver for JDBC and SQLJ provides methods that allow you to establish and use trusted connections in Java programs.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to:
  - DB2 Database for Linux, UNIX, and Windows Version 9.5 or later
  - DB2 for z/OS Version 9.1 or later
  - IBM Informix Version 11.70 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 9.1 or later

A three-tiered application model consists of a database server, a middleware server such as WebSphere Application Server, and end users. With this model, the middleware server is responsible for accessing the database server on behalf of end users. Trusted context support ensures that an end user's database identity and database privileges are used when the middleware server performs any database requests on behalf of that end user.

A trusted context is an object that the database administrator defines that contains a system authorization ID and a set of trust attributes. Currently, for IDS database servers, a database connection is the only type of context that is supported. The trust attributes identify a set of characteristics of a connection that are required for the connection to be considered a trusted connection. The relationship between a database connection and a trusted context is established when the connection to the database server is first created, and that relationship remains for the life of the database connection.

After a trusted context is defined, and an initial trusted connection to the data server is made, the middleware server can use that database connection under a different user without reauthenticating the new user at the database server.

To avoid vulnerability to security breaches, an application server that uses these trusted methods should not use untrusted connection methods.

The DB2ConnectionPoolDataSource class provides several versions of the getDB2TrustedPooledConnection method, and the DB2XADataSource class

provides several versions of the getDB2TrustedXAConnection method, which allow an application server to establish the initial trusted connection. You choose a method based on the types of connection properties that you pass and whether you use Kerberos security. When an application server calls one of these methods, the IBM Data Server Driver for JDBC and SQLJ returns an Object[] array with two elements:

- The first element contains a connection instance for the initial connection.
- The second element contains a unique cookie for the connection instance. The cookie is generated by the JDBC driver and is used for authentication during subsequent connection reuse.

The DB2PooledConnection class provides several versions of the getDB2Connection method, and the DB2Connection class provides several versions of the reuseDB2Connection method, which allow an application server to reuse an existing trusted connection on behalf of a new user. The application server uses the method to pass the following items to the new user:

- The cookie from the initial connection
- New connection properties for the reused connection

The JDBC driver checks that the supplied cookie matches the cookie of the underlying trusted physical connection, to ensure that the connection request originates from the application server that established the trusted physical connection. If the cookies match, the connection becomes available for immediate use by this new user, with the new properties.

Example: Obtain the initial trusted connection:

```
// Create a DB2ConnectionPoolDataSource instance
com.ibm.db2.jcc.DB2ConnectionPoolDataSource dataSource =
  new com.ibm.db2.jcc.DB2ConnectionPoolDataSource();
// Set properties for this instance
dataSource.setDatabaseName ("STLEC1");
dataSource.setServerName ("v7ec167.svl.ibm.com");
dataSource.setDriverType (4);
dataSource.setPortNumber(446);
java.util.Properties properties = new java.util.Properties();
// Set other properties using
// properties.put("property", "value");
// Supply the user ID and password for the connection
String user = "user";
String password = "password";
// Call getDB2TrustedPooledConnection to get the trusted connection
// instance and the cookie for the connection
Object[] objects = dataSource.getDB2TrustedPooledConnection(
                         user,password, properties);
```

Example: Reuse an existing trusted connection:

```
// The first item that was obtained from the previous getDB2TrustedPooledConnection
// call is a connection object. Cast it to a PooledConnection object.
javax.sql.PooledConnection pooledCon =
  (javax.sql.PooledConnection)objects[0];
properties = new java.util.Properties();
// Set new properties for the reused object using
// properties.put("property", "value");
// The second item that was obtained from the previous getDB2TrustedPooledConnection
// call is the cookie for the connection. Cast it as a byte array.
byte[] cookie = ((byte[])(objects[1]);
// Supply the user ID for the new connection.
String newuser = "newuser";
// Supply the name of a mapping service that maps a workstation user
// ID to a z/OS RACF ID
```

```
String userRegistry = "registry";
// Do not supply any security token data to be traced.
byte[] userSecTkn = null;
// Do not supply a previous user ID.
String originalUser = null;
// Call getDB2Connection to get the connection object for the new
// user.
java.sql.Connection con =
  ((com.ibm.db2.jcc.DB2PooledConnection)pooledCon).getDB2Connection(
     cookie,newuser,password,userRegistry,userSecTkn,originalUser,properties);
```

IBM Data Server Driver for JDBC and SQLJ support for SSL

The IBM Data Server Driver for JDBC and SQLJ provides support for the Secure Sockets Layer (SSL) through the Java Secure Socket Extension (JSSE).

You can use SSL support in your Java applications if you use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS Version 9 or later, to DB2 Database for Linux, UNIX, and Windows Version 9.1, Fix Pack 2 or later, or to IBM Informix Version 11.50 or later.

If you use SSL support for a connection to a DB2 for z/OS data server, and the z/OS version is V1.8, V1.9, or V1.10, the appropriate PTF for APAR PK72201 must be applied to Communication Server for z/OS IP Services.

To use SSL connections, you need to:

- Configure connections to the data server to use SSL.
- Configure your Java Runtime Environment to use SSL.

Configuring connections under the IBM Data Server Driver for JDBC and SQLJ to use SSL

To configure database connections under the IBM Data Server Driver for JDBC and SQLJ to use SSL, you need to set the DB2BaseDataSource.sslConnection property to `true`.

Before a connection to a data source can use SSL, the port to which the application connects must be configured in the database server as the SSL listener port.

1. Set DB2BaseDataSource.sslConnection on a Connection or DataSource instance.
2. Optional: Set DB2BaseDataSource.sslTrustStoreLocation on a Connection or DataSource instance to identify the location of the truststore. Setting the sslTrustStoreLocation property is an alternative to setting the Java javax.net.ssl.trustStore property. If you set DB2BaseDataSource.sslTrustStoreLocation, javax.net.ssl.trustStore is not used.
3. Optional: Set DB2BaseDataSource.sslTrustStorePassword on a Connection or DataSource instance to identify the truststore password. Setting the sslTrustStorePassword property is an alternative to setting the Java javax.net.ssl.trustStorePassword property. If you set DB2BaseDataSource.sslTrustStorePassword, javax.net.ssl.trustStorePassword is not used.

The following example demonstrates how to set the sslConnection property on a Connection instance:

```
java.util.Properties properties = new java.util.Properties();
properties.put("user", "xxxx");
properties.put("password", "yyyy");
```

```
properties.put("sslConnection", "true");
java.sql.Connection con =
  java.sql.DriverManager.getConnection(url, properties);
```

Configuring the Java Runtime Environment to use SSL

Before you can use Secure Sockets Layer (SSL) connections in your JDBC and SQLJ applications, you need to configure the Java Runtime Environment to use SSL.

Before you can configure your Java Runtime Environment for SSL, you need to satisfy the following prerequisites:

- The Java Runtime Environment must include a Java security provider. The IBM JSSE provider or the SunJSSE provider must be installed. The IBM JSSE provider is automatically installed with the IBM SDK for Java.

  Restriction: You can only use the SunJSSE provider only with an Oracle Java Runtime Environment. The SunJSSE provider does not work with an IBM Java Runtime Environment.
- SSL support must be configured on the database server.

To configure your Java Runtime Environment to use SSL, follow these steps.

1. Import a certificate from the database server to a Java truststore on the client.

   Use the Java keytool utility to import the certificate into the truststore.

   For example, suppose that the server certificate is stored in a file named jcc.cacert. Issue the following keytool utility statement to read the certificate from file jcc.cacert, and store it in a truststore named cacerts.

   ```
   keytool -import -file jcc.cacert -keystore cacerts
   ```
2. Configure the Java Runtime Environment for the Java security providers by adding entries to the java.security file.

   The format of a security provider entry is:

   ```
   security.provider.n=provider-package-name
   ```

   A provider with a lower value of *n* takes precedence over a provider with a higher value of *n*.

   The Java security provider entries that you add depend on whether you use the IBM JSSE provider or the SunJSSE provider.

   - If you use the SunJSSE provider, add entries for the Oracle security providers to your java.security file.
   - If you use the IBM JSSE provider, use one of the following methods:
     - **Use the IBMJSSE2 provider (supported for the IBM SDK for Java 1.4.2 and later):**

       Recommendation: Use the IBMJSSE2 provider, and use it in FIPS mode.

       - If you do not need to operate in FIPS-compliant mode:
         - For the IBM SDK for Java 1.4.2, add an entry for the IBMJSSE2Provider to the java.security file. Ensure that an entry for the IBMJCE provider is in the java.security file. The java.security file that is shipped with the IBM SDK for Java contains an entry for entries for IBMJCE.
         - For later versions of the IBM SDK for Java, ensure that entries for the IBMJSSE2Provider and the IBMJCE provider are in the java.security file. The java.security file that is shipped with the IBM SDK for Java contains entries for those providers.
       - If you need to operate in FIPS-compliant mode:

- Add an entry for the IBMJCEFIPS provider to your java.security file before the entry for the IBMJCE provider. Do not remove the entry for the IBMJCE provider.
- Enable FIPS mode in the IBMJSSE2 provider. See step 3.

– **Use the IBMJSSE provider (supported for the IBM SDK for Java 1.4.2 only):**
  - If you do not need to operate in FIPS-compliant mode, ensure that entries for the IBMJSSEProvider and the IBMJCE provider are in the java.security file. The java.security file that is shipped with the IBM SDK for Java contains entries for those providers.
  - If you need to operate in FIPS-compliant mode, add entries for the FIPS-approved provider IBMJSSEFIPSProvider and the IBMJCEFIPS provider to your java.security file, before the entry for the IBMJCE provider.

Restriction: If you use the IBMJSSE provider on the Solaris operating system, you need to include an entry for the SunJSSE provider before entries for the IBMJCE, IBMJCEFIPS, IBMJSSE, or IBMJSSE2 providers.

Example: Use a java.security file similar to this one if you need to run in FIPS-compliant mode, and you enable FIPS mode in the IBMJSSE2 provider:

```
# Set the Java security providers
security.provider.1=com.ibm.jsse2.IBMJSSEProvider2
security.provider.2=com.ibm.crypto.fips.provider.IBMJCEFIPS
security.provider.3=com.ibm.crypto.provider.IBMJCE
security.provider.4=com.ibm.security.jgss.IBMJGSSProvider
security.provider.5=com.ibm.security.cert.IBMCertPath
security.provider.6=com.ibm.security.sasl.IBMSASL
```

Example: Use a java.security file similar to this one if you need to run in FIPS-compliant mode, and you use the IBMJSSE provider:

```
# Set the Java security providers
security.provider.1=com.ibm.fips.jsse.IBMJSSEFIPSProvider
security.provider.2=com.ibm.crypto.fips.provider.IBMJCEFIPS
security.provider.3=com.ibm.crypto.provider.IBMJCE
security.provider.4=com.ibm.security.jgss.IBMJGSSProvider
security.provider.5=com.ibm.security.cert.IBMCertPath
security.provider.6=com.ibm.security.sasl.IBMSASL
```

Example: Use a java.security file similar to this one if you use the SunJSSE provider:

```
# Set the Java security providers
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.rsajca.Provider
security.provider.3=com.sun.crypto.provider.SunJCE
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
```

3. If you plan to use the IBM Data Server Driver for JDBC and SQLJ in FIPS-compliant mode, you need to set the `com.ibm.jsse2.JSSEFIPS` Java system property:

   ```
   com.ibm.jsse2.JSSEFIPS=true
   ```

   Restriction: Non-FIPS-mode JSSE applications cannot run in a JVM that is in FIPS mode.

   Restriction: When the IBMJSSE2 provider runs in FIPS mode, it cannot use hardware cryptography.
4. Configure the Java Runtime Environment for the SSL socket factory providers by adding entries to the java.security file.

The format of SSL socket factory provider entries are:

```
ssl.SocketFactory.provider=provider-package-name
ssl.ServerSocketFactory.provider=provider-package-name
```

Specify the SSL socket factory provider for the Java security provider that you are using.

Example: Include SSL socket factory provider entries like these in the java.security file when you enable FIPS mode in the IBMJSSE2 provider:

```
# Set the SSL socket factory provider
ssl.SocketFactory.provider=com.ibm.jsse2.SSLSocketFactoryImpl
ssl.ServerSocketFactory.provider=com.ibm.jsse2.SSLServerSocketFactoryImpl
```

Example: Include SSL socket factory provider entries like these in the java.security file when you enable FIPS mode in the IBMJSSE provider:

```
# Set the SSL socket factory provider
ssl.SocketFactory.provider=com.ibm.fips.jsse.JSSESocketFactory
ssl.ServerSocketFactory.provider=com.ibm.fips.jsse.JSSEServerSocketFactory
```

Example: Include SSL socket factory provider entries like these when you use the SunJSSE provider:

```
# Set the SSL socket factory provider
ssl.SocketFactory.provider=com.sun.net.ssl.internal.ssl.SSLSocketFactoryImpl
ssl.ServerSocketFactory.provider=com.sun.net.ssl.internal.ssl.SSLServerSocketFactoryImpl
```

5. Configure Java system properties to use the truststore.

   To do that, set the following Java system properties:

   javax.net.ssl.trustStore
   Specifies the name of the truststore that you specified with the -keystore parameter in the keytool utility in step 1 on page 8-9.

   If the IBM Data Server Driver for JDBC and SQLJ property DB2BaseDataSource.sslTrustStoreLocation is set, its value overrides the javax.net.ssl.trustStore property value.

   javax.net.ssl.trustStorePassword (optional)
   Specifies the password for the truststore. You do not need to set a truststore password. However, if you do not set the password, you cannot protect the integrity of the truststore.

   If the IBM Data Server Driver for JDBC and SQLJ property DB2BaseDataSource.sslTrustStorePassword is set, its value overrides the javax.net.ssl.trustStorePassword property value.

   Example: One way that you can set Java system properties is to specify them as the arguments of the -D option when you run a Java application. Suppose that you want to run a Java application named MySSL.java, which accesses a data source using an SSL connection. You have defined a truststore named cacerts. The following command sets the truststore name when you run the application.

   ```
   java -Djavax.net.ssl.trustStore=cacerts MySSL
   ```

Chapter 9. Problem diagnosis with the IBM Data Server Driver for JDBC and SQLJ

The IBM Data Server Driver for JDBC and SQLJ includes diagnostic tools and traces for diagnosing problems during connection and SQL statement execution.

Testing a data server connection

Run the DB2Jcc utility to test a connection to a data server. You provide DB2Jcc with the URL for the data server, for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity. DB2Jcc attempts to connect to the data server, and to execute an SQL statement and a DatabaseMetaData method. If the connection or statement execution fails, DB2Jcc provides diagnostic information about the failure.

Collecting JDBC trace data

Use one of the following procedures to start the trace:

Procedure 1: For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, the recommended method is to start the trace by setting the db2.jcc.override.traceFile property or the db2.jcc.override.traceDirectory property in the IBM Data Server Driver for JDBC and SQLJ configuration properties file. You can set the db2.jcc.tracePolling and db2.jcc.tracePollingInterval properties before you start the driver to allow you to change global configuration trace properties while the driver is running.

Procedure 2: If you use the DataSource interface to connect to a data source, follow this method to start the trace:

1. Invoke the DB2BaseDataSource.setTraceLevel method to set the type of tracing that you need. The default trace level is `TRACE_ALL`. See "Properties for the IBM Data Server Driver for JDBC and SQLJ" for information on how to specify more than one type of tracing.
2. Invoke the DB2BaseDataSource.setJccLogWriter method to specify the trace destination and turn the trace on.

Procedure 3:

If you use the DataSource interface to connect to a data source, invoke the javax.sql.DataSource.setLogWriter method to turn the trace on. With this method, `TRACE_ALL` is the only available trace level.

If you use the DriverManager interface to connect to a data source, follow this procedure to start the trace.

1. Invoke the DriverManager.getConnection method with the `traceLevel` property set in the *info* parameter or *url* parameter for the type of tracing that you need. The default trace level is `TRACE_ALL`. See "Properties for the IBM Data Server Driver for JDBC and SQLJ" for information on how to specify more than one type of tracing.
2. Invoke the DriverManager.setLogWriter method to specify the trace destination and turn the trace on.

After a connection is established, you can turn the trace off or back on, change the trace destination, or change the trace level with the DB2Connection.setJccLogWriter method. To turn the trace off, set the logWriter value to `null`.

The logWriter property is an object of type java.io.PrintWriter. If your application cannot handle java.io.PrintWriter objects, you can use the `traceFile` property to specify the destination of the trace output. To use the `traceFile` property, set the logWriter property to `null`, and set the `traceFile` property to the name of the file to which the driver writes the trace data. This file and the directory in which it resides must be writable. If the file already exists, the driver overwrites it.

Procedure 4: If you are using the DriverManager interface, specify the `traceFile` and `traceLevel` properties as part of the URL when you load the driver. For example:

```
String url = "jdbc:ids://sysmvs1.stl.ibm.com:5021/san_jose" +
 ":traceFile=/u/db2p/jcctrace;" +
 "traceLevel=" + com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS + ";";
```

Procedure 5: Use DB2TraceManager methods. The DB2TraceManager class provides the ability to suspend and resume tracing of any type of log writer.

Example of starting a trace using configuration properties: For a complete example of using configuration parameters to collect trace data, see "Example of using configuration properties to start a JDBC trace".

Trace example program: For a complete example of a program for tracing under the IBM Data Server Driver for JDBC and SQLJ, see "Example of a trace program under the IBM Data Server Driver for JDBC and SQLJ".

DB2Jcc - IBM Data Server Driver for JDBC and SQLJ diagnostic utility

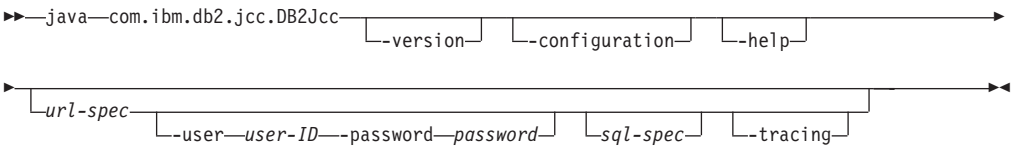
DB2Jcc verifies that a data server is configured for database access.

To verify the connection, DB2Jcc connects to the specified data server, executes an SQL statement, and executes a java.sql.DatabaseMetadata method.

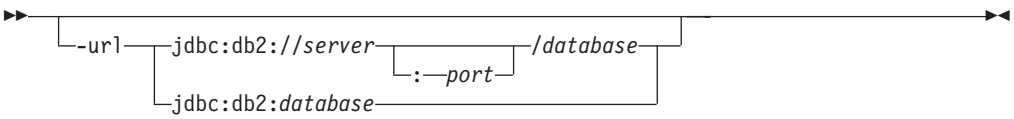
Authorization

The user ID under which DB2Jcc runs must have the authority to connect to the specified data server and to execute the specified SQL statement.

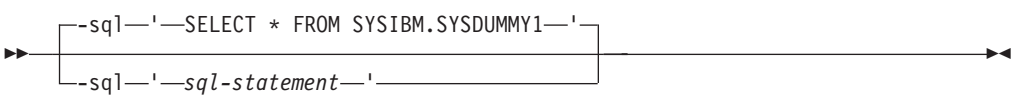
DB2Jcc Syntax



url-spec:



sql-spec:



DB2Jcc parameters

-help
Specifies that DB2Jcc describes each of the options that it supports. If any other options are specified with -help, they are ignored.

-version
Specifies that DB2Jcc displays the driver name and version.

-configuration
Specifies that DB2Jcc displays driver configuration information.

-url
Specifies the URL for the data server for which the connection is being tested. The URL can be a URL for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity or IBM Data Server Driver for JDBC and SQLJ type 4 connectivity. The variable parts of the `-url` value are:

server
The domain name or IP address of the operating system on which the database server resides. *server* is used only for type 4 connectivity.

port
The TCP/IP server port number that is assigned to the data server. The default is 446. *port* is used only for type 4 connectivity.

database
A name for the database server for which the profile is to be customized.

If the connection is to a DB2 for z/OS server, *database* is the DB2 location name that is defined during installation. All characters in this value must be uppercase characters. You can determine the location name by executing the following SQL statement on the server:

```
SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
```

If the connection is to a DB2 Database for Linux, UNIX, and Windows server, *database* is the database name that is defined during installation.

If the connection is to an IBM Informix data server, *database* is the database name. The name is case-insensitive. The server converts the name to lowercase.

If the connection is to an IBM Cloudscape server, the *database* is the fully-qualified name of the file that contains the database. This name must be enclosed in double quotation marks ("). For example:

```
"c:/databases/testdb"
```

-user *user-ID*
Specifies the user ID that is to be used to test the connection to the data server.

-password *password*
Specifies the password for the user ID that is to be used to test the connection to the data server.

-sql '*sql-statement***'**
Specifies the SQL statement that is sent to the data server to verify the connection. If the -sql parameter is not specified, this SQL statement is sent to the data server:

```
SELECT * FROM SYSIBM.SYSDUMMY1
```

-tracing
Specifies that tracing is enabled. The trace destination is System.out.

If you omit the -tracing parameter, tracing is disabled.

Examples

Example: Test the connection to a data server using IBM Data Server Driver for JDBC and SQLJ type 4 connectivity. Use the default SQL statement to test the connection. Enable tracing for the test.

```
java com.ibm.db2.jcc.DB2Jcc
-url jdbc:db2://mysys.myloc.svl.ibm.com:446/MYDB
-user db2user -password db2pass -tracing
```

Example: Test the connection to a data server using IBM Data Server Driver for JDBC and SQLJ type 2 connectivity. Use the following SQL statement to test the connection:

```
SELECT COUNT(*) FROM EMPLOYEE
```

Disable tracing for the test.

```
java com.ibm.db2.jcc.DB2Jcc
-url jdbc:db2:MYDB
-user db2user -password db2pass
-sql 'SELECT COUNT(*) FROM EMPLOYEE'
```

Examples of using configuration properties to start a JDBC trace

You can control tracing of JDBC applications without modifying those applications.

Example of writing trace data to one trace file for each connection

Suppose that you want to collect trace data for a program named Test.java, which uses IBM Data Server Driver for JDBC and SQLJ type 4 connectivity. Test.java does no tracing, and you do not want to modify the program, so you enable tracing using configuration properties. You want your trace output to have the following characteristics:

- Trace information for each connection on the same DataSource is written to a separate trace file. Output goes into a directory named /Trace.
- Each trace file name begins with jccTrace1.
- If trace files with the same names already exist, the trace data is appended to them.

Although Test.java does not contain any code to do tracing, you want to set the configuration properties so that if the application is modified in the future to do tracing, the settings within the program will take precedence over the settings in the configuration properties. To do that, use the set of configuration properties that begin with db2.jcc, not db2.jcc.override.

The configuration property settings look like this:

- db2.jcc.traceDirectory=/Trace
- db2.jcc.traceFile=jccTrace1
- db2.jcc.traceFileAppend=true

You want the trace settings to apply only to your stand-alone program Test.java, so you create a file with these settings, and then refer to the file when you invoke the Java program by specifying the -Ddb2.jcc.propertiesFile option. Suppose that the file that contains the settings is /Test/jcc.properties. To enable tracing when you run Test.java, you issue a command like this:

```
java -Ddb2.jcc.propertiesFile=/Test/jcc.properties Test
```

Suppose that Test.java creates two connections for one DataSource. The program does not define a logWriter object, so the driver creates a global logWriter object for the trace output. When the program completes, the following files contain the trace data:

- /Trace/jccTrace1_global_0
- /Trace/jccTrace1_global_1

Example of a trace program under the IBM Data Server Driver for JDBC and SQLJ

You might want to write a single class that includes methods for tracing under the DriverManager interface, as well as the DataSource interface.

The following example shows such a class. The example uses IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Figure 9-1. Example of tracing under the IBM Data Server Driver for JDBC and SQLJ

```
public class TraceExample
{

  public static void main(String[] args)
  {
    sampleConnectUsingSimpleDataSource();
    sampleConnectWithURLUsingDriverManager();
  }

  private static void sampleConnectUsingSimpleDataSource()
  {
    java.sql.Connection c = null;
    java.io.PrintWriter printWriter =
     new java.io.PrintWriter(System.out, true);
                                       // Prints to console, true means
                                       // auto-flush so you don't lose trace
    try {
      javax.sql.DataSource ds =
       new com.ibm.db2.jcc.DB2SimpleDataSource();
      ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setServerName("sysmvs1.stl.ibm.com");
      ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setPortNumber(5021);
      ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setDatabaseName("san_jose");
      ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setDriverType(4);

     ds.setLogWriter(printWriter);    // This turns on tracing

     // Refine the level of tracing detail
     ((com.ibm.db2.jcc.DB2BaseDataSource) ds).
      setTraceLevel(com.ibm.db2.jcc.DB2SimpleDataSource.TRACE_CONNECTS |
      com.ibm.db2.jcc.DB2SimpleDataSource.TRACE_DRDA_FLOWS);
```

```
    // This connection request is traced using trace level
    // TRACE_CONNECTS | TRACE_DRDA_FLOWS
    c = ds.getConnection("myname", "mypass");

    // Change the trace level to TRACE_ALL
    // for all subsequent requests on the connection
    ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(printWriter,
     com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL);
    // The following INSERT is traced using trace level TRACE_ALL
    java.sql.Statement s1 = c.createStatement();
    s1.executeUpdate("INSERT INTO sampleTable(sampleColumn) VALUES(1)");
    s1.close();

    // This code disables all tracing on the connection
    ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(null);

    // The following INSERT statement is not traced
    java.sql.Statement s2 = c.createStatement();
    s2.executeUpdate("INSERT INTO sampleTable(sampleColumn) VALUES(1)");
    s2.close();

    c.close();
   }
   catch(java.sql.SQLException e) {
    com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e,
     printWriter, "[TraceExample]");
   }
   finally {
     cleanup(c, printWriter);
     printWriter.flush();
   }
}

// If the code ran successfully, the connection should
// already be closed. Check whether the connection is closed.
// If so, just return.
// If a failure occurred, try to roll back and close the connection.

private static void cleanup(java.sql.Connection c,
 java.io.PrintWriter printWriter)
{
  if(c == null) return;

  try {
   if(c.isClosed()) {
      printWriter.println("[TraceExample] " +
       "The connection was successfully closed");
      return;
   }

   // If we get to here, something has gone wrong.
   // Roll back and close the connection.
   printWriter.println("[TraceExample] Rolling back the connection");
   try {
     c.rollback();
   }
   catch(java.sql.SQLException e) {
     printWriter.println("[TraceExample] " +
      "Trapped the following java.sql.SQLException while trying to roll back:");
     com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e, printWriter,
      "[TraceExample]");
     printWriter.println("[TraceExample] " +
      "Unable to roll back the connection");
    }
    catch(java.lang.Throwable e) {
      printWriter.println("[TraceExample] Trapped the " +
       "following java.lang.Throwable while trying to roll back:");
```

```
      com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e,
       printWriter, "[TraceExample]");
      printWriter.println("[TraceExample] Unable to " +
       "roll back the connection");
    }

    // Close the connection
    printWriter.println("[TraceExample] Closing the connection");
    try {
      c.close();
    }
    catch(java.sql.SQLException e) {
      printWriter.println("[TraceExample] Exception while " +
       "trying to close the connection");
      printWriter.println("[TraceExample] Deadlocks could " +
       "occur if the connection is not closed.");
      com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e, printWriter,
       "[TraceExample]");
    }
    catch(java.lang.Throwable e) {
      printWriter.println("[TraceExample] Throwable caught " +
       "while trying to close the connection");
      printWriter.println("[TraceExample] Deadlocks could " +
       "occur if the connection is not closed.");
      com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e, printWriter,
       "[TraceExample]");
    }
  }
  catch(java.lang.Throwable e) {
    printWriter.println("[TraceExample] Unable to " +
     "force the connection to close");
    printWriter.println("[TraceExample] Deadlocks " +
     "could occur if the connection is not closed.");
    com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e, printWriter,
     "[TraceExample]");
  }
}
private static void sampleConnectWithURLUsingDriverManager()
{
  java.sql.Connection c = null;

  // This time, send the printWriter to a file.
  java.io.PrintWriter printWriter = null;
  try {
    printWriter =
      new java.io.PrintWriter(
        new java.io.BufferedOutputStream(
          new java.io.FileOutputStream("/temp/driverLog.txt"), 4096), true);
  }
  catch(java.io.FileNotFoundException e) {
    java.lang.System.err.println("Unable to establish a print writer for trace");
    java.lang.System.err.flush();
    return;
  }

  try {
    Class.forName("com.ibm.db2.jcc.DB2Driver");
  }
  catch(ClassNotFoundException e) {
    printWriter.println("[TraceExample] " +
     "IBM Data Server Driver for JDBC and SQLJ type 4 connectivity " +
     "is not in the application classpath. Unable to load driver.");
    printWriter.flush();
    return;
  }

  // This URL describes the target data source for Type 4 connectivity.
```

```
    // The traceLevel property is established through the URL syntax,
    // and driver tracing is directed to file "/temp/driverLog.txt"
    // The traceLevel property has type int. The constants
    // com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS and
    // com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS represent
    // int values. Those constants cannot be used directly in the
    // first getConnection parameter. Resolve the constants to their
    // int values by assigning them to a variable. Then use the
    // variable as the first parameter of the getConnection method.
    String databaseURL =
     "jdbc:ids://sysmvs1.stl.ibm.com:5021" +
     "/sample:traceFile=/temp/driverLog.txt;traceLevel=" +
     (com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS |
     com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS) + ";";

    // Set other properties
    java.util.Properties properties = new java.util.Properties();
    properties.setProperty("user", "myname");
    properties.setProperty("password", "mypass");

    try {
     // This connection request is traced using trace level
     // TRACE_CONNECTS | TRACE_DRDA_FLOWS
     c = java.sql.DriverManager.getConnection(databaseURL, properties);

     // Change the trace level for all subsequent requests
     // on the connection to TRACE_ALL
     ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(printWriter,
      com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL);

     // The following INSERT is traced using trace level TRACE_ALL
     java.sql.Statement s1 = c.createStatement();
     s1.executeUpdate("INSERT INTO sampleTable(sampleColumn) VALUES(1)");
     s1.close();

     // Disable all tracing on the connection
     ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(null);

      // The following SQL insert code is not traced
      java.sql.Statement s2 = c.createStatement();
      s2.executeUpdate("insert into sampleTable(sampleColumn) values(1)");
      s2.close();

      c.close();
    }
    catch(java.sql.SQLException e) {
     com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e, printWriter,
      "[TraceExample]");
    }
    finally {
      cleanup(c, printWriter);
      printWriter.flush();
    }
  }
}
```

Techniques for monitoring IBM Data Server Driver for JDBC and SQLJ Sysplex support

To monitor IBM Data Server Driver for JDBC and SQLJ Sysplex support, you need to monitor the global transport objects pool.

You can monitor the global transport objects pool in either of the following ways:

- Using traces that you start by setting IBM Data Server Driver for JDBC and SQLJ configuration properties
- Using an application programming interface

Configuration properties for monitoring the global transport objects pool

The db2.jcc.dumpPool, db2.jcc.dumpPoolStatisticsOnSchedule, and db2.jcc.dumpPoolStatisticsOnScheduleFile configuration properties control tracing of the global transport objects pool.

For example, the following set of configuration property settings cause error messages and dump pool error messages to be written every 60 seconds to a file named /home/WAS/logs/srv1/poolstats:

```
db2.jcc.dumpPool=DUMP_SYSPLEX_MSG|DUMP_POOL_ERROR
db2.jcc.dumpPoolStatisticsOnSchedule=60
db2.jcc.dumpPoolStatisticsOnScheduleFile=/home/WAS/logs/srv1/poolstats
```

An entry in the pool statistics file looks like this:

```
time Scheduled PoolStatistics npr:2575 nsr:2575 lwroc:439 hwroc:1764 coc:372
aooc:362 rmoc:362 nbr:2872 tbt:857520 tpo:10
```

The meanings of the fields are:

npr
The total number of requests that the IBM Data Server Driver for JDBC and SQLJ has made to the pool since the pool was created.

nsr
The number of successful requests that the IBM Data Server Driver for JDBC and SQLJ has made to the pool since the pool was created. A successful request means that the pool returned an object.

lwroc
The number of objects that were reused but were not in the pool. This can happen if a Connection object releases a transport object at a transaction boundary. If the Connection object needs a transport object later, and the original transport object has not been used by any other Connection object, the Connection object can use that transport object.

hwroc
The number of objects that were reused from the pool.

coc
The number of objects that the IBM Data Server Driver for JDBC and SQLJ created since the pool was created.

aooc
The number of objects that exceeded the idle time that was specified by db2.jcc.maxTransportObjectIdleTime and were deleted from the pool.

rmoc
The number of objects that have been deleted from the pool since the pool was created.

nbr
The number of requests that the IBM Data Server Driver for JDBC and SQLJ made to the pool that the pool blocked because the pool reached its maximum

capacity. A blocked request might be successful if an object is returned to the pool before the db2.jcc.maxTransportObjectWaitTime is exceeded and an exception is thrown.

tbt
The total time in milliseconds for requests that were blocked by the pool. This time can be much larger than the elapsed execution time of the application if the application uses multiple threads.

sbt
The shortest time in milliseconds that a thread waited to get a transport object from the pool. If the time is under one millisecond, the value in this field is zero.

lbt
The longest time in milliseconds that a thread waited to get a transport object from the pool.

abt
The average amount of time in milliseconds that threads waited to get a transport object from the pool. This value is tbt/nbr.

tpo
The number of objects that are currently in the pool.

Application programming interfaces for monitoring the global transport objects pool

You can write applications to gather statistics on the global transport objects pool. Those applications create objects in the DB2PoolMonitor class and invoke methods to retrieve information about the pool.

For example, the following code creates an object for monitoring the global transport objects pool:

```
import com.ibm.db2.jcc.DB2PoolMonitor;
DB2PoolMonitor transportObjectPoolMonitor =
        DB2PoolMonitor.getPoolMonitor (DB2PoolMonitor.TRANSPORT_OBJECT);
```

After you create the DB2PoolMonitor object, you can use methods in the DB2PoolMonitor class to monitor the pool.

Chapter 10. System monitoring for the IBM Data Server Driver for JDBC and SQLJ

To assist you in monitoring the performance of your applications with the IBM Data Server Driver for JDBC and SQLJ, the driver provides two methods to collect information for a connection.

That information is:

Core driver time
The sum of elapsed monitored API times that were collected while system monitoring was enabled, in microseconds. In general, only APIs that might result in network I/O or database server interaction are monitored.

Network I/O time
The sum of elapsed network I/O times that were collected while system monitoring was enabled, in microseconds.

Server time
The sum of all reported database server elapsed times that were collected while system monitoring was enabled, in microseconds.

Application time
The sum of the application, JDBC driver, network I/O, and database server elapsed times, in milliseconds.

The two methods are:

- The DB2SystemMonitor interface
- The TRACE_SYSTEM_MONITOR trace level

To collect system monitoring data using the DB2SystemMonitor interface: Perform these basic steps:

1. Invoke the DB2Connection.getDB2SystemMonitor method to create a DB2SystemMonitor object.
2. Invoke the DB2SystemMonitor.enable method to enable the DB2SystemMonitor object for the connection.
3. Invoke the DB2SystemMonitor.start method to start system monitoring.
4. When the activity that is to be monitored is complete, invoke DB2SystemMonitor.stop to stop system monitoring.
5. Invoke the DB2SystemMonitor.getCoreDriverTimeMicros, DB2SystemMonitor.getNetworkIOTimeMicros, `DB2SystemMonitor.getServerTimeMicros`, or DB2SystemMonitor.getApplicationTimeMillis methods to retrieve the elapsed time data.

   The server time that is returned by `DB2SystemMonitor.getServerTimeMicros` does not include commit or rollback time.

For example, the following code demonstrates how to collect each type of elapsed time data. The numbers to the right of selected statements correspond to the previously described steps.

```
import java.sql.*;
import com.ibm.db2.jcc.*;
public class TestSystemMonitor
{
  public static void main(String[] args)
  {
    String url = "jdbc:ids://sysmvs1.svl.ibm.com:5021/san_jose";
    String user="db2adm";
    String password="db2adm";
    try
    {
      // Load the IBM Data Server Driver for JDBC and SQLJ
      Class.forName("com.ibm.db2.jcc.DB2Driver");
      System.out.println("**** Loaded the JDBC driver");

      // Create the connection using the IBM Data Server Driver for JDBC and SQLJ
      Connection conn = DriverManager.getConnection (url,user,password);
      // Commit changes manually
      conn.setAutoCommit(false);
      System.out.println("**** Created a JDBC connection to the data source");
      DB2SystemMonitor systemMonitor =                                    1
        ((DB2Connection)conn).getDB2SystemMonitor();
      systemMonitor.enable(true);                                         2
      systemMonitor.start(DB2SystemMonitor.RESET_TIMES);                  3
      Statement stmt = conn.createStatement();
      int numUpd = stmt.executeUpdate(
        "UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'");
      systemMonitor.stop();                                               4
      System.out.println("Server elapsed time (microseconds)="
        + systemMonitor.getServerTimeMicros());                           5
      System.out.println("Network I/O elapsed time (microseconds)="
        + systemMonitor.getNetworkIOTimeMicros());
      System.out.println("Core driver elapsed time (microseconds)="
        + systemMonitor.getCoreDriverTimeMicros());
      System.out.println("Application elapsed time (milliseconds)="
        + systemMonitor.getApplicationTimeMillis());
      conn.rollback();
      stmt.close();
      conn.close();
    }
    // Handle errors
    catch(ClassNotFoundException e)
    {
      System.err.println("Unable to load the driver, " + e);
    }
    catch(SQLException e)
    {
      System.out.println("SQLException: " + e);
      e.printStackTrace();
    }
  }
}
```

Figure 10-1. Example of using DB2SystemMonitor methods to collect system monitoring data

To collect system monitoring information using the trace method: Start a JDBC trace, using configuration properties or Connection or DataSource properties. Include TRACE_SYSTEM_MONITOR when you set the traceLevel property. For example:

```
String url = "jdbc:ids://sysmvs1.stl.ibm.com:5021/san_jose" +
 ":traceFile=/u/db2p/jcctrace;" +
 "traceLevel=" + com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SYSTEM_MONITOR + ";";
```

The trace records with system monitor information look similar to this:

```
[jcc][SystemMonitor:start]
...
[jcc][SystemMonitor:stop] core: 565.67ms | network: 211.695ms | server: 207.771ms
```

IBM Data Server Driver for JDBC and SQLJ remote trace controller

The IBM Data Server Driver for JDBC and SQLJ provides a facility for controlling IBM Data Server Driver for JDBC and SQLJ traces dynamically.

This remote trace controller lets you perform operations like these for multiple driver instances:

- Start, stop, or resume a trace
- Change the output trace file or directory location
- Change the trace level

The remote trace controller uses the Java Management Extensions (JMX) architecture, which is part of the Java Standard Edition, Version 6, or later. The JMX consists of:

- A set of built-in management utilities, which let you do monitoring from a management console such as the Java Monitoring and Management Console (JConsole).
- A set of APIs that let you write applications to perform the same functions.

Enabling the remote trace controller

Enabling the remote trace controller involves enabling Java Management Extensions (JMX) in the IBM Data Server Driver for JDBC and SQLJ, and making the JMX agent available to clients.

The remote trace controller requires Java Standard Edition, Version 6 or later.

The steps for enabling the remote trace controller are:

1. Enable JMX to the IBM Data Server Driver for JDBC and SQLJ by setting the db2.jcc.jmxEnabled global configuration property to true or yes.

   For example, include this string in DB2JccConfiguration.properties:

   ```
   db2.jcc.jmxEnabled=true
   ```

2. Make the JMX agent (the platform MBean server) available to local or remote clients.
   - For local clients:

     Monitoring and management capabilities are automatically made available when the JVM is started. After your application is started, you can use a JMX client such as JConsole to connect locally to your Java process.
   - For remote clients, use one of the following methods:
     - Use the out-of-the-box JMX agent.

       Out-of-the-box management uses JMX built-in management utilities. To enable out-of-the-box management, you need to set a number of Java system properties. You must at least set the following property:

       ```
       com.sun.management.jmxremote.port=portNum
       ```

       In addition, you should ensure that authentication and SSL are properly configured.

       Full information on enabling out-of-the-box management is at the following URL:

```
http://download.oracle.com/javase/6/docs/technotes/guides/management/agent.html
```

– Write a JMX agent. This technique is also discussed at:

http://download.oracle.com/javase/6/docs/technotes/guides/management/agent.html

In the following example, an RMI connector server is created for the PlatformMBeanServer using the MyCustomJMXAuthenticator object. The MyCustomJMXAuthenticator class defines how remote credentials are converted into a JAAS Subject by implementing the JMXAuthenticator interface:

```
...
HashMap<String> env = new HashMap<String>();
env.put(JMXConnectorServer.AUTHENTICATOR, new MyCustomJMXAuthenticator());
env.put("jmx.remote.x.access.file", "my.access.file");

MBeanServer mbs =
  java.lang.management.ManagementFactory.getPlatformMBeanServer();
JMXServiceURL url =
  new JMXServiceURL("service:jmx:rmi:///jndi/rmi://:9999/jmxrmi");

JMXConnectorServer cs =
  JMXConnectorServerFactory.newJMXConnectorServer(url, env, mbs);
cs.start();
...
public class MyCustomJMXAuthenticator implements JMXAuthenticator {

  public Subject authenticate(Object credentials) {
    // the hash contains username, password, etc...
    Hashtable <String> credentialsHash
      = (Hashtable <String>) credentials;

    ...
    // Authenticate using the provided credentials
    ...
    if (authentication-successful) {
      return new Subject(true,
        Collections.singleton
          (new JMXPrincipal(credentialsHash.get("username"))),
        Collections.EMPTY_SET,
        Collections.EMPTY_SET);
    }
    throw new SecurityException("Invalid credentials");
  }
}
```

Accessing the remote trace controller

You can access the remote trace controller through out-of-the-box management tools, or through an application.

You use out-of-the-box management through a JMX-compliant management client, such as JConsole, which is part of Java Standard Edition, Version 6. Information on using JConsole for out-of-the-box management is at the following URL:

http://download.oracle.com/javase/6/docs/technotes/guides/management/jconsole.html

In an application that accesses the remote trace controller, the remote trace controller is a managed bean (MBean). JMX manages resources through JMX agents. A JMX agent is an MBean server. Each MBean represents a resource. Every MBean has a name, which you define through an object of class javax.management.ObjectName. You use the ObjectName object to register and retrieve MBeans in the MBeanServer.

The MBean name has two parts: the domain and the key properties. For the ObjectName for the IBM Data Server Driver for JDBC and SQLJ remote trace controller, the domain is com.ibm.db2.jcc, and the key properties are name=DB2TraceManager.

An application that accesses the remote trace controller must include these steps:

1. Establish a Remote Method Invocation (RMI) connection to an MBean server.
2. Perform a lookup on the remote trace controller in the MBean server.
3. Invoke trace operations on the MBean.

   You can operate on the MBean in the following ways:

   - Using an MBean proxy
   - Without a proxy, through an MBeanServerConnection.

Example: accessing the remote trace controller without proxies: This example demonstrates accessing MBeans directly from an MBeanServerConnection. This method is the most generic because it does not require matching interface definitions on the JMX client application.

```
Hashtable<String> env = new Hashtable<String>();
env.put(Context.INITIAL_CONTEXT_FACTORY,
 "com.sun.jndi.fscontext.RefFSContextFactory");

try {
  System.out.println ("");
  System.out.println ("-------------------------------------------------");
  System.out.println ("Establish an RMI connection to an MBeanServer");
  System.out.println ("-------------------------------------------------");
  JMXServiceURL url =
    new JMXServiceURL ("service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi");
  JMXConnector jmxc = JMXConnectorFactory.connect (url, env);
  MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();

  System.out.println ("");
  System.out.println ("-------------------------------------------------");
  System.out.println ("Processing MBean");
  System.out.println ("-------------------------------------------------");
  String objectNameString = "com.ibm.db2.jcc:name=DB2TraceManager";
  ObjectName name = new ObjectName(objectNameString);
  System.out.println ("ObjectName="+objectNameString);

  System.out.println ("");
  System.out.println ("-------------------------------------------------");
  System.out.println ("Print all attributes of the MBean");
  System.out.println ("-------------------------------------------------");

  System.out.println(
    "TraceDirectory  = "+mbsc.getAttribute (name, "TraceDirectory"));
  System.out.println(
    "TraceFile       = "+mbsc.getAttribute (name, "TraceFile"));
  System.out.println(
    "TraceFileAppend = "+mbsc.getAttribute (name, "TraceFileAppend"));
  System.out.println(
    "TraceLevel      = "+mbsc.getAttribute (name, "TraceLevel"));

  System.out.println ("");
  System.out.println ("-------------------------------------------------");
  System.out.println ("Invoke some operations on the MBean");
  System.out.println ("-------------------------------------------------");
  System.out.print ("Invoking suspendTrace()...");
  mbsc.invoke (name, "suspendTrace", null , null);
  System.out.println ("success");

  System.out.print ("Invoking resumeTrace()...");
```

```
  mbsc.invoke (name, "resumeTrace", null , null);
  System.out.println ("success");
}
catch (Exception e) {
  System.out.println ("failure");
  e.printStackTrace ();
}
```

Example: accessing the remote trace controller with proxies: This example demonstrates the creation of a proxy to an MBean. The proxy implements the com.ibm.db2.jcc.mx.DB2TraceManagerMXBean interface. The application makes calls directly on the proxy, and the underlying proxy implementation invokes the MBean operation on the remote MBean server.

```
Hashtable<String> env = new Hashtable<String>();
env.put(Context.INITIAL_CONTEXT_FACTORY,
 "com.sun.jndi.fscontext.RefFSContextFactory");

try {
  System.out.println ("");
  System.out.println ("-------------------------------------------------");
  System.out.println ("Establish an RMI connection to an MBeanServer");
  System.out.println ("-------------------------------------------------");
  JMXServiceURL url =
    new JMXServiceURL ("service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi");
  JMXConnector jmxc = JMXConnectorFactory.connect (url, env);
  MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();

  System.out.println ("");
  System.out.println ("-------------------------------------------------");
  System.out.println ("Processing MBean");
  System.out.println ("-------------------------------------------------");
  String objectNameString = "com.ibm.db2.jcc:name=DB2TraceManager";
  ObjectName name = new ObjectName(objectNameString);
  System.out.println ("ObjectName="+objectNameString);

  System.out.println ("");
  System.out.println ("-------------------------------------------------");
  System.out.println ("Print all attributes of the MBean");
  System.out.println ("-------------------------------------------------");
  com.ibm.db2.jcc.mx.DB2TraceManagerMXBean mbeanProxy =
    JMX.newMBeanProxy(mbsc, name,
      com.ibm.db2.jcc.mx.DB2TraceManagerMXBean.class, true);
  System.out.println ("TraceDirectory  = "+mbeanProxy.getTraceDirectory ());
  System.out.println ("TraceFile       = "+mbeanProxy.getTraceFile ());
  System.out.println ("TraceFileAppend = "+mbeanProxy.getTraceFileAppend ());
  System.out.println ("TraceLevel      = "+mbeanProxy.getTraceLevel ());
  System.out.println ("");
  System.out.println ("-------------------------------------------------");
  System.out.println ("Invoke some operations on the MBean");
  System.out.println ("-------------------------------------------------");
  System.out.print ("Invoking suspendTrace()...");
  mbeanProxy.suspendTrace();
  System.out.println ("success");
  System.out.print ("Invoking resumeTrace()...");
  mbeanProxy.resumeTrace();
  System.out.println ("success");
}
catch (Exception e) {
  System.out.println ("failure");
  e.printStackTrace ();
}
```

Chapter 11. Java client support for high availability on IBM data servers

Client applications that connect to DB2 Database for Linux, UNIX, and Windows, DB2 for z/OS, or IBM Informix can easily take advantage of the high availability features of those data servers.

Client applications can use the following high availability features:

- Automatic client reroute

  Automatic client reroute capability is available on all IBM data servers. Automatic client reroute uses information that is provided by the data servers to redirect client applications from a server that experiences an outage to an alternate server. Automatic client reroute enables applications to continue their work with minimal interruption. Redirection of work to an alternate server is called *failover*.

  For connections to DB2 for z/OS data servers, automatic client reroute is part of the workload balancing feature. In general, for DB2 for z/OS, automatic client reroute should not be enabled without workload balancing.
- Client affinities

  Client affinities is a failover solution that is controlled completely by the client. It is intended for situations in which you need to connect to a particular primary server. If an outage occurs during the connection to the primary server, you use client affinities to enforce a specific order for failover to alternate servers.

  Client affinities is not applicable to a DB2 for z/OS data sharing environment, because all members of a data sharing group can access data concurrently. Data sharing is the recommended solution for high availability for DB2 for z/OS.
- Workload balancing

  Workload balancing is available on all IBM data servers. Workload balancing ensures that work is distributed efficiently among servers in an IBM Informix high-availability cluster, DB2 for z/OS data sharing group, or DB2 Database for Linux, UNIX, and Windows DB2 pureScale® instance.

The following table provides links to server-side information about these features.

Table 11-1. Server-side information on high availability

Data server	Related topics
DB2 Database for Linux, UNIX, and Windows	• DB2 pureScale: Road map to DB2 pureScale Feature documentation • Automatic client reroute: Automatic client reroute roadmap
IBM Informix	Manage Cluster Connections with the Connection Manager
DB2 for z/OS	Communicating with data sharing groups

Important: For connections to DB2 for z/OS, this information discusses direct connections to DB2 for z/OS. For information about high availability for connections through DB2 Connect™ Server, see the DB2 Connect documentation.

Java client support for high availability for connections to DB2 Database for Linux, UNIX, and Windows servers

DB2 Database for Linux, UNIX, and Windows servers provide high availability for client applications, through workload balancing and automatic client reroute. This support is available for applications that use Java clients (JDBC, SQLJ, or pureQuery), as well as non-Java clients (ODBC, CLI, .NET, OLE DB, PHP, Ruby, or embedded SQL).

For Java clients, you need to use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to take advantage of DB2 Database for Linux, UNIX, and Windows high-availability support. You need IBM Data Server Driver for JDBC and SQLJ version 3.58 or 4.8, or later.

High availability support for connections to DB2 Database for Linux, UNIX, and Windows servers includes:

Automatic client reroute
This support enables a client to recover from a failure by attempting to reconnect to the database through an alternate server. Reconnection to another server is called *failover*. For Java clients, automatic client reroute support is always enabled.

Servers can provide automatic client reroute capability in any of the following ways:

- Several servers are configured in a DB2 pureScale instance. A connection to a database is a connection to a member of that DB2 pureScale instance. Failover involves reconnection to another member of the DB2 pureScale instance. This environment requires that clients use TCP/IP to connect to the DB2 pureScale instance.
- A DB2 pureScale instance and an alternate server are defined for a database. Failover first involves reconnection to another member of the DB2 pureScale instance. Failover to the alternate server is attempted only if no member of the DB2 pureScale instance is available.
- A DB2 pureScale instance is defined for the primary server, and another DB2 pureScale instance is defined for the alternate server. Failover first involves reconnection to another member of the primary DB2 pureScale instance. Failover to the alternate DB2 pureScale instance is attempted only if no member of the primary DB2 pureScale instance is available.
- A database is defined on a single server. The configuration for that database includes specification of an alternate server. Failover involves reconnection to the alternate server.

For Java, client applications, failover for automatic client reroute can be *seamless* or *non-seamless*. With non-seamless failover, when the client application reconnects to another server, an error is always returned to the application, to indicate that failover (connection to the alternate server) occurred. With seamless failover, the driver does not return an error if a connection failure and successful reconnection to an alternate server occur during execution of the first SQL statement in a transaction.

In a DB2 pureScale instance, automatic client reroute support can be used without workload balancing or with workload balancing.

Workload balancing
Workload balancing can improve availability of a DB2 pureScale instance.

With workload balancing, a DB2 pureScale instance ensures that work is distributed efficiently among members.

Java clients on any operating system support workload balancing. The connection from the client to the DB2 pureScale instance must use TCP/IP.

When workload balancing is enabled, the client gets frequent status information about the members of the DB2 pureScale instance through a server list. The client caches the server list and uses the information in it to determine the member to which the next transaction should be routed.

For Java applications, when JNDI is used, the cached server list can be shared by multiple JVMs for the first connection. However workload balancing is always performed within the context of a single JVM.

DB2 Database for Linux, UNIX, and Windows supports two types of workload balancing:

Connection-level workload balancing
Connection-level workload balancing is performed at connection boundaries. It is not supported for Java clients.

Transaction-level workload balancing
Transaction-level workload balancing is performed at transaction boundaries. Client support for transaction-level workload balancing is disabled by default for clients that connect to DB2 Database for Linux, UNIX, and Windows.

Client affinities
Client affinities is an automatic client reroute solution that is controlled completely by the client. It is intended for situations in which you need to connect to a particular primary server. If an outage occurs during the connection to the primary server, you use client affinities to enforce a specific order for failover to alternate servers.

Configuration of DB2 Database for Linux, UNIX, and Windows automatic client reroute support for Java clients

For connections to DB2 Database for Linux, UNIX, and Windows databases, the process for configuration of automatic client reroute support on Java clients is the same for connections to a non-DB2 pureScale environment and a DB2 pureScale environment.

Automatic client reroute support for Java client applications that connect to DB2 Database for Linux, UNIX, and Windows works for connections that are obtained using the javax.sql.DataSource, javax.sql.ConnectionPoolDataSource, `javax.sql.XADataSource`, or java.sql.DriverManager interface.

To configure automatic client reroute on a IBM Data Server Driver for JDBC and SQLJ client:

1. Set the appropriate properties to specify the primary and alternate server addresses to use if the first connection fails.
   - If your application is using the DriverManager interface for connections:
     a. Specify the server name and port number of the primary server that you want to use in the connection URL.
     b. Set the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties to the server name and port number of the alternate server that you want to use.

Restriction: Automatic client reroute support for connections that are made with the DriverManager interface has the following restrictions:

- Alternate server information is shared between DriverManager connections only if you create the connections with the same URL and properties.
- You cannot set the clientRerouteServerListJNDIName property or the clientRerouteServerListJNDIContext properties for a DriverManager connection.
- Automatic client reroute is not enabled for default connections (jdbc:default:connection).

- If your application is using the DataSource interface for connections, use one or both of the following techniques:
  - Set the server names and port numbers in DataSource properties:
    a. Set the serverName and portNumber properties to the server name and port number of the primary server that you want to use.
    b. Set the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties to the server name and port number of the alternate server that you want to use.
  - Configure JNDI for automatic client reroute by using a DB2ClientRerouteServerList instance to identify the primary server and alternate server.
    a. Create an instance of DB2ClientRerouteServerList.

       DB2ClientRerouteServerList is a serializable Java bean with the following properties:

Property name	Data type
com.ibm.db2.jcc.DB2ClientRerouteServerList.alternateServerName	String[]
com.ibm.db2.jcc.DB2ClientRerouteServerList.alternatePortNumber	int[]
com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryServerName	String[]
com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryPortNumber	int[]

       getXXX and setXXX methods are defined for each property.
    b. Set the com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryServerName and com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryPortNumber properties to the server name and port number of the primary server that you want to use.
    c. Set the com.ibm.db2.jcc.DB2ClientRerouteServerList.alternateServerName and com.ibm.db2.jcc.DB2ClientRerouteServerList.alternatePortNumber properties to the server names and port numbers of the alternate server that you want to use.
    d. To make the DB2ClientRerouteServerList persistent:
       1) Bind the DB2ClientRerouteServerList instance to the JNDI registry.
       2) Assign the JNDI name of the DB2ClientRerouteServerList object to the IBM Data Server Driver for JDBC and SQLJ clientRerouteServerListJNDIName property.
       3) Assign the name of the JNDI context that is used for binding and lookup of the DB2ClientRerouteServerList instance to the clientRerouteServerListJNDIContext property.

When a DataSource is configured to use JNDI for storing automatic client reroute alternate information, the standard server and port properties of the DataSource are not used for a getConnection request. Instead, the primary server address is obtained from the transient clientRerouteServerList information. If the JNDI store is not available due to a JNDI bind or lookup failure, the IBM Data Server Driver for JDBC and SQLJ attempts to make a connection using the standard server and port properties of the DataSource. Warnings are accumulated to indicate that a JNDI bind or lookup failure occurred.

After a failover:

- The IBM Data Server Driver for JDBC and SQLJ attempts to propagate the updated server information to the JNDI store.
- primaryServerName and primaryPortNumber values that are specified in DB2ClientRerouteServerList are used for the connection. If primaryServerName is not specified, the serverName and portNumber values for the DataSource instance are used.

If you configure DataSource properties as well as configuring JNDI for automatic client reroute, the DataSource properties have precedence over the JNDI configuration.

2. Set properties to control the number of retries, time between retries, and the frequency with which the server list is refreshed.

   The following properties control retry behavior for automatic client reroute.

   maxRetriesForClientReroute
   The maximum number of connection retries for automatic client reroute.

   When client affinities support is not configured, if maxRetriesForClientReroute or retryIntervalForClientReroute is not set, the default behavior is that the connection is retried for 10 minutes, with a wait time between retries that increases as the length of time from the first retry increases.

   When client affinities is configured, the default for maxRetriesForClientReroute is 3.

   retryIntervalForClientReroute
   The number of seconds between consecutive connection retries.

   When client affinities support is not configured, if retryIntervalForClientReroute or maxRetriesForClientReroute is not set, the default behavior is that the connection is retried for 10 minutes, with a wait time between retries that increases as the length of time from the first retry increases.

   When client affinities is configured, the default for retryIntervalForClientReroute is 0 (no wait).

Example of enabling DB2 Database for Linux, UNIX, and Windows automatic client reroute support in Java applications

Java client setup for DB2 Database for Linux, UNIX, and Windows automatic client reroute support includes setting several IBM Data Server Driver for JDBC and SQLJ properties.

The following example demonstrates setting up Java client applications for DB2 Database for Linux, UNIX, and Windows automatic client reroute support.

Suppose that your installation has a primary server and an alternate server with the following server names and port numbers:

Server name	Port number
srv1.sj.ibm.com	50000
srv3.sj.ibm.com	50002

The following code sets up DataSource properties in an application so that the application connects to srv1.sj.ibm.com as the primary server, and srv3.sj.ibm.com as the alternative server. That is, if srv1.sj.ibm.com is down during the initial connection, the driver should connect to srv3.sj.ibm.com.

```
ds.setDriverType(4);
ds.setServerName("srv1.sj.ibm.com");
ds.setPortNumber("50000");
ds.setClientRerouteAlternateServerName("srv3.sj.ibm.com");
ds.setClientRerouteAlternatePortNumber("50002");
```

The following code configures JNDI for automatic client reroute. It creates an instance of DB2ClientRerouteServerList, binds that instance to the JNDI registry, and assigns the JNDI name of the DB2ClientRerouteServerList object to the clientRerouteServerListJNDIName property.

```
// Create a starting context for naming operations
InitialContext registry = new InitialContext();
// Create a DB2ClientRerouteServerList object
DB2ClientRerouteServerList address = new DB2ClientRerouteServerList();

// Set the port number and server name for the primary server
address.setPrimaryPortNumber(50000);
address.setPrimaryServerName("srv1.sj.ibm.com");

// Set the  port number and server name for the alternate server
int[] port = {50002};
String[] server = {"srv3.sj.ibm.com"};
address.setAlternatePortNumber(port);
address.setAlternateServerName(server);

registry.rebind("serverList", address);
// Assign the JNDI name of the DB2ClientRerouteServerList object to the
// clientRerouteServerListJNDIName property
datasource.setClientRerouteServerListJNDIName("serverList");
```

Configuration of DB2 Database for Linux, UNIX, and Windows workload balancing support for Java clients

To configure a IBM Data Server Driver for JDBC and SQLJ client application that connects to a DB2 Database for Linux, UNIX, and Windows DB2 pureScale instance for workload balancing, you need to connect to a member of the DB2 pureScale instance, and set the properties that enable workload balancing and the maximum number of connections.

Java client applications support transaction-level workload balancing. They do not support connection-level workload balancing. Workload balancing is supported only for connections to a DB2 pureScale instance.

Workload balancing support for Java client applications that connect to DB2 Database for Linux, UNIX, and Windows works for connections that are obtained using the javax.sql.DataSource, javax.sql.ConnectionPoolDataSource, `javax.sql.XADataSource`, or java.sql.DriverManager interface.

Restriction: Workload balancing support for connections that are made with the DriverManager interface has the following restrictions:

- Alternate server information is shared between DriverManager connections only if you create the connections with the same URL and properties.
- You cannot set the clientRerouteServerListJNDIName property or the clientRerouteServerListJNDIContext properties for a DriverManager connection.
- Workload balancing is not enabled for default connections (jdbc:default:connection).

The following table describes the basic property settings for enabling DB2 Database for Linux, UNIX, and Windows workload balancing for Java applications.

Table 11-2. Basic settings to enable workload support in Java applications

IBM Data Server Driver for JDBC and SQLJ setting	Value
enableSysplexWLB property	true
maxTransportObjects property	The maximum number of connections that the requester can make to the DB2 pureScale instance
Connection address:	
server	The IP address of a member of a DB2 pureScale instance[1]
port	The SQL port number for the DB2 pureScale instance[1]
database	The database name

Note:

1. Alternatively, you can use a distributor, such as Websphere Application Server Network Deployment, or multihomed DNS to establish the initial connection to the database.
   - For a distributor, you specify the IP address and port number of the distributor. The distributor analyzes the current workload distribution, and uses that information to forward the connection request to one of the members of the DB2 pureScale instance.
   - For multihomed DNS, you specify an IP address and port number that can resolve to the IP address and port number of any member of the DB2 pureScale instance. Multihomed DNS processing selects a member based on some criterion, such as simple round-robin selection or member workload distribution.

If you want to fine-tune DB2 Database for Linux, UNIX, and Windows workload balancing support, global configuration properties are available. The properties for the IBM Data Server Driver for JDBC and SQLJ are listed in the following table.

Table 11-3. Configuration properties for fine-tuning DB2 Database for Linux, UNIX, and Windows workload balancing support for connections from the IBM Data Server Driver for JDBC and SQLJ

IBM Data Server Driver for JDBC and SQLJ configuration property	Description
db2.jcc.maxRefreshInterval	Specifies the maximum amount of time in seconds between refreshes of the client copy of the server list that is used for workload balancing. The default is 30. The minimum valid value is 1.
db2.jcc.maxTransportObjectIdleTime	Specifies the maximum elapsed time in number of seconds before an idle transport is dropped. The default is 60. The minimum supported value is 0.

Table 11-3. Configuration properties for fine-tuning DB2 Database for Linux, UNIX, and Windows workload balancing support for connections from the IBM Data Server Driver for JDBC and SQLJ (continued)

IBM Data Server Driver for JDBC and SQLJ configuration property	Description
db2.jcc.maxTransportObjectWaitTime	Specifies the number of seconds that the client will wait for a transport to become available. The default is -1 (unlimited). The minimum supported value is 0.
db2.jcc.minTransportObjects	Specifies the lower limit for the number of transport objects in a global transport object pool. The default value is 0. Any value that is less than or equal to 0 means that the global transport object pool can become empty.

Example of enabling DB2 Database for Linux, UNIX, and Windows workload balancing support in Java applications

Java client setup for DB2 Database for Linux, UNIX, and Windows workload balancing support includes setting several IBM Data Server Driver for JDBC and SQLJ properties.

The following example demonstrates setting up Java client applications for DB2 Database for Linux, UNIX, and Windows workload balancing support.

Before you can set up the client, the servers to which the client connects must be configured in a DB2 pureScale instance.

Follow these steps to set up the client:

1. Verify that the IBM Data Server Driver for JDBC and SQLJ is at the correct level to support workload balancing by following these steps:
   a. Issue the following command in a command line window:
      ```
      java com.ibm.db2.jcc.DB2Jcc -version
      ```
   b. Find a line in the output like this, and check that *nnn* is 3.58 or later.
   c.
      ```
      [jcc] Driver: IBM Data Server Driver for JDBC and SQLJ Architecture nnn xxx
      ```
2. Set IBM Data Server Driver for JDBC and SQLJ properties to enable the connection concentrator or workload balancing:
   a. Set these Connection or DataSource properties:
      - enableSysplexWLB
      - maxTransportObjects
   b. Set the db2.jcc.maxRefreshInterval global configuration property in a DB2JccConfiguration.properties file to set the maximum refresh interval for all DataSource or Connection instances that are created under the driver.

   Start with settings similar to these:

Table 11-4. Example of property settings for workload balancing for DB2 Database for Linux, UNIX, and Windows

Property	Setting
enableSysplexWLB	true
maxTransportObjects	80
db2.jcc.maxRefreshInterval	30

The values that are specified are not intended to be recommended values. You need to determine values based on factors such as the number of physical

connections that are available. The number of transport objects must be equal to or greater than the number of connection objects.

3. To fine-tune workload balancing for all DataSource or Connection instances that are created under the driver, set the db2.jcc.maxTransportObjects configuration property in a DB2JccConfiguration.properties file.

   Start with a setting similar to this one:

   ```
   db2.jcc.maxTransportObjects=500
   ```

Operation of automatic client reroute for connections to DB2 Database for Linux, UNIX, and Windows from Java clients

When IBM Data Server Driver for JDBC and SQLJ client reroute support is enabled, a Java application that is connected to a DB2 Database for Linux, UNIX, and Windows database can continue to run when the primary server has a failure.

Automatic client reroute for a Java application that is connected to a DB2 Database for Linux, UNIX, and Windows database operates in the following way when support for client affinities is disabled:

1. During each connection to the data source, the IBM Data Server Driver for JDBC and SQLJ obtains primary and alternate server information.
   - For the first connection to a DB2 Database for Linux, UNIX, and Windows database:
     a. If the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties are set, the IBM Data Server Driver for JDBC and SQLJ loads those values into memory as the alternate server values, along with the primary server values serverName and portNumber.
     b. If the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties are not set, and a JNDI store is configured by setting the property clientRerouteServerListJNDIName on the DB2BaseDataSource, the IBM Data Server Driver for JDBC and SQLJ loads the primary and alternate server information from the JNDI store into memory.
     c. If no DataSource properties are set for the alternate servers, and JNDI is not configured, the IBM Data Server Driver for JDBC and SQLJ checks DNS tables for primary and alternate server information. If DNS information exists, the IBM Data Server Driver for JDBC and SQLJ loads those values into memory.

        In a DB2 pureScale environment, regardless of the outcome of the DNS lookup:

        1) If configuration property db2.jcc.outputDirectory is set, the IBM Data Server Driver for JDBC and SQLJ searches the directory that is specified by db2.jcc.outputDirectory for a file named jccServerListCache.bin.
        2) If db2.jcc.outputDirectory is not set, and the java.io.tmpdir system property is set, the IBM Data Server Driver for JDBC and SQLJ searches the directory that is specified by java.io.tmpdir for a file named jccServerListCache.bin.
        3) If jccServerListCache.bin can be accessed, the IBM Data Server Driver for JDBC and SQLJ loads the cache into memory, and obtains the alternate server information from jccServerListCache.bin for the serverName value that is defined for the DataSource object.

d. If no primary or alternate server information is available, a connection cannot be established, and the IBM Data Server Driver for JDBC and SQLJ throws an exception.

- For subsequent connections, the IBM Data Server Driver for JDBC and SQLJ obtains primary and alternate server values from driver memory.

2. The IBM Data Server Driver for JDBC and SQLJ attempts to connect to the data source using the primary server name and port number.

   In a non-DB2 pureScale environment, the primary server is a stand-alone server. In a DB2 pureScale environment, the primary server is a member of a DB2 pureScale instance.

   If the connection is through the DriverManager interface, the IBM Data Server Driver for JDBC and SQLJ creates an internal DataSource object for automatic client reroute processing.

3. If the connection to the primary server fails:
   a. If this is the first connection, the IBM Data Server Driver for JDBC and SQLJ attempts to reconnect to a server using information that is provided by driver properties such as clientRerouteAlternateServerName and clientRerouteAlternatePortNumber.
   b. If this is not the first connection, the IBM Data Server Driver for JDBC and SQLJ attempts to make a connection using the information from the latest server list that is returned from the server.

   Connection to an alternate server is called *failover*.

   The IBM Data Server Driver for JDBC and SQLJ uses the maxRetriesForClientReroute and retryIntervalForClientReroute properties to determine how many times to retry the connection and how long to wait between retries. An attempt to connect to the primary server and alternate servers counts as one retry.

4. If the connection is not established, maxRetriesForClientReroute and retryIntervalForClientReroute are not set, and the original serverName and portNumber values that are defined on the DataSource are different from the serverName and portNumber values that were used for the current connection, the connection is retried with the serverName and portNumber values that are defined on the DataSource.

5. If failover is successful during the initial connection, the driver generates an SQLWarning. If a successful failover occurs after the initial connection:
   - If seamless failover is enabled, and the following conditions are satisfied, the driver retries the transaction on the new server, without notifying the application.
     – The enableSeamlessFailover property is set to DB2BaseDataSource.YES (1).
     – The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
     – There are no global temporary tables in use on the server.
     – There are no open, held cursors.
   - If seamless failover is not in effect, the driver throws an SQLException to the application with error code -4498, to indicate to the application that the connection was automatically reestablished and the transaction was implicitly rolled back. The application can then retry its transaction without doing an explicit rollback first.

     A reason code that is returned with error code -4498 indicates whether any database server special registers that were modified during the original connection are reestablished in the failover connection.

You can determine whether alternate server information was used in establishing the initial connection by calling the DB2Connection.alternateWasUsedOnConnect method.

6. After failover, driver memory is updated with new primary and alternate server information that is returned from the new primary server.

Examples

Example: Automatic client reroute to a DB2 Database for Linux, UNIX, and Windows server when maxRetriesForClientReroute and retryIntervalForClientReroute are not set: Suppose that the following properties are set for a connection to a database:

Property	Value
enableClientAffinitiesList	DB2BaseDataSource.NO (2)
serverName	host1
portNumber	port1
clientRerouteAlternateServerName	host2
clientRerouteAlternatePortNumber	port2

The following steps demonstrate an automatic client reroute scenario for a connection to a DB2 Database for Linux, UNIX, and Windows server:

1. The IBM Data Server Driver for JDBC and SQLJ loads host1:port1 into its memory as the primary server address, and host2:port2 into its memory as the alternate server address.
2. On the initial connection, the driver tries to connect to host1:port1.
3. The connection to host1:port1 fails, so the driver tries another connection to host1:port1.
4. The reconnection to host1:port1 fails, so the driver tries to connect to host2:port2.
5. The connection to host2:port2 succeeds.
6. The driver retrieves alternate server information that was received from server host2:port2, and updates its memory with that information.

   Assume that the driver receives a server list that contains host2:port2, host2a:port2a. host2:port2 is stored as the new primary server, and host2a:port2a is stored as the new alternate server. If another communication failure is detected on this same connection, or on another connection that is created from the same DataSource, the driver tries to connect to host2:port2 as the new primary server. If that connection fails, the driver tries to connect to the new alternate server host2a:port2a.
7. A communication failure occurs during the connection to host2:port2.
8. The driver tries to connect to host2a:port2a.
9. The connection to host2a:port2a is successful.
10. The driver retrieves alternate server information that was received from server host2a:port2a, and updates its memory with that information.

Example: Automatic client reroute to a DB2 Database for Linux, UNIX, and Windows server in a DB2 pureScale environment, when maxRetriesForClientReroute and retryIntervalForClientReroute are not set, and configuration property db2.jcc.outputDirectory is set: Suppose that the following properties are set for a connection that is established from DataSource A:

Property	Value
enableClientAffinitiesList	DB2BaseDataSource.NO (2)
serverName	host1
portNumber	port1
db2.jcc.outputDirectory (configuration property)	/home/tmp

The following steps demonstrate an automatic client reroute scenario for a connection to a DB2 Database for Linux, UNIX, and Windows server:

1. Using the information in DataSource A, the IBM Data Server Driver for JDBC and SQLJ loads host1:port1 into its memory as the primary server address. The driver searches for cache file jccServerListCache.bin in /home/tmp, but the cache file does not exist.
2. The connection to host1:port1 succeeds. Suppose that the server returns a server list that contains host1:port1 and host2:port2.
3. The driver creates a cache in memory, with an entry that specifies host2:port2 as the alternate server list for host1:port1. The driver then creates the cache file /home/tmp/jccServerListCache.bin, and writes the cache from memory to this file.
4. The connection of Application A to host1:port1 fails, so the driver tries to connect to host2:port2.
5. The connection of Application A to host2:port2 succeeds. Suppose that the server returns a server list that contains host2:port2 and host2a:port2a. host2:port2 is the new primary server, and host2a:port2a is the new alternate server.
6. The driver looks for alternate server information for host2:port2 in the in-memory cache, but does not find any. It creates a new entry in the in-memory cache for host2:port2, with host2a:port2a as the alternate server list. The driver updates cache file /home/tmp/jccServerListCache.bin with the new entry that was added to the in-memory cache.
7. Application A completes, and the JVM exits.
8. Application B, which also uses DataSource A, starts.
9. The driver loads the server list from cache file /home/tmp/jccServerListCache.bin into memory, and finds the entry for host1:port1, which specifies host2:port2 as the alternate server list. The driver sets host2:port2 as the alternate server list for host1:port1.
10. A communication failure occurs when Application B tries to connect to host1:port1.
11. Application B attempts to connect to alternate server host2:port2.
12. The connection to host2:port2 succeeds. Application B continues.

Example: Automatic client reroute to a DB2 Database for Linux, UNIX, and Windows server when maxRetriesForClientReroute and retryIntervalForClientReroute are set for multiple retries: Suppose that the following properties are set for a connection to a database:

Property	Value
enableClientAffinitiesList	DB2BaseDataSource.NO (2)
serverName	host1

Property	Value
portNumber	port1
clientRerouteAlternateServerName	host2
clientRerouteAlternatePortNumber	port2
maxRetriesForClientReroute	3
retryIntervalForClientReroute	2

The following steps demonstrate an automatic client reroute scenario for a connection to a DB2 Database for Linux, UNIX, and Windows server:

1. The IBM Data Server Driver for JDBC and SQLJ loads host1:port1 into its memory as the primary server address, and host2:port2 into its memory as the alternate server address.
2. On the initial connection, the driver tries to connect to host1:port1.
3. The connection to host1:port1 fails, so the driver tries another connection to host1:port1.
4. The connection to host1:port1 fails again, so the driver tries to connect to host2:port2.
5. The connection to host2:port2 fails.
6. The driver waits two seconds.
7. The driver tries to connect to host1:port1 and fails.
8. The driver tries to connect to host2:port2 and fails.
9. The driver waits two seconds.
10. The driver tries to connect to host1:port1 and fails.
11. The driver tries to connect to host2:port2 and fails.
12. The driver waits two seconds.
13. The driver throws an SQLException with error code -4499.

Operation of workload balancing for connections to DB2 Database for Linux, UNIX, and Windows

Workload balancing (also called transaction-level workload balancing) for connections to DB2 Database for Linux, UNIX, and Windows contributes to high availability by balancing work among servers in a DB2 pureScale instance at the start of a transaction.

The following overview describes the steps that occur when a client connects to a DB2 Database for Linux, UNIX, and Windows DB2 pureScale instance, and transaction-level workload balancing is enabled:

1. When the client first establishes a connection to the DB2 pureScale instance, the member to which the client connects returns a server list with the connection details (IP address, port, and weight) for the members of the DB2 pureScale instance.

   The server list is cached by the client. The default lifespan of the cached server list is 30 seconds.
2. At the start of a new transaction, the client reads the cached server list to identify a server that has unused capacity, and looks in the transport pool for an idle transport that is tied to the under-utilized server. (An idle transport is a transport that has no associated connection object.)

- If an idle transport is available, the client associates the connection object with the transport.
- If, after a user-configurable timeout period (db2.jcc.maxTransportObjectWaitTime for a Java client or maxTransportWaitTime for a non-Java client), no idle transport is available in the transport pool and no new transport can be allocated because the transport pool has reached its limit, an error is returned to the application.

3. When the transaction runs, it accesses the server that is tied to the transport.
4. When the transaction ends, the client verifies with the server that transport reuse is still allowed for the connection object.
5. If transport reuse is allowed, the server returns a list of SET statements for special registers that apply to the execution environment for the connection object.

   The client caches these statements, which it replays in order to reconstruct the execution environment when the connection object is associated with a new transport.
6. The connection object is then dissociated from the transport, if the client determines that it needs to do so.
7. The client copy of the server list is refreshed when a new connection is made, or every 30 seconds, or the user-configured interval.
8. When transaction-level workload balancing is required for a new transaction, the client uses the previously described process to associate the connection object with a transport.

Application programming requirements for high availability for connections to DB2 Database for Linux, UNIX, and Windows servers

Failover for automatic client reroute can be seamless or non-seamless. If failover for connections to DB2 Database for Linux, UNIX, and Windows is not seamless, you need to add code to account for the errors that are returned when failover occurs.

If failover is non-seamless, and a connection is reestablished with the server, SQLCODE -4498 (for Java clients) or SQL30108N (for non-Java clients) is returned to the application. All work that occurred within the current transaction is rolled back. In the application, you need to:

- Check the reason code that is returned with the error. Determine whether special register settings on the failing data sharing member are carried over to the new (failover) data sharing member. Reset any special register values that are not current.
- Execute all SQL operations that occurred during the previous transaction.

The following conditions must be satisfied for failover for connections to DB2 Database for Linux, UNIX, and Windows to be seamless:

- The application programming language is Java, CLI, or .NET.
- The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
- If transaction-level load balancing is enabled, the data server allows transport reuse at the end of the previous transaction.
- All global session data is closed or dropped.
- There are no open, held cursors.

- If the application uses CLI, the application cannot perform actions that require the driver to maintain a history of previously called APIs in order to replay the SQL statement. Examples of such actions are specifying data at execution time, performing compound SQL, or using array input.
- The application is not a stored procedure.
- Autocommit is not enabled. Seamless failover can occur when autocommit is enabled. However, the following situation can cause problems: Suppose that SQL work is successfully executed and committed at the data server, but the connection or server goes down before acknowledgment of the commit operation is sent back to the client. When the client re-establishes the connection, it replays the previously committed SQL statement. The result is that the SQL statement is executed twice. To avoid this situation, turn autocommit off when you enable seamless failover.

Client affinities for DB2 Database for Linux, UNIX, and Windows

Client affinities is a client-only method for providing automatic client reroute capability.

Client affinities is available for applications that use CLI, .NET, or Java (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity). All rerouting is controlled by the driver.

Client affinities is intended for situations in which you need to connect to a particular primary server. If an outage occurs during the connection to the primary server, you need to enforce a specific order for failover to alternate servers. You should use client affinities for automatic client reroute only if automatic client reroute that uses server failover capabilities does not work in your environment.

As part of configuration of client affinities, you specify a list of alternate servers, and the order in which connections to the alternate servers are tried. When client affinities is in use, connections are established based on the list of alternate servers instead of the host name and port number that are specified by the application. For example, if an application specifies that a connection is made to server1, but the configuration process specifies that servers should be tried in the order (server2, server3, server1), the initial connection is made to server2 instead of server1.

Failover with client affinities is seamless, if the following conditions are true:

- The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
- There are no global temporary tables in use on the server.
- There are no open, held cursors.

When you use client affinities, you can specify that if the primary server returns to operation after an outage, connections return from an alternate server to the primary server on a transaction boundary. This activity is known as *failback*.

Configuration of client affinities for Java clients for DB2 Database for Linux, UNIX, and Windows connections

To enable support for client affinities in Java applications, you set properties to indicate that you want to use client affinities, and to specify the primary and alternate servers.

The following table describes the property settings for enabling client affinities for Java applications.

Table 11-5. Property settings to enable client affinities for Java applications

IBM Data Server Driver for JDBC and SQLJ setting	Value
enableClientAffinitiesList	DB2BaseDataSource.YES (1)
clientRerouteAlternateServerName	A comma-separated list of the primary server and alternate servers
clientRerouteAlternatePortNumber	A comma-separated list of the port numbers for the primary server and alternate servers
enableSeamlessFailover	DB2BaseDataSource.YES (1) for seamless failover; DB2BaseDataSource.NO (2) or enableSeamlessFailover not specified for no seamless failover
maxRetriesForClientReroute	The number of times to retry the connection to each server, including the primary server, after a connection to the primary server fails. The default is 3.
retryIntervalForClientReroute	The number of seconds to wait between retries. The default is no wait.
affinityFailbackInterval	The number of seconds to wait after the first transaction boundary to fail back to the primary server. Set this value if you want to fail back to the primary server.

Example of enabling client affinities in Java clients for DB2 Database for Linux, UNIX, and Windows connections

Before you can use client affinities for automatic client reroute in Java applications, you need to set properties to indicate that you want to use client affinities, and to identify the primary alternate servers.

The following example shows how to enable client affinities for failover without failback.

Suppose that you set the following properties for a connection to a database:

Property	Value
enableClientAffinitiesList	DB2BaseDataSource.YES (1)
clientRerouteAlternateServername	host1,host2,host3
clientRerouteAlternatePortNumber	port1,port2,port3
maxRetriesForClientReroute	3
retryIntervalForClientReroute	2

Suppose that a communication failure occurs during a connection to the server that is identified by host1:port1. The following steps demonstrate automatic client reroute with client affinities.

1. The driver tries to connect to host1:port1.
2. The connection to host1:port1 fails.
3. The driver waits two seconds.
4. The driver tries to connect to host1:port1.

5. The connection to host1:port1 fails.
6. The driver waits two seconds.
7. The driver tries to connect to host1:port1.
8. The connection to host1:port1 fails.
9. The driver waits two seconds.
10. The driver tries to connect to host2:port2.
11. The connection to host2:port2 fails.
12. The driver waits two seconds.
13. The driver tries to connect to host2:port2.
14. The connection to host2:port2 fails.
15. The driver waits two seconds.
16. The driver tries to connect to host2:port2.
17. The connection to host2:port2 fails.
18. The driver waits two seconds.
19. The driver tries to connect to host3:port3.
20. The connection to host3:port3 fails.
21. The driver waits two seconds.
22. The driver tries to connect to host3:port3.
23. The connection to host3:port3 fails.
24. The driver waits two seconds.
25. The driver tries to connect to host3:port3.
26. The connection to host3:port3 fails.
27. The driver waits two seconds.
28. The driver throws an SQLException with error code -4499.

The following example shows how to enable client affinities for failover with failback.

Suppose that you set the following properties for a connection to a database:

Property	Value
enableClientAffinitiesList	DB2BaseDataSource.YES (1)
clientRerouteAlternateServername	host1,host2,host3
clientRerouteAlternatePortNumber	port1,port2,port3
maxRetriesForClientReroute	3
retryIntervalForClientReroute	2
affinityFailbackInterval	300

Suppose that the database administrator takes the server that is identified by host1:port1 down for maintenance after a connection is made to host1:port1. The following steps demonstrate failover to an alternate server and failback to the primary server after maintenance is complete.

1. The driver successfully connects to host1:port1 on behalf of an application.
2. The database administrator brings down host1:port1.
3. The application tries to do work on the connection.
4. The driver successfully fails over to host2:port2.

5. After a total of 200 seconds have elapsed, the work is committed.
6. After a total of 300 seconds have elapsed, the failback interval has elasped. The driver checks whether the primary server is up. It is not up, so no failback occurs.
7. After a total of 350 seconds have elapsed, host1:port1 is brought back online.
8. The application continues to do work on host2:port2, because the latest failback interval has not elapsed.
9. After a total of 600 seconds have elapsed, the failback interval has elapsed again. The driver checks whether the primary server is up. It is now up.
10. After a total of 650 seconds have elapsed, the work is committed.
11. After a total of 651 seconds have elapsed, the application tries to start a new transaction on host2:port2. Failback to host1:port1 occurs, so the new transaction starts on host1:port1.

Java client support for high availability for connections to IBM Informix servers

High-availability cluster support on IBM Informix servers provides high availability for client applications, through workload balancing and automatic client reroute. This support is available for applications that use Java clients (JDBC, SQLJ, or pureQuery), or non-Java clients (ODBC, CLI, .NET, OLE DB, PHP, Ruby, or embedded SQL).

For Java clients, you need to use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to take advantage of IBM Informix high-availability cluster support.

For non-Java clients, you need to use one of the following clients or client packages to take advantage of high-availability cluster support:

- IBM Data Server Client
- IBM Data Server Runtime Client
- IBM Data Server Driver Package
- IBM Data Server Driver for ODBC and CLI

Cluster support for high availability for connections to IBM Informix servers includes:

Automatic client reroute

This support enables a client to recover from a failure by attempting to reconnect to the database through any available server in a high-availability cluster. Reconnection to another server is called *failover*. You enable automatic client reroute on the client by enabling workload balancing on the client.

In an IBM Informix environment, primary and standby servers correspond to members of a high-availability cluster that is controlled by a Connection Manager. If multiple Connection Managers exist, the client can use them to determine primary and alternate server information. The client uses alternate Connection Managers only for the initial connection.

Failover for automatic client reroute can be *seamless* or *non-seamless*. With non-seamless failover, when the client application reconnects to an alternate server, the server always returns an error to the application, to indicate that failover (connection to the alternate server) occurred.

For Java, CLI, or .NET client applications, failover for automatic client reroute can be seamless or non-seamless. Seamless failover means that when the application successfully reconnects to an alternate server, the server does not return an error to the application.

Workload balancing
Workload balancing can improve availability of an IBM Informix high-availability cluster. When workload balancing is enabled, the client gets frequent status information about the members of a high-availability cluster. The client uses this information to determine the server to which the next transaction should be routed. With workload balancing, IBM Informix Connection Managers ensure that work is distributed efficiently among servers and that work is transferred to another server if a server has a failure.

Connection concentrator
This support is available for Java applications that connect to IBM Informix. The connection concentrator reduces the resources that are required on IBM Informix database servers to support large numbers of workstation and web users. With the connection concentrator, only a few concurrent, active physical connections are needed to support many applications that concurrently access the database server. When you enable workload balancing on a Java client, you automatically enable the connection concentrator.

Client affinities
Client affinities is an automatic client reroute solution that is controlled completely by the client. It is intended for situations in which you need to connect to a particular primary server. If an outage occurs during the connection to the primary server, you use client affinities to enforce a specific order for failover to alternate servers.

Configuration of IBM Informix high-availability support for Java clients

To configure a IBM Data Server Driver for JDBC and SQLJ client application that connects to an IBM Informix high-availability cluster, you need to connect to an address that represents a Connection Manager, and set the properties that enable workload balancing and the maximum number of connections.

High availability support for Java clients that connect to IBM Informix works for connections that are obtained using the javax.sql.DataSource, javax.sql.ConnectionPoolDataSource, `javax.sql.XADataSource`, or java.sql.DriverManager interface.

Restriction: High availability support for connections that are made with the DriverManager interface has the following restrictions:

- Alternate server information is shared between DriverManager connections only if you create the connections with the same URL and properties.
- You cannot set the clientRerouteServerListJNDIName property or the clientRerouteServerListJNDIContext properties for a DriverManager connection.
- High availability support is not enabled for default connections (jdbc:default:connection).

Before you can enable IBM Data Server Driver for JDBC and SQLJ for high availability for connections to IBM Informix, your installation must have one or more Connection Managers, a primary server, and one or more alternate servers.

The following table describes the basic property settings for enabling workload balancing for Java applications.

Table 11-6. Basic settings to enable IBM Informix high availability support in Java applications

IBM Data Server Driver for JDBC and SQLJ setting	Value
enableSysplexWLB property	true
maxTransportObjects property	The maximum number of connections that the requester can make to the high-availability cluster
Connection address:	
server	The IP address of a Connection Manager. See "Setting server and port properties for connecting to a Connection Manager" on page 11-21.
port	The SQL port number for the Connection Manager. See "Setting server and port properties for connecting to a Connection Manager" on page 11-21.
database	The database name

If you want to enable the connection concentrator, but you do not want to enable workload balancing, you can use these properties.

Table 11-7. Settings to enable the IBM Informix connection concentrator without workload balancing in Java applications

IBM Data Server Driver for JDBC and SQLJ setting	Value
enableSysplexWLB property	false
enableConnectionConcentrator property	true

If you want to fine-tune IBM Informix high-availability support, additional properties are available. The properties for the IBM Data Server Driver for JDBC and SQLJ are listed in the following table. Those properties are configuration properties, and not Connection or DataSource properties.

Table 11-8. Properties for fine-tuning IBM Informix high-availability support for connections from the IBM Data Server Driver for JDBC and SQLJ

IBM Data Server Driver for JDBC and SQLJ configuration property	Description
db2.jcc.maxTransportObjectIdleTime	Specifies the maximum elapsed time in number of seconds before an idle transport is dropped. The default is 60. The minimum supported value is 0.
db2.jcc.maxTransportObjectWaitTime	Specifies the number of seconds that the client will wait for a transport to become available. The default is -1 (unlimited). The minimum supported value is 0.
db2.jcc.minTransportObjects	Specifies the lower limit for the number of transport objects in a global transport object pool. The default value is 0. Any value that is less than or equal to 0 means that the global transport object pool can become empty.

Setting server and port properties for connecting to a Connection Manager

To set the server and port number for connecting to a Connection Manager, follow this process:

- If your high-availability cluster is using a single Connection Manager, and your application is using the DataSource interface for connections, set the serverName and portNumber properties to the server name and port number of the Connection Manager.
- If your high-availability cluster is using a single Connection Manager, and your application is using the DriverManager interface for connections, specify the server name and port number of the Connection manager in the connection URL.
- If your high-availability cluster is using more than one Connection manager, and your application is using the DriverManager interface for connections:
  1. Specify the server name and port number of the main Connection Manager that you want to use in the connection URL.
  2. Set the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties to the server names and port numbers of the alternative Connection Managers that you want to use.
- If your high-availability cluster is using more than one Connection Manager, and your application is using the DataSource interface for connections, use one of the following techniques:
  - Set the server names and port numbers in DataSource properties:
    1. Set the serverName and portNumber properties to the server name and port number of the main Connection Manager that you want to use.
    2. Set the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties to the server names and port numbers of the alternative Connection Managers that you want to use.
  - Configure JNDI for high availability by using a DB2ClientRerouteServerList instance to identify the main Connection Manager and alternative Connection Managers.
    1. Create an instance of DB2ClientRerouteServerList.

       DB2ClientRerouteServerList is a serializable Java bean with the following properties:

Property name	Data type
com.ibm.db2.jcc.DB2ClientRerouteServerList.alternateServerName	String[]
com.ibm.db2.jcc.DB2ClientRerouteServerList.alternatePortNumber	int[]
com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryServerName	String[]
com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryPortNumber	int[]

       getXXX and setXXX methods are defined for each property.
    2. Set the com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryServerName and com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryPortNumber properties to the server name and port number of the main Connection Manager that you want to use.
    3. Set the com.ibm.db2.jcc.DB2ClientRerouteServerList.alternateServerName and com.ibm.db2.jcc.DB2ClientRerouteServerList.alternatePortNumber

properties to the server names and port numbers of the alternative Connection Managers that you want to use.

4. To make the DB2ClientRerouteServerList persistent:
   a. Bind the DB2ClientRerouteServerList instance to the JNDI registry.
   b. Assign the JNDI name of the DB2ClientRerouteServerList object to the IBM Data Server Driver for JDBC and SQLJ clientRerouteServerListJNDIName property.
   c. Assign the name of the JNDI context that is used for binding and lookup of the DB2ClientRerouteServerList instance to the clientRerouteServerListJNDIContext property.

When a DataSource is configured to use JNDI for storing automatic client reroute alternate information, the standard server and port properties of the DataSource are not used for a getConnection request. Instead, the primary server address is obtained from the transient clientRerouteServerList information. If the JNDI store is not available due to a JNDI bind or lookup failure, the IBM Data Server Driver for JDBC and SQLJ attempts to make a connection using the standard server and port properties of the DataSource. Warnings are accumulated to indicate that a JNDI bind or lookup failure occurred.

After a failover:

- The IBM Data Server Driver for JDBC and SQLJ attempts to propagate the updated server information to the JNDI store.
- primaryServerName and primaryPortNumber values that are specified in DB2ClientRerouteServerList are used for the connection. If primaryServerName is not specified, the serverName value for the DataSource instance is used.

Example of enabling IBM Informix high availability support in Java applications

Java client setup for IBM Informix high availability support includes setting several IBM Data Server Driver for JDBC and SQLJ properties.

The following example demonstrates setting up Java client applications for IBM Informix high availability support.

Before you can set up the client, you need to configure one or more high availability clusters that are controlled by Connection Managers.

Follow these steps to set up the client:

1. Verify that the IBM Data Server Driver for JDBC and SQLJ is at the correct level to support workload balancing by following these steps:
   a. Issue the following command in a command line window:

      ```
      java com.ibm.db2.jcc.DB2Jcc -version
      ```

   b. Find a line in the output like this, and check that *nnn* is 3.52 or later.
   c.

      ```
      [jcc] Driver: IBM Data Server Driver for JDBC and SQLJ Architecture nnn xxx
      ```

2. Set IBM Data Server Driver for JDBC and SQLJ Connection or DataSource properties to enable workload balancing:
   - enableSysplexWLB
   - maxTransportObjects
   - maxRefreshInterval

   Start with settings similar to these:

Table 11-9. Example of Connection or DataSource property settings for workload balancing for IBM Informix

Property	Setting
enableSysplexWLB	true
maxTransportObjects	80
maxRefreshInterval	30

Enabling workload balancing enables the connection concentrator by default.

The values that are specified are not intended to be recommended values. You need to determine values based on factors such as the number of transport objects that are available. The number of transport objects must be equal to or greater than the number of connection objects.

3. Set IBM Data Server Driver for JDBC and SQLJ configuration properties to fine-tune the workload balancing for all DataSource or Connection instances that are created under the driver. Set the configuration properties in a DB2JccConfiguration.properties file by following these steps:
   a. Create a DB2JccConfiguration.properties file or edit the existing DB2JccConfiguration.properties file.
   b. Set the following configuration property:
      - db2.jcc.maxTransportObjects

        Start with a setting similar to this one:

        ```
        db2.jcc.maxTransportObjects=500
        ```
   c. Include the directory that contains DB2JccConfiguration.properties in the CLASSPATH concatenation.

Operation of automatic client reroute for connections to IBM Informix from Java clients

When IBM Data Server Driver for JDBC and SQLJ client reroute support is enabled, a Java application that is connected to an IBM Informix high-availability cluster can continue to run when the primary server has a failure.

Automatic client reroute for a Java application that is connected to an IBM Informix server operates in the following way when automatic client reroute is enaabled:

1. During each connection to the data source, the IBM Data Server Driver for JDBC and SQLJ obtains primary and alternate server information.
   - For the first connection to IBM Informix:
     a. The application specifies a server and port for the initial connection. Those values identify a Connection Manager.
     b. The IBM Data Server Driver for JDBC and SQLJ uses the information from the Connection Manager to obtain information about the primary and alternate servers. IBM Data Server Driver for JDBC and SQLJ loads those values into memory.
     c. If the initial connection to the Connection Manager fails:
        – If the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties are set, the IBM Data Server Driver for JDBC and SQLJ connects to the Connection Manager that is identified by clientRerouteAlternateServerName and clientRerouteAlternatePortNumber, and obtains information about primary and alternate servers from that Connection Manager. The IBM

Data Server Driver for JDBC and SQLJ loads those values into memory as the primary and alternate server values.

– If the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties are not set, and a JNDI store is configured by setting the property clientRerouteServerListJNDIName on the DB2BaseDataSource, the IBM Data Server Driver for JDBC and SQLJ connects to the Connection Manager that is identified by DB2ClientRerouteServerList.alternateServerName and DB2ClientRerouteServerList.alternatePortNumber, and obtains information about primary and alternate servers from that Connection Manager. IBM Data Server Driver for JDBC and SQLJ loads the primary and alternate server information from the Connection Manager into memory.

d. If clientRerouteAlternateServerName and clientRerouteAlternatePortNumber are not set, and JNDI is not configured, the IBM Data Server Driver for JDBC and SQLJ checks DNS tables for Connection Manager server and port information. If DNS information exists, the IBM Data Server Driver for JDBC and SQLJ connects to the Connection Manager, obtains information about primary and alternate servers, and loads those values into memory.

e. If no primary or alternate server information is available, a connection cannot be established, and the IBM Data Server Driver for JDBC and SQLJ throws an exception.

• For subsequent connections, the IBM Data Server Driver for JDBC and SQLJ obtains primary and alternate server values from driver memory.

2. The IBM Data Server Driver for JDBC and SQLJ attempts to connect to the data source using the primary server name and port number.

   If the connection is through the DriverManager interface, the IBM Data Server Driver for JDBC and SQLJ creates an internal DataSource object for automatic client reroute processing.

3. If the connection to the primary server fails:

   a. If this is the first connection, the IBM Data Server Driver for JDBC and SQLJ attempts to reconnect to the original primary server.

   b. If this is not the first connection, the IBM Data Server Driver for JDBC and SQLJ attempts to reconnect to the new primary server, whose server name and port number were provided by the server.

   c. If reconnection to the primary server fails, the IBM Data Server Driver for JDBC and SQLJ attempts to connect to the alternate servers.

      If this is not the first connection, the latest alternate server list is used to find the next alternate server.

   Connection to an alternate server is called *failover*.

   The IBM Data Server Driver for JDBC and SQLJ uses the maxRetriesForClientReroute and retryIntervalForClientReroute properties to determine how many times to retry the connection and how long to wait between retries. An attempt to connect to the primary server and alternate servers counts as one retry.

4. If the connection is not established, maxRetriesForClientReroute and retryIntervalForClientReroute are not set, and the original serverName and portNumber values that are defined on the DataSource are different from the

serverName and portNumber values that were used for the original connection, retry the connection with the serverName and portNumber values that are defined on the DataSource.

5. If failover is successful during the initial connection, the driver generates an SQLWarning. If a successful failover occurs after the initial connection:
   - If seamless failover is enabled, the driver retries the transaction on the new server, without notifying the application.

     The following conditions must be satisfied for seamless failover to occur:
     - The enableSeamlessFailover property is set to DB2BaseDataSource.YES (1).

       If Sysplex workload balancing is in effect (the value of the enableSysplexWLB is `true`), seamless failover is attempted, regardless of the enableSeamlessFailover setting.
     - The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
     - There are no global temporary tables in use on the server.
     - There are no open, held cursors.
   - If seamless failover is not in effect, the driver throws an SQLException to the application with error code -4498, to indicate to the application that the connection was automatically reestablished and the transaction was implicitly rolled back. The application can then retry its transaction without doing an explicit rollback first.

     A reason code that is returned with error code -4498 indicates whether any database server special registers that were modified during the original connection are reestablished in the failover connection.

   You can determine whether alternate server information was used in establishing the initial connection by calling the DB2Connection.alternateWasUsedOnConnect method.
6. After failover, driver memory is updated with new primary and alternate server information from the new primary server.

Examples

Example: Automatic client reroute to an IBM Informix server when maxRetriesForClientReroute and retryIntervalForClientReroute are not set: Suppose that the following properties are set for a connection to a database:

Property	Value
enableClientAffinitiesList	DB2BaseDataSource.NO (2)
serverName	host1
portNumber	port1
clientRerouteAlternateServerName	host2
clientRerouteAlternatePortNumber	port2

The following steps demonstrate an automatic client reroute scenario for a connection to IBM Informix:

1. The IBM Data Server Driver for JDBC and SQLJ tries to connect to the Connection Manager that is identified by host1:port1.
2. The connection to host1:port1 fails, so the driver tries to connect to the Connection Manager that is identified by host2:port2.
3. The connection to host2:port2 succeeds.

4. The driver retrieves alternate server information that was received from server host2:port2, and updates its memory with that information.

   Assume that the driver receives a server list that contains host2:port2, host2a:port2a. host2:port2 is stored as the new primary server, and host2a:port2a is stored as the new alternate server. If another communication failure is detected on this same connection, or on another connection that is created from the same DataSource, the driver tries to connect to host2:port2 as the new primary server. If that connection fails, the driver tries to connect to the new alternate server host2a:port2a.
5. The driver connects to host1a:port1a.
6. A failure occurs during the connection to host1a:port1a.
7. The driver tries to connect to host2a:port2a.
8. The connection to host2a:port2a is successful.
9. The driver retrieves alternate server information that was received from server host2a:port2a, and updates its memory with that information.

Example: Automatic client reroute to an IBM Informix server when maxRetriesForClientReroute and retryIntervalForClientReroute are set for multiple retries: Suppose that the following properties are set for a connection to a database:

Property	Value
enableClientAffinitiesList	DB2BaseDataSource.NO (2)
serverName	host1
portNumber	port1
clientRerouteAlternateServerName	host2
clientRerouteAlternatePortNumber	port2
maxRetriesForClientReroute	3
retryIntervalForClientReroute	2

The following steps demonstrate an automatic client reroute scenario for a connection to IBM Informix:

1. The IBM Data Server Driver for JDBC and SQLJ tries to connect to the Connection Manager that is identified by host1:port1.
2. The connection to host1:port1 fails, so the driver tries to connect to the Connection Manager that is identified by host2:port2.
3. The connection to host2:port2 succeeds.
4. The driver retrieves alternate server information from the connection manager that is identified by host2:port2, and updates its memory with that information. Assume that the Connection Manager identifies host1a:port1a as the new primary server, and host2a:port2a as the new alternate server.
5. The driver tries to connect to host1a:port1a.
6. The connection to host1a:port1a fails.
7. The driver tries to connect to host2a:port2a.
8. The connection to host2a:port2a fails.
9. The driver waits two seconds.
10. The driver tries to connect to host1a:port1a.
11. The connection to host1a:port1a fails.
12. The driver tries to connect to host2a:port2a.

13. The connection to host2a:port2a fails.
14. The driver waits two seconds.
15. The driver tries to connect to host1a:port1a.
16. The connection to host1a:port1a fails.
17. The driver tries to connect to host2a:port2a.
18. The connection to host2a:port2a fails.
19. The driver waits two seconds.
20. The driver throws an SQLException with error code -4499.

Operation of workload balancing for connections to IBM Informix from Java clients

Workload balancing (also called transaction-level workload balancing) for connections to IBM Informix contributes to high availability by balancing work among servers in a high-availability cluster at the start of a transaction.

The following overview describes the steps that occur when a client connects to an IBM Informix Connection Manager, and workload balancing is enabled:

1. When the client first establishes a connection using the IP address of the Connection Manager, the Connection Manager returns the server list and the connection details (IP address, port, and weight) for the servers in the cluster.

   The server list is cached by the client. The default lifespan of the cached server list is 30 seconds.
2. At the start of a new transaction, the client reads the cached server list to identify a server that has untapped capacity, and looks in the transport pool for an idle transport that is tied to the under-utilized server. (An idle transport is a transport that has no associated connection object.)
   - If an idle transport is available, the client associates the connection object with the transport.
   - If, after a user-configurable timeout, no idle transport is available in the transport pool and no new transport can be allocated because the transport pool has reached its limit, an error is returned to the application.
3. When the transaction runs, it accesses the server that is tied to the transport.
4. When the transaction ends, the client verifies with the server that transport reuse is still allowed for the connection object.
5. If transport reuse is allowed, the server returns a list of SET statements for special registers that apply to the execution environment for the connection object.

   The client caches these statements, which it replays in order to reconstruct the execution environment when the connection object is associated with a new transport.
6. The connection object is then dissociated from the transport, if the client determines that it needs to do so.
7. The client copy of the server list is refreshed when a new connection is made, or every 30 seconds, or at the user-configured interval.
8. When workload balancing is required for a new transaction, the client uses the previously described process to associate the connection object with a transport.

Application programming requirements for high availability for connections from Java clients to IBM Informix servers

Failover for automatic client reroute can be seamless or non-seamless. If failover for connections to IBM Informix is not seamless, you need to add code to account for the errors that are returned when failover occurs.

If failover is non-seamless, and a connection is reestablished with the server, SQLCODE -4498 (for Java clients) or SQL30108N (for non-Java clients) is returned to the application. All work that occurred within the current transaction is rolled back. In the application, you need to:

- Check the reason code that is returned with the error. Determine whether special register settings on the failing data sharing member are carried over to the new (failover) data sharing member. Reset any special register values that are not current.
- Execute all SQL operations that occurred during the previous transaction.

The following conditions must be satisfied for seamless failover to occur during connections to IBM Informix databases:

- The application programming language is Java, CLI, or .NET.
- The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
- The data server must allow transport reuse at the end of the previous transaction.
- All global session data is closed or dropped.
- There are no open held cursors.
- If the application uses CLI, the application cannot perform actions that require the driver to maintain a history of previously called APIs in order to replay the SQL statement. Examples of such actions are specifying data at execution time, performing compound SQL, or using array input.
- The application is not a stored procedure.
- Autocommit is not enabled. Seamless failover can occur when autocommit is enabled. However, the following situation can cause problems: Suppose that SQL work is successfully executed and committed at the data server, but the connection or server goes down before acknowledgment of the commit operation is sent back to the client. When the client re-establishes the connection, it replays the previously committed SQL statement. The result is that the SQL statement is executed twice. To avoid this situation, turn autocommit off when you enable seamless failover.

In addition, seamless automatic client reroute might not be successful if the application has autocommit enabled. With autocommit enabled, a statement might be executed and committed multiple times.

Client affinities for connections to IBM Informix from Java clients

Client affinities is a client-only method for providing automatic client reroute capability.

Client affinities is available for applications that use CLI, .NET, or Java (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity). All rerouting is controlled by the driver.

Client affinities is intended for situations in which you need to connect to a particular primary server. If an outage occurs during the connection to the primary server, you need to enforce a specific order for failover to alternate servers. You should use client affinities for automatic client reroute only if automatic client reroute that uses server failover capabilities does not work in your environment.

As part of configuration of client affinities, you specify a list of alternate servers, and the order in which connections to the alternate servers are tried. When client affinities is in use, connections are established based on the list of alternate servers instead of the host name and port number that are specified by the application. For example, if an application specifies that a connection is made to server1, but the configuration process specifies that servers should be tried in the order (server2, server3, server1), the initial connection is made to server2 instead of server1.

Failover with client affinities is seamless, if the following conditions are true:

- The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
- There are no global temporary tables in use on the server.
- There are no open, held cursors.

When you use client affinities, you can specify that if the primary server returns to operation after an outage, connections return from an alternate server to the primary server on a transaction boundary. This activity is known as *failback*.

Configuration of client affinities for Java clients for IBM Informix connections

To enable support for client affinities in Java applications, you set properties to indicate that you want to use client affinities, and to specify the primary and alternate servers.

The following table describes the property settings for enabling client affinities for Java applications.

Table 11-10. Property settings to enable client affinities for Java applications

IBM Data Server Driver for JDBC and SQLJ setting	Value
enableClientAffinitiesList	DB2BaseDataSource.YES (1)
clientRerouteAlternateServerName	A comma-separated list of the primary server and alternate servers
clientRerouteAlternatePortNumber	A comma-separated list of the port numbers for the primary server and alternate servers
enableSeamlessFailover	DB2BaseDataSource.YES (1) for seamless failover; DB2BaseDataSource.NO (2) or enableSeamlessFailover not specified for no seamless failover
maxRetriesForClientReroute	The number of times to retry the connection to each server, including the primary server, after a connection to the primary server fails. The default is 3.
retryIntervalForClientReroute	The number of seconds to wait between retries. The default is no wait.

Table 11-10. Property settings to enable client affinities for Java applications (continued)

IBM Data Server Driver for JDBC and SQLJ setting	Value
affinityFailbackInterval	The number of seconds to wait after the first transaction boundary to fail back to the primary server. Set this value if you want to fail back to the primary server.

Example of enabling client affinities in Java clients for IBM Informix connections

Before you can use client affinities for automatic client reroute in Java applications, you need to set properties to indicate that you want to use client affinities, and to identify the primary alternate servers.

The following example shows how to enable client affinities for failover without failback.

Suppose that you set the following properties for a connection to a database:

Property	Value
enableClientAffinitiesList	DB2BaseDataSource.YES (1)
clientRerouteAlternateServername	host1,host2,host3
clientRerouteAlternatePortNumber	port1,port2,port3
maxRetriesForClientReroute	3
retryIntervalForClientReroute	2

Suppose that a communication failure occurs during a connection to the server that is identified by host1:port1. The following steps demonstrate automatic client reroute with client affinities.

1. The driver tries to connect to host1:port1.
2. The connection to host1:port1 fails.
3. The driver waits two seconds.
4. The driver tries to connect to host1:port1.
5. The connection to host1:port1 fails.
6. The driver waits two seconds.
7. The driver tries to connect to host1:port1.
8. The connection to host1:port1 fails.
9. The driver waits two seconds.
10. The driver tries to connect to host2:port2.
11. The connection to host2:port2 fails.
12. The driver waits two seconds.
13. The driver tries to connect to host2:port2.
14. The connection to host2:port2 fails.
15. The driver waits two seconds.
16. The driver tries to connect to host2:port2.
17. The connection to host2:port2 fails.
18. The driver waits two seconds.
19. The driver tries to connect to host3:port3.

20. The connection to host3:port3 fails.
21. The driver waits two seconds.
22. The driver tries to connect to host3:port3.
23. The connection to host3:port3 fails.
24. The driver waits two seconds.
25. The driver tries to connect to host3:port3.
26. The connection to host3:port3 fails.
27. The driver waits two seconds.
28. The driver throws an SQLException with error code -4499.

The following example shows how to enable client affinities for failover with failback.

Suppose that you set the following properties for a connection to a database:

Property	Value
enableClientAffinitiesList	DB2BaseDataSource.YES (1)
clientRerouteAlternateServername	host1,host2,host3
clientRerouteAlternatePortNumber	port1,port2,port3
maxRetriesForClientReroute	3
retryIntervalForClientReroute	2
affinityFailbackInterval	300

Suppose that the database administrator takes the server that is identified by host1:port1 down for maintenance after a connection is made to host1:port1. The following steps demonstrate failover to an alternate server and failback to the primary server after maintenance is complete.

1. The driver successfully connects to host1:port1 on behalf of an application.
2. The database administrator brings down host1:port1.
3. The application tries to do work on the connection.
4. The driver successfully fails over to host2:port2.
5. After a total of 200 seconds have elapsed, the work is committed.
6. After a total of 300 seconds have elapsed, the failback interval has elasped. The driver checks whether the primary server is up. It is not up, so no failback occurs.
7. After a total of 350 seconds have elapsed, host1:port1 is brought back online.
8. The application continues to do work on host2:port2, because the latest failback interval has not elapsed.
9. After a total of 600 seconds have elapsed, the failback interval has elapsed again. The driver checks whether the primary server is up. It is now up.
10. After a total of 650 seconds have elapsed, the work is committed.
11. After a total of 651 seconds have elapsed, the application tries to start a new transaction on host2:port2. Failback to host1:port1 occurs, so the new transaction starts on host1:port1.

Java client direct connect support for high availability for connections to DB2 for z/OS servers

Sysplex workload balancing functionality on DB2 for z/OS servers provides high availability for client applications that connect directly to a data sharing group. Sysplex workload balancing functionality provides workload balancing and automatic client reroute capability. This support is available for applications that use Java clients (JDBC, SQLJ, or pureQuery) that use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, or non-Java clients (ODBC, CLI, .NET, OLE DB, PHP, Ruby, or embedded SQL). Workload balancing is transparent to applications.

A Sysplex is a set of z/OS systems that communicate and cooperate with each other through certain multisystem hardware components and software services to process customer workloads. DB2 for z/OS subsystems on the z/OS systems in a Sysplex can be configured to form a data sharing group. With data sharing, applications that run on more than one DB2 for z/OS subsystem can read from and write to the same set of data concurrently. One or more coupling facilities provide high-speed caching and lock processing for the data sharing group. The Sysplex, together with the Workload Manager (WLM), dynamic virtual IP address (DVIPA), and the Sysplex Distributor, allow a client to access a DB2 for z/OS database over TCP/IP with network resilience, and distribute transactions for an application in a balanced manner across members within the data sharing group.

Central to these capabilities is a server list that the data sharing group returns on connection boundaries and optionally on transaction boundaries. This list contains the IP address and WLM weight for each data sharing group member. With this information, a client can distribute transactions in a balanced manner, or identify the member to use when there is a communication failure.

The server list is returned on the first successful connection to the DB2 for z/OS data server. After the client has received the server list, the client directly accesses a data sharing group member based on information in the server list.

DB2 for z/OS provides several methods for clients to access a data sharing group. The access method that is set up for communication with the data sharing group determines whether Sysplex workload balancing is possible. The following table lists the access methods and indicates whether Sysplex workload balancing is possible.

Table 11-11. Data sharing access methods and Sysplex workload balancing

Data sharing access method[1]	Description	Sysplex workload balancing possible?
Group access	A requester uses the group's dynamic virtual IP address (DVIPA) to make an initial connection to the DB2 for z/OS location. A connection to the data sharing group that uses the group IP address and SQL port is always successful if at least one member is started. The server list that is returned by the data sharing group contains: • A list of members that are currently active and can perform work • The WLM weight for each member The group IP address is configured using the z/OS Sysplex distributor. To clients that are outside the Sysplex, the Sysplex distributor provides a single IP address that represents a DB2 location. In addition to providing fault tolerance, the Sysplex distributor can be configured to provide connection load balancing.	Yes
Member-specific access	A requester uses a location alias to make an initial connection to one of the members that is represented by the alias. A connection to the data sharing group that uses the group IP address and alias SQL port is always successful if at least one member is started. The server list that is returned by the data sharing group contains: • A list of members that are currently active, can perform work, and have been configured as an alias • The WLM weight for each member The requester uses this information to connect to the member or members with the most capacity that are also associated with the location alias. Member-specific access is used when requesters need to take advantage of Sysplex workload balancing among a subset of members of a data sharing group.	Yes
Single-member access	Single-member access is used when requesters need to access only one member of a data sharing group. For single-member access, the connection uses the member-specific IP address.	No

Note:

1. For more information on data sharing access methods, see http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db29.doc.dshare/db2z_tcpipaccessmethods.htm.

Sysplex workload balancing includes automatic client reroute: Automatic client reroute support enables a client to recover from a failure by attempting to reconnect to the database through any available member of a Sysplex. Reconnection to another member is called *failover*.

Sysplex workload balancing during migration of a data sharing group to DB2 9.1 for z/OS or DB2 10 for z/OS: In general, if you use IBM Data Server Driver for JDBC and SQLJ Version 3.61 or 4.11, migration of a data sharing group from DB2 for z/OS Version 8 or Version 9.1 to Version 10, or DB2 for z/OS Version 8 to Version 9.1 does not cause an outage for Java applications that connect to the data sharing group using IBM Data Server Driver for JDBC and SQLJ type 4 connectivity. You do not need to restart all members of the data sharing group or JVMs to maintain balanced connections. In addition, if you use IBM Data Server Driver for JDBC and SQLJ Version 3.62 or 4.12 or later, in any mode during the migration from DB2 for z/OS Version 8 or Version 9.1 new-function mode to Version 10 new-function mode, or from DB2 for z/OS Version 8 new-function mode to Version 9.1 new-function mode, new applications that use features that require a higher DRDA level can coexist with old applications that use a lower DRDA level, if they use the same DataSource. This coexistence includes reversion from a mode to a previous mode, such as reversion from Version 10 ENFM9 to CM9*. For coexistence of DRDA levels, you need to have APAR PM24292 installed on the DB2 for z/OS Version 9.1 and DB2 for z/OS Version 10 data servers.

Sysplex workload balancing during migration of a data sharing group to DB2 9.1 for z/OS: When you migrate a data sharing group to DB2 9.1 for z/OS new-function mode, you need to take these steps:

1. Restart all members of the data group.
2. Restart the JVMs under which applications that connect to the data sharing group using IBM Data Server Driver for JDBC and SQLJ type 4 connectivity run.

Stopping and starting all members prevents applications that use Sysplex workload balancing from having unbalanced connections.

For Java, CLI, or .NET client applications, failover for automatic client reroute can be *seamless* or *non-seamless*. Seamless failover means that when the application successfully reconnects to an alternate server, the server does not return an error to the application.

Client direct connect support for high availability with a DB2 Connect server: Client direct connect support for high availability requires a DB2 Connect license, but does not need a DB2 Connect server. The client connects directly to DB2 for z/OS. If you use a DB2 Connect server, but set up your environment for client high availability, you cannot take advantage of some of the features that a direct connection to DB2 for z/OS provides, such as transaction-level workload balancing or automatic client reroute capability that is provided by the Sysplex.

Do not use client affinities: Client affinities should not be used as a high availability solution for direct connections to DB2 for z/OS. Client affinities is not applicable to a DB2 for z/OS data sharing environment, because all members of a data sharing group can access data concurrently. A major disadvantage of client affinities in a data sharing environment is that if failover occurs because a data sharing group member fails, the member that fails might have retained locks that can severely affect transactions on the member to which failover occurs.

Configuration of Sysplex workload balancing at a Java client

To configure a IBM Data Server Driver for JDBC and SQLJ client application that connects directly to DB2 for z/OS to use Sysplex workload balancing, you need to use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity. You also need to connect to an address that represents the data sharing group (for group access)

or a subset of the data sharing group (for member-specific access), and set the properties that enable workload balancing and the maximum number of connections.

The following table describes the basic property settings for Java applications.

Table 11-12. Basic settings to enable Sysplex high availability support in Java applications

Data sharing access method	IBM Data Server Driver for JDBC and SQLJ setting	Value
Group access	enableSysplexWLB property	true
	Connection address:	
	server	The group IP address or domain name of the data sharing group
	port	The SQL port number for the DB2 location
	database	The DB2 location name that is defined during installation
Member-specific access	enableSysplexWLB property	true
	Connection address:	
	server	The group IP address or domain name of the data sharing group
	port	The port number for the DB2 location alias
	database	The name of the DB2 location alias that represents a subset of the members of the data sharing group

If you want to fine-tune Sysplex workload balancing, additional properties are available.

The following IBM Data Server Driver for JDBC and SQLJ Connection or DataSource properties control Sysplex workload balancing.

Table 11-13. Connection or DataSource properties for fine-tuning Sysplex workload balancing for direct connections from the IBM Data Server Driver for JDBC and SQLJ to DB2 for z/OS

IBM Data Server Driver for JDBC and SQLJ property	Description
blockingReadConnectionTimeout	Specifies the amount of time in seconds before a connection socket read times out. This property affects all requests that are sent to the data source after a connection is successfully established. The default is 0, which means that there is no timeout. Set this property to a value greater by a few seconds than the time that is required to execute the longest query in the application.
loginTimeout	Specifies the maximum time in seconds to wait for a new connection to a data source. After the number of seconds that are specified by loginTimeout have elapsed, the driver closes the connection to the data source. The default is 0, which means that the timeout value is the default system timeout value. The recommended value is five seconds.

Table 11-13. Connection or DataSource properties for fine-tuning Sysplex workload balancing for direct connections from the IBM Data Server Driver for JDBC and SQLJ to DB2 for z/OS (continued)

IBM Data Server Driver for JDBC and SQLJ property	Description
maxRefreshInterval	Specifies the maximum amount of time in seconds between refreshes of the client copy of the server list. The default is 30. The minimum valid value is 1.
maxTransportObjects	Specifies the maximum number of connections that the requester can make to the data sharing group. The default is -1, which means an unlimited number.

The following IBM Data Server Driver for JDBC and SQLJ configuration properties also control Sysplex workload balancing.

Table 11-14. Configuration properties for fine-tuning Sysplex workload balancing for direct connections from the IBM Data Server Driver for JDBC and SQLJ to DB2 for z/OS

IBM Data Server Driver for JDBC and SQLJ configuration property	Description
db2.jcc.maxTransportObjectIdleTime	Specifies the maximum elapsed time in number of seconds before an idle transport is dropped. The default is 60. The minimum supported value is 0.
db2.jcc.maxTransportObjectWaitTime	Specifies the number of seconds that the client will wait for a transport to become available. The default is -1 (unlimited). The minimum supported value is 0.
db2.jcc.minTransportObjects	Specifies the lower limit for the number of transport objects in a global transport object pool. The default value is 0. Any value that is less than or equal to 0 means that the global transport object pool can become empty.

Example of enabling DB2 for z/OS Sysplex workload balancing in Java applications

Java client setup for Sysplex workload balancing includes setting several IBM Data Server Driver for JDBC and SQLJ properties.

The following examples demonstrate setting up Java client applications for Sysplex workload balancing for high availability.

Before you can set up the client, you need to configure the following server software:

- WLM for z/OS

  For workload balancing to work efficiently, DB2 work needs to be classified. Classification applies to the first non-SET SQL statement in each transaction. Among the areas by which you need to classify the work are:
  - Authorization ID
  - Client info properties
  - Stored procedure name

    The stored procedure name is used for classification only if the first statement that is issued by the client in the transaction is an SQL CALL statement.

  For a complete list of classification attributes, see the information on classification attributes at the following URL:

  `http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/-com.ibm.db29.doc.perf/db2z_classificationattributes.htm`

- DB2 for z/OS, set up for data sharing

Example of setup with WebSphere Application Server

This example assumes that you are using WebSphere Application Server. The minimum level of WebSphere Application Server is Version 5.1.

Follow these steps to set up the client:

1. Verify that the IBM Data Server Driver for JDBC and SQLJ is at the correct level to support the Sysplex workload balancing by following these steps:
   a. Issue the following command

      ```
      java com.ibm.db2.jcc.DB2Jcc -version
      ```

   b. Find a line in the output like this, and check that *nnn* is 3.50 or later.

      ```
      [jcc] Driver: IBM Data Server Driver for JDBC and SQLJ Architecture nnn xxx
      ```

2. Set the IBM Data Server Driver for JDBC and SQLJ data source property enableSysplexWLB to enable the Sysplex workload balancing.

   In the WebSphere Application Server administrative console, set the following properties for the DataSource that your application uses to connect to the data source. Start with settings similar to these:

Table 11-15. Example of data source property settings for IBM Data Server Driver for JDBC and SQLJ Sysplex workload balancing for DB2 for z/OS

Property	Setting
enableSysplexWLB	true
maxRefreshInterval	30
maxTransportObjects	80

   Use property maxTransportObjects to limit the total number of connections to the DB2 for z/OS data sharing group.

   Recommendation: Set maxTransportObjects to a value that is larger than the MaxConnections value for the WebSphere Application Server connection pool. Doing so allows workload balancing to occur between data sharing members without the need to open and close connections to DB2.

3. Set IBM Data Server Driver for JDBC and SQLJ configuration properties to fine-tune workload balancing for all DataSource or Connection instances that are created under the driver. Set the configuration properties in a DB2JccConfiguration.properties file by following these steps:
   a. Create a DB2JccConfiguration.properties file or edit the existing DB2JccConfiguration.properties file.
   b. Set the db2.jcc.maxTransportObjects configuration property only if multiple DataSource objects are defined that point to the same data sharing group, and the number of connections across the different DataSource objects needs to be limited.

      Start with a setting similar to this one:

      ```
      db2.jcc.maxTransportObjects=500
      ```

   c. Add the directory path for DB2JccConfiguration.properties to the WebSphere Application Server IBM Data Server Driver for JDBC and SQLJ classpath.
   d. Restart WebSphere Application Server.

Example of setup for DriverManager connections

This example assumes that you are using the DriverManager interface to establish a connection.

Follow these steps to set up the client:

1. Verify that the IBM Data Server Driver for JDBC and SQLJ is at the correct level to support the Sysplex workload balancing by following these steps:
   a. Issue the following command

      ```
      java com.ibm.db2.jcc.DB2Jcc -version
      ```

   b. Find a line in the output like this, and check that *nnn* is 3.50 or later. A minimum driver level of 3.50 is required for using Sysplex workload balancing for DriverManager connections.
   c.

      ```
      [jcc] Driver: IBM Data Server Driver for JDBC and SQLJ Architecture nnn xxx
      ```

2. Set the IBM Data Server Driver for JDBC and SQLJ Connection property enableSysplexWLB to enable workload balancing. You can also use property maxTransportObjects to limit the total number of connections to the DB2 for z/OS data sharing group, and maxRefreshInterval to specify the maximum amount of time between refreshes of the client copy of the server list. A minimum driver level of 3.58 is required for using maxRefreshInterval.

   ```
   java.util.Properties properties = new java.util.Properties();
   properties.put("user", "xxxx");
   properties.put("password", "yyyy");
   properties.put("enableSysplexWLB", "true");
   properties.put("maxTransportObjects", "80");
   properties.put("maxRefreshInterval", "30");
   java.sql.Connection con =
     java.sql.DriverManager.getConnection(url, properties);
   ```

3. Set IBM Data Server Driver for JDBC and SQLJ configuration properties to fine-tune workload balancing for all DataSource or Connection instances that are created under the driver. Set the configuration properties in a DB2JccConfiguration.properties file by following these steps:
   a. Create a DB2JccConfiguration.properties file or edit the existing DB2JccConfiguration.properties file.
   b. Set the db2.jcc.maxTransportObjects configuration property only if multiple DataSource objects are defined that point to the same data sharing group, and the number of connections across the different DataSource objects needs to be limited.

      Start with a setting similar to this one:

      ```
      db2.jcc.maxTransportObjects=500
      ```

   c. Include the directory that contains DB2JccConfiguration.properties in the CLASSPATH concatenation.

Operation of Sysplex workload balancing for connections from Java clients to DB2 for z/OS servers

Sysplex workload balancing (also called transaction-level workload balancing) for connections to DB2 for z/OS contributes to high availability by balancing work among members of a data sharing group at the start of a transaction.

The following overview describes the steps that occur when a client connects to a DB2 for z/OS Sysplex, and Sysplex workload balancing is enabled:

1. When the client first establishes a connection using the sysplex-wide IP address called the group IP address, or when a connection is reused by another connection object, the server returns member workload distribution information.

   The default lifespan of the cached server list is 30 seconds.
2. At the start of a new transaction, the client reads the cached server list to identify a member that has untapped capacity, and looks in the transport pool for an idle transport that is tied to the under-utilized member. (An idle transport is a transport that has no associated connection object.)
   - If an idle transport is available, the client associates the connection object with the transport.
   - If, after a user-configurable timeout, no idle transport is available in the transport pool and no new transport can be allocated because the transport pool has reached its limit, an error is returned to the application.
3. When the transaction runs, it accesses the member that is tied to the transport.
4. When the transaction ends, the client verifies with the server that transport reuse is still allowed for the connection object.
5. If transport reuse is allowed, the server returns a list of SET statements for special registers that apply to the execution environment for the connection object.

   The client caches these statements, which it replays in order to reconstruct the execution environment when the connection object is associated with a new transport.
6. The connection object is then disassociated from the transport.
7. The client copy of the server list is refreshed when a new connection is made, or every 30 seconds.
8. When workload balancing is required for a new transaction, the client uses the same process to associate the connection object with a transport.

Operation of automatic client reroute for connections from Java clients to DB2 for z/OS

Automatic client reroute support provides failover support when an IBM data server client loses connectivity to a member of a DB2 for z/OS Sysplex. Automatic client reroute enables the client to recover from a failure by attempting to reconnect to the database through any available member of the Sysplex.

Automatic client reroute is enabled by default when Sysplex workload balancing is enabled.

Client support for automatic client reroute is available in IBM data server clients that have a DB2 Connect license. The DB2 Connect server is not required to perform automatic client reroute.

Automatic client reroute for connections to DB2 for z/OS operates in the following way:

1. As part of the response to a COMMIT request from the client, the data server returns:
   - An indicator that specifies whether transports can be reused. Transports can be reused if there are no resources remaining, such as held cursors.
   - SET statements that the client can use to replay the connection state during transport reuse.

2. If the first SQL statement in a transaction fails, and transports can be reused:
   - No error is reported to the application.
   - The failing SQL statement is executed again.
   - The SET statements that are associated with the logical connection are replayed to restore the connection state.
3. If an SQL statement that is not the first SQL statement in a transaction fails, and transports can be reused:
   - The transaction is rolled back.
   - The application is reconnected to the data server.
   - The SET statements that are associated with the logical connection are replayed to restore the connection state.
   - SQL error -30108 (for Java) or SQL30108N (for non-Java clients) is returned to the application to notify it of the rollback and successful reconnection. The application needs to include code to retry the failed transaction.
4. If an SQL statement that is not the first SQL statement in a transaction fails, and transports cannot be reused:
   - The logical connection is returned to its initial, default state.
   - SQL error -30081 (for Java) or SQL30081N (for non-Java clients) is returned to the application to notify it that reconnection was unsuccessful. The application needs to reconnect to the data server, reestablish the connection state, and retry the failed transaction.
5. If connections to all members of the data sharing member list have been tried, and none have succeeded, a connection is tried using the URL that is associated with the data sharing group, to determine whether any members are now available.

Application programming requirements for high availability for connections from Java clients to DB2 for z/OS servers

Failover for automatic client reroute can be seamless or non-seamless. If failover for connections to DB2 for z/OS is not seamless, you need to add code to account for the errors that are returned when failover occurs.

If failover is not seamless, and a connection is reestablished with the server, SQLCODE -30108 (SQL30108N) is returned to the application. All work that occurred within the current transaction is rolled back. In the application, you need to:

- Check the reason code that is returned with the -30108 error to determine whether special register settings on the failing data sharing member are carried over to the new (failover) data sharing member. Reset any special register values that are not current.
- Execute all SQL operations that occurred since the previous commit operation.

The following conditions must be satisfied for seamless failover to occur for direct connections to DB2 for z/OS:

- The application language is Java, CLI, or .NET.
- The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
- The data server allows transport reuse at the end of the previous transaction. An exception to this condition is if transport reuse is not granted because the application was bound with KEEPDYNAMIC(YES).
- All global session data is closed or dropped.

- There are no open, held cursors.
- If the application uses CLI, the application cannot perform actions that require the driver to maintain a history of previously called APIs in order to replay the SQL statement. Examples of such actions are specifying data at execution time, performing compound SQL, or using array input.
- The application is not a stored procedure.
- The application is not running in a Federated environment.
- Two-phase commit is used, if transactions are dependent on the success of previous transactions. When a failure occurs during a commit operation, the client has no information about whether work was committed or rolled back at the server. If each transaction is dependent on the success of the previous transaction, use two-phase commit. Two-phase commit requires the use of XA support.

Chapter 12. Java 2 Platform, Enterprise Edition

The Java 2 Platform, Enterprise Edition (J2EE), reduces the cost and complexity of developing multi-tier services.

In today's global business environment, organizations need to extend their reach, lower their costs, and lower their response times by providing services that are easily accessible to their customers, employees, suppliers, and other business partners. These services need to have the following characteristics:

- Highly available, to meet the requirements of global business environment
- Secure, to protect the privacy of the users and the integrity of the enterprise
- Reliable and scalable, so that business transactions are accurately and promptly processed

In most cases, these services are provided with the help of multi-tier applications with each tier serving a specific purpose.

J2EE achieves these benefits by defining a standard architecture that is delivered as the following elements:

- J2EE Application Model, a standard application model for developing multi-tier, thin-client services
- J2EE Platform, a standard platform for hosting J2EE applications
- J2EE Compatibility Test Suite for verifying that a J2EE platform product complies with the J2EE platform standard
- J2EE Reference Implementation for demonstrating the capabilities of J2EE, and for providing an operational definition of the J2EE platform

Application components of Java 2 Platform, Enterprise Edition support

The Java 2 Platform, Enterprise Edition (J2EE) provides the runtime environment for hosting J2EE applications.

The runtime environment defines four application component types that a J2EE product must support:

- Application clients are Java programming language programs that are typically GUI programs that execute on a desktop computer. Application clients have access to all of the facilities of the J2EE middle tier.
- Applets are GUI components that typically execute in a web browser, but can execute in a variety of other applications or devices that support the applet programming model.
- Servlets, JavaServer Pages (JSPs), filters, and web event listeners typically execute in a web server and might respond to HTTP requests from web clients. Servlets, JSPs, and filters can be used to generate HTML pages that are an application's user interface. They can also be used to generate XML or other format data that is consumed by other application components. Servlets, pages created with the JSP technology, web filters, and web event listeners are referred to collectively in this specification as *web components*. Web applications are composed of web components and other data such as HTML pages.

- Enterprise JavaBeans (EJB) components execute in a managed environment that supports transactions. Enterprise beans typically contain the business logic for a J2EE application.

The application components listed above can divided into three categories, based on how they can be deployed and managed:

- Components that are deployed, managed, and executed on a J2EE server.
- Components that are deployed, managed on a J2EE server, but are loaded to and executed on a client machine.
- Components whose deployment and management are not completely defined by this specification. Application clients can be under this category.

The runtime support for these components is provided by *containers*.

Java 2 Platform, Enterprise Edition containers

A container provides a federated view of the underlying Java 2 Platform, Enterprise Edition (J2EE) APIs to the application components.

A typical J2EE product will provide a container for each application component type; application client container, applet container, web container, and enterprise bean container. The container tools also understand the file formats for packaging the application components for deployment.

The specification requires that these containers provide a Java-compatible runtime environment. This specification defines a set of standard services that each J2EE product must support. These standard services are:

- HTTP service
- HTTPS service
- Java transaction API
- Remote invocation method
- Java IDL
- JDBC API
- Java message service
- Java naming and directory interface
- JavaMail
- JavaBeans activation framework
- Java API for XML parsing
- Connector architecture
- Java authentication and authorization service

Java 2 Platform, Enterprise Edition Server

One part of a Java 2 Platform, Enterprise Edition (J2EE) container is a server.

A J2EE Product Provider typically implements the J2EE server-side functionality. The J2EE client functionality is typically built on J2SE technology.

The IBM WebSphere Application Server is a J2EE-compliant server.

Java 2 Platform, Enterprise Edition database requirements

Java 2 Platform, Enterprise Edition requires a data server to store business data. The data server must be accessible through the JDBC API.

The database is accessible from web components, enterprise beans, and application client components. The database need not be accessible from applets.

Java Naming and Directory Interface (JNDI)

JNDI enables Java platform-based applications to access multiple naming and directory services.

It is a part of the Java Enterprise application programming interface (API) set. JNDI makes it possible for developers to create portable applications that are enabled for a number of different naming and directory services, including: file systems; directory services such as Lightweight Directory Access Protocol (LDAP) and Novell Directory Services, and distributed object systems such as the Common Object Request Broker Architecture (CORBA), Java Remote Method Invocation (RMI), and Enterprise JavaBeans (EJB).

The JNDI API has two parts: an application-level interface used by the application components to access naming and directory services and a service provider interface to attach a provider of a naming and directory service.

Java transaction management

Java 2 Platform, Enterprise Edition (J2EE) simplifies application programming for distributed transaction management.

J2EE includes support for distributed transactions through two specifications, Java Transaction API (JTA) and Java Transaction Service (JTS). JTA is a high-level, implementation-independent, protocol-independent API that allows applications and application servers to access transactions. In addition, the JTA is always enabled.

The IBM Data Server Driver for JDBC and SQLJ and the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows implement the JTA and JTS specifications.

For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity distributed transactions are supported to DB2 Database for Linux, UNIX, and Windows, DB2 for z/OS, and DB2 for i servers.

JTA 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.

JTS specifies the implementation of a Transaction Manager which supports JTA and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 specification at the level below the API. JTS propagates transactions using IIOP.

JTA and JTS allow application J2EE servers to take the burden of transaction management off of the component developer. Developers can define the transactional properties of EJB technology based components during design or deployment using declarative statements in the deployment descriptor. The application server takes over the transaction management responsibilities.

In the IDS and WebSphere Application Server environment, WebSphere Application Server assumes the role of transaction manager, and IDS acts as a resource manager. WebSphere Application Server implements JTS and part of JTA, and the

JDBC drivers also implement part of JTA so that WebSphere Application Server and IDS can provide coordinated distributed transactions.

It is not necessary to configure IDS to be JTA-enabled in the WebSphere Application Server environment because the JDBC drivers automatically detect this environment.

The IBM Data Server Driver for JDBC and SQLJ provides these two DataSource classes:
- com.ibm.db2.jcc.DB2ConnectionPoolDataSource
- com.ibm.db2.jcc.DB2XADataSource

WebSphere Application Server provides pooled connections to databases. If the application will be involved in a distributed transaction, the com.ibm.db2.jdbc.DB2XADataSource class should be used when defining IDS data sources within the WebSphere Application Server.

For the detail information about how to configure the WebSphere Application Server with IDS, refer to WebSphere Application Server InfoCenter at:

`http://www.ibm.com/software/webservers/appserv/library.html`

Example of a distributed transaction that uses JTA methods

Distributed transactions typically involve multiple connections to the same data source or different data sources, which can include data sources from different manufacturers.

The best way to demonstrate distributed transactions is to contrast them with local transactions. With local transactions, a JDBC application makes changes to a database permanent and indicates the end of a unit of work in one of the following ways:
- By calling the Connection.commit or Connection.rollback methods after executing one or more SQL statements
- By calling the Connection.setAutoCommit(true) method at the beginning of the application to commit changes after every SQL statement

Figure 12-1 outlines code that executes local transactions.

```
con1.setAutoCommit(false);  // Set autocommit off
// execute some SQL
...
con1.commit();              // Commit the transaction
// execute some more SQL
...
con1.rollback();            // Roll back the transaction
con1.setAutoCommit(true);   // Enable commit after every SQL statement
...
// Execute some more SQL, which is automatically committed after
// every SQL statement.
```

Figure 12-1. Example of a local transaction

In contrast, applications that participate in distributed transactions cannot call the Connection.commit, Connection.rollback, or Connection.setAutoCommit(true) methods within the distributed transaction. With distributed transactions, the Connection.commit or Connection.rollback methods do not indicate transaction boundaries. Instead, your applications let the application server manage transaction boundaries.

Figure 12-2 demonstrates an application that uses distributed transactions. While the code in the example is running, the application server is also executing other EJBs that are part of this same distributed transaction. When all EJBs have called utx.commit(), the entire distributed transaction is committed by the application server. If any of the EJBs are unsuccessful, the application server rolls back all the work done by all EJBs that are associated with the distributed transaction.

```
javax.transaction.UserTransaction utx;
// Use the begin method on a UserTransaction object to indicate
// the beginning of a distributed transaction.
utx.begin();
...
// Execute some SQL with one Connection object.
// Do not call Connection methods commit or rollback.
...
// Use the commit method on the UserTransaction object to
// drive all transaction branches to commit and indicate
// the end of the distributed transaction.

utx.commit();
...
```

Figure 12-2. Example of a distributed transaction under an application server

Figure 12-3 illustrates a program that uses JTA methods to execute a distributed transaction. This program acts as the transaction manager and a transactional application. Two connections to two different data sources do SQL work under a single distributed transaction.

Figure 12-3. Example of a distributed transaction that uses the JTA

```
class XASample
{
  javax.sql.XADataSource xaDS1;
  javax.sql.XADataSource xaDS2;
  javax.sql.XAConnection xaconn1;
  javax.sql.XAConnection xaconn2;
  javax.transaction.xa.XAResource xares1;
  javax.transaction.xa.XAResource xares2;
  java.sql.Connection conn1;
  java.sql.Connection conn2;

  public static void main (String args []) throws java.sql.SQLException
  {
    XASample xat = new XASample();
    xat.runThis(args);
  }
  // As the transaction manager, this program supplies the global
  // transaction ID and the branch qualifier. The global
  // transaction ID and the branch qualifier must not be
  // equal to each other, and the combination must be unique for
  // this transaction manager.
  public void runThis(String[] args)
  {
    byte[] gtrid = new byte[] { 0x44, 0x11, 0x55, 0x66 };
    byte[] bqual = new byte[] { 0x00, 0x22, 0x00 };
    int rc1 = 0;
    int rc2 = 0;

    try
    {

      javax.naming.InitialContext context = new javax.naming.InitialContext();
```

```
        /*
          * Note that javax.sql.XADataSource is used instead of a specific
          * driver implementation such as com.ibm.db2.jcc.DB2XADataSource.
          */
        xaDS1 = (javax.sql.XADataSource)context.lookup("checkingAccounts");
        xaDS2 = (javax.sql.XADataSource)context.lookup("savingsAccounts");

        // The XADatasource contains the user ID and password.
        // Get the XAConnection object from each XADataSource
        xaconn1 = xaDS1.getXAConnection();
        xaconn2 = xaDS2.getXAConnection();

        // Get the java.sql.Connection object from each XAConnection
        conn1 = xaconn1.getConnection();
        conn2 = xaconn2.getConnection();

        // Get the XAResource object from each XAConnection
        xares1 = xaconn1.getXAResource();
        xares2 = xaconn2.getXAResource();
        // Create the Xid object for this distributed transaction.
        // This example uses the com.ibm.db2.jcc.DB2Xid implementation
        // of the Xid interface. This Xid can be used with any JDBC driver
        // that supports JTA.
        javax.transaction.xa.Xid xid1 =
                                 new com.ibm.db2.jcc.DB2Xid(100, gtrid, bqual);

        // Start the distributed transaction on the two connections.
        // The two connections do NOT need to be started and ended together.
        // They might be done in different threads, along with their SQL operations.
        xares1.start(xid1, javax.transaction.xa.XAResource.TMNOFLAGS);
        xares2.start(xid1, javax.transaction.xa.XAResource.TMNOFLAGS);
...
        // Do the SQL operations on connection 1.
        // Do the SQL operations on connection 2.
...
        // Now end the distributed transaction on the two connections.
        xares1.end(xid1, javax.transaction.xa.XAResource.TMSUCCESS);
        xares2.end(xid1, javax.transaction.xa.XAResource.TMSUCCESS);

        // If connection 2 work had been done in another thread,
        // a thread.join() call would be needed here to wait until the
        // connection 2 work is done.

        try
        { // Now prepare both branches of the distributed transaction.
          // Both branches must prepare successfully before changes
          // can be committed.
          // If the distributed transaction fails, an XAException is thrown.
          rc1 = xares1.prepare(xid1);
          if(rc1 == javax.transaction.xa.XAResource.XA_OK)
          { // Prepare was successful. Prepare the second connection.
            rc2 = xares2.prepare(xid1);
            if(rc2 == javax.transaction.xa.XAResource.XA_OK)
            { // Both connections prepared successfully and neither was read-only.
              xares1.commit(xid1, false);
              xares2.commit(xid1, false);
            }
            else if(rc2 == javax.transaction.xa.XAException.XA_RDONLY)
            { // The second connection is read-only, so just commit the
              // first connection.
              xares1.commit(xid1, false);
            }
          }
          else if(rc1 == javax.transaction.xa.XAException.XA_RDONLY)
          { // SQL for the first connection is read-only (such as a SELECT).
            // The prepare committed it. Prepare the second connection.
            rc2 = xares2.prepare(xid1);
```

```
      if(rc2 == javax.transaction.xa.XAResource.XA_OK)
      { // The first connection is read-only but the second is not.
        // Commit the second connection.
        xares2.commit(xid1, false);
      }
      else if(rc2 == javax.transaction.xa.XAException.XA_RDONLY)
      { // Both connections are read-only, and both already committed,
        // so there is nothing more to do.
      }
    }
  }      catch (javax.transaction.xa.XAException xae)
  { // Distributed transaction failed, so roll it back.
    // Report XAException on prepare/commit.
    System.out.println("Distributed transaction prepare/commit failed. " +
                       "Rolling it back.");
    System.out.println("XAException error code = " + xae.errorCode);
    System.out.println("XAException message = " + xae.getMessage());
    xae.printStackTrace();
    try
    {
      xares1.rollback(xid1);
    }
    catch (javax.transaction.xa.XAException xae1)
    { // Report failure of rollback.
      System.out.println("distributed Transaction rollback xares1 failed");
      System.out.println("XAException error code = " + xae1.errorCode);
      System.out.println("XAException message = " + xae1.getMessage());
    }
    try
    {
      xares2.rollback(xid1);
    }
    catch (javax.transaction.xa.XAException xae2)
    { // Report failure of rollback.
      System.out.println("distributed Transaction rollback xares2 failed");
      System.out.println("XAException error code = " + xae2.errorCode);
      System.out.println("XAException message = " + xae2.getMessage());
    }
  }

  try
  {
    conn1.close();
    xaconn1.close();
  }
  catch (Exception e)
  {
    System.out.println("Failed to close connection 1: " + e.toString());
    e.printStackTrace();
  }
  try
  {
    conn2.close();
    xaconn2.close();
  }
  catch (Exception e)
  {
    System.out.println("Failed to close connection 2: " + e.toString());
    e.printStackTrace();
  }
}
catch (java.sql.SQLException sqe)
{
  System.out.println("SQLException caught: " + sqe.getMessage());
  sqe.printStackTrace();
}
catch (javax.transaction.xa.XAException xae)
```

```
    {
      System.out.println("XA error is " + xae.getMessage());
      xae.printStackTrace();
    }
    catch (javax.naming.NamingException nme)
    {
      System.out.println(" Naming Exception: " + nme.getMessage());
    }
  }
}
```

Recommendation: For better performance, complete a distributed transaction before you start another distributed or local transaction.

Enterprise Java Beans

The Enterprise Java beans architecture is a component architecture for the development and deployment of component-based distributed business applications.

Applications that are written using the Enterprise Java beans architecture can be written once, and then deployed on any server platform that supports the Enterprise Java beans specification. Java 2 Platform, Enterprise Edition (J2EE) applications implement server-side business components using Enterprise Java beans (EJBs) that include session beans and entity beans.

Session beans represent business services and are not shared between users. Entity beans are multi-user, distributed transactional objects that represent persistent data. The transactional boundaries of a EJB application can be set by specifying either container-managed or bean-managed transactions.

The sample program `AccessEmployee.ear` uses Enterprise Java beans to implement a J2EE application to access a data source. You can find this sample in the `SQLLIB/samples/websphere` directory.

The EJB sample application provides two business services. One service allows the user to access information about an employee (which is stored in the EMPLOYEE table of the **sample** database) through that employee's employee number. The other service allows the user to retrieve a list of the employee numbers, so that the user can obtain an employee number to use for querying employee data.

The following sample uses EJBs to implement a J2EE application to access a data source. The sample utilizes the Model-View-Controller (MVC) architecture, which is a commonly-used GUI architecture. The JSP is used to implement the view (the presentation component). A servlet acts as the controller in the sample. It controls the workflow and delegates the user's request to the model, which is implemented using EJBs. The model component of the sample consists of two EJBs, one session bean and one entity bean. The container-managed persistence (CMP) bean, Employee, represents the distributed transactional objects that represent the persistent data in the EMPLOYEE table of the sample database. The term container-managed persistence means that the EJB container handles all database access required by the entity bean. The bean's code contains no database access (SQL) calls. As a result, the bean's code is not tied to a specific persistent storage mechanism (database). The session bean, AccessEmployee, acts as the Façade of the entity bean and provides provide a uniform client access strategy. This Façade design reduces the network traffic between the EJB client and the entity bean and is more efficient in distributed transactions than if the EJB client accesses the entity

bean directly. Access to the database server can be provided from the session bean or entity bean. The two services of the sample application demonstrate both approaches to accessing the database server. In the first service, the entity bean is used:

```
//=====================================================
// This method returns an employee's information by
// interacting with the entity bean located by the
// provided employee number
public EmployeeInfo getEmployeeInfo(String empNo)
throws java.rmi.RemoteException
}
Employee employee = null;
try
}
employee = employeeHome.findByPrimaryKey(new EmployeeKey(empNo));
EmployeeInfo empInfo = new EmployeeInfo(empNo);
//set the employee's information to the dependent value object
empInfo.setEmpno(employee.getEmpno());
empInfo.setFirstName (employee.getFirstName());
empInfo.setMidInit(employee.getMidInit());
empInfo.setLastName(employee.getLastName());
empInfo.setWorkDept(employee.getWorkDept());
empInfo.setPhoneNo(employee.getPhoneNo());
empInfo.setHireDate(employee.getHireDate());
empInfo.setJob(employee.getJob());
empInfo.setEdLevel(employee.getEdLevel());
empInfo.setSex(employee.getSex());
empInfo.setBirthDate(employee.getBirthDate());
empInfo.setSalary(employee.getSalary());
empInfo.setBonus(employee.getBonus());
empInfo.setComm(employee.getComm());
return empInfo;
}
catch (java.rmi.RemoteException rex)
{
......
```

In the second service, which displays employee numbers, the session bean, AccessEmployee, directly accesses the database table.

```
/=============================================
* Get the employee number list.
* @return Collection
*/
public Collection getEmpNoList()
{
ResultSet rs = null;
PreparedStatement ps = null;
Vector list = new Vector();
DataSource ds = null;
Connection con = null;
try
{
ds = getDataSource();
con = ds.getConnection();
String schema = getEnvProps(DBschema);
String query = "Select EMPNO from " + schema + ".EMPLOYEE";
ps = con.prepareStatement(query);
ps.executeQuery();
rs = ps.getResultSet();
EmployeeKey pk;
while (rs.next())
{
pk = new EmployeeKey();
pk.employeeId = rs.getString(1);
```

```
list.addElement(pk.employeeId);
}
rs.close();
return list;
```

Chapter 13. JDBC and SQLJ connection pooling support

Connection pooling is part of JDBC DataSource support, and is supported by the IBM Data Server Driver for JDBC and SQLJ.

The IBM Data Server Driver for JDBC and SQLJ provides a factory of pooled connections that are used by WebSphere Application Server or other application servers. The application server actually does the pooling. Connection pooling is completely transparent to a JDBC or SQLJ application.

Connection pooling is a framework for caching physical data source connections, which are equivalent to IBM Informix sessions. With connection pooling, session connections are reused. When JDBC reuses data source connections, the expensive operations that are required for the creation and subsequent closing of java.sql.Connection objects are minimized.

Without connection pooling, each java.sql.Connection object represents a single, session-level connection to the data source, which is not reused. When the application establishes a connection to a data source, IBM Informix creates a new physical connection to the data source. When the application calls the java.sql.Connection.close method, IBM Informix terminates the connection to the data source.

Connection pooling can be *homogeneous* or *heterogeneous*.

With homogeneous pooling, all Connection objects that come from a connection pool should have the same properties. The first logical Connection that is created with the DataSource has the properties that were defined for the DataSource. However, an application can change those properties. When a Connection is returned to the connection pool, an application server or a pooling module should reset the properties to their original values. However, an application server or pooling module might not reset the changed properties. The JDBC driver does not modify the properties. Therefore, depending on the application server or pool module design, a reused logical Connection might have the same properties as those that are defined for the DataSource or different properties.

With heterogeneous pooling, Connection objects with different properties can share the same connection pool.

Chapter 14. JDBC and SQLJ reference information

The IBM implementations of JDBC and SQLJ provide a number of application programming interfaces, properties, and commands for developing JDBC and SQLJ applications.

Data types that map to database data types in Java applications

To write efficient JDBC and SQLJ programs, you need to use the best mappings between Java data types and table column data types.

The following tables summarize the mappings of Java data types to JDBC and database data types for a DB2 Database for Linux, UNIX, and Windows, DB2 for z/OS, or IBM Informix system.

Data types for updating table columns

The following table summarizes the mappings of Java data types to database data types for PreparedStatement.setXXX or ResultSet.updateXXX methods in JDBC programs. When more than one Java data type is listed, the first data type is the recommended data type.

Table 14-1. Mappings of Java data types to database server data types for updating database tables

Java data type	Database data type
short, java.lang.Short	SMALLINT
short, java.lang.Short	BOOLEAN
boolean, java.lang.Boolean	BOOLEAN
int, java.lang.Integer	INTEGER
int, java.lang.Integer	SERIAL
long, java.lang.Long	INT8
long, java.lang.Long	BIGINT
long, java.lang.Long	SERIAL8
long, java.lang.Long	BIGSERIAL
float, java.lang.Float	SMALLFLOAT
double, java.lang.Double	FLOAT
java.math.BigDecimal	DECIMAL(*p*,*s*)[1]
java.math.BigDecimal	DECIMAL(*p*)[2]
java.math.BigDecimal	DECIMAL[3]
java.math.BigDecimal	MONEY(*p*,*s*)[1]
java.lang.String	CHAR(*n*)[4]
java.lang.String	NCHAR(*n*)[4]
java.lang.String	VARCHAR(*m*,*r*)[5]
java.lang.String	LVARCHAR(*m*,*r*)[6]
java.lang.String	NVARCHAR(*m*,*r*)[6]
java.lang.String	INTERVAL

Table 14-1. Mappings of Java data types to database server data types for updating database tables (continued)

Java data type	Database data type
java.lang.String	CLOB[7]
byte[]	BYTE
byte[]	BLOB[7]
java.sql.Blob	BLOB
java.sql.Clob	CLOB
java.lang.Clob	TEXT
java.sql.Date	DATE
java.sql.Time	DATETIME HOUR TO SECOND
java.sql.Timestamp	DATETIME YEAR TO FRACTION(5)
java.io.ByteArrayInputStream	BLOB
java.io.StringReader	CLOB
java.io.ByteArrayInputStream	CLOB

Notes:

1. *p* is the decimal precision and *s* is the scale of the table column.
2. For an ANSI-compliant database, *p* is the precision, and the scale is 0. For a database that is not ANSI-compliant, if you specify only *p*, the data type is DECIMAL floating point.
3. For an ANSI-compliant database, if you specify no parameters, the precision is 16 and the scale is 0. For a database that is not ANSI-compliant, if you specify no parameters, the data type is DECIMAL floating point.
4. *n*<=32767.
5. 0<=*r*<=*m*<=255.
6. 0<=*r*<=*m*<=32739.
7. This mapping is valid only if the database server can determine the data type of the column.

Data types for retrieval from table columns

The following table summarizes the mappings of DB2 or IBM Informix data types to Java data types for ResultSet.getXXX methods in JDBC programs, and for iterators in SQLJ programs. This table does not list Java numeric wrapper object types, which are retrieved using ResultSet.getObject.

Table 14-2. Mappings of database server data types to Java data types for retrieving data from database server tables

SQL data type	Recommended Java data type or Java object type	Other supported Java data types
SMALLINT	short	byte, int, long, float, double, java.math.BigDecimal, boolean, java.lang.String
BOOLEAN	boolean	short
INTEGER	int	short, byte, long, float, double, java.math.BigDecimal, boolean, java.lang.String
SERIAL	int	short, byte, long, float, double, java.math.BigDecimal, boolean, java.lang.String
INT8	long	int, short, byte, float, double, java.math.BigDecimal, boolean, java.lang.String

Table 14-2. Mappings of database server data types to Java data types for retrieving data from database server tables (continued)

SQL data type	Recommended Java data type or Java object type	Other supported Java data types
BIGINT	long	int, short, byte, float, double, java.math.BigDecimal, boolean, java.lang.String
SERIAL8	long	int, short, byte, float, double, java.math.BigDecimal, boolean, java.lang.String
BIGSERIAL	long	int, short, byte, float, double, java.math.BigDecimal, boolean, java.lang.String
DECIMAL(*p*,*s*)	java.math.BigDecimal	long, int, short, byte, float, double, boolean, java.lang.String
DECIMAL(*p*)	java.math.BigDecimal	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.lang.String
MONEY(*p*,*s*)	java.math.BigDecimal	long, int, short, byte, float, double, boolean, java.lang.String
SMALLFLOAT	float	long, int, short, byte, double, java.math.BigDecimal, boolean, java.lang.String
FLOAT	double	long, int, short, byte, float, java.math.BigDecimal, boolean, java.lang.String
CHAR(*n*)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
NCHAR(*n*)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
VARCHAR(*m*,*r*)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
LVARCHAR(*m*,*r*)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
NVARCHAR(*n*)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader

Table 14-2. Mappings of database server data types to Java data types for retrieving data from database server tables (continued)

SQL data type	Recommended Java data type or Java object type	Other supported Java data types
INTERVAL	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
BYTE	byte[]	None
CLOB(*n*)	java.sql.Clob	java.lang.String
TEXT	java.sql.Clob	java.lang.String
BLOB(*n*)	java.sql.Blob	byte[][1]
DATE	java.sql.Date	java.sql.String, java.sql.Timestamp
DATETIME HOUR TO SECOND	java.sql.Time	java.sql.String, java.sql.Timestamp
DATETIME YEAR TO FRACTION(5)	java.sql.Timestamp	java.sql.String, java.sql.Date, java.sql.Time, java.sql.Timestamp

Notes:

1. This mapping is valid only if the database server can determine the data type of the column.

Data types for calling stored procedures and user-defined functions

The following table summarizes mappings of Java data types to JDBC data types and DB2 or IBM Informix data types for calling user-defined function and stored procedure parameters. The mappings of Java data types to JDBC data types are for CallableStatement.registerOutParameter methods in JDBC programs. The mappings of Java data types to database server data types are for parameters in stored procedure or user-defined function invocations.

If more than one Java data type is listed in the following table, the first data type is the **recommended** data type.

Table 14-3. Mappings of Java, JDBC, and SQL data types for calling stored procedures and user-defined functions

Java data type	JDBC data type	SQL data type
boolean, java.lang.Boolean	BIT	BOOLEAN
boolean, java.lang.Boolean	BOOLEAN	BOOLEAN
byte[1], java.lang.Byte	TINYINT	SMALLINT
short, java.lang.Short	SMALLINT	SMALLINT
int, java.lang.Integer	INTEGER	INTEGER
int, java.lang.Integer	INTEGER	SERIAL
long, java.lang.Long	BIGINT	INT8
long, java.lang.Long	BIGINT	BIGINT
long, java.lang.Long	BIGINT	SERIAL8
long, java.lang.Long	BIGINT	BIGSERIAL
float, java.lang.Float	REAL	SMALLFLOAT
float, java.lang.Float	FLOAT	SMALLFLOAT

Table 14-3. Mappings of Java, JDBC, and SQL data types for calling stored procedures and user-defined functions (continued)

Java data type	JDBC data type	SQL data type
double, java.lang.Double	DOUBLE	FLOAT
java.math.BigDecimal	NUMERIC	DECIMAL
java.math.BigDecimal	DECIMAL	DECIMAL
java.math.BigDecimal	NUMERIC	MONEY
java.lang.String	CHAR	CHAR
java.lang.String	CHAR	INTERVAL
java.lang.String	CHAR	NCHAR
java.lang.String	VARCHAR	VARCHAR
java.lang.String	VARCHAR	NVARCHAR
java.lang.String	LONGVARCHAR	VARCHAR
java.lang.String	LONGVARCHAR	LVARCHAR
java.lang.String	VARCHAR	CLOB
java.lang.String	LONGVARCHAR	CLOB
java.lang.String	CLOB	CLOB
java.lang.String	CLOB	TEXT
byte[]	BINARY	BYTE
byte[]	VARBINARY	BYTE
byte[]	VARBINARY	BYTE
byte[]	LONGVARBINARY	BYTE
byte[]	LONGVARBINARY	BLOB[2]
java.sql.Date	DATE	DATE
java.sql.Time	TIME	DATETIME HOUR TO SECOND
java.sql.Timestamp	TIMESTAMP	DATETIME YEAR TO FRACTION(5)
java.sql.Blob	BLOB	BLOB
java.sql.Clob	CLOB	CLOB
java.io.ByteArrayInputStream	None	BLOB
java.io.StringReader	None	CLOB
java.io.ByteArrayInputStream	None	CLOB

Notes:

1. A stored procedure or user-defined function that is defined with a SMALLINT parameter can be invoked with a boolean or byte parameter. However, this is not recommended.
2. This mapping is valid only if the database server can determine the data type of the column.

IBM Informix SQL types with limited support

The SET, MULTISET, LIST, ROW, and UDT data types have limited support. You cannot retrieve data from columns with those data types, but you can use DatabaseMetaData methods to retrieve the column data types and type names. The following table lists the data types and type names that are returned from DatabaseMetaData.getColumns and `DatabaseMetaData.getTypeInfo`.

Table 14-4. Data types returned for DatabaseMetaData calls against SET, MULTISET, LIST, ROW, and UDT columns

Column data type	JDBC data type	Type name
SET	java.sql.Types.OTHER	set
MULTISET	java.sql.Types.OTHER	multiset
LIST	java.sql.Types.OTHER	list
ROW	java.sql.Types.STRUCTURE	row
UDT	java.sql.Types.JAVA_OBJECT	The fully qualified type name that was specified when the UDT was created.

Retrieval of special values from DECFLOAT columns in Java applications

Special handling is necessary if you retrieve values from DECFLOAT columns into java.math.BigDecimal variables, and the DECFLOAT columns contain the values NaN, Infinity, or -Infinity.

The recommended Java data type for retrieval of DECFLOAT column values is java.math.BigDecimal. However, if you receive SQL error code -4231 if you perform either of these operations:

- Retrieve the value NaN, Infinity, or -Infinity from a DECFLOAT column using the JDBC java.sql.ResultSet.getBigDecimal or java.sql.ResultSet.getObject method
- Retrieve the value NaN, Infinity, or -Infinity from a DECFLOAT column into a java.math.BigDecimal variable in an SQLJ clause of an SQLJ program

You can circumvent this restriction by testing for the -4231 error, and retrieving the special value using the java.sql.ResultSet.getDouble method.

Suppose that the following SQL statements were used to create and populate a table.

```
CREATE TABLE TEST.DECFLOAT_TEST
(
 INT_VAL INT,
 DECFLOAT_VAL DECFLOAT
);
INSERT INTO TEST.DECFLOAT_TEST (INT_VAL, DECFLOAT_VAL) VALUES (1, 123.456),
INSERT INTO TEST.DECFLOAT_TEST (INT_VAL, DECFLOAT_VAL) VALUES (2, INFINITY),
INSERT INTO TEST.DECFLOAT_TEST (INT_VAL, DECFLOAT_VAL) VALUES (3, -123.456),
INSERT INTO TEST.DECFLOAT_TEST (INT_VAL, DECFLOAT_VAL) VALUES (4, -INFINITY),
INSERT INTO TEST.DECFLOAT_TEST (INT_VAL, DECFLOAT_VAL) VALUES (5, NaN);
```

The following code retrieves the contents of the DECFLOAT column using the java.sql.ResultSet.getBigDecimal method. If retrieval fails because the column value is NaN, INFINITY, or -INFINITY, the program retrieves the value using the java.sql.ResultSet.getBigDouble method.

```
final static int DECFLOAT_SPECIALVALUE_ENCOUNTERED = -4231;
java.sql.Connection con =
  java.sql.DriverManager.getConnection("jdbc:db2://localhost:50000/sample"
    , "userid", "password");
java.sql.Statement stmt = con.createStatement();
java.sql.ResultSet rs = stmt.executeQuery(
 "SELECT INT_VAL, DECFLOAT_VAL FROM TEST.DECFLOAT_TEST ORDER BY INT_VAL");
int i = 0;
while (rs.next()) {
```

```
try {
 System.out.println("\nRow " + ++i);
 System.out.println("INT_VAL      = " + rs.getInt(1));
 System.out.println("DECFLOAT_VAL = " + rs.getBigDecimal(2));
}
catch (java.sql.SQLException e) {
 System.out.println("Catched SQLException" + e.getMessage());
 if (e.getErrorCode() == DECFLOAT_SPECIALVALUE_ENCOUNTERED) {
 // getBigDecimal failed because the retrieved value is NaN,
 // INFINITY, or -INFINITY, so retry with getDouble.
    double d = rs.getDouble(2);
    if (d == Double.POSITIVE_INFINITY) {
      System.out.println("DECFLOAT_VAL = +INFINITY");
    } else if (d == Double.NEGATIVE_INFINITY) {
      System.out.println("DECFLOAT_VAL = -INFINITY");
    } else if (d == Double.NaN) {
      System.out.println("DECFLOAT_VAL = NaN");
    } else {
      System.out.println("DECFLOAT_VAL = " + d);
    }
 } else {
    e.printStackTrace();
}
```

Properties for the IBM Data Server Driver for JDBC and SQLJ

IBM Data Server Driver for JDBC and SQLJ properties define how the connection to a particular data source should be made. Most properties can be set for a DataSource object or for a Connection object.

Methods for setting the properties

Properties can be set in one of the following ways:

- Using setXXX methods, where `XXX` is the unqualified property name, with the first character capitalized.

  Properties are applicable to the following IBM Data Server Driver for JDBC and SQLJ-specific implementations that inherit from com.ibm.db2.jcc.DB2BaseDataSource:
  - com.ibm.db2.jcc.DB2SimpleDataSource
  - com.ibm.db2.jcc.DB2ConnectionPoolDataSource
  - com.ibm.db2.jcc.DB2XADataSource
- In a java.util.Properties value in the *info* parameter of a DriverManager.getConnection call.
- In a `java.lang.String` value in the *url* parameter of a DriverManager.getConnection call.

  Some properties with an int data type have predefined constant field values. You must resolve constant field values to their integer values before you can use those values in the *url* parameter. For example, you cannot use com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL in a *url* parameter. However, you can build a URL string that includes com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL, and assign the URL string to a String variable. Then you can use the String variable in the *url* parameter:

```
String url =
 "jdbc:ids://sysmvs1.stl.ibm.com:5021" +
 "user=dbadm;password=dbadm;" +
 "traceLevel=" +
 (com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL) + ";";
```

```
Connection con =
  java.sql.DriverManager.getConnection(url);
```

Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products

Most of the IBM Data Server Driver for JDBC and SQLJ properties apply to all database products that the driver supports.

Unless otherwise noted, all properties are in com.ibm.db2.jcc.DB2BaseDataSource.

Those properties are:

affinityFailbackInterval
Specifies the length of the interval, in seconds, that the IBM Data Server Driver for JDBC and SQLJ waits between attempts to fail back an existing connection to the primary server. A value that is less than or equal to 0 means that the connection does not fail back. The default is `DB2BaseDataSource.NOT_SET` (0).

Attempts to fail back connections to the primary server are made at transaction boundaries, after the specified interval elapses.

affinityFailbackInterval is used only if the values of properties enableSeamlessFailover and enableClientAffinitiesList are `DB2BaseDataSource.YES` (1).

affinityFailbackInterval applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

allowNextOnExhaustedResultSet
Specifies how the IBM Data Server Driver for JDBC and SQLJ handles a ResultSet.next() call for a forward-only cursor that is positioned after the last row of the ResultSet. The data type of this property is int.

Possible values are:

DB2BaseDataSource.YES (1)
For a ResultSet that is defined as TYPE_FORWARD_ONLY, ResultSet.next() returns `false` if the cursor was previously positioned after the last row of the ResultSet. `false` is returned, regardless of whether the cursor is open or closed.

DB2BaseDataSource.NO (2)
For a ResultSet that is defined as TYPE_FORWARD_ONLY, when ResultSet.next() is called, and the cursor was previously positioned after the last row of the ResultSet, the driver throws a java.sql.SQLException with error text "Invalid operation: result set is closed." This is the default.

allowNullResultSetForExecuteQuery
Specifies whether the IBM Data Server Driver for JDBC and SQLJ returns null when Statement.executeQuery, PreparedStatement.executeQuery, or CallableStatement.executeQuery is used to execute a CALL statement for a stored procedure that does not return any result sets.

Possible values are:

DB2BaseDataSource.NOT_SET (0)
The behavior is the same as for `DB2BaseDataSource.NO`.

DB2BaseDataSource.YES (1)
The IBM Data Server Driver for JDBC and SQLJ returns null when

Statement.executeQuery, PreparedStatement.executeQuery, or CallableStatement.executeQuery is used to execute a CALL statement for a stored procedure that does not return any result sets. This behavior does not conform to the JDBC standard.

DB2BaseDataSource.NO (2)
The IBM Data Server Driver for JDBC and SQLJ throws an SQLException when Statement.executeQuery, PreparedStatement.executeQuery, or CallableStatement.executeQuery is used to execute a CALL statement for a stored procedure that does not return any result sets. This behavior conforms to the JDBC standard.

atomicMultiRowInsert
Specifies whether batch operations that use PreparedStatement methods to modify a table are atomic or non-atomic. The data type of this property is int.

For connections to DB2 for z/OS, this property applies only to batch INSERT operations.

For connections to DB2 Database for Linux, UNIX, and Windows or IBM Informix, this property applies to batch INSERT, MERGE, UPDATE or DELETE operations.

Possible values are:

DB2BaseDataSource.YES (1)
Batch operations are atomic. Insertion of all rows in the batch is considered to be a single operation. If insertion of a single row fails, the entire operation fails with a BatchUpdateException. Use of a batch statement that returns auto-generated keys fails with a BatchUpdateException.

If atomicMultiRowInsert is set to `DB2BaseDataSource.YES` (1):

- Execution of statements in a heterogeneous batch is not allowed.
- If the target data source is DB2 for z/OS the following operations are not allowed:
  - Insertion of more than 32767 rows in a batch results in a BatchUpdateException.
  - Calling more than one of the following methods against the same parameter in different rows results in a BatchUpdateException:
    - PreparedStatement.setAsciiStream
    - PreparedStatement.setCharacterStream
    - PreparedStatement.setUnicodeStream

DB2BaseDataSource.NO (2)
Batch inserts are non-atomic. Insertion of each row is considered to be a separate execution. Information on the success of each insert operation is provided by the int[] array that is returned by Statement.executeBatch.

DB2BaseDataSource.NOT_SET (0)
Batch inserts are non-atomic. Insertion of each row is considered to be a separate execution. Information on the success of each insert operation is provided by the int[] array that is returned by Statement.executeBatch. This is the default.

blockingReadConnectionTimeout
The amount of time in seconds before a connection socket read times out. This property applies only to IBM Data Server Driver for JDBC and SQLJ type 4

connectivity, and affects all requests that are sent to the data source after a connection is successfully established. The default is 0. A value of 0 means that there is no timeout.

clientDebugInfo
Specifies a value for the CLIENT DEBUGINFO connection attribute, to notify the data server that stored procedures and user-defined functions that are using the connection are running in debug mode. CLIENT DEBUGINFO is used by the DB2 Unified Debugger. The data type of this property is String. The maximum length is 254 bytes.

This property applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

clientRerouteAlternateServerName
Specifies one or more server names for client reroute. The data type of this property is String.

When enableClientAffinitiesList=DB2BaseDataSource.YES (1), clientRerouteAlternateServerName must contain the name of the primary server as well as alternate server names. The server that is identified by serverName and portNumber is the primary server. That server name must appear at the beginning of the clientRerouteAlternateServerName list.

If more than one server name is specified, delimit the server names with commas (,) or spaces. The number of values that is specified for clientRerouteAlternateServerName must match the number of values that is specified for clientRerouteAlternatePortNumber.

clientRerouteAlternateServerName applies to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows and IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

clientRerouteAlternatePortNumber
Specifies one or more port numbers for client reroute. The data type of this property is String.

When enableClientAffinitiesList=DB2BaseDataSource.YES (1), clientRerouteAlternatePortNumber must contain the port number for the primary server as well as port numbers for alternate servers. The server that is identified by serverName and portNumber is the primary server. That port number must appear at the beginning of the clientRerouteAlternatePortNumber list.

If more than one port number is specified, delimit the port numbers with commas (,) or spaces. The number of values that is specified for clientRerouteAlternatePortNumber must match the number of values that is specified for clientRerouteAlternateServerName.

clientRerouteAlternatePortNumber applies to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows and IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

clientRerouteServerListJNDIName
Identifies a JNDI reference to a DB2ClientRerouteServerList instance in a JNDI repository of reroute server information. clientRerouteServerListJNDIName applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, and to connections that are established through the DataSource interface.

If the value of clientRerouteServerListJNDIName is not null, clientRerouteServerListJNDIName provides the following functions:
- Allows information about reroute servers to persist across JVMs

- Provides an alternate server location if the first connection to the data source fails

clientRerouteServerListJNDIContext
Specifies the JNDI context that is used for binding and lookup of the DB2ClientRerouteServerList instance. clientRerouteServerListJNDIContext applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, and to connections that are established through the DataSource interface.

If clientRerouteServerListJNDIContext is not set, the IBM Data Server Driver for JDBC and SQLJ creates an initial context using system properties or the jndi.properties file.

clientRerouteServerListJNDIContext can be set **only** by using the following method:

```
public void setClientRerouteServerListJNDIContext(javax.naming.Context registry)
```

connectionCloseWithInFlightTransaction
Specifies whether the IBM Data Server Driver for JDBC and SQLJ throws an SQLException or rolls back a transaction without throwing an SQLException when a connection is closed in the middle of the transaction. Possible values are:

DB2BaseDataSource.NOT_SET (0)
The behavior is the same as for DB2BaseDataSource.CONNECTION_CLOSE_WITH_EXCEPTION.

DB2BaseDataSource.CONNECTION_CLOSE_WITH_EXCEPTION (1)
When a connection is closed in the middle of a transaction, an SQLException with error -4471 is thrown.

DB2BaseDataSource.CONNECTION_CLOSE_WITH_ROLLBACK (2)
When a connection is closed in the middle of a transaction, the transaction is rolled back, and no SQLException is thrown.

databaseName
Specifies the name for the data source. This name is used as the *database* portion of the connection URL. The name depends on whether IBM Data Server Driver for JDBC and SQLJ type 4 connectivity or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity is used.

For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity:

- If the connection is to a DB2 for z/OS server, the databaseName value is the DB2 location name that is defined during installation. All characters in this value must be uppercase characters. You can determine the location name by executing the following SQL statement on the server:

  ```
  SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
  ```

- If the connection is to a DB2 Database for Linux, UNIX, and Windows server, the databaseName value is the database name that is defined during installation.
- If the connection is to an IBM Informix server, *database* is the database name. The name is case-insensitive. The server converts the name to lowercase.
- If the connection is to an IBM Cloudscape server, the databaseName value is the fully-qualified name of the file that contains the database. This name must be enclosed in double quotation marks ("). For example:

  ```
  "c:/databases/testdb"
  ```

If this property is not set, connections are made to the local site.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity:

decimalSeparator
Specifies the decimal separator for input and output, for decimal, floating point, or decimal floating-point data values. The data type of this property is int.

If the value of the sendDataAsIs property is `true`, decimalSeparator affects only output values.

Possible values are:

DB2BaseDataSource.DECIMAL_SEPARATOR_NOT_SET (0)
A period is used as the decimal separator. This is the default.

DB2BaseDataSource.DECIMAL_SEPARATOR_PERIOD (1)
A period is used as the decimal separator.

DB2BaseDataSource.DECIMAL_SEPARATOR_COMMA (2)
A comma is used as the decimal separator.

When DECIMAL_SEPARATOR_COMMA is set, the result of ResultSet.getString on a decimal, floating point, or decimal floating-point value has a comma as a separator. However, if the toString method is executed on a value that is retrieved with a ResultSet.getXXX method that returns a decimal, floating point, or decimal floating-point value, the result has a decimal point as the decimal separator.

decimalStringFormat
Specifies the string format for data that is retrieved from a DECIMAL or DECFLOAT column when the SDK for Java is Version 1.5 or later. The data type of this property is int. Possible values are:

DB2BaseDataSource.DECIMAL_STRING_FORMAT_NOT_SET (0)
The IBM Data Server Driver for JDBC and SQLJ returns decimal values in the format that the java.math.BigDecimal.toString method returns them. This is the default.

For example, the value 0.0000000004 is returned as 4E-10.

DB2BaseDataSource.DECIMAL_STRING_FORMAT_TO_STRING (1)
The IBM Data Server Driver for JDBC and SQLJ returns decimal values in the format that the java.math.BigDecimal.toString method returns them.

For example, the value 0.0000000004 is returned as 4E-10.

DB2BaseDataSource.DECIMAL_STRING_FORMAT_TO_PLAIN_STRING (2)
The IBM Data Server Driver for JDBC and SQLJ returns decimal values in the format that the java.math.BigDecimal.toPlainString method returns them.

For example, the value 0.0000000004 is returned as 0.0000000004.

This property has no effect for earlier versions of the SDK for Java. For those versions, the IBM Data Server Driver for JDBC and SQLJ returns decimal values in the format that the java.math.BigDecimal.toString method returns them.

defaultIsolationLevel
Specifies the default transaction isolation level for new connections. The data type of this property is int. When defaultIsolationLevel is set on a DataSource, all connections that are created from that DataSource have the default isolation level that is specified by defaultIsolationLevel.

For DB2 data sources, the default is `java.sql.Connection.TRANSACTION_READ_COMMITTED`.

For IBM Informix databases, the default depends on the type of data source. The following table shows the defaults.

Table 14-5. Default isolation levels for IBM Informix databases

Type of data source	Default isolation level
ANSI-compliant database with logging	java.sql.Connection.TRANSACTION_SERIALIZABLE
Database without logging	java.sql.Connection.TRANSACTION_READ_UNCOMMITTED
Non-ANSI-compliant database with logging	java.sql.Connection.TRANSACTION_READ_COMMITTED

deferPrepares

Specifies whether invocation of the Connection.prepareStatement method results in immediate preparation of an SQL statement on the data source, or whether statement preparation is deferred until the PreparedStatement.execute method is executed. The data type of this property is boolean.

deferPrepares is supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows, and for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Possible values are:

true Statement preparation on the data source does not occur until the PreparedStatement.execute method is executed. This is the default.

false Statement preparation on the data source occurs when the Connection.prepareStatement method is executed.

Deferring prepare operations can reduce network delays. However, if you defer prepare operations, you need to ensure that input data types match table column types.

description

A description of the data source. The data type of this property is String.

downgradeHoldCursorsUnderXa

Specifies whether cursors that are defined WITH HOLD can be opened under XA connections.

downgradeHoldCursorsUnderXa applies to:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS servers.
- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows servers.

The default is `false`, which means that a cursor that is defined WITH HOLD cannot be opened under an XA connection. An exception is thrown when an attempt is made to open that cursor.

If downgradeHoldCursorsUnderXa is set to `true`, a cursor that is defined WITH HOLD can be opened under an XA connection. However, the cursor has the following restrictions:

- When the cursor is opened under an XA connection, the cursor does not have WITH HOLD behavior. The cursor is closed at XA End.

- A cursor that is open before XA Start on a local transaction is closed at XA Start.

driverType
For the DataSource interface, determines which driver to use for connections. The data type of this property is int. Valid values are 2 or 4. 2 is the default.

enableClientAffinitiesList
Specifies whether the IBM Data Server Driver for JDBC and SQLJ enables client affinities for cascaded failover support. The data type of this property is int. Possible values are:

DB2BaseDataSource.YES (1)
The IBM Data Server Driver for JDBC and SQLJ enables client affinities for cascaded failover support. This means that only servers that are specified in the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties are retried. The driver does not attempt to reconnect to any other servers.

For example, suppose that clientRerouteAlternateServerName contains the following string:

```
host1,host2,host3
```

Also suppose that clientRerouteAlternatePortNumber contains the following string:

```
port1,port2,port3
```

When client affinities are enabled, the retry order is:
1. host1:port1
2. host2:port2
3. host3:port3

DB2BaseDataSource.NO (2)
The IBM Data Server Driver for JDBC and SQLJ does not enable client affinities for cascaded failover support.

DB2BaseDataSource.NOT_SET (0)
The IBM Data Server Driver for JDBC and SQLJ does not enable client affinities for cascaded failover support. This is the default.

The effect of the maxRetriesForClientReroute and retryIntervalForClientReroute properties differs depending on whether enableClientAffinitiesList is enabled.

This property applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

enableNamedParameterMarkers
Specifies whether support for named parameter markers is enabled in the IBM Data Server Driver for JDBC and SQLJ. The data type of this property is int. Possible values are:

DB2BaseDataSource.YES (1)
Named parameter marker support is enabled in the IBM Data Server Driver for JDBC and SQLJ.

DB2BaseDataSource.NO (2)
Named parameter marker support is not enabled in the IBM Data Server Driver for JDBC and SQLJ.

The driver sends an SQL statement with named parameter markers to the target data source without modification. The success or failure of the statement depends on a number of factors, including the following ones:

- Whether the target data source supports named parameter markers
- Whether the deferPrepares property value is true of false
- Whether the sendDataAsIs property value is true of false

Recommendation: To avoid unexpected behavior in an application that uses named parameter markers, set enableNamedParameterMarkers to YES.

DB2BaseDataSource.NOT_SET (0)
The behavior is the same as the behavior for DB2BaseDataSource.NO (2). This is the default.

enableSeamlessFailover
Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses seamless failover for client reroute. The data type of this property is int.

For connections to DB2 for z/OS, if enableSysplexWLB is set to true, enableSeamlessFailover has no effect. The IBM Data Server Driver for JDBC and SQLJ uses seamless failover regardless of the enableSeamlessFailover setting.

Possible values of enableSeamlessFailover are:

DB2BaseDataSource.YES (1)
The IBM Data Server Driver for JDBC and SQLJ uses seamless failover. This means that the driver does not throw an SQLException with error code -4498 after a failed connection has been successfully re-established if the following conditions are true:

- The connection was not being used for a transaction at the time the failure occurred.
- There are no outstanding global resources, such as global temporary tables or open, held cursors, or connection states that prevent a seamless failover to another server.

When seamless failover occurs, after the connection to a new data source has been established, the driver re-issues the SQL statement that was being processed when the original connection failed.

Recommendation: Set the queryCloseImplicit property to DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_NO (2) when you set enableSeamlessFailover to DB2BaseDataSource.YES, if the application uses held cursors.

DB2BaseDataSource.NO (2)
The IBM Data Server Driver for JDBC and SQLJ does not use seamless failover.

When this setting is in effect, if a server goes down, the driver tries to fail back or fail over to an alternate server. If failover or failback is successful, the driver throws an SQLException with error code -4498, which indicates that a connection failed but was successfully reestablished. An SQLException with error code -4498 informs the application that it should retry the transaction during which the connection failure occurred. If the driver cannot reestablish a connection, it throws an SQLException with error code -4499.

DB2BaseDataSource.NOT_SET (0)
The IBM Data Server Driver for JDBC and SQLJ does not use seamless failover. This is the default.

enableSysplexWLB
Indicates whether the Sysplex workload balancing function of the IBM Data Server Driver for JDBC and SQLJ is enabled. The data type of enableSysplexWLB is boolean. The default is `false`.

enablSysplexWLB applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

fetchSize
Specifies the default fetch size for ResultSet objects that are generated from Statement objects. The data type of this property is int.

The fetchSize default can be overridden by the Statement.setFetchSize method. The fetchSize property does not affect Statement objects that already exist when fetchSize is set.

Possible values of fetchSize are:

0 or *positive-integer*
The default *fetchSize* value for newly created Statement objects. If the fetchSize property value is invalid, the IBM Data Server Driver for JDBC and SQLJ sets the default *fetchSize* value to 0.

DB2BaseDataSource.FETCHSIZE_NOT_SET (-1)
Indicates that the default *fetchSize* value for Statement objects is 0. This is the property default.

The fetchSize property differs from the queryDataSize property. fetchSize affects the number of rows that are returned, and queryDataSize affects the number of bytes that are returned.

floatingPointStringFormat
Specifies the format for data that is retrieved from a DOUBLE, FLOAT, or REAL column with the ResultSet.getString method. The data type of this property is int. Possible values are:

DB2BaseDataSource.NOT_SET (0)
The IBM Data Server Driver for JDBC and SQLJ returns double-precision floating point values in the string format that the java.lang.String.valueOf(double) method returns them. The IBM Data Server Driver for JDBC and SQLJ returns single-precision floating point values in the string format that the java.lang.String.valueOf(float) method returns them. This is the default.

For example, suppose that the value 71256.789 is retrieved from a DOUBLE column. If floatingPointStringFormat is not set, the string format of the retrieved value is 71256.789. If the value 71256.789 is retrieved from a REAL column, the string format of the retrieved value is 71256.79.

DB2BaseDataSource.JCC_DRIVER_FLOATING_POINT_STRING_FORMAT (1)
The IBM Data Server Driver for JDBC and SQLJ returns double-precision floating point values in the string format that the java.lang.String.valueOf(double) method returns them. The IBM Data Server Driver for JDBC and SQLJ returns single-precision floating point values in the string format that the java.lang.String.valueOf(float) method returns them. This is the default.

For example, suppose that the value 71256.789 is retrieved from a DOUBLE column. If floatingPointStringFormat is DB2BaseDataSource.JCC_DRIVER_FLOATING_POINT_STRING_FORMAT, the string format of the retrieved value is 71256.789. If the value 71256.789 is retrieved from a REAL column, the string format of the retrieved value is 71256.79.

DB2BaseDataSource.LUW_TYPE2_DRIVER_FLOATING_POINT_STRING_FORMAT (2)
The IBM Data Server Driver for JDBC and SQLJ returns DOUBLE, FLOAT, or REAL values in the same format that the DB2 JDBC Type 2 Driver for Linux, UNIX, and Windows returns them.

For example, suppose that the value 71256.789 is retrieved from a DOUBLE column. If floatingPointStringFormat is DB2BaseDataSource.LUW_TYPE2_DRIVER_FLOATING_POINT_STRING_FORMAT, the string format of the retrieved value is 7.12567890000000E+004. If the value 71256.789 is retrieved from a REAL column, the string format of the retrieved value is 7.125679E+04.

fullyMaterializeLobData
Indicates whether the driver retrieves LOB locators for FETCH operations. The data type of this property is boolean.

The effect of fullyMaterializeLobData depends on whether the data source supports progressive streaming, which is also known as dynamic data format:

- If the data source does not support progressive streaming:

  If the value of fullyMaterializeLobData is `true`, LOB data is fully materialized within the JDBC driver when a row is fetched. If the value is `false`, LOB data is streamed. The driver uses locators internally to retrieve LOB data in chunks on an as-needed basis It is highly recommended that you set this value to `false` when you retrieve LOBs that contain large amounts of data. The default is `true`.
- If the data source supports progressive streaming:

  The JDBC driver ignores the value of fullyMaterializeLobData if the progressiveStreaming property is set to `DB2BaseDataSource.YES` or `DB2BaseDataSource.NOT_SET`.

This property has no effect on stored procedure parameters or on LOBs that are fetched using scrollable cursors. LOB stored procedure parameters are always fully materialized. LOBs that are fetched using scrollable cursors use LOB locators if progressive streaming is not in effect.

interruptProcessingMode
Specifies the behavior of the IBM Data Server Driver for JDBC and SQLJ when an application executes the Statement.cancel method. Possible values are:

DB2BaseDataSource.INTERRUPT_PROCESSING_MODE_DISABLED (0)
Interrupt processing is disabled. When an application executes Statement.cancel, the IBM Data Server Driver for JDBC and SQLJ does nothing.

DB2BaseDataSource.INTERRUPT_PROCESSING_MODE_STATEMENT_CANCEL (1)
When an application executes Statement.cancel, the IBM Data Server Driver for JDBC and SQLJ cancels the currently executing statement, if the data server supports interrupt processing. If the data server does not support interrupt processing, the IBM Data Server Driver for JDBC and SQLJ throws an SQLException that indicates that the feature is not

supported.
INTERRUPT_PROCESSING_MODE_STATEMENT_CANCEL is the default.

For DB2 Database for Linux, UNIX, and Windows clients, when interruptProcessingMode is set to INTERRUPT_PROCESSING_MODE_STATEMENT_CANCEL, the DB2 Connect setting for INTERRUPT_ENABLED and the DB2 registry variable setting for DB2CONNECT_DISCONNECT_ON_INTERRUPT override this value.

DB2BaseDataSource.INTERRUPT_PROCESSING_MODE_CLOSE_SOCKET (2)
When an application executes Statement.cancel, the IBM Data Server Driver for JDBC and SQLJ performs one of the following actions:

- If automatic client reroute or client affinities is not enabled, the IBM Data Server Driver for JDBC and SQLJ drops the underlying socket, closes the connection, and throws an SQLException that indicates that the application is being disconnected from the data server. Any subsequent operations that are invoked on any Statement objects that are created from the same connection receive an SQLException that indicates that the connection is closed.
- If automatic client reroute or client affinities is enabled, the IBM Data Server Driver for JDBC and SQLJ drops the underlying socket, closes the connection, and then attempts to re-establish the connection. If re-connection is successful, the driver throws an SQLException that indicates that the connection was re-established. the driver does not re-execute any SQL statements, even if the enableSeamlessFailover property is set to DB2BaseDataSource.YES.

loginTimeout
The maximum time in seconds to wait for a connection to a data source. After the number of seconds that are specified by loginTimeout have elapsed, the driver closes the connection to the data source. The data type of this property is int. The default is 0. A value of 0 means that the timeout value is the default system timeout value. This property is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

logWriter
The character output stream to which all logging and trace messages for the DataSource object are printed. The data type of this property is java.io.PrinterWriter. The default value is null, which means that no logging or tracing for the DataSource is output.

maxRetriesForClientReroute
During automatic client reroute, limit the number of retries if the primary connection to the data source fails.

The data type of this property is int.

The IBM Data Server Driver for JDBC and SQLJ uses the maxRetriesForClientReroute property only if the retryIntervalForClientReroute property is also set.

If the enableClientAffinitiesList is set to DB2BaseDataSource.NO (2), an attempt to connect to the primary server and alternate servers counts as one retry. If enableClientAffinitiesList is set to DB2BaseDataSource.YES (1), each server that is specified by the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber values is retried the number of times that is specified by maxRetriesForClientReroute.

The default value for maxRetriesForClientReroute is 0 if enableClientAffinitiesList is DB2BaseDataSource.NO (2), or 3 if enableClientAffinitiesList is DB2BaseDataSource.YES (1).

If the value of maxRetriesForClientReroute is 0, client reroute processing does not occur.

password
The password to use for establishing connections. The data type of this property is String. When you use the DataSource interface to establish a connection, you can override this property value by invoking this form of the DataSource.getConnection method:

```
getConnection(user, password);
```

portNumber
The port number where the DRDA server is listening for requests. The data type of this property is int.

progressiveStreaming
Specifies whether the JDBC driver uses progressive streaming when progressive streaming is supported on the data source.

DB2 for z/OS Version 9.1 and later supports progressive streaming for LOBs and XML objects. DB2 Database for Linux, UNIX, and Windows Version 9.5 and later, and IBM Informix Version 11.50 and later support progressive streaming for LOBs.

With progressive streaming, also known as dynamic data format, the data source dynamically determines the most efficient mode in which to return LOB or XML data, based on the size of the LOBs or XML objects. The value of the streamBufferSize parameter determines whether the data is materialized when it is returned.

The data type of progressiveStreaming is int. Valid values are `DB2BaseDataSource.YES` (1) and `DB2BaseDataSource.NO` (2). If the progressiveStreaming property is not specified, the progressiveStreaming value is `DB2BaseDataSource.NOT_SET` (0).

If the connection is to a data source that supports progressive streaming, and the value of progressiveStreaming is `DB2BaseDataSource.YES` or `DB2BaseDataSource.NOT_SET`, the JDBC driver uses progressive streaming to return LOBs and XML data.

If the value of progressiveStreaming is `DB2BaseDataSource.NO`, or the data source does not support progressive streaming, the way in which the JDBC driver returns LOB or XML data depends on the value of the fullyMaterializeLobData property.

queryCloseImplicit
Specifies whether cursors are closed immediately after all rows are fetched. queryCloseImplicit applies only to connections to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS Version 8 or later, and IBM Data Server Driver for JDBC and SQLJ type 4 connectivity or IBM Data Server Driver for JDBC and SQLJ type 2 connectivityDB2 Database for Linux, UNIX, and Windows Version 9.7 or later. Possible values are:

DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_YES (1)
Close cursors immediately after all rows are fetched.

A value of `DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_YES` can provide better performance because this setting results in less network traffic.

DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_NO (2)
Do not close cursors immediately after all rows are fetched.

DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_COMMIT (3)
Perform these actions:

- Implicitly close the cursor after all rows are fetched.
- If the application is in autocommit mode, implicitly send a commit request to the data source for the current unit of work.

Important: When this value is set, there might be impacts on other resources, just as an explicit commit operation might impact other resources. For example, other non-held cursors are closed, LOB locators go out of scope, progressive references are reset, and scrollable cursors lose their position.

Restriction: The following restrictions apply to QUERY_CLOSE_IMPLICIT_COMMIT behavior:

- This behavior applies only to SELECT statements that are issued by the application. It does not apply to SELECT statements that are generated by the IBM Data Server Driver for JDBC and SQLJ.
- If QUERY_CLOSE_IMPLICIT_COMMIT is set, and the application is not in autocommit mode, the driver uses the default behavior (QUERY_CLOSE_IMPLICIT_NOT_SET behavior). If QUERY_CLOSE_IMPLICIT_COMMIT is the default behavior, the driver uses QUERY_CLOSE_IMPLICIT_YES behavior.
- If QUERY_CLOSE_IMPLICIT_COMMIT is set, and the data source does not support QUERY_CLOSE_IMPLICIT_COMMIT behavior, the driver uses QUERY_CLOSE_IMPLICIT_YES behavior.
- This behavior is not supported for batched statements.
- This behavior is supported on an XA Connection only when the connection is in a local transaction.

DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_NOT_SET (0)
This is the default. The following table describes the behavior for a connection to each type of data source.

Data source	Version	Data sharing environment	Behavior
DB2 for z/OS	Version 10	Data sharing or non-data sharing	QUERY_CLOSE_IMPLICIT_COMMIT
DB2 for z/OS	Version 9 with APAR PK68746	Non-data sharing, or in a data sharing group but not in coexistence mode with Version 8 members	QUERY_CLOSE_IMPLICIT_COMMIT
DB2 for z/OS	Version 9 without APAR PK68746	Non-data sharing, or in a data sharing group but not in coexistence mode with Version 8 members	QUERY_CLOSE_IMPLICIT_YES
DB2 for z/OS	Version 9 with APAR PK68746	In a data sharing group in coexistence mode with Version 8 members	QUERY_CLOSE_IMPLICIT_COMMIT
DB2 for z/OS	Version 9 without APAR PK68746	In a data sharing group in coexistence mode with Version 8 members	QUERY_CLOSE_IMPLICIT_YES

Data source	Version	Data sharing environment	Behavior
DB2 for z/OS	Version 8 with or without APAR PK68746		QUERY_CLOSE_IMPLICIT_YES
DB2 Database for Linux, UNIX, and Windows	Version 9.7		QUERY_CLOSE_IMPLICIT_YES

queryDataSize

Specifies a hint that is used to control the amount of query data, in bytes, that is returned from the data source on each fetch operation. This value can be used to optimize the application by controlling the number of trips to the data source that are required to retrieve data.

Use of a larger value for queryDataSize can result in less network traffic, which can result in better performance. For example, if the result set size is 50 KB, and the value of queryDataSize is 32767 (32KB), two trips to the database server are required to retrieve the result set. However, if queryDataSize is set to 65535 (64 KB), only one trip to the data source is required to retrieve the result set.

The following table lists minimum, maximum, and default values of queryDataSize for each data source.

Table 14-6. Default, minimum, and maximum values of queryDataSize

Data source	Product Version	Default	Minimum	Maximum	Valid values
DB2 Database for Linux, UNIX, and Windows	All	32767	4096	262143	4096-32767, 98303, 131071, 163839, 196607, 229375, 262143[1]
IBM Informix	All	32767	4096	10485760	4096-10485760
DB2 for i	V5R4	32767	4096	65535	4096-65535
	V6R1	32767	4096	262143	4096-65535, 98303, 131071, 163839, 196607, 229375, 262143[1]
DB2 for z/OS	Version 8 (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	32767	32767	32767	32767
	Version 9 (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	32767	32767	65535	32767, 65535
	Version 10 (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	32767	32767	262143	32767, 65535, 98303, 131071, 163839, 196607, 229375, 262143[1]

Table 14-6. Default, minimum, and maximum values of queryDataSize (continued)

Data source	Product Version	Default	Minimum	Maximum	Valid values
	Version 10 (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity)	32767	32767	1048575	32767, 65535, 98303, 131071, 163839, 196607, 229375, 262143, 294911, 327679, 360447, 393215, 425983, 458751, 491519, 524287, 557055, 589823, 622591, 655359, 688127, 720895, 753663, 786431, 819199, 851967, 884735, 917503, 950271, 983039, 1015807, 1048575[1]

Note:

1. If you specify a value between the minimum and maximum value that is not a valid value, the IBM Data Server Driver for JDBC and SQLJ sets queryDataSize to the nearest valid value.

queryTimeoutProcessingMode

Specifies what happens when the query timeout interval for a Statement object expires. Valid values are:

DB2BaseDataSource.-INTERRUPT_PROCESSING_MODE_STATEMENT_CANCEL (1)

Specifies that when the query timeout interval for a Statement object expires, the IBM Data Server Driver for JDBC and SQLJ cancels the currently executing SQL statement, if the data server supports interruption of SQL statements. If the data server does not support interruption of SQL statements, the driver throws an Exception that indicates that the feature is not supported.

DB2BaseDataSource.-INTERRUPT_PROCESSING_MODE_STATEMENT_CANCEL is the default.

DB2BaseDataSource.INTERRUPT_PROCESSING_MODE_CLOSE_SOCKET (2)

Specifies that the underlying socket is dropped and the connection is closed when the query timeout interval for a Statement object expires.

When the Statement object times out, and automatic client reroute or client affinities is not configured, a DisconnectException with error code -4499 is thrown. Any subsequent operations on the Statement object, or on any other Statement objects that were created from the same connection receive an Exception that indicates that the connection is closed. After a Statement object times out, the application must establish a new connection before it can execute a new transaction.

If automatic client reroute or client affinities is configured, the IBM Data Server Driver for JDBC and SQLJ tries to re-establish a connection according to the reroute mechanism in effect. If a new connection is successfully re-established, the driver returns an error code of -4498 or -30108, instead of -4499. However, the driver does not execute the timed-out SQL statements again, even if enableSeamlessFailover is set to DB2BaseDataSource.YES (1).

resultSetHoldability

Specifies whether cursors remain open after a commit operation. The data type of this property is int. Valid values are:

DB2BaseDataSource.HOLD_CURSORS_OVER_COMMIT (1)

Leave cursors open after a commit operation.

This setting is not valid for a connection that is part of a distributed (XA) transaction.

DB2BaseDataSource.CLOSE_CURSORS_AT_COMMIT (2)
Close cursors after a commit operation.

DB2BaseDataSource.NOT_SET (0)
This is the default value. The behavior is:

- For connections that are part of distributed (XA) transactions, cursors are closed after a commit operation.
- For connections that are not part of a distributed transaction:
  - For connections to all versions of DB2 for z/OS, DB2 Database for Linux, UNIX, and Windows, or DB2 for i servers, or to Cloudscape Version 8.1 or later servers, cursors remain open after a commit operation.
  - For connections to all versions of IBM Informix, or to Cloudscape versions earlier than Version 8.1, cursors are closed after a commit operation.

retrieveMessagesFromServerOnGetMessage
Specifies whether JDBC SQLException.getMessage or SQLWarning.getMessage calls cause the IBM Data Server Driver for JDBC and SQLJ to invoke a DB2 for z/OS stored procedure that retrieves the message text for the error. The data type of this property is boolean. The default is `false`, which means that the full message text is not returned to the client.

For example, if retrieveMessagesFromServerOnGetMessage is set to `true`, a message similar to this one is returned by SQLException.getMessage after an attempt to perform an SQL operation on nonexistent table ADMF001.NO_TABLE:

```
ADMF001.NO_TABLE IS AN UNDEFINED NAME. SQLCODE=-204,
SQLSTATE=42704, DRIVER=3.50.54
```

If retrieveMessagesFromServerOnGetMessage is set to `false`, a message similar to this one is returned:

```
DB2 SQL Error: SQLCODE=-204, SQLSTATE=42704, DRIVER=3.50.54
```

An alternative to setting this property to `true` is to use the IBM Data Server Driver for JDBC and SQLJ-only `DB2Sqlca.getMessage` method in applications. Both techniques result in a stored procedure call, which starts a unit of work.

retryIntervalForClientReroute
For automatic client reroute, specifies the amount of time in seconds between connection retries.

The data type of this property is int.

The IBM Data Server Driver for JDBC and SQLJ uses the retryIntervalForClientReroute property only if the maxRetriesForClientReroute property is also set.

If maxRetriesForClientReroute or retryIntervalForClientReroute is not set, the IBM Data Server Driver for JDBC and SQLJ performs retries for 10 minutes.

If the enableClientAffinitiesList is set to DB2BaseDataSource.NO (2), an attempt to connect to the primary server and alternate servers counts as one retry. The driver waits the number of seconds that is specified by retryIntervalForClientReroute before retrying the connection. If enableClientAffinitiesList is set to DB2BaseDataSource.YES (1), each server that is specified by the clientRerouteAlternateServerName and

clientRerouteAlternatePortNumber values is retried after the number of seconds that is specified by retryIntervalForClientReroute.

The default value for retryIntervalForClientReroute is DB2BaseDataSource.NOT_SET (-1). The default behavior is that there is no wait between retries.

securityMechanism
Specifies the DRDA security mechanism. The data type of this property is int. Possible values are:

CLEAR_TEXT_PASSWORD_SECURITY (3)
User ID and password

USER_ONLY_SECURITY (4)
User ID only

ENCRYPTED_PASSWORD_SECURITY (7)
User ID, encrypted password

ENCRYPTED_USER_AND_PASSWORD_SECURITY (9)
Encrypted user ID and password

KERBEROS_SECURITY (11)
Kerberos. This value does not apply to connections to IBM Informix.

ENCRYPTED_USER_AND_DATA_SECURITY (12)
Encrypted user ID and encrypted security-sensitive data. This value applies to connections to DB2 for z/OS only.

ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY (13)
Encrypted user ID and password, and encrypted security-sensitive data. This value does not apply to connections to IBM Informix.

PLUGIN_SECURITY (15)
Plug-in security. This value applies to connections to DB2 Database for Linux, UNIX, and Windows only.

ENCRYPTED_USER_ONLY_SECURITY (16)
Encrypted user ID. This value does not apply to connections to IBM Informix.

If this property is specified, the specified security mechanism is the only mechanism that is used. If the security mechanism is not supported by the connection, an exception is thrown.

The default value for securityMechanism is CLEAR_TEXT_PASSWORD_SECURITY. If the server does not support CLEAR_TEXT_PASSWORD_SECURITY but supports ENCRYPTED_USER_AND_PASSWORD_SECURITY, the IBM Data Server Driver for JDBC and SQLJ driver updates the security mechanism to ENCRYPTED_USER_AND_PASSWORD_SECURITY and attempts to connect to the server. Any other mismatch in security mechanism support between the requester and the server results in an error.

sendDataAsIs
Specifies that the IBM Data Server Driver for JDBC and SQLJ does not convert input parameter values to the target column data types. The data type of this property is boolean. The default is `false`.

You should use this property only for applications that always ensure that the data types in the application match the data types in the corresponding database tables.

serverName
The host name or the TCP/IP address of the data source. The data type of this property is String.

sslConnection
Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses an SSL socket to connect to the data source. If sslConnection is set to `true`, the connection uses an SSL socket. If sslConnection is set to `false`, the connection uses a plain socket.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

sslTrustStoreLocation
Specifies the name of the Java truststore on the client that contains the server certificate for an SSL connection.

The IBM Data Server Driver for JDBC and SQLJ uses this option only if the sslConnection property is set to `true`.

If sslTrustStore is set, and sslConnection is set to `true`, the IBM Data Server Driver for JDBC and SQLJ uses the sslTrustStoreLocation value instead of the value in the javax.net.ssl.trustStore Java property.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

sslTrustStorePassword
Specifies the password for the Java truststore on the client that contains the server certificate for an SSL connection.

The IBM Data Server Driver for JDBC and SQLJ uses this option only if the sslConnection property is set to `true`.

If sslTrustStorePassword is set, and sslConnection is set to `true`, the IBM Data Server Driver for JDBC and SQLJ uses the sslTrustStorePassword value instead of the value in the javax.net.ssl.trustStorePassword Java property.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

stripTrailingZerosForDecimalNumbers
Specifies whether the IBM Data Server Driver for JDBC and SQLJ removes trailing zeroes when it retrieves data from a DECFLOAT, DECIMAL, or NUMERIC column. This property is meaningful only if the SDK for Java is Version 1.5 or later. The data type of this property is int. Possible values are:

DB2BaseDataSource.NOT_SET (0)
The IBM Data Server Driver for JDBC and SQLJ does not remove trailing zeroes from the retrieved value. This is the default.

DB2BaseDataSource.YES (1)
The IBM Data Server Driver for JDBC and SQLJ removes trailing zeroes when it retrieves a value from a DECFLOAT, DECIMAL, or NUMERIC column as a java.math.BigDecimal object.

For example, when the driver retrieves the value 234.04000, it returns the value 234.04 to the application.

DB2BaseDataSource.NO (2)
The IBM Data Server Driver for JDBC and SQLJ does not remove trailing zeroes from the retrieved value.

timestampFormat
Specifies the format in which the result of the `ResultSet.getString` or `CallableStatement.getString` method against a TIMESTAMP column is returned. The data type of timestampFormat is int.

Possible values of timestampFormat are:

Constant	Integer value	Format
`com.ibm.db2.jcc.DB2BaseDataSource.ISO`	1	*yyyy-mm-dd-hh.mm.ss.nnnnnn*
`com.ibm.db2.jcc.DB2BaseDataSource.JDBC`	5	*yyyy-mm-dd hh:mm:ss.nnnnnn*

Note:

The default is `com.ibm.db2.jcc.DB2BaseDataSource.JDBC`.

timestampFormat affects the format of output only.

timestampPrecisionReporting
Specifies whether trailing zeroes are truncated in the result of a Resultset.getString call for a TIMESTAMP value. The data type of this property is int. Possible values are:

TIMESTAMP_JDBC_STANDARD (1)
Trailing zeroes are truncated in the result of a Resultset.getString call for a TIMESTAMP value. This is the default.

For example:

- A TIMESTAMP value of 2009-07-19-10.12.00.000000 is truncated to 2009-07-19-10.12.00.0 after retrieval.
- A TIMESTAMP value of 2009-12-01-11.30.00.100000 is truncated to 2009-12-01-11.30.00.1 after retrieval.

TIMESTAMP_ZERO_PADDING (2)
Trailing zeroes are not truncated in the result of a Resultset.getString call for a TIMESTAMP value.

traceDirectory
Specifies a directory into which trace information is written. The data type of this property is `String`. When traceDirectory is specified, trace information for multiple connections on the same DataSource is written to multiple files.

When traceDirectory is specified, a connection is traced to a file named traceFile_*origin*_*n*.

n is the *n*th connection for a DataSource.

origin indicates the origin of the log writer that is in use. Possible values of *origin* are:

cpds The log writer for a DB2ConnectionPoolDataSource object.

driver The log writer for a DB2Driver object.

global The log writer for a DB2TraceManager object.

sds The log writer for a DB2SimpleDataSource object.

xads The log writer for a DB2XADataSource object.

If the traceFile property is also specified, the traceDirectory value is not used.

traceFile
Specifies the name of a file into which the IBM Data Server Driver for JDBC and SQLJ writes trace information. The data type of this property is `String`. The traceFile property is an alternative to the logWriter property for directing the output trace stream to a file.

traceFileAppend
Specifies whether to append to or overwrite the file that is specified by the traceFile property. The data type of this property is boolean. The default is `false`, which means that the file that is specified by the traceFile property is overwritten.

traceLevel
Specifies what to trace. The data type of this property is int.

You can specify one or more of the following traces with the traceLevel property:
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_NONE` (X'00')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTION_CALLS` (X'01')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_STATEMENT_CALLS` (X'02')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_CALLS` (X'04')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRIVER_CONFIGURATION` (X'10')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS` (X'20')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS` (X'40')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_META_DATA` (X'80')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_PARAMETER_META_DATA` (X'100')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DIAGNOSTICS` (X'200')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SQLJ` (X'400')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_META_CALLS` (X'2000')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DATASOURCE_CALLS` (X'4000')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_LARGE_OBJECT_CALLS` (X'8000')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SYSTEM_MONITOR` (X'20000')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_TRACEPOINTS` (X'40000')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL` (X'FFFFFFFF')

To specify more than one trace, use one of these techniques:
- Use bitwise OR (`|`) operators with two or more trace values. For example, to trace DRDA flows and connection calls, specify this value for traceLevel:
  ```
  TRACE_DRDA_FLOWS|TRACE_CONNECTION_CALLS
  ```
- Use a bitwise complement (~) operator with a trace value to specify all except a certain trace. For example, to trace everything except DRDA flows, specify this value for traceLevel:
  ```
  ~TRACE_DRDA_FLOWS
  ```

user
The user ID to use for establishing connections. The data type of this property is String. When you use the DataSource interface to establish a connection, you can override this property value by invoking this form of the DataSource.getConnection method:

```
getConnection(user, password);
```

xaNetworkOptimization
Specifies whether XA network optimization is enabled for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity. You might need to disable XA network optimization in an environment in which an XA Start and XA End are issued from one Java process, and an XA Prepare and an XA Commit are issued from another Java process. With XA network optimization, the XA

Prepare can reach the data source before the XA End, which results in an XAER_PROTO error. To prevent the XAER_PROTO error, disable XA network optimization.

The default is `true`, which means that XA network optimization is enabled. If xaNetworkOptimization is `false`, which means that XA network optimization is disabled, the driver closes any open cursors at XA End time.

xaNetworkOptimization can be set on a DataSource object, or in the *url* parameter in a getConnection call. The value of xaNetworkOptimization cannot be changed after a connection is obtained.

com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements
Controls an internal statement cache that is associated with a PooledConnection. The data type of this property is int. Possible values are:

positive integer
Enables the internal statement cache for a PooledConnection, and specifies the number of statements that the IBM Data Server Driver for JDBC and SQLJ keeps open in the cache.

0 or negative integer
Disables internal statement caching for the PooledConnection. 0 is the default.

com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements controls the internal statement cache that is associated with a PooledConnection only when the PooledConnection object is created. com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements has no effect on caching in an already existing PooledConnection object.

com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Common IBM Data Server Driver for JDBC and SQLJ properties for DB2 servers

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply to DB2 for z/OS and DB2 Database for Linux, UNIX, and Windows only.

Unless otherwise noted, all properties are in com.ibm.db2.jcc.DB2BaseDataSource.

Those properties are:

clientAccountingInformation
Specifies accounting information for the current client for the connection. This information is for client accounting purposes. This value can change during a connection. The data type of this property is String. The maximum length is 255 bytes. A Java empty string ("") is valid for this value, but a Java `null` value is not valid.

clientApplicationInformation
Specifies the application or transaction name of the end user's application. You can use this property to provide the identity of the client end user for accounting and monitoring purposes. This value can change during a connection. The data type of this property is String. For a DB2 for z/OS server, the maximum length is 32 bytes. For a DB2 Database for Linux, UNIX, and Windows server, the maximum length is 255 bytes. A Java empty string ("") is valid for this value, but a Java `null` value is not valid.

clientProgramId
Specifies a value for the client program ID that can be used to identify the end user. The data type of this property is String, and the length is 80 bytes. If the program ID value is less than 80 bytes, the value must be padded with blanks.

clientProgramName
Specifies an application ID that is fixed for the duration of a physical connection for a client. The value of this property becomes the correlation ID on a DB2 for z/OS server. Database administrators can use this property to correlate work on a DB2 for z/OS server to client applications. The data type of this property is String. The maximum length is 12 bytes. If this value is `null`, the IBM Data Server Driver for JDBC and SQLJ supplies a value of db2jcc*thread-name*.

This property applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

concurrentAccessResolution
Specifies whether the IBM Data Server Driver for JDBC and SQLJ requests that a read transaction can access a committed and consistent image of rows that are incompatibly locked by write transactions, if the data source supports accessing currently committed data, and the application isolation level is cursor stability (CS) or read stability (RS). This option has the same effect as the DB2 CONCURRENTACCESSRESOLUTION bind option. Possible values are:

DB2BaseDataSource.-
CONCURRENTACCESS_USE_CURRENTLY_COMMITTED (1)
The IBM Data Server Driver for JDBC and SQLJ requests that:

- Read transactions access the currently committed data when the data is being updated or deleted.
- Read transactions skip rows that are being inserted.

DB2BaseDataSource.CONCURRENTACCESS_WAIT_FOR_OUTCOME (2)
The IBM Data Server Driver for JDBC and SQLJ requests that:

- Read transactions wait for a commit or rollback operation when they encounter data that is being updated or deleted.
- Read transactions do not skip rows that are being inserted.

DB2BaseDataSource.CONCURRENTACCESS_NOT_SET (0)
Enables the data server's default behavior for read transactions when lock contention occurs. This is the default value.

currentDegree
Specifies the degree of parallelism for the execution of queries that are dynamically prepared. The type of this property is String. The currentDegree value is used to set the CURRENT DEGREE special register on the data source. If currentDegree is not set, no value is passed to the data source.

currentExplainMode
Specifies the value for the CURRENT EXPLAIN MODE special register. The CURRENT EXPLAIN MODE special register enables and disables the Explain facility. The data type of this property is String. The maximum length is 254 bytes. This property applies only to connections to data sources that support the CURRENT EXPLAIN MODE special register.

currentFunctionPath
Specifies the SQL path that is used to resolve unqualified data type names and function names in SQL statements that are in JDBC programs. The data type of this property is String. For a DB2 Database for Linux, UNIX, and Windows

server, the maximum length is 254 bytes. For a DB2 for z/OS server, the maximum length is 2048 bytes. The value is a comma-separated list of schema names. Those names can be ordinary or delimited identifiers.

currentMaintainedTableTypesForOptimization
Specifies a value that identifies the types of objects that can be considered when the data source optimizes the processing of dynamic SQL queries. This register contains a keyword representing table types. The data type of this property is String.

Possible values of currentMaintainedTableTypesForOptimization are:

ALL
Indicates that all materialized query tables will be considered.

NONE
Indicates that no materialized query tables will be considered.

SYSTEM
Indicates that only system-maintained materialized query tables that are refresh deferred will be considered.

USER
Indicates that only user-maintained materialized query tables that are refresh deferred will be considered.

currentPackagePath
Specifies a comma-separated list of collections on the server. The database server searches these collections for JDBC and SQLJ packages.

The precedence rules for the currentPackagePath and currentPackageSet properties follow the precedence rules for the CURRENT PACKAGESET and CURRENT PACKAGE PATH special registers.

currentPackageSet
Specifies the collection ID to search for JDBC and SQLJ packages. The data type of this property is String. If currentPackageSet is set, its value overrides the value of jdbcCollection.

Multiple instances of the IBM Data Server Driver for JDBC and SQLJ can be installed at a database server by running the DB2Binder utility multiple times. The DB2binder utility includes a -collection option that lets the installer specify the collection ID for each IBM Data Server Driver for JDBC and SQLJ instance. To choose an instance of the IBM Data Server Driver for JDBC and SQLJ for a connection, you specify a currentPackageSet value that matches the collection ID for one of the IBM Data Server Driver for JDBC and SQLJ instances.

The precedence rules for the currentPackagePath and currentPackageSet properties follow the precedence rules for the CURRENT PACKAGESET and CURRENT PACKAGE PATH special registers.

currentRefreshAge
Specifies a timestamp duration value that is the maximum duration since a REFRESH TABLE statement was processed on a system-maintained REFRESH DEFERRED materialized query table such that the materialized query table can be used to optimize the processing of a query. This property affects dynamic statement cache matching. The data type of this property is long.

currentSchema
Specifies the default schema name that is used to qualify unqualified database objects in dynamically prepared SQL statements. The value of this property

sets the value in the CURRENT SCHEMA special register on the database server. The schema name is case-sensitive, and must be specified in uppercase characters.

cursorSensitivity
Specifies whether the java.sql.ResultSet.TYPE_SCROLL_SENSITIVE value for a JDBC ResultSet maps to the SENSITIVE DYNAMIC attribute, the SENSITIVE STATIC attribute, or the ASENSITIVE attribute for the underlying database cursor. The data type of this property is int. Possible values are TYPE_SCROLL_SENSITIVE_STATIC (0), TYPE_SCROLL_SENSITIVE_DYNAMIC (1), or TYPE_SCROLL_ASENSITIVE (2). The default is TYPE_SCROLL_SENSITIVE_STATIC.

If the data source does not support sensitive dynamic scrollable cursors, and TYPE_SCROLL_SENSITIVE_DYNAMIC is requested, the JDBC driver accumulates a warning and maps the sensitivity to SENSITIVE STATIC. For DB2 for i database servers, which do not support sensitive static cursors, java.sql.ResultSet.TYPE_SCROLL_SENSITIVE always maps to SENSITIVE DYNAMIC.

dateFormat
Specifies:

- The format in which the String argument of the PreparedStatement.setString method against a DATE column must be specified.
- The format in which the result of the ResultSet.getString or CallableStatement.getString method against a DATE column is returned.

The data type of dateFormat is int.

Possible values of dateFormat are:

Constant	Integer value	Format
com.ibm.db2.jcc.DB2BaseDataSource.ISO	1	*yyyy-mm-dd*
com.ibm.db2.jcc.DB2BaseDataSource.USA	2	*mm/dd/yyyy*
com.ibm.db2.jcc.DB2BaseDataSource.EUR	3	*dd.mm.yyyy*
com.ibm.db2.jcc.DB2BaseDataSource.JIS	4	*yyyy-mm-dd*

The default is com.ibm.db2.jcc.DB2BaseDataSource.ISO.

decimalRoundingMode
Specifies the rounding mode for assignment to decimal floating-point variables or DECFLOAT columns on DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows data servers.

Possible values are:

DB2BaseDataSource.ROUND_DOWN (1)
Rounds the value towards 0 (truncation). The discarded digits are ignored.

DB2BaseDataSource.ROUND_CEILING (2)
Rounds the value towards positive infinity. If all of the discarded digits are zero or if the sign is negative the result is unchanged other than the removal of the discarded digits. Otherwise, the result coefficient is incremented by 1.

DB2BaseDataSource.ROUND_HALF_EVEN (3)
Rounds the value to the nearest value; if the values are equidistant, rounds the value so that the final digit is even. If the discarded digits represents greater than half (0.5) of the value of one in the next left

position then the result coefficient is incremented by 1. If they represent less than half, then the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise the result coefficient is unaltered if its rightmost digit is even, or is incremented by 1 if its rightmost digit is odd (to make an even digit).

DB2BaseDataSource.ROUND_HALF_UP (4)
Rounds the value to the nearest value; if the values are equidistant, rounds the value away from zero. If the discarded digits represent greater than or equal to half (0.5) of the value of one in the next left position then the result coefficient is incremented by 1. Otherwise the discarded digits are ignored.

DB2BaseDataSource.ROUND_FLOOR (6)
Rounds the value towards negative infinity. If all of the discarded digits are zero or if the sign is positive the result is unchanged other than the removal of discarded digits. Otherwise, the sign is negative and the result coefficient is incremented by 1.

DB2BaseDataSource.ROUND_UNSET (-2147483647)
No rounding mode was explicitly set. The IBM Data Server Driver for JDBC and SQLJ does not use the decimalRoundingMode to set the rounding mode on the data server. The rounding mode is ROUND_HALF_EVEN.

If you explicitly set the decimalRoundingMode value, that value updates the CURRENT DECFLOAT ROUNDING MODE special register value on a DB2 for z/OS data server.

If you explicitly set the decimalRoundingMode value, that value does not update the CURRENT DECFLOAT ROUNDING MODE special register value on a DB2 Database for Linux, UNIX, and Windows data server. If the value to which you set decimalRoundingMode is not the same as the value of the CURRENT DECFLOAT ROUNDING MODE special register, an Exception is thrown. To change the data server value, you need to set that value with the decflt_rounding database configuration parameter.

decimalRoundingMode does not affect decimal value assignments. The IBM Data Server Driver for JDBC and SQLJ always rounds decimal values down.

enableExtendedIndicators
Specifies whether support for extended indicators is enabled in the IBM Data Server Driver for JDBC and SQLJ. Possible values are:

DB2BaseDataSource.YES (1)
Support for extended indicators is enabled in the IBM Data Server Driver for JDBC and SQLJ.

DB2BaseDataSource.NO (2)
Support for extended indicators is disabled in the IBM Data Server Driver for JDBC and SQLJ.

DB2BaseDataSource.NOT_SET (0)
Support for extended indicators is enabled in the IBM Data Server Driver for JDBC and SQLJ. This is the default value.

enableRowsetSupport
Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses multiple-row FETCH for forward-only cursors or scrollable cursors, if the data source supports multiple-row FETCH. The data type of this property is int.

When enableRowsetSupport is set, its value overrides the useRowsetCursor property value.

Possible values are:

DB2BaseDataSource.YES (1)
Specifies that:

- For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS, multiple-row FETCH is used for scrollable cursors and forward-only cursors, if the data source supports multiple-row FETCH.
- For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows, multiple-row fetch is used for scrollable cursors, if the data source supports multiple-row FETCH.

DB2BaseDataSource.NO (2)
Specifies that multiple-row fetch is not used.

DB2BaseDataSource.NOT_SET (0)
Specifies that if the enableRowsetSupport property is not set:

- For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS, multiple-row fetch is not used.
- For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows, the useRowsetCursor property determines whether multiple-row fetch is used for scrollable cursors.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS, multiple-row fetch is not compatible with progressive streaming. Therefore, if progressive streaming is used for a FETCH operation, multiple-row FETCH is not used.

encryptionAlgorithm
Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses 56-bit DES (weak) encryption or 256-bit AES (strong) encryption. The data type of this property is int. Possible values are:

1 The driver uses 56-bit DES encryption.

2 The driver uses 256-bit AES encryption, if the database server supports it. 256-bit AES encryption is available for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity only.

encryptionAlgorithm can be specified only if the securityMechanism value is `ENCRYPTED_PASSWORD_SECURITY` (7) or `ENCRYPTED_USER_AND_PASSWORD_SECURITY` (9).

fullyMaterializeInputStreams
Indicates whether streams are fully materialized before they are sent from the client to a data source. The data type of this property is boolean. The default is `false`.

If the value of `fullyMaterializeInputStreams` is `true`, the JDBC driver fully materialized the streams before sending them to the server.

gssCredential
For a data source that uses Kerberos security, specifies a delegated credential that is passed from another principal. The data type of this property is

org.ietf.jgss.GSSCredential. Delegated credentials are used in multi-tier environments, such as when a client connects to WebSphere Application Server, which, in turn, connects to the data source. You obtain a value for this property from the client, by invoking the GSSContext.getDelegCred method. GSSContext is part of the IBM Java Generic Security Service (GSS) API. If you set this property, you also need to set the Mechanism and KerberosServerPrincipal properties.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

For more information on using Kerberos security with the IBM Data Server Driver for JDBC and SQLJ, see "Using Kerberos security under the IBM Data Server Driver for JDBC and SQLJ".

kerberosServerPrincipal
For a data source that uses Kerberos security, specifies the name that is used for the data source when it is registered with the Kerberos Key Distribution Center (KDC). The data type of this property is String.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

pdqProperties
Specifies properties that control the interaction between the IBM Data Server Driver for JDBC and SQLJ and the client optimization feature of pureQuery.

The data type of this property is String.

Set the pdqProperties property **only** if you are using the client optimization feature of pureQuery. See the Integrated Data Management Information Center for information about valid values for pdqProperties.

readOnly
Specifies whether the connection is read-only. The data type of this property is boolean. The default is `false`.

resultSetHoldabilityForCatalogQueries
Specifies whether cursors for queries that are executed on behalf of DatabaseMetaData methods remain open after a commit operation. The data type of this property is int.

When an application executes DatabaseMetaData methods, the IBM Data Server Driver for JDBC and SQLJ executes queries against the catalog of the target data source. By default, the holdability of those cursors is the same as the holdability of application cursors. To use different holdability for catalog queries, use the resultSetHoldabilityForCatalogQueries property. Possible values are:

DB2BaseDataSource.HOLD_CURSORS_OVER_COMMIT (1)
Leave cursors for catalog queries open after a commit operation, regardless of the resultSetHoldability setting.

DB2BaseDataSource.CLOSE_CURSORS_AT_COMMIT (2)
Close cursors for catalog queries after a commit operation, regardless of the resultSetHoldability setting.

DB2BaseDataSource.NOT_SET (0)
Use the resultSetHoldability setting for catalog queries. This is the default value.

returnAlias
Specifies whether the JDBC driver returns rows for table aliases and synonyms

for DatabaseMetaData methods that return table information, such as getTables. The data type of returnAlias is int. Possible values are:

0 Do not return rows for aliases or synonyms of tables in output from DatabaseMetaData methods that return table information.

1 For tables that have aliases or synonyms, return rows for aliases and synonyms of those tables, as well as rows for the tables, in output from DatabaseMetaData methods that return table information. This is the default.

statementConcentrator
Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses the data source's statement concentrator functionality. The statement concentrator is the ability to bypass preparation of a statement when it is the same as a statement in the dynamic statement cache, except for literal values. Statement concentrator functionality applies only to SQL statements that have literals but no parameter markers. Possible values are:

DB2BaseDataSource.STATEMENT_CONCENTRATOR_OFF (1)
The IBM Data Server Driver for JDBC and SQLJ does not use the data source's statement concentrator functionality.

DB2BaseDataSource.STATEMENT_CONCENTRATOR_WITH_LITERALS (2)
The IBM Data Server Driver for JDBC and SQLJ uses the data source's statement concentrator functionality.

DB2BaseDataSource.STATEMENT_CONCENTRATOR_NOT_SET (0)
Enables the data server's default behavior for statement concentrator functionality. This is the default value.

For DB2 Database for Linux, UNIX, and Windows data sources that support statement concentrator functionality, the functionality is used if the STMT_CONC configuration parameter is set to ON at the data source. Otherwise, statement concentrator functionality is not used.

For DB2 for z/OS data sources that support statement concentrator functionality, the functionality is not used if statementConcentrator is not set.

streamBufferSize
Specifies the size, in bytes, of the JDBC driver buffers for chunking LOB or XML data. The JDBC driver uses the streamBufferSize value whether or not it uses progressive streaming. The data type of streamBufferSize is int. The default is 1048576.

If the JDBC driver uses progressive streaming, LOB or XML data is materialized if it fits in the buffers, and the driver does not use the fullyMaterializeLobData property.

DB2 for z/OS Version 9.1 and later supports progressive streaming for LOBs and XML objects. DB2 Database for Linux, UNIX, and Windows Version 9.5 and later, and IBM Informix Version 11.50 and later support progressive streaming for LOBs.

supportsAsynchronousXARollback
Specifies whether the IBM Data Server Driver for JDBC and SQLJ supports asynchronous XA rollback operations. The data type of this property is int. The default is `DB2BaseDataSource.NO` (2). If the application runs against a BEA WebLogic Server application server, set supportsAsynchronousXARollback to `DB2BaseDataSource.YES` (1).

sysSchema
Specifies the schema of the shadow catalog tables or views that are searched when an application invokes a DatabaseMetaData method. The sysSchema property was formerly called cliSchema.

timeFormat
Specifies:

- The format in which the String argument of the PreparedStatement.setString method against a TIME column must be specified.
- The format in which the result of the ResultSet.getString or CallableStatement.getString method against a TIME column is returned.

The data type of timeFormat is int.

Possible values of timeFormat are:

Constant	Integer value	Format
`com.ibm.db2.jcc.DB2BaseDataSource.ISO`	1	*hh:mm:ss*
`com.ibm.db2.jcc.DB2BaseDataSource.USA`	2	*hh:mm* am or *hh:mm* pm
`com.ibm.db2.jcc.DB2BaseDataSource.EUR`	3	*hh.mm.ss*
`com.ibm.db2.jcc.DB2BaseDataSource.JIS`	4	*hh:mm:ss*

The default is `com.ibm.db2.jcc.DB2BaseDataSource.ISO`.

timestampOutputType
Specifies whether the IBM Data Server Driver for JDBC and SQLJ returns a java.sql.Timestamp object or a com.ibm.db2.jcc.DBTimestamp when the standard JDBC interfaces ResultSet.getTimestamp, CallableStatement.getTimestamp, ResultSet.getObject, or CallableStatement.getObject are called to return timestamp information.

Possible values are:

DB2BaseDataSource.JDBC_TIMESTAMP (1)
The IBM Data Server Driver for JDBC and SQLJ returns java.sql.Timestamp objects from ResultSet.getTimestamp, CallableStatement.getTimestamp, ResultSet.getObject, or CallableStatement.getObject calls.

DB2BaseDataSource.JCC_DBTIMESTAMP (2)
The IBM Data Server Driver for JDBC and SQLJ returns com.ibm.db2.jcc.DBTimestamp objects from ResultSet.getTimestamp, CallableStatement.getTimestamp, ResultSet.getObject, or CallableStatement.getObject calls.

DB2BaseDataSource.NOT_SET (0)
This is the default behavior.

The behavior is the same as the behavior for DB2BaseDataSource.JDBC_TIMESTAMP.

useCachedCursor
Specifies whether the underlying cursor for PreparedStatement objects is cached and reused on subsequent executions of the PreparedStatement. The data type of useCachedCursor is boolean.

If useCachedCursor is set to `true`, the cursor for PreparedStatement objects is cached, which can improve performance. `true` is the default.

Set useCachedCursor to `false` if PreparedStatement objects access tables whose column types or lengths change between executions of those PreparedStatement objects.

useJDBC4ColumnNameAndLabelSemantics

Specifies how the IBM Data Server Driver for JDBC and SQLJ handles column labels in ResultSetMetaData.getColumnName, ResultSetMetaData.getColumnLabel, and ResultSet.findColumn method calls.

Possible values are:

DB2BaseDataSource.YES (1)

The IBM Data Server Driver for JDBC and SQLJ uses the following rules, which conform to the JDBC 4.0 specification, to determine the value that ResultSetMetaData.getColumnName, ResultSetMetaData.getColumnLabel, and ResultSet.findColumn return:

- The column name that is returned by ResultSetMetaData.getColumnName is its name from the database.
- The column label that is returned by ResultSetMetaData.getColumnLabel is the label that is specified with the SQL AS clause. If the SQL AS clause is not specified, the label is the name of the column.
- ResultSet.findColumn takes the label for the column, as specified with the SQL AS clause, as input. If the SQL AS clause was not specified, the label is the column name.
- The IBM Data Server Driver for JDBC and SQLJ does not use a column label that is assigned by the SQL LABEL ON statement.

These rules apply to IBM Data Server Driver for JDBC and SQLJ version 3.50 and later, for connections to the following database systems:

- DB2 for z/OS Version 8 or later
- DB2 Database for Linux, UNIX, and Windows Version 8.1 or later
- DB2 UDB for iSeries® V5R3 or later

For earlier versions of the driver or the database systems, the rules for a useJDBC4ColumnNameAndLabelSemantics value of DB2BaseDataSource.NO apply, even if useJDBC4ColumnNameAndLabelSemantics is set to DB2BaseDataSource.YES.

DB2BaseDataSource.NO (2)

The IBM Data Server Driver for JDBC and SQLJ uses the following rules to determine the values that ResultSetMetaData.getColumnName, ResultSetMetaData.getColumnLabel, and ResultSet.findColumn return:

If the data source does not support the LABEL ON statement, or the source column is not defined with the LABEL ON statement:

- The value that is returned by ResultSetMetaData.getColumnName is its name from the database, if no SQL AS clause is specified. If the SQL AS clause is specified, the value that is returned is the column label.
- The value that is returned by ResultSetMetaData.getColumnLabel is the label that is specified with the SQL AS clause. If the SQL AS clause is not specified, the value that is returned is the name of the column.
- ResultSet.findColumn takes the column name as input.

If the source column is defined with the LABEL ON statement:

- The value that is returned by ResultSetMetaData.getColumnName is the column name from the database, if no SQL AS clause is specified. If the SQL AS clause is specified, the value that is returned is the column label that is specified in the AS clause.
- The value that is returned by ResultSetMetaData.getColumnLabel is the label that is specified in the LABEL ON statement.
- ResultSet.findColumn takes the column name as input.

These rules conform to the behavior of the IBM Data Server Driver for JDBC and SQLJ before Version 3.50.

DB2BaseDataSource.NOT_SET (0)
This is the default behavior.

For the IBM Data Server Driver for JDBC and SQLJ version 3.50 and earlier, the default behavior for useJDBC4ColumnNameAndLabelSemantics is the same as the behavior for DB2BaseDataSource.NO.

For the IBM Data Server Driver for JDBC and SQLJ version 4.0 and later:

- The default behavior for useJDBC4ColumnNameAndLabelSemantics is the same as the behavior for DB2BaseDataSource.YES, for connections to the following database systems:
  - DB2 for z/OS Version 8 or later
  - DB2 Database for Linux, UNIX, and Windows Version 8.1 or later
  - DB2 UDB for iSeries V5R3 or later
- For connections to earlier versions of these database systems, the default behavior for useJDBC4ColumnNameAndLabelSemantics is DB2BaseDataSource.NO.

com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements
Controls an internal statement cache that is associated with a PooledConnection. The data type of this property is int. Possible values are:

positive integer
Enables the internal statement cache for a PooledConnection, and specifies the number of statements that the IBM Data Server Driver for JDBC and SQLJ keeps open in the cache.

0 or negative integer
Disables internal statement caching for the PooledConnection. 0 is the default.

maxStatements controls the internal statement cache that is associated with a PooledConnection only when the PooledConnection object is created. maxStatements has no effect on caching in an already existing PooledConnection object.

maxStatements applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, and toIBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Common IBM Data Server Driver for JDBC and SQLJ properties for DB2 for z/OS and IBM Informix

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply to IBM Informix and DB2 for z/OS database servers.

Properties that apply to IBM Informix and DB2 for z/OS are:

enableConnectionConcentrator
Indicates whether the connection concentrator function of the IBM Data Server Driver for JDBC and SQLJ is enabled.

The data type of enableConnectionConcentrator is boolean. The default is `false`. However, if enableSysplexWLB is set to `true`, the default is `true`.

enablConnectionConcentrator applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

keepDynamic
Specifies whether the data source keeps already prepared dynamic SQL statements in the dynamic statement cache after commit points so that those prepared statements can be reused. The data type of this property is int. Valid values are `DB2BaseDataSource.YES` (1) and `DB2BaseDataSource.NO` (2).

If the keepDynamic property is not specified, the keepDynamic value is `DB2BaseDataSource.NOT_SET` (0). If the connection is to a DB2 for z/OS server, caching of dynamic statements for a connection is not done if the property is not set. If the connection is to an IBM Informix data source, caching of dynamic statements for a connection is done if the property is not set.

keepDynamic is used with the DB2Binder -keepdynamic option. The keepDynamic property value that is specified must match the -keepdynamic value that was specified when DB2Binder was run.

For a DB2 for z/OS database server, dynamic statement caching can be done only if the EDM dynamic statement cache is enabled on the data source. The CACHEDYN subsystem parameter must be set to `DB2BaseDataSource.YES` to enable the dynamic statement cache.

maxTransportObjects
Specifies the maximum number of transport objects that can be used for all connections with the associated DataSource object. The IBM Data Server Driver for JDBC and SQLJ uses transport objects and a global transport objects pool to support the connection concentrator and Sysplex workload balancing. There is one transport object for each physical connection to the data source.

The data type of this property is int.

The maxTransportObjects value is ignored if the enableConnectionConcentrator or enableSysplexWLB properties are not set to enable the use of the connection concentrator or Sysplex workload balancing.

If the maxTransportObjects value has not been reached, and a transport object is not available in the global transport objects pool, the pool creates a new transport object. If the maxTransportObjects value has been reached, the application waits for the amount of time that is specified by the db2.jcc.maxTransportObjectWaitTime configuration property. After that amount of time has elapsed, if there is still no available transport object in the pool, the pool throws an SQLException.

maxTransportObjects does **not** override the db2.jcc.maxTransportObjects configuration property. maxTransportObjects has no effect on connections from other DataSource objects. If the maxTransportObjects value is larger than the db2.jcc.maxTransportObjects value, maxTransportObjects does not increase the db2.jcc.maxTransportObjects value.

The default value for maxTransportObjects is -1, which means that the number of transport objects for the DataSource is limited only by the db2.jcc.maxTransportObjects value for the driver.

Common IBM Data Server Driver for JDBC and SQLJ properties for IBM Informix and DB2 Database for Linux, UNIX, and Windows

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply to IBM Informix and DB2 Database for Linux, UNIX, and Windows database servers.

Properties that apply to IBM Informix and DB2 Database for Linux, UNIX, and Windows are:

currentLockTimeout
Specifies whether DB2 Database for Linux, UNIX, and Windows servers wait for a lock when the lock cannot be obtained immediately. The data type of this property is int. Possible values are:

integer Wait for integer *seconds*. *integer* is between -1 and 32767, inclusive.

LOCK_TIMEOUT_NO_WAIT
Do not wait for a lock. This is the default.

LOCK_TIMEOUT_WAIT_INDEFINITELY
Wait indefinitely for a lock.

LOCK_TIMEOUT_NOT_SET
Use the default for the data source.

IBM Data Server Driver for JDBC and SQLJ properties for DB2 Database for Linux, UNIX, and Windows

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply only to DB2 Database for Linux, UNIX, and Windows servers.

Those properties are:

connectNode
Specifies the target database partition server that an application connects to. The data type of this property is int. The value can be between 0 and 999. The default is database partition server that is defined with port 0. connectNode applies to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 Database for Linux, UNIX, and Windows servers only.

currentExplainSnapshot
Specifies the value for the CURRENT EXPLAIN SNAPSHOT special register. The CURRENT EXPLAIN SNAPSHOT special register enables and disables the Explain snapshot facility. The data type of this property is String. The maximum length is eight bytes. This property applies only to connections to data sources that support the CURRENT EXPLAIN SNAPSHOT special register, such as DB2 Database for Linux, UNIX, and Windows.

currentQueryOptimization
Specifies a value that controls the class of query optimization that is performed by the database manager when it binds dynamic SQL statements. The data type of this property is int. The possible values of currentQueryOptimization are:

0 Specifies that a minimal amount of optimization is performed to generate an access plan. This class is most suitable for simple dynamic SQL access to well-indexed tables.

1 Specifies that optimization roughly comparable to DB2 Database for Linux, UNIX, and Windows Version 1 is performed to generate an access plan.

2 Specifies a level of optimization higher than that of DB2 Database for Linux, UNIX, and Windows Version 1, but at significantly less optimization cost than levels 3 and above, especially for very complex queries.

3 Specifies that a moderate amount of optimization is performed to generate an access plan.

5 Specifies a significant amount of optimization is performed to generate an access plan. For complex dynamic SQL queries, heuristic rules are used to limit the amount of time spent selecting an access plan. Where possible, queries will use materialized query tables instead of the underlying base tables.

7 Specifies a significant amount of optimization is performed to generate an access plan. This value is similar to 5 but without the heuristic rules.

9 Specifies the maximum amount of optimization is performed to generate an access plan. This can greatly expand the number of possible access plans that are evaluated. This class should be used to determine if a better access plan can be generated for very complex and very long-running queries using large tables. Explain and performance measurements can be used to verify that a better plan has been generated.

optimizationProfile
Specifies an optimization profile that is used during SQL optimization. The data type of this property is String. The optimizationProfile value is used to set the OPTIMIZATION PROFILE special register. The default is null.

optimizationProfile applies to DB2 Database for Linux, UNIX, and Windows servers only.

optimizationProfileToFlush
Specifies the name of an optimization profile that is to be removed from the optimization profile cache. The data type of this property is String. The default is null.

plugin
The name of a client-side JDBC security plug-in. This property has the `Object` type and contains a new instance of the JDBC security plug-in method.

pluginName
The name of a server-side security plug-in module.

retryWithAlternativeSecurityMechanism
Specifies whether the IBM Data Server Driver for JDBC and SQLJ retries a connection with an alternative security mechanism if the security mechanism that is specified by property securityMechanism is not supported by the data source. The data type of this property is int. Possible values are:

com.ibm.db2.jcc.DB2BaseDataSource.YES (1)
Retry the connection using an alternative security mechanism. The IBM

Data Server Driver for JDBC and SQLJ issues warning code +4222 and retries the connection with the most secure available security mechanism.

com.ibm.db2.jcc.DB2BaseDataSource.NO (2) or com.ibm.db2.jcc.DB2BaseDataSource.NOT_SET (0)
Do not retry the connection using an alternative security mechanism.

retryWithAlternativeSecurityMechanism applies to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity connections to DB2 Database for Linux, UNIX, and Windows only.

useTransactionRedirect
Specifies whether the DB2 system directs SQL statements to different database partitions for better performance. The data type of this property is boolean. The default is `false`.

This property is applicable only under the following conditions:

- The connection is to a DB2 Database for Linux, UNIX, and Windows server that uses the Database Partitioning Feature (DPF).
- The partitioning key remains constant throughout a transaction.

If useTransactionRedirect is `true`, the IBM Data Server Driver for JDBC and SQLJ sends connection requests to the DPF node that contains the target data of the first directable statement in the transaction. DB2 Database for Linux, UNIX, and Windows then directs the SQL statement to different partitions as needed.

IBM Data Server Driver for JDBC and SQLJ properties for DB2 for z/OS

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply only to DB2 for z/OS servers.

Those properties are:

accountingInterval
Specifies whether DB2 accounting records are produced at commit points or on termination of the physical connection to the data source. The data type of this property is String.

If the value of accountingInterval is "COMMIT", and there are no open, held cursors, DB2 writes an accounting record each time that the application commits work. If the value of accountingInterval is "COMMIT", and the application performs a commit operation while a held cursor is open, the accounting interval spans that commit point and ends at the next valid accounting interval end point. If the value of accountingInterval is not "COMMIT", accounting records are produced on termination of the physical connection to the data source.

The accountingInterval property sets the *accounting-interval* parameter for an underlying RRSAF signon call. If the value of subsystem parameter ACCUMACC is not NO, the ACCUMACC value overrides the accountingInterval setting.

accountingInterval applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. accountingInterval is not applicable to connections under CICS® or IMS™, or for Java stored procedures.

The accountingInterval property overrides the db2.jcc.accountingInterval configuration property.

charOutputSize

Specifies the maximum number of bytes to use for INOUT or OUT stored procedure parameters that are registered as `Types.CHAR` charOutputSize applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS database servers.

Because DESCRIBE information for stored procedure INOUT and OUT parameters is not available at run time, by default, the IBM Data Server Driver for JDBC and SQLJ sets the maximum length of each character INOUT or OUT parameter to 32767. For stored procedures with many `Types.CHAR` parameters, this maximum setting can result in allocation of much more storage than is necessary.

To use storage more efficiently, set charOutputSize to the largest expected length for any `Types.CHAR` INOUT or OUT parameter.

charOutputSize has no effect on INOUT or OUT parameters that are registered as `Types.VARCHAR` or `Types.LONGVARCHAR`. The driver uses the default length of 32767 for `Types.VARCHAR` and `Types.LONGVARCHAR` parameters.

The value that you choose for charOutputSize needs to take into account the possibility of expansion during character conversion. Because the IBM Data Server Driver for JDBC and SQLJ has no information about the server-side CCSID that is used for output parameter values, the driver requests the stored procedure output data in UTF-8 Unicode. The charOutputSize value needs to be the maximum number of bytes that are needed after the parameter value is converted to UTF-8 Unicode. UTF-8 Unicode characters can require up to three bytes. (The euro symbol is an example of a three-byte UTF-8 character.) To ensure that the value of charOutputSize is large enough, if you have no information about the output data, set charOutputSize to three times the defined length of the largest CHAR parameter.

clientUser

Specifies the current client user name for the connection. This information is for client accounting purposes. Unlike the JDBC connection user name, this value can change during a connection. For a DB2 for z/OS server, the maximum length is 16 bytes.

This property applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

clientWorkstation

Specifies the workstation name for the current client for the connection. This information is for client accounting purposes. This value can change during a connection. The data type of this property is String. For a DB2 for z/OS server, the maximum length is 18 bytes. A Java empty string ("") is valid for this value, but a Java `null` value is not valid.

This property applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

currentSQLID

Specifies:

- The authorization ID that is used for authorization checking on dynamically prepared CREATE, GRANT, and REVOKE SQL statements.
- The owner of a table space, database, storage group, or synonym that is created by a dynamically issued CREATE statement.
- The implicit qualifier of all table, view, alias, and index names specified in dynamic SQL statements.

currentSQLID sets the value in the CURRENT SQLID special register on a DB2 for z/OS server. If the currentSQLID property is not set, the default schema name is the value in the CURRENT SQLID special register.

enableMultiRowInsertSupport
Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses multi-row INSERT for batched INSERT or MERGE operations, when the target data server is a DB2 for z/OS server that supports multi-row INSERT. The batch operations must be PreparedStatement calls with parameter markers. The data type of this property is boolean. The default is `true`.

The enableMultiRowInsertSupport value cannot be changed for the duration of a connection. enableMultiRowInsertSupport must be set to false if INSERT FROM SELECT statements are executed in a batch. Otherwise, the driver throws a BatchUpdateException.

jdbcCollection
Specifies the collection ID for the packages that are used by an instance of the IBM Data Server Driver for JDBC and SQLJ at run time. The data type of jdbcCollection is String. The default is NULLID.

This property is used with the DB2Binder -collection option. The DB2Binder utility must have previously bound IBM Data Server Driver for JDBC and SQLJ packages at the server using a -collection value that matches the jdbcCollection value.

The jdbcCollection setting does not determine the collection that is used for SQLJ applications. For SQLJ, the collection is determined by the -collection option of the SQLJ customizer.

jdbcCollection does not apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

maxRowsetSize
Specifies the maximum number of bytes that are used for rowset buffering for each statement, when the IBM Data Server Driver for JDBC and SQLJ uses multiple-row FETCH for cursors. The data type of this property is int. The default is 32767.

maxRowsetSize applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

reportLongTypes
Specifies whether DatabaseMetaData methods report LONG VARCHAR and LONG VARGRAPHIC column data types as long data types. The data type of this property is short. Possible values are:

com.ibm.db2.jcc.DB2BaseDataSource.NO (2) or com.ibm.db2.jcc.DB2BaseDataSource.NOT_SET (0)
Specifies that DatabaseMetaData methods that return information about a LONG VARCHAR or LONG VARGRAPHIC column return java.sql.Types.VARCHAR in the DATA_TYPE column and VARCHAR or VARGRAPHIC in the TYPE_NAME column of the result set. This is the default for DB2 for z/OS Version 9 or later.

com.ibm.db2.jcc.DB2BaseDataSource.YES (1)
Specifies that DatabaseMetaData methods that return information about a LONG VARCHAR or LONG VARGRAPHIC column return java.sql.Types.LONGVARCHAR in the DATA_TYPE column and LONG VARCHAR or LONG VARGRAPHIC in the TYPE_NAME column of the result set.

sendCharInputsUTF8

Specifies whether the IBM Data Server Driver for JDBC and SQLJ converts character input data to the CCSID of the DB2 for z/OS database server, or sends the data in UTF-8 encoding for conversion by the database server. sendCharInputsUTF8 applies to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS database servers only. The data type of this property is int. If this property is also set at the driver level (db2.jcc.sendCharInputsUTF8), this value overrides the driver-level value.

Possible values are:

com.ibm.db2.jcc.DB2BaseDataSource.NO (2)

Specifies that the IBM Data Server Driver for JDBC and SQLJ converts character input data to the target encoding before the data is sent to the DB2 for z/OS database server. com.ibm.db2.jcc.DB2BaseDataSource.NO is the default.

com.ibm.db2.jcc.DB2BaseDataSource.YES (1)

Specifies that the IBM Data Server Driver for JDBC and SQLJ sends character input data to the DB2 for z/OS database server in UTF-8 encoding. The database server converts the data from UTF-8 encoding to the target CCSID.

Specify com.ibm.db2.jcc.DB2BaseDataSource.YES only if conversion to the target CCSID by the SDK for Java causes character conversion problems. The most common problem occurs when you use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to insert a Unicode line feed character (U+000A) into a table column that has CCSID 37, and then retrieve that data from a non-z/OS client. If the SDK for Java does the conversion during insertion of the character into the column, the line feed character is converted to the EBCDIC new line character X'15'. However, during retrieval, some SDKs for Java on operating systems other than z/OS convert the X'15' character to the Unicode next line character (U+0085) instead of the line feed character (U+000A). The next line character causes unexpected behavior for some XML parsers. If you set sendCharInputsUTF8 to com.ibm.db2.jcc.DB2BaseDataSource.YES, the DB2 for z/OS database server converts the U+000A character to the EBCDIC line feed character X'25' during insertion into the column, so the character is always retrieved as a line feed character.

Conversion of data to the target CCSID on the database server might cause the IBM Data Server Driver for JDBC and SQLJ to use more memory than conversion by the driver. The driver allocates memory for conversion of character data from the source encoding to the encoding of the data that it sends to the database server. The amount of space that the driver allocates for character data that is sent to a table column is based on the maximum possible length of the data. UTF-8 data can require up to three bytes for each character. Therefore, if the driver sends UTF-8 data to the database server, the driver needs to allocate three times the maximum number of characters in the input data. If the driver does the conversion, and the target CCSID is a single-byte CCSID, the driver needs to allocate only the maximum number of characters in the input data.

sessionTimeZone

Specifies the setting for the CURRENT SESSION TIME ZONE special register. The data type of this property is String.

The sessionTimeZone value is a time zone value that is in the format of *sth:tm*. *s* is the sign, *th* is the time zone hour, and *tm* is time zone minutes. The range of valid values is -12:59 to +14:00.

sqljEnableClassLoaderSpecificProfiles
Specifies whether the IBM Data Server Driver for JDBC and SQLJ allows using and loading of SQLJ profiles with the same Java name in multiple J2EE application (.ear) files. The data type of this property is boolean. The default is `false`. sqljEnableClassLoaderSpecificProfiles is a DataSource property. This property is primarily intended for use with WebSphere Application Server.

ssid
Specifies the name of the local DB2 for z/OS subsystem to which a connection is established using IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. The data type of this property is String.

The ssid property overrides the db2.jcc.ssid configuration property.

ssid can be the subsystem name for a local subsystem or a group attachment name or subgroup attachment name.

Specification of a single local subsystem name allows more than one subsystem on a single LPAR to be accessed as a local subsystem for connections that use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

Specification of a group attachment name or subgroup attachment name allows failover processing to occur if a data sharing group member fails. If the DB2 subsystem to which an application is connected fails, the connection terminates. However, when new connections use that group attachment name or subgroup attachment name, DB2 for z/OS uses group or subgroup attachment processing to find an active DB2 subsystem to which to connect.

ssid applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS.

useIdentityValLocalForAutoGeneratedKeys
Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses only the SQL built-in function IDENTITY_VAL_LOCAL to determine automatically generated key values. The data type of this property is boolean. Possible values are:

true Specifies that the IBM Data Server Driver for JDBC and SQLJ always uses the SQL built-in function IDENTITY_VAL_LOCAL to determine automatically generated key values. The driver uses IDENTITY_VAL_LOCAL even if it is possible to use SELECT FROM INSERT.

Specify `true` if the target data server supports SELECT FROM INSERT, but the target objects do not. For example, SELECT FROM INSERT is not valid for a table on which a trigger is defined.

false Specifies that the IBM Data Server Driver for JDBC and SQLJ determines whether to use SELECT FROM INSERT or IDENTITY_VAL_LOCAL to determine automatically generated keys. `false` is the default.

useRowsetCursor
Specifies whether the IBM Data Server Driver for JDBC and SQLJ always uses multiple-row FETCH for scrollable cursors if the data source supports multiple-row fetch. The data type of this property is boolean.

This property applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, or to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS. If the enableRowsetSupport property is not set, the default for useRowsetCursor is `true`. If the enableRowsetSupport property is set, the useRowsetCursor property is not used.

Applications that use the JDBC 1 technique for performing positioned update or delete operations should set useRowSetCursor to `false`. Those applications do not operate properly if the IBM Data Server Driver for JDBC and SQLJ uses multiple-row FETCH.

xmlFormat
Specifies the format that is used to send XML data to the data server or retrieve XML data from the data server. The XML format cannot be modified after a connection is established. Possible values are:

com.ibm.db2.jcc.DB2BaseDataSource.XML_FORMAT_NOT_SET (-Integer.MAX_VALUE)
Specifies that binary XML format (Extensible Dynamic Binary XML DB2 Client/Server Binary XML Format) is used if the data server supports it. If the data server does not support binary XML format, textual XML format is used. This is the default.

com.ibm.db2.jcc.DB2BaseDataSource.XML_FORMAT_TEXTUAL (0)
Specifies that the XML textual format is used.

com.ibm.db2.jcc.DB2BaseDataSource.XML_FORMAT_BINARY (1)
Specifies that the binary XML format is used.

When binary XML is used, the XML data that is passed to the IBM Data Server Driver for JDBC and SQLJ cannot refer to external entities, internal entities, or internal DTDs. External DTDs are supported only if those DTDs were previously registered in the data source.

IBM Data Server Driver for JDBC and SQLJ properties for IBM Informix

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply only to IBM Informix databases. Those properties correspond to IBM Informix environment variables.

Properties that are shown in uppercase characters in the following information must be specified in uppercase. For those properties, getXXX and setXXX methods are formed by prepending the uppercase property name with get or set. For example:

```
boolean dbDate = DB2BaseDateSource.getDBDATE();
```

The IBM Informix-specific properties are:

DBANSIWARN
Specifies whether the IBM Data Server Driver for JDBC and SQLJ instructs the IBM Informix database to return an SQLWarning to the application if an SQL statement does not use ANSI-standard syntax. The data type of this property is boolean. Possible values are:

false or 0
Do not send a value to the IBM Informix database that instructs the database to return an SQLWarning to the application if an SQL statement does not use ANSI-standard syntax. This is the default.

true or 1
Send a value to the IBM Informix database that instructs the database to return an SQLWarning to the application if an SQL statement does not use ANSI-standard syntax.

You can use the DBANSIWARN IBM Data Server Driver for JDBC and SQLJ property to set the DBANSIWARN IBM Informix property, but you cannot use the DBANSIWARN IBM Data Server Driver for JDBC and SQLJ property to reset the DBANSIWARN IBM Informix property.

DBDATE
Specifies the end-user format of DATE values. The data type of this property is String. Possible values are in the description of the DBDATE environment variable in *IBM Informix Guide to SQL: Reference*.

The default value is "Y4MD-".

DBPATH
Specifies a colon-separated list of values that identify the database servers that contain databases. The date type of this property is String. Each value can be:
- A full path name
- A relative path name
- The server name of an IBM Informix database server
- A server name and full path name

The default ".".

DBSPACETEMP
Specifies a comma-separated or colon-separated list of existing dbspaces in which temporary tables are placed. The data type of this property is String.

If this property is not set, no value is sent to the server. The value for the DBSPACETEMP environment variable is used.

DBTEMP
Specifies the full path name of an existing directory in which temporary files and temporary tables are placed. The data type of this property is String. The default is "/tmp".

DBUPSPACE
Specifies the maximum amount of system disk space and maximum amount of memory, in kilobytes, that the UPDATE STATISTICS statement can use when it constructs multiple column distributions simultaneously. The data type of this property is String.

The format of DBUPSPACE is "*maximum-disk-space*:*maximum-memory*".

If this property is not set, no value is sent to the server. The value for the DBUPSPACE environment variable is used.

DB_LOCALE
Specifies the database locale, which the database server uses to process locale-sensitive data. The data type of this property is String. Valid values are the same as valid values for the DB_LOCALE environment variable. The default value is null.

DELIMIDENT
Specifies whether delimited SQL identifiers can be used in an application. The data type of this property is boolean. Possible values are:

false The application cannot contain delimited SQL identifiers. Double quotation marks (") or single quotation marks (') delimit literal strings. This is the default.

true The application can contain delimited SQL identifiers. Delimited SQL identifiers must be enclosed in double quotation marks ("). Single quotation marks (') delimit literal strings.

IFX_DIRECTIVES
Specifies whether the optimizer allows query optimization directives from within a query. The data type of this property is String. Possible values are:

"1" or "ON"
Optimization directives are accepted.

"0" or "OFF"
Optimization directives are not accepted.

If this property is not set, no value is sent to the server. The value for the IFX_DIRECTIVES environment variable is used.

IFX_EXTDIRECTIVES
Specifies whether the optimizer allows external query optimization directives from the sysdirectives system catalog table to be applied to queries in existing applications. Possible values are:

"1" or "ON"
External query optimization directives are accepted.

"0" or "OFF"
External query optimization are not accepted.

If this property is not set, no value is sent to the server. The value for the IFX_EXTDIRECTIVES environment variable is used.

IFX_UPDDESC
Specifies whether a DESCRIBE of an UPDATE statement is permitted. The data type of this property is String.

Any non-null value indicates that a DESCRIBE of an UPDATE statement is permitted. The default is "1".

IFX_XASTDCOMPLIANCE_XAEND
Specifies whether global transactions are freed only after an explicit rollback, or after any rollback. The data type of this property is String. Possible values are:

"0" Global transactions are freed only after an explicit rollback. This behavior conforms to the X/Open XA standard.

"1" Global transactions are freed after any rollback.

If this property is not set, no value is sent to the server. The value for the IFX_XASTDCOMPLIANCE_XAEND environment variable is used.

INFORMIXOPCACHE
Specifies the size of the memory cache, in kilobytes, for the staging-area blobspace of the client application. The data type of this property is String. A value of "0" indicates that the cache is not used.

If this property is not set, no value is sent to the server. The value for the INFORMIXOPCACHE environment variable is used.

INFORMIXSTACKSIZE
Specifies the stack size, in kilobytes, that the database server uses for the primary thread of a client session. The data type of this property is String.

If this property is not set, no value is sent to the server. The value for the INFORMIXSTACKSIZE environment variable is used.

NODEFDAC

Specifies whether the database server prevents default table privileges (SELECT, INSERT, UPDATE, and DELETE) from being granted to PUBLIC when a new table is created during the current session, in a database that is not ANSI compliant. The data type of this property is String. Possible values are:

"yes" The database server prevents default table privileges from being granted to PUBLIC when a new table is created during the current session, in a database that is not ANSI compliant.

"no" The database server does not prevent default table privileges from being granted to PUBLIC when a new table is created during the current session, in a database that is not ANSI compliant. This is the default.

OPTCOMPIND

Specifies the preferred method for performing a join operation on an ordered pair of tables. The data type of this property is String. Possible values are:

"0" The optimizer chooses a nested-loop join, where possible, over a sort-merge join or a hash join.

"1" When the isolation level is repeatable read, the optimizer chooses a nested-loop join, where possible, over a sort-merge join or a hash join. When the isolation level is not repeatable read, the optimizer chooses a join method based on costs.

"2" The optimizer chooses a join method based on costs, regardless of the transaction isolation mode.

If this property is not set, no value is sent to the server. The value for the OPTCOMPIND environment variable is used.

OPTOFC

Specifies whether to enable optimize-OPEN-FETCH-CLOSE functionality. The data type of this property is String. Possible values are:

"0" Disable optimize-OPEN-FETCH-CLOSE functionality for all threads of applications.

"1" Enable optimize-OPEN-FETCH-CLOSE functionality for all cursors in all threads of applications.

If this property is not set, no value is sent to the server. The value for the OPTOFCD environment variable is used.

PDQPRIORITY

Specifies the degree of parallelism that the database server uses. The PDQPRIORITY value affects how the database server allocates resources, including memory, processors, and disk reads. The data type of this property is String. Possible values are:

"HIGH"
When the database server allocates resources among all users, it gives as many resources as possible to queries.

"LOW" or "1"
The database server fetches values from fragmented tables in parallel.

"OFF" or "0"
Parallel processing is disabled.

If this property is not set, no value is sent to the server. The value for the PDQPRIORITY environment variable is used.

PSORT_DBTEMP
Specifies the full path name of a directory in which the database server writes temporary files that are used for a sort operation. The data type of this property is String.

If this property is not set, no value is sent to the server. The value for the PSORT_DBTEMP environment variable is used.

PSORT_NPROCS
Specifies the maximum number of threads that the database server can use to sort a query. The data type of this property is String. The maximum value of PSORT_NPROCS is "10".

If this property is not set, no value is sent to the server. The value for the PSORT_NPROCS environment variable is used.

STMT_CACHE
Specifies whether the shared-statement cache is enabled. The data type of this property is String. Possible values are:

"0" The shared-statement cache is disabled.

"1" A 512 KB shared-statement cache is enabled.

If this property is not set, no value is sent to the server. The value for the STMT_CACHE environment variable is used.

dumpPool
Specifies the types of statistics on global transport pool events that are written, in addition to summary statistics. The global transport pool is used for the connection concentrator and Sysplex workload balancing.

The data type of dumpPool is int. dumpPoolStatisticsOnSchedule and dumpPoolStatisticsOnScheduleFile must also be set for writing statistics before any statistics are written.

You can specify one or more of the following types of statistics with the db2.jcc.dumpPool property:
- `DUMP_REMOVE_OBJECT` (hexadecimal: X'01', decimal: 1)
- `DUMP_GET_OBJECT` (hexadecimal: X'02', decimal: 2)
- `DUMP_WAIT_OBJECT` (hexadecimal: X'04', decimal: 4)
- `DUMP_SET_AVAILABLE_OBJECT` (hexadecimal: X'08', decimal: 8)
- `DUMP_CREATE_OBJECT` (hexadecimal: X'10', decimal: 16)
- `DUMP_SYSPLEX_MSG` (hexadecimal: X'20', decimal: 32)
- `DUMP_POOL_ERROR` (hexadecimal: X'80', decimal: 128)

To trace more than one type of event, add the values for the types of events that you want to trace. For example, suppose that you want to trace DUMP_GET_OBJECT and DUMP_CREATE_OBJECT events. The numeric equivalents of these values are 2 and 16, so you specify 18 for the dumpPool value.

The default is 0, which means that only summary statistics for the global transport pool are written.

This property does not have a setXXX or a getXXX method.

dumpPoolStatisticsOnSchedule
Specifies how often, in seconds, global transport pool statistics are written to

the file that is specified by dumpPoolStatisticsOnScheduleFile. The global transport object pool is used for the connection concentrator and Sysplex workload balancing.

The default is -1. -1 means that global transport pool statistics are not written.

This property does not have a setXXX or a getXXX method.

dumpPoolStatisticsOnScheduleFile
Specifies the name of the file to which global transport pool statistics are written. The global transport pool is used for the connection concentrator and Sysplex workload balancing.

If dumpPoolStatisticsOnScheduleFile is not specified, global transport pool statistics are not written.

This property does not have a setXXX or a getXXX method.

maxTransportObjectIdleTime
Specifies the amount of time in seconds that an unused transport object stays in a global transport object pool before it can be deleted from the pool. Transport objects are used for the connection concentrator and Sysplex workload balancing.

The default value for maxTransportObjectIdleTime is 60. Setting maxTransportObjectIdleTime to a value less than 0 causes unused transport objects to be deleted from the pool immediately. Doing this is **not** recommended because it can cause severe performance degradation.

This property does not have a setXXX or a getXXX method.

maxTransportObjectWaitTime
Specifies the maximum amount of time in seconds that an application waits for a transport object if the maxTransportObjects value has been reached. Transport objects are used for the connection concentrator and Sysplex workload balancing. When an application waits for longer than the maxTransportObjectWaitTime value, the global transport object pool throws an SQLException.

The default value for maxTransportObjectWaitTime is -1. Any negative value means that applications wait forever.

This property does not have a setXXX or a getXXX method.

minTransportObjects
Specifies the lower limit for the number of transport objects in a global transport object pool for the connection concentrator and Sysplex workload balancing. When a JVM is created, there are no transport objects in the pool. Transport objects are added to the pool as they are needed. After the minTransportObjects value is reached, the number of transport objects in the global transport object pool never goes below the minTransportObjects value for the lifetime of that JVM.

The default value for minTransportObjects is 0. Any value that is less than or equal to 0 means that the global transport object pool can become empty.

This property does not have a setXXX or a getXXX method.

IBM Data Server Driver for JDBC and SQLJ configuration properties

The IBM Data Server Driver for JDBC and SQLJ configuration properties have driver-wide scope.

The following table summarizes the configuration properties and corresponding Connection or DataSource properties, if they exist.

Table 14-7. Summary of Configuration properties and corresponding Connection and DataSource properties

Configuration property name	Connection or DataSource property name: com.ibm.db2.jcc.DB2BaseDataSource. ...	Notes
db2.jcc.accountingInterval	accountingInterval	1 on page 14-54, 4 on page 14-54
db2.jcc.allowSqljDuplicateStaticQueries		4 on page 14-54
db2.jcc.charOutputSize	charOutputSize	1 on page 14-54, 4 on page 14-54
db2.jcc.currentSchema	currentSchema	1 on page 14-54, 4 on page 14-54, 6 on page 14-54
db2.jcc.override.currentSchema	currentSchema	2 on page 14-54, 4 on page 14-54, 6 on page 14-54
db2.jcc.currentSQLID	currentSQLID	1 on page 14-54, 4 on page 14-54
db2.jcc.override.currentSQLID	currentSQLID	2 on page 14-54, 4 on page 14-54
db2.jcc.defaultSQLState		4 on page 14-54
db2.jcc.disableSQLJProfileCaching		4 on page 14-54
db2.jcc.dumpPool	dumpPool	1 on page 14-54, 3 on page 14-54, 4 on page 14-54, 5 on page 14-54
db2.jcc.dumpPoolStatisticsOnSchedule	dumpPoolStatisticsOnSchedule	1 on page 14-54, 3 on page 14-54, 4 on page 14-54, 5 on page 14-54
db2.jcc.dumpPoolStatisticsOnScheduleFile	dumpPoolStatisticsOnScheduleFile	1 on page 14-54, 3 on page 14-54, 4 on page 14-54, 5 on page 14-54
db2.jcc.jmxEnabled		4 on page 14-54, 5 on page 14-54, 6 on page 14-54
db2.jcc.lobOutputSize		4 on page 14-54
db2.jcc.maxRefreshInterval		4 on page 14-54, 5 on page 14-54, 6 on page 14-54
db2.jcc.maxTransportObjectIdleTime	maxTransportObjectIdleTime	1 on page 14-54, 4 on page 14-54, 5 on page 14-54
db2.jcc.maxTransportObjectWaitTime	maxTransportObjectWaitTime	1 on page 14-54, 4 on page 14-54, 5 on page 14-54
db2.jcc.maxTransportObjects	maxTransportObjects	1 on page 14-54, 4 on page 14-54, 5 on page 14-54
db2.jcc.minTransportObjects	minTransportObjects	1 on page 14-54, 4 on page 14-54, 5 on page 14-54
db2.jcc.outputDirectory		6 on page 14-54
db2.jcc.pkList	pkList	1 on page 14-54, 4 on page 14-54
db2.jcc.planName	planName	1 on page 14-54, 4 on page 14-54
db2.jcc.progressiveStreaming	progressiveStreaming	1 on page 14-54, 4 on page 14-54, 5 on page 14-54, 6 on page 14-54
db2.jcc.override.progressiveStreaming	progressiveStreaming	2 on page 14-54, 4 on page 14-54, 5 on page 14-54, 6 on page 14-54
db2.jcc.rollbackOnShutdown		4 on page 14-54
db2.jcc.sendCharInputsUTF8	sendCharInputsUTF8	4 on page 14-54

Table 14-7. Summary of Configuration properties and corresponding Connection and DataSource properties (continued)

Configuration property name	Connection or DataSource property name: com.ibm.db2.jcc.DB2BaseDataSource. ...	Notes
db2.jcc.sqljToolsExitJVMOnCompletion		4, 6
db2.jcc.sqljUncustomizedWarningOrException		4, 6
db2.jcc.ssid	ssid	1, 4
db2.jcc.traceDirectory	traceDirectory	1, 4, 5, 6
db2.jcc.override.traceDirectory	traceDirectory	2, 4, 5, 6
db2.jcc.traceFile	traceFile	1, 4, 5, 6
db2.jcc.override.traceFile	traceFile	2, 4, 5, 6
db2.jcc.traceFileAppend	traceFileAppend	1, 4, 5, 6
db2.jcc.override.traceFileAppend	traceFileAppend	2, 4, 5, 6
db2.jcc.traceLevel	traceLevel	1, 4, 5, 6
db2.jcc.override.traceLevel	traceLevel	2, 4, 5, 6
db2.jcc.tracePolling		4, 5, 6
db2.jcc.tracePollingInterval		4, 5, 6
db2.jcc.t2zosTraceFile		4
db2.jcc.t2zosTraceBufferSize		4
db2.jcc.t2zosTraceWrap		4
db2.jcc.useCcsid420ShapedConverter		4

Note:

1. The Connection or DataSource property setting overrides the configuration property setting. The configuration property provides a default value for the Connection or DataSource property.
2. The configuration property setting overrides the Connection or DataSource property.
3. The corresponding Connection or DataSource property is defined only for IBM Informix.
4. The configuration property applies to DB2 for z/OS.
5. The configuration property applies to IBM Informix.
6. The configuration property applies to DB2 Database for Linux, UNIX, and Windows.

The meanings of the configuration properties are:

db2.jcc.accountingInterval

Specifies whether IDS accounting records are produced at commit points or on termination of the physical connection to the data source. If the value of db2.jcc.accountingInterval is COMMIT, IDS accounting records are produced at commit points. For example:

```
db2.jcc.accountingInterval=COMMIT
```

Otherwise, accounting records are produced on termination of the physical connection to the data source.

db2.jcc.accountingInterval applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. db2.jcc.accountingInterval is not applicable to connections under CICS or IMS, or for Java stored procedures.

You can override db2.jcc.accountingInterval by setting the `accountingInterval` property for a Connection or DataSource object.

This configuration property applies only to DB2 for z/OS.

db2.jcc.allowSqljDuplicateStaticQueries
Specifies whether multiple open iterators on a single SELECT statement in an SQLJ application are allowed under IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

To enable this support, set db2.jcc.allowSqljDuplicateStaticQueries to YES or true.

db2.jcc.charOutputSize
Specifies the maximum number of bytes to use for INOUT or OUT stored procedure parameters that are registered as `Types.CHAR`.

Because DESCRIBE information for stored procedure INOUT and OUT parameters is not available at run time, by default, the IBM Data Server Driver for JDBC and SQLJ sets the maximum length of each character INOUT or OUT parameter to 32767. For stored procedures with many `Types.CHAR` parameters, this maximum setting can result in allocation of much more storage than is necessary.

To use storage more efficiently, set db2.jcc.charOutputSize to the largest expected length for any `Types.CHAR` INOUT or OUT parameter.

db2.jcc.charOutputSize has no effect on INOUT or OUT parameters that are registered as `Types.VARCHAR` or `Types.LONGVARCHAR`. The driver uses the default length of 32767 for `Types.VARCHAR` and `Types.LONGVARCHAR` parameters.

The value that you choose for db2.jcc.charOutputSize needs to take into account the possibility of expansion during character conversion. Because the IBM Data Server Driver for JDBC and SQLJ has no information about the server-side CCSID that is used for output parameter values, the driver requests the stored procedure output data in UTF-8 Unicode. The db2.jcc.charOutputSize value needs to be the maximum number of bytes that are needed after the parameter value is converted to UTF-8 Unicode. UTF-8 Unicode characters can require up to three bytes. (The euro symbol is an example of a three-byte UTF-8 character.) To ensure that the value of db2.jcc.charOutputSize is large enough, if you have no information about the output data, set db2.jcc.charOutputSize to three times the defined length of the largest CHAR parameter.

This configuration property applies only to DB2 for z/OS.

db2.jcc.currentSchema or db2.jcc.override.currentSchema
Specifies the default schema name that is used to qualify unqualified database objects in dynamically prepared SQL statements. This value of this property sets the value in the CURRENT SCHEMA special register on the database server. The schema name is case-sensitive, and must be specified in uppercase characters.

This configuration property applies only to DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows.

db2.jcc.currentSQLID or db2.jcc.override.currentSQLID
Specifies:

- The authorization ID that is used for authorization checking on dynamically prepared CREATE, GRANT, and REVOKE SQL statements.
- The owner of a table space, database, storage group, or synonym that is created by a dynamically issued CREATE statement.
- The implicit qualifier of all table, view, alias, and index names specified in dynamic SQL statements.

currentSQLID sets the value in the CURRENT SQLID special register on a DB2 for z/OS server. If the currentSQLID property is not set, the default schema name is the value in the CURRENT SQLID special register.

This configuration property applies only to DB2 for z/OS.

db2.jcc.decimalRoundingMode or db2.jcc.override.decimalRoundingMode
Specifies the rounding mode for assignment to decimal floating-point variables or DECFLOAT columns on DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows data servers.

Possible values are:

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_DOWN (1)
Rounds the value towards 0 (truncation). The discarded digits are ignored.

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_CEILING (2)
Rounds the value towards positive infinity. If all of the discarded digits are zero or if the sign is negative the result is unchanged other than the removal of the discarded digits. Otherwise, the result coefficient is incremented by 1.

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_HALF_EVEN (3)
Rounds the value to the nearest value; if the values are equidistant, rounds the value so that the final digit is even. If the discarded digits represents greater than half (0.5) of the value of one in the next left position then the result coefficient is incremented by 1. If they represent less than half, then the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise the result coefficient is unaltered if its rightmost digit is even, or is incremented by 1 if its rightmost digit is odd (to make an even digit).

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_HALF_UP (4)
Rounds the value to the nearest value; if the values are equidistant, rounds the value away from zero. If the discarded digits represent greater than or equal to half (0.5) of the value of one in the next left position then the result coefficient is incremented by 1. Otherwise the discarded digits are ignored.

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_FLOOR (6)
Rounds the value towards negative infinity. If all of the discarded digits are zero or if the sign is positive the result is unchanged other than the removal of discarded digits. Otherwise, the sign is negative and the result coefficient is incremented by 1.

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_UNSET (-2147483647)
No rounding mode was explicitly set. The IBM Data Server Driver for JDBC and SQLJ does not use the decimalRoundingMode to set the rounding mode on the database server. The rounding mode is ROUND_HALF_EVEN.

If you explicitly set the db2.jcc.decimalRoundingMode or db2.jcc.override.decimalRoundingMode value, that value updates the CURRENT DECFLOAT ROUNDING MODE special register value on a DB2 for z/OS data server.

If you explicitly set the db2.jcc.decimalRoundingMode or db2.jcc.override.decimalRoundingMode value, that value does not update the CURRENT DECFLOAT ROUNDING MODE special register value on a DB2 Database for Linux, UNIX, and Windows data server. If the value to which you

set db2.jcc.decimalRoundingMode or db2.jcc.override.decimalRoundingMode is not the same as the value of the CURRENT DECFLOAT ROUNDING MODE special register, an Exception is thrown. To change the data server value, you need to set that value with the decflt_rounding database configuration parameter.

decimalRoundingMode does not affect decimal value assignments. The IBM Data Server Driver for JDBC and SQLJ always rounds decimal values down.

db2.jcc.defaultSQLState
Specifies the SQLSTATE value that the IBM Data Server Driver for JDBC and SQLJ returns to the client for SQLException or SQLWarning objects that have null SQLSTATE values. This configuration property can be specified in the following ways:

db2.jcc.defaultSQLState
If db2.jcc.defaultSQLState is specified with no value, the IBM Data Server Driver for JDBC and SQLJ returns 'FFFFF'.

db2.jcc.defaultSQLState=*xxxxx*
xxxxx is the value that the IBM Data Server Driver for JDBC and SQLJ returns when the SQLSTATE value is null. If *xxxxx* is longer than five bytes, the driver truncates the value to five bytes. If *xxxxx* is shorter than five bytes, the driver pads *xxxxx* on the right with blanks.

If db2.jcc.defaultSQLState is not specified, the IBM Data Server Driver for JDBC and SQLJ returns a null SQLSTATE value.

This configuration property applies only to DB2 for z/OS.

db2.jcc.disableSQLJProfileCaching
Specifies whether serialized profiles are cached when the JVM under which their application is running is reset. db2.jcc.disableSQLJProfileCaching applies only to applications that run in a resettable JVM (applications that run in the CICS, IMS, or Java stored procedure environment), and use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. Possible values are:

YES SQLJ serialized profiles are not cached every time the JVM is reset, so that new versions of the serialized profiles are loaded when the JVM is reset. Use this option when an application is under development, and new versions of the application and its serialized profiles are produced frequently.

NO SQLJ serialized profiles are cached when the JVM is reset. NO is the default.

This configuration property applies only to DB2 for z/OS.

db2.jcc.dumpPool
Specifies the types of statistics on global transport pool events that are written, in addition to summary statistics. The global transport pool is used for the connection concentrator and Sysplex workload balancing.

db2.jcc.dumpPoolStatisticsOnSchedule and db2.jcc.dumpPoolStatisticsOnScheduleFile must also be set for writing statistics before any statistics are written.

You can specify one or more of the following types of statistics with the db2.jcc.dumpPool property:
- `DUMP_REMOVE_OBJECT` (hexadecimal: X'01', decimal: 1)
- `DUMP_GET_OBJECT` (hexadecimal: X'02', decimal: 2)

- DUMP_WAIT_OBJECT (hexadecimal: X'04', decimal: 4)
- DUMP_SET_AVAILABLE_OBJECT (hexadecimal: X'08', decimal: 8)
- DUMP_CREATE_OBJECT (hexadecimal: X'10', decimal: 16)
- DUMP_SYSPLEX_MSG (hexadecimal: X'20', decimal: 32)
- DUMP_POOL_ERROR (hexadecimal: X'80', decimal: 128)

To trace more than one type of event, add the values for the types of events that you want to trace. For example, suppose that you want to trace DUMP_GET_OBJECT and DUMP_CREATE_OBJECT events. The numeric equivalents of these values are 2 and 16, so you specify 18 for the db2.jcc.dumpPool value.

The default is 0, which means that only summary statistics for the global transport pool are written.

This configuration property applies only to DB2 for z/OS or IBM Informix.

db2.jcc.dumpPoolStatisticsOnSchedule
Specifies how often, in seconds, global transport pool statistics are written to the file that is specified by db2.jcc.dumpPoolStatisticsOnScheduleFile. The global transport object pool is used for the connection concentrator and Sysplex workload balancing.

The default is -1. -1 means that global transport pool statistics are not written.

This configuration property applies only to DB2 for z/OS or IBM Informix.

db2.jcc.dumpPoolStatisticsOnScheduleFile
Specifies the name of the file to which global transport pool statistics are written. The global transport pool is used for the connection concentrator and Sysplex workload balancing.

If db2.jcc.dumpPoolStatisticsOnScheduleFile is not specified, global transport pool statistics are not written.

This configuration property applies only to DB2 for z/OS or IBM Informix.

db2.jcc.jmxEnabled
Specifies whether the Java Management Extensions (JMX) is enabled for the IBM Data Server Driver for JDBC and SQLJ instance. JMX must be enabled before applications can use the remote trace controller.

Possible values are:

true or yes
Indicates that JMX is enabled.

Any other value
Indicates that JMX is disabled. This is the default.

db2.jcc.lobOutputSize
Specifies the number of bytes of storage that the IBM Data Server Driver for JDBC and SQLJ needs to allocate for output LOB values when the driver cannot determine the size of those LOBs. This situation occurs for LOB stored procedure output parameters. db2.jcc.lobOutputSize applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

The default value for db2.jcc.lobOutputSize is 1048576. For systems with storage limitations and smaller LOBs, set the db2.jcc.lobOutputSize value to a lower number.

For example, if you know that the output LOB size is at most 64000, set db2.jcc.lobOutputSize to 64000.

This configuration property applies only to DB2 for z/OS.

db2.jcc.maxRefreshInterval
For workload balancing, specifies the maximum amount of time in seconds between refreshes of the client copy of the server list. The default is 30. The minimum valid value is 1.

db2.jcc.maxTransportObjectIdleTime
Specifies the amount of time in seconds that an unused transport object stays in a global transport object pool before it can be deleted from the pool. Transport objects are used for the connection concentrator and Sysplex workload balancing.

The default value for db2.jcc.maxTransportObjectIdleTime is 60. Setting db2.jcc.maxTransportObjectIdleTime to a value less than 0 causes unused transport objects to be deleted from the pool immediately. Doing this is **not** recommended because it can cause severe performance degradation.

This configuration property applies only to DB2 for z/OS or IBM Informix.

db2.jcc.maxTransportObjects
Specifies the upper limit for the number of transport objects in a global transport object pool for the connection concentrator and Sysplex workload balancing. When the number of transport objects in the pool reaches the db2.jcc.maxTransportObjects value, transport objects that have not been used for longer than the db2.jcc.maxTransportObjectIdleTime value are deleted from the pool.

The default value for db2.jcc.maxTransportObjects is -1. Any value that is less than or equal to 0 means that there is no limit to the number of transport objects in the global transport object pool.

This configuration property applies only to DB2 for z/OS or IBM Informix.

db2.jcc.maxTransportObjectWaitTime
Specifies the maximum amount of time in seconds that an application waits for a transport object if the db2.jcc.maxTransportObjects value has been reached. Transport objects are used for the connection concentrator and Sysplex workload balancing. When an application waits for longer than the db2.jcc.maxTransportObjectWaitTime value, the global transport object pool throws an SQLException.

The default value for db2.jcc.maxTransportObjectWaitTime is -1. Any negative value means that applications wait forever.

This configuration property applies only to DB2 for z/OS or IBM Informix.

db2.jcc.minTransportObjects
Specifies the lower limit for the number of transport objects in a global transport object pool for the connection concentrator and Sysplex workload balancing. When a JVM is created, there are no transport objects in the pool. Transport objects are added to the pool as they are needed. After the db2.jcc.minTransportObjects value is reached, the number of transport objects in the global transport object pool never goes below the db2.jcc.minTransportObjects value for the lifetime of that JVM.

The default value for db2.jcc.minTransportObjects is 0. Any value that is less than or equal to 0 means that the global transport object pool can become empty.

This configuration property applies only to DB2 for z/OS or IBM Informix.

db2.jcc.outputDirectory
Specifies where the IBM Data Server Driver for JDBC and SQLJ stores temporary log or cache files.

If this property is set, the IBM Data Server Driver for JDBC and SQLJ stores the following files in the specified directory:

jccServerListCache.bin
Contains a copy of the primary and alternate server information for automatic client reroute in a DB2 pureScale environment.

This file applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 Database for Linux, UNIX, and Windows.

If db2.jcc.outputDirectory is not specified, the IBM Data Server Driver for JDBC and SQLJ searches for a directory that is specified by the java.io.tmpdir system property. If the java.io.tmpdir system property is also not specified, the driver uses only the in-memory cache for the primary and alternate server information. If a directory is specified, but jccServerListCache.bin cannot be accessed, the driver uses only the in-memory cache for the server list.

jccdiag.log
Contains diagnostic information that is written by the IBM Data Server Driver for JDBC and SQLJ.

If db2.jcc.outputDirectory is not specified, the IBM Data Server Driver for JDBC and SQLJ searches for a directory that is specified by the java.io.tmpdir system property. If the java.io.tmpdir system property is also not specified, the driver does not write diagnostic information to jccdiag.log. If a directory is specified, but jccdiag.log cannot be accessed, the driver does not write diagnostic information to jccdiag.log.

connlicj.bin
Contains information about IBM Data Server Driver for JDBC and SQLJ license verification, for direct connections to DB2 for z/OS. The IBM Data Server Driver for JDBC and SQLJ writes this file when server license verification is performed successfully for a data server. When a copy of the license verification information is stored at the client, performance of license verification on subsequent connections can be improved.

If db2.jcc.outputDirectory is not specified, the IBM Data Server Driver for JDBC and SQLJ searches for a directory that is specified by the java.io.tmpdir system property. If the java.io.tmpdir system property is also not specified, the driver does not store a copy of server license verification information at the client. If a directory is specified, but connlicj.bin cannot be accessed, the driver does not store a copy of server license verification information at the client.

The IBM Data Server Driver for JDBC and SQLJ does not create the directory. You must create the directory and assign the required file permissions.

db2.jcc.outputDirectory can specify an absolute path or a relative path. However, an absolute path is recommended.

db2.jcc.pkList
Specifies a package list that is used for the underlying RRSAF CREATE THREAD call when a JDBC or SQLJ connection to a data source is established.

Specify this property if you do not bind plans for your SQLJ programs or for the JDBC driver. If you specify this property, **do not specify db2.jcc.planName.**

db2.jcc.pkList applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. db2.jcc.pkList does not apply to applications that run under CICS or IMS, or to Java stored procedures. The JDBC driver ignores the db2.jcc.pkList setting in those cases.

Recommendation: Use db2.jcc.pkList instead of db2.jcc.planName.

The format of the package list is:

The default value of db2.jcc.pkList is NULLID.*.

If you specify the -collection parameter when you run `com.ibm.db2.jcc.DB2Binder`, the collection ID that you specify for IBM Data Server Driver for JDBC and SQLJ packages when you run `com.ibm.db2.jcc.DB2Binder` must also be in the package list for the db2.jcc.pkList property.

You can override db2.jcc.pkList by setting the `pkList` property for a Connection or DataSource object.

The following example specifies a package list for a IBM Data Server Driver for JDBC and SQLJ instance whose packages are in collection JDBCCID. SQLJ applications that are prepared under this driver instance are bound into collections SQLJCID1, SQLJCID2, or SQLJCID3.

```
db2.jcc.pkList=JDBCCID.*,SQLJCID1.*,SQLJCID2.*,SQLJCID3.*
```

This configuration property applies only to DB2 for z/OS.

db2.jcc.planName

Specifies a DB2 for z/OS plan name that is used for the underlying RRSAF CREATE THREAD call when a JDBC or SQLJ connection to a data source is established. Specify this property if you bind plans for your SQLJ programs and for the JDBC driver packages. If you specify this property, **do not specify db2.jcc.pkList.**

db2.jcc.planName applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. db2.jcc.planName does not apply to applications that run under CICS or IMS, or to Java stored procedures. The JDBC driver ignores the db2.jcc.planName setting in those cases.

If you do not specify this property or the db2.jcc.pkList property, the IBM Data Server Driver for JDBC and SQLJ uses the db2.jcc.pkList default value of NULLID.*.

If you specify db2.jcc.planName, you need to bind the packages that you produce when you run `com.ibm.db2.jcc.DB2Binder` into a plan whose name is the value of this property. You also need to bind all SQLJ packages into a plan whose name is the value of this property.

You can override db2.jcc.planName by setting the `planName` property for a Connection or DataSource object.

The following example specifies a plan name of MYPLAN for the IBM Data Server Driver for JDBC and SQLJ JDBC packages and SQLJ packages.

```
db2.jcc.planName=MYPLAN
```

This configuration property applies only to DB2 for z/OS.

db2.jcc.progressiveStreaming or db2.jcc.override.progressiveStreaming
Specifies whether the JDBC driver uses progressive streaming when progressive streaming is supported on the data source.

With progressive streaming, also known as dynamic data format, the data source dynamically determines the most efficient mode in which to return LOB or XML data, based on the size of the LOBs or XML objects.

Valid values are:

1 Use progressive streaming, if the data source supports it.

2 Do not use progressive streaming.

db2.jcc.rollbackOnShutdown
Specifies whether DB2 for z/OS forces a rollback operation and disables further operations on JDBC connections that are in a unit of work during processing of JVM shutdown hooks.

db2.jcc.rollbackOnShutdown applies to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity only.

db2.jcc.rollbackOnShutdown does not apply to the CICS, IMS, stored procedure, or WebSphere Application Server environments.

Possible values are:

yes or true
The IBM Data Server Driver for JDBC and SQLJ directs DB2 for z/OS to force a rollback operation and disables further operations on JDBC connections that are in a unit of work during processing of JVM shutdown hooks.

Any other value
The IBM Data Server Driver for JDBC and SQLJ takes no action with respect to rollback processing during processing of JVM shutdown hooks. This is the default.

This configuration property applies only to DB2 for z/OS.

db2.jcc.sendCharInputsUTF8
Specifies whether the IBM Data Server Driver for JDBC and SQLJ converts character input data to the CCSID of the DB2 for z/OS database server, or sends the data in UTF-8 encoding for conversion by the database server. db2.jcc.sendCharInputsUTF8 applies to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS database servers only. If this property is also set at the connection level, the connection-level setting overrides this value.

Possible values are:

no, false, or 2
Specifies that the IBM Data Server Driver for JDBC and SQLJ converts character input data to the target encoding before the data is sent to the DB2 for z/OS database server. This is the default.

yes, true, or 1
Specifies that the IBM Data Server Driver for JDBC and SQLJ sends

character input data to the DB2 for z/OS database server in UTF-8 encoding. The data source converts the data from UTF-8 encoding to the target CCSID.

Specify yes, true, or 1 only if conversion to the target CCSID by the SDK for Java causes character conversion problems. The most common problem occurs when you use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to insert a Unicode line feed character (U+000A) into a table column that has CCSID 37, and then retrieve that data from a non-z/OS client. If the SDK for Java does the conversion during insertion of the character into the column, the line feed character is converted to the EBCDIC new line character X'15'. However, during retrieval, some SDKs for Java on operating systems other than z/OS convert the X'15' character to the Unicode next line character (U+0085) instead of the line feed character (U+000A). The next line character causes unexpected behavior for some XML parsers. If you set db2.jcc.sendCharInputsUTF8 to yes, the DB2 for z/OS database server converts the U+000A character to the EBCDIC line feed character X'25' during insertion into the column, so the character is always retrieved as a line feed character.

Conversion of data to the target CCSID on the data source might cause the IBM Data Server Driver for JDBC and SQLJ to use more memory than conversion by the driver. The driver allocates memory for conversion of character data from the source encoding to the encoding of the data that it sends to the data source. The amount of space that the driver allocates for character data that is sent to a table column is based on the maximum possible length of the data. UTF-8 data can require up to three bytes for each character. Therefore, if the driver sends UTF-8 data to the data source, the driver needs to allocate three times the maximum number of characters in the input data. If the driver does the conversion, and the target CCSID is a single-byte CCSID, the driver needs to allocate only the maximum number of characters in the input data.

For example, any of the following settings for db2.jcc.sendCharInputsUTF8 causes the IBM Data Server Driver for JDBC and SQLJ to convert input character strings to UTF-8, rather than the target encoding, before sending the data to the data source:

```
db2.jcc.sendCharInputsUTF8=yes
db2.jcc.sendCharInputsUTF8=true
db2.jcc.sendCharInputsUTF8=1
```

This configuration property applies only to DB2 for z/OS.

db2.jcc.sqljToolsExitJVMOnCompletion
Specifies whether the Java programs that underlie SQLJ tools such as db2sqljcustomize and db2sqljbind issue the System.exit call on return to the calling programs.

Possible values are:

true Specifies that the Java programs that underlie SQLJ tools issue the System.exit call upon completion. `true` is the default.

false Specifies that the Java programs that underlie SQLJ tools do not issue the System.exit call.

db2.jcc.sqljUncustomizedWarningOrException
Specifies the action that the IBM Data Server Driver for JDBC and SQLJ takes

when an uncustomized SQLJ application runs.
db2.jcc.sqljUncustomizedWarningOrException can have the following values:

0 The IBM Data Server Driver for JDBC and SQLJ does not throw a Warning or Exception when an uncustomized SQLJ application is run. This is the default.

1 The IBM Data Server Driver for JDBC and SQLJ throws a Warning when an uncustomized SQLJ application is run.

2 The IBM Data Server Driver for JDBC and SQLJ throws an Exception when an uncustomized SQLJ application is run.

This configuration property applies only to DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows.

db2.jcc.traceDirectory or db2.jcc.override.traceDirectory
Enables the IBM Data Server Driver for JDBC and SQLJ trace for Java driver code, and specifies a directory into which trace information is written. When db2.jcc.override.traceDirectory is specified, trace information for multiple connections on the same DataSource is written to multiple files.

When db2.jcc.override.traceDirectory is specified, a connection is traced to a file named *file-name_origin_n*.

- *n* is the *n*th connection for a DataSource.
- If neither db2.jcc.traceFileName nor db2.jcc.override.traceFileName is specified, *file-name* is traceFile. If db2.jcc.traceFileName or db2.jcc.override.traceFileName is also specified, *file-name* is the value of db2.jcc.traceFileName or db2.jcc.override.traceFileName.
- *origin* indicates the origin of the log writer that is in use. Possible values of *origin* are:

  cpds The log writer for a DB2ConnectionPoolDataSource object.

  driver The log writer for a DB2Driver object.

  global The log writer for a DB2TraceManager object.

  sds The log writer for a DB2SimpleDataSource object.

  xads The log writer for a DB2XADataSource object.

The db2.jcc.override.traceDirectory property overrides the `traceDirectory` property for a Connection or DataSource object.

For example, specifying the following setting for db2.jcc.override.traceDirectory enables tracing of the IBM Data Server Driver for JDBC and SQLJ Java code to files in a directory named /SYSTEM/tmp:

```
db2.jcc.override.traceDirectory=/SYSTEM/tmp
```

You should set the trace properties under the direction of IBM Software Support.

db2.jcc.traceLevel or db2.jcc.override.traceLevel
Specifies what to trace.

The db2.jcc.override.traceLevel property overrides the `traceLevel` property for a Connection or DataSource object.

You specify one or more trace levels by specifying a decimal value. The trace levels are the same as the trace levels that are defined for the traceLevel property on a Connection or DataSource object.

To specify more than one trace level, do an OR (|) operation on the values, and specify the result in decimal in the db2.jcc.traceLevel or db2.jcc.override.traceLevel specification.

For example, suppose that you want to specify TRACE_DRDA_FLOWS and TRACE_CONNECTIONS for db2.jcc.override.traceLevel. TRACE_DRDA_FLOWS has a hexadecimal value of X'40'. TRACE_CONNECTION_CALLS has a hexadecimal value of X'01'. To specify both traces, do a bitwise OR operation on the two values, which results in X'41'. The decimal equivalent is 65, so you specify:

```
db2.jcc.override.traceLevel=65
```

db2.jcc.ssid

Specifies the DB2 for z/OS subsystem to which applications make connections with IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

The db2.jcc.ssid value can be the name of the local DB2 subsystem or a group attachment name or subgroup attachment name.

For example:

```
db2.jcc.ssid=DB2A
```

The ssid Connection and DataSource property overrides db2.jcc.ssid.

If you specify a group attachment name or subgroup attachment name, and the DB2 subsystem to which an application is connected fails, the connection terminates. However, when new connections use that group attachment name or subgroup attachment name, DB2 for z/OS uses group attachment or subgroup attachment processing to find an active DB2 subsystem to which to connect.

If you do not specify the db2.jcc.ssid property, the IBM Data Server Driver for JDBC and SQLJ uses the SSID value from the application defaults load module. When you install DB2 for z/OS, an application defaults load module is created in the *prefix*.SDSNEXIT data set and the *prefix*.SDSNLOAD data set. Other application defaults load modules might be created in other data sets for selected applications.

The IBM Data Server Driver for JDBC and SQLJ must load an application defaults load module before it can read the SSID value. z/OS searches data sets in the following places, and in the following order, for the application defaults load module:

1. Job pack area (JPA)
2. TASKLIB
3. STEPLIB or JOBLIB
4. LPA
5. Libraries in the link list

You need to ensure that if your system has more than one copy of the application defaults load module, z/OS finds the data set that contains the correct copy for the IBM Data Server Driver for JDBC and SQLJ first.

This configuration property applies only to DB2 for z/OS.

db2.jcc.traceFile or db2.jcc.override.traceFile

Enables the IBM Data Server Driver for JDBC and SQLJ trace for Java driver code, and specifies the name on which the trace file names are based.

Specify a fully qualified z/OS UNIX System Services file name for the db2.jcc.override.traceFile property value.

The db2.jcc.override.traceFile property overrides the traceFile property for a Connection or DataSource object.

For example, specifying the following setting for db2.jcc.override.traceFile enables tracing of the IBM Data Server Driver for JDBC and SQLJ Java code to a file named /SYSTEM/tmp/jdbctrace:

```
db2.jcc.override.traceFile=/SYSTEM/tmp/jdbctrace
```

You should set the trace properties under the direction of IBM Software Support.

db2.jcc.traceFileAppend or db2.jcc.override.traceFileAppend
Specifies whether to append to or overwrite the file that is specified by the db2.jcc.override.traceFile property. Valid values are true or false. The default is false, which means that the file that is specified by the traceFile property is overwritten.

The db2.jcc.override.traceFileAppend property overrides the traceFileAppend property for a Connection or DataSource object.

For example, specifying the following setting for db2.jcc.override.traceFileAppend causes trace data to be added to the existing trace file:

```
db2.jcc.override.traceFileAppend=true
```

You should set the trace properties under the direction of IBM Software Support.

db2.jcc.tracePolling
Indicates whether the IBM Data Server Driver for JDBC and SQLJ polls the global configuration file for changes in trace directives and modifies the trace behavior to match the new trace directives. Possible values are true or false. False is the default.

The IBM Data Server Driver for JDBC and SQLJ modifies the trace behavior at the beginning of the next polling interval after the configuration properties file is changed. If db2.jcc.tracePolling is set to true while an application is running, the trace is enabled, and information about all the PreparedStatement objects that were created by the application before the trace was enabled are dumped to the trace destination.

db2.jcc.tracePolling polls the following global configuration properties:
- db2.jcc.override.traceLevel
- db2.jcc.override.traceFile
- db2.jcc.override.traceDirectory
- db2.jcc.override.traceFileAppend

db2.jcc.tracePollingInterval
Specifies the interval, in seconds, for polling the IBM Data Server Driver for JDBC and SQLJ global configuration file for changes in trace directives. The property value is a positive integer. The default is 60. For the specified trace polling interval to be used, the db2.jcc.tracePollingInterval property must be set *before* the driver is loaded and initialized. Changes to db2.jcc.tracePollingInterval after the driver is loaded and initialized have no effect.

db2.jcc.t2zosTraceFile
Enables the IBM Data Server Driver for JDBC and SQLJ trace for C/C++ native driver code for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity, and specifies the name on which the trace file names are based. This property is required for collecting trace data for C/C++ native driver code.

Specify a fully qualified z/OS UNIX System Services file name for the db2.jcct.t2zosTraceFile property value.

For example, specifying the following setting for db2.jcct.t2zosTraceFile enables tracing of the IBM Data Server Driver for JDBC and SQLJ C/C++ native code to a file named /SYSTEM/tmp/jdbctraceNative:

```
db2.jcc.t2zosTraceFile=/SYSTEM/tmp/jdbctraceNative
```

You should set the trace properties under the direction of IBM Software Support.

This configuration property applies only to DB2 for z/OS.

db2.jcc.t2zosTraceBufferSize

Specifies the size, in kilobytes, of a trace buffer in virtual storage that is used for tracing the processing that is done by the C/C++ native driver code. This value is also the maximum amount of C/C++ native driver trace information that can be collected.

Specify an integer between 64 (64 KB) and 4096 (4096 KB). The default is 256 (256 KB).

The JDBC driver determines the trace buffer size as shown in the following table:

Specified value (*n*)	Trace buffer size (KB)
<64	64
64<=*n*<128	64
128<=*n*<256	128
256<=*n*<512	256
512<=*n*<1024	512
1024<=*n*<2048	1024
2048<=*n*<4096	2048
n>=4096	4096

db2.jcc.t2zosTraceBufferSize is used only if the db2.jcc.t2zosTraceFile property is set.

Recommendation: To avoid a performance impact, specify a value of 1024 or less.

For example, to set a trace buffer size of 1024 KB, use this setting:

```
db2.jcc.t2zosTraceBufferSize=1024
```

You should set the trace properties under the direction of IBM Software Support.

This configuration property applies only to DB2 for z/OS.

db2.jcc.t2zosTraceWrap

Enables or disables wrapping of the SQLJ trace. db2.jcc.t2zosTraceWrap can have one of the following values:

1 Wrap the trace

0 Do not wrap the trace

The default is 1. This parameter is optional. For example:

```
DB2SQLJ_TRACE_WRAP=0
```

You should set db2.jcc.t2zosTraceWrap only under the direction of IBM Software Support.

This configuration property applies only to DB2 for z/OS.

db2.jcc.useCcsid420ShapedConverter
Specifies whether Arabic character data that is in EBCDIC CCSID 420 maps to Cp420S encoding.

db2.jcc.useCcsid420ShapedConverter applies only to connections to DB2 for z/OS database servers.

If the value of db2.jcc.useCcsid420ShapedConverter is `true`, CCSID 420 maps to Cp420S encoding. If the value of db2.jcc.useCcsid420ShapedConverter is `false`, CCSID 420 maps to Cp420 encoding. `false` is the default.

This configuration property applies only to DB2 for z/OS.

Driver support for JDBC APIs

The JDBC drivers that are supported by DB2 and IBM Informix database systems have different levels of support for JDBC methods.

The following tables list the JDBC interfaces and indicate which drivers supports them. The drivers and their supported platforms are:

Table 14-8. JDBC drivers for DB2 and IBM Informix database systems

JDBC driver name	Associated data source
IBM Data Server Driver for JDBC and SQLJ	DB2 Database for Linux, UNIX, and Windows, DB2 for z/OS, or IBM Informix
IBM Informix JDBC Driver (IBM Informix JDBC Driver)	IBM Informix

If a method has JDBC 2.0 and JDBC 3.0 forms, the IBM Data Server Driver for JDBC and SQLJ supports all forms. The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows supports only the JDBC 2.0 forms.

Table 14-9. Support for Array methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ1 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
free[2]	Yes	No	No
getArray	Yes	No	Yes
getBaseType	Yes	No	Yes
getBaseTypeName	Yes	No	Yes
getResultSet	Yes	No	Yes

Notes:

1. Under the IBM Data Server Driver for JDBC and SQLJ, `Array` methods are supported for connections to DB2 Database for Linux, UNIX, and Windows data sources only.
2. This is a JDBC 4.0 method.

Table 14-10. Support for BatchUpdateException methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Exception	Yes	Yes	Yes
getUpdateCounts	Yes	Yes	Yes

Table 14-11. Support for Blob methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
free[1]	Yes	No	No
getBinaryStream	Yes[2]	Yes	Yes
getBytes	Yes	Yes	Yes
length	Yes	Yes	Yes
position	Yes	Yes	Yes
setBinaryStream[3]	Yes	No	No
setBytes[3]	Yes	No	No
truncate[3]	Yes	No	No

Notes:

1. This is a JDBC 4.0 method.
2. Supported forms of this method include the following JDBC 4.0 form:
   `getBinaryStream(long pos, long length)`
3. For versions of the IBM Data Server Driver for JDBC and SQLJ before version 3.50, these methods cannot be used if a Blob is passed to a stored procedure as an IN or INOUT parameter, and the methods are used on the Blob in the stored procedure.

Table 14-12. Support for CallableStatement methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.sql.Statement	Yes	Yes	Yes
Methods inherited from java.sql.PreparedStatement	Yes[1]	Yes	Yes
getArray	No	No	No
getBigDecimal	Yes[3]	Yes	Yes
getBlob	Yes[3]	Yes	Yes
getBoolean	Yes[3]	Yes	Yes
getByte	Yes[3]	Yes	Yes
getBytes	Yes[3]	Yes	Yes
getClob	Yes[3]	Yes	Yes
getDate	Yes[3,4]	Yes[4]	Yes
getDouble	Yes[3]	Yes	Yes
getFloat	Yes[3]	Yes	Yes

Table 14-12. Support for CallableStatement methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
getInt	Yes[3]	Yes	Yes
getLong	Yes[3]	Yes	Yes
getObject	Yes[3,5]	Yes[5]	Yes
getRef	No	No	No
getRowId[2]	Yes	No	No
getShort	Yes[3]	Yes	Yes
getString	Yes[3]	Yes	Yes
getTime	Yes[3,4]	Yes[4]	Yes
getTimestamp	Yes[3,4]	Yes[4]	Yes
getURL	Yes	No	No
registerOutParameter	Yes[6]	Yes[6]	Yes[6]
setAsciiStream	Yes[7]	No	Yes
setBigDecimal	Yes[7]	No	Yes
setBinaryStream	Yes[7]	No	Yes
setBoolean	Yes[7]	No	Yes
setByte	Yes[7]	No	Yes
setBytes	Yes[7]	No	Yes
setCharacterStream	Yes[7]	No	Yes
setDate	Yes[7]	No	Yes
setDouble	Yes[7]	No	Yes
setFloat	Yes[7]	No	Yes
setInt	Yes[7]	No	Yes
setLong	Yes[7]	No	Yes
setNull	Yes[7,8]	No	Yes
setObject	Yes[7,]	No	Yes
setShort	Yes[7]	No	Yes
setString	Yes[7]	No	Yes
setTime	Yes[7]	No	Yes
setTimestamp	Yes[7]	No	Yes
setURL	Yes	No	No
wasNull	Yes	Yes	Yes

Table 14-12. Support for CallableStatement methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support

Notes:

1. The inherited getParameterMetaData method is not supported if the data source is DB2 for z/OS.
2. This is a JDBC 4.0 method.
3. The following forms of CallableStatement.getXXX methods are not supported if the data source is DB2 for z/OS:
   `get`*`XXX`*`(String` *`parameterName`*`)`
4. The database server does no timezone adjustment for datetime values. The JDBC driver adjusts a value for the local timezone after retrieving the value from the server if you specify a form of the getDate, getTime, or getTimestamp method that includes a java.util.Calendar parameter.
5. The following form of the getObject method is not supported:
   `getObject(int` *`parameterIndex`*`, java.util.Map` *`map`*`)`
6. The following form of the registerOutParameter method is not supported:
   `registerOutParameter(int` *`parameterIndex`*`, int` *`jdbcType`*`, String` *`typeName`*`)`
7. The following forms of CallableStatement.setXXX methods are not supported if the data source is DB2 for z/OS:
   `set`*`XXX`*`(String` *`parameterName`*`,...)`
8. The following form of setNull is not supported:
   `setNull(int` *`parameterIndex`*`, int` *`jdbcType`*`, String` *`typeName`*`)`

Table 14-13. Support for Clob methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
free[1]	Yes	No	No
getAsciiStream	Yes	Yes	Yes
getCharacterStream	Yes2	Yes	Yes
getSubString	Yes	Yes	Yes
length	Yes	Yes	Yes
position	Yes	Yes	Yes
setAsciiStream[3]	Yes	No	Yes
setCharacterStream[3]	Yes	No	Yes
setString[3]	Yes	No	Yes
truncate[3]	Yes	No	Yes

Notes:

1. This is a JDBC 4.0 method.
2. Supported forms of this method include the following JDBC 4.0 form:
   `getCharacterStream(long` *`pos`*`, long` *`length`*`)`
3. For versions of the IBM Data Server Driver for JDBC and SQLJ before version 3.50, these methods cannot be used if a Clob is passed to a stored procedure as an IN or INOUT parameter, and the methods are used on the Clob in the stored procedure.

Table 14-14. Support for Connection methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
clearWarnings	Yes	Yes	Yes
close	Yes	Yes	Yes
commit	Yes	Yes	Yes
createBlob[1]	Yes	No	No
createClob[1]	Yes	No	No
createStatement	Yes	Yes[2]	Yes
getAutoCommit	Yes	Yes	Yes
getCatalog	Yes	Yes	Yes
getClientInfo[1]	Yes	No	No
getHoldability	Yes	No	No
getMetaData	Yes	Yes	Yes
getTransactionIsolation	Yes	Yes	Yes
getTypeMap	No	No	Yes
getWarnings	Yes	Yes	Yes
isClosed	Yes	Yes	Yes
isReadOnly	Yes	Yes	Yes
isValid[1,3]	Yes	No	No
nativeSQL	Yes	Yes	Yes
prepareCall	Yes[4]	Yes	Yes
prepareStatement	Yes	Yes[2]	Yes
releaseSavepoint	Yes	No	No
rollback	Yes	Yes[2]	Yes
setAutoCommit	Yes	Yes	Yes
setCatalog	Yes	Yes	No
setClientInfo[1]	Yes	No	No
setReadOnly	Yes[5]	Yes	No
setSavepoint	Yes	No	No
setTransactionIsolation	Yes	Yes	Yes
setTypeMap	No	No	Yes

Notes:

1. This is a JDBC 4.0 method.
2. The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows does not support the JDBC 3.0 forms of this method.
3. Under IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, an SQLException is thrown if the *timeout* parameter value is less than 0. Under IBM Data Server Driver for JDBC and SQLJ type 2 connectivity, an SQLException is thrown if the if the *timeout* parameter value is not 0.
4. If the stored procedure in the CALL statement is on DB2 for z/OS, the parameters of the CALL statement cannot be expressions.
5. The driver does not use the setting. For the IBM Data Server Driver for JDBC and SQLJ, a connection can be set as read-only through the `readOnly` property for a Connection or DataSource object.

Table 14-15. Support for ConnectionEvent methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.util.EventObject	Yes	Yes	Yes
getSQLException	Yes	Yes	Yes

Table 14-16. Support for ConnectionEventListener methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
connectionClosed	Yes	Yes	Yes
connectionErrorOccurred	Yes	Yes	Yes

Table 14-17. Support for ConnectionPoolDataSource methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
getLoginTimeout	Yes	Yes	Yes
getLogWriter	Yes	Yes	Yes
getPooledConnection	Yes	Yes	Yes
setLoginTimeout	Yes[1]	Yes	Yes
setLogWriter	Yes	Yes	Yes

Note:

1. This method is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

Table 14-18. Support for DatabaseMetaData methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
allProceduresAreCallable	Yes	Yes	Yes
allTablesAreSelectable	Yes[1]	Yes	Yes[1]
dataDefinitionCausesTransactionCommit	Yes	Yes	Yes
dataDefinitionIgnoredInTransactions	Yes	Yes	Yes
deletesAreDetected	Yes	Yes	Yes
doesMaxRowSizeIncludeBlobs	Yes	Yes	Yes
getAttributes	Yes[2]	No	No
getBestRowIdentifier	Yes	Yes	Yes
getCatalogs	Yes	Yes	Yes
getCatalogSeparator	Yes	Yes	Yes
getCatalogTerm	Yes	Yes	Yes
getClientInfoProperties[6]	Yes	No	No

Table 14-18. Support for DatabaseMetaData methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
getColumnPrivileges	Yes	Yes	Yes
getColumns	Yes[7]	Yes[10]	Yes[10]
getConnection	Yes	Yes	Yes
getCrossReference	Yes	Yes	Yes
getDatabaseMajorVersion	Yes	No	No
getDatabaseMinorVersion	Yes	No	No
getDatabaseProductName	Yes	Yes	Yes
getDatabaseProductVersion	Yes	Yes	Yes
getDefaultTransactionIsolation	Yes	Yes	Yes
getDriverMajorVersion	Yes	Yes	Yes
getDriverMinorVersion	Yes	Yes	Yes
getDriverName	Yes[8]	Yes	Yes
getDriverVersion	Yes	Yes	Yes
getExportedKeys	Yes	Yes	Yes
getFunctionColumns[6]	Yes	No	No
getFunctions[6]	Yes	No	No
getExtraNameCharacters	Yes	Yes	Yes
getIdentifierQuoteString	Yes	Yes	Yes
getImportedKeys	Yes	Yes	Yes
getIndexInfo	Yes	Yes	Yes
getJDBCMajorVersion	Yes	No	No
getJDBCMinorVersion	Yes	No	No
getMaxBinaryLiteralLength	Yes	Yes	Yes
getMaxCatalogNameLength	Yes	Yes	Yes
getMaxCharLiteralLength	Yes	Yes	Yes
getMaxColumnNameLength	Yes	Yes	Yes
getMaxColumnsInGroupBy	Yes	Yes	Yes
getMaxColumnsInIndex	Yes	Yes	Yes
getMaxColumnsInOrderBy	Yes	Yes	Yes
getMaxColumnsInSelect	Yes	Yes	Yes
getMaxColumnsInTable	Yes	Yes	Yes
getMaxConnections	Yes	Yes	Yes
getMaxCursorNameLength	Yes	Yes	Yes
getMaxIndexLength	Yes	Yes	Yes
getMaxProcedureNameLength	Yes	Yes	Yes
getMaxRowSize	Yes	Yes	Yes

Table 14-18. Support for DatabaseMetaData methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
getMaxSchemaNameLength	Yes	Yes	Yes
getMaxStatementLength	Yes	Yes	Yes
getMaxStatements	Yes	Yes	Yes
getMaxTableNameLength	Yes	Yes	Yes
getMaxTablesInSelect	Yes	Yes	Yes
getMaxUserNameLength	Yes	Yes	Yes
getNumericFunctions	Yes	Yes	Yes
getPrimaryKeys	Yes	Yes	Yes
getProcedureColumns	Yes7 on page 14-78	Yes	Yes
getProcedures	Yes7 on page 14-78	Yes	Yes
getProcedureTerm	Yes	Yes	Yes
getResultSetHoldability	Yes	No	No
getRowIdLifetime[6]	Yes	No	No
getSchemas	Yes9 on page 14-78	Yes[10]	Yes[10]
getSchemaTerm	Yes	Yes	Yes
getSearchStringEscape	Yes	Yes	Yes
getSQLKeywords	Yes	Yes	Yes
getSQLStateType	Yes	No	No
getStringFunctions	Yes	Yes	Yes
getSuperTables	Yes[2]	No	No
getSuperTypes	Yes[2]	No	No
getSystemFunctions	Yes	Yes	Yes
getTablePrivileges	Yes	Yes	Yes
getTables	Yes	Yes[10]	Yes[10]
getTableTypes	Yes	Yes	Yes
getTimeDateFunctions	Yes	Yes	Yes
getTypeInfo	Yes	Yes	Yes
getUDTs	No	Yes[11]	Yes[11]
getURL	Yes	Yes	Yes
getUserName	Yes	Yes	Yes
getVersionColumns	Yes	Yes	Yes
insertsAreDetected	Yes	Yes	Yes
isCatalogAtStart	Yes	Yes	Yes
isReadOnly	Yes	Yes	Yes

Table 14-18. Support for DatabaseMetaData methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
locatorsUpdateCopy	Yes[3]	Yes	Yes[3]
nullPlusNonNullIsNull	Yes	Yes	Yes
nullsAreSortedAtEnd	Yes[4]	Yes	Yes[4]
nullsAreSortedAtStart	Yes	Yes	Yes
nullsAreSortedHigh	Yes[5]	Yes	Yes[5]
nullsAreSortedLow	Yes[1]	Yes	Yes[1]
othersDeletesAreVisible	Yes	Yes	Yes
othersInsertsAreVisible	Yes	Yes	Yes
othersUpdatesAreVisible	Yes	Yes	Yes
ownDeletesAreVisible	Yes	Yes	Yes
ownInsertsAreVisible	Yes	Yes	Yes
ownUpdatesAreVisible	Yes	Yes	Yes
storesLowerCaseIdentifiers	Yes[1]	Yes	Yes[1]
storesLowerCaseQuotedIdentifiers	Yes[4]	Yes	Yes[4]
storesMixedCaseIdentifiers	Yes	Yes	Yes
storesMixedCaseQuotedIdentifiers	Yes	Yes	Yes
storesUpperCaseIdentifiers	Yes[5]	Yes	Yes[5]
storesUpperCaseQuotedIdentifiers	Yes	Yes	Yes
supportsAlterTableWithAddColumn	Yes	Yes	Yes
supportsAlterTableWithDropColumn	Yes[1]	Yes	Yes[1]
supportsANSI92EntryLevelSQL	Yes	Yes	Yes
supportsANSI92FullSQL	Yes	Yes	Yes
supportsANSI92IntermediateSQL	Yes	Yes	Yes
supportsBatchUpdates	Yes	Yes	Yes
supportsCatalogsInDataManipulation	Yes[1]	Yes	Yes[1]
supportsCatalogsInIndexDefinitions	Yes	Yes	Yes
supportsCatalogsInPrivilegeDefinitions	Yes	Yes	Yes
supportsCatalogsInProcedureCalls	Yes[1]	Yes	Yes[1]
supportsCatalogsInTableDefinitions	Yes	Yes	Yes
SupportsColumnAliasing	Yes	Yes	Yes
supportsConvert	Yes	Yes	Yes
supportsCoreSQLGrammar	Yes	Yes	Yes
supportsCorrelatedSubqueries	Yes	Yes	Yes
supportsDataDefinitionAndDataManipulationTransactions	Yes	Yes	Yes
supportsDataManipulationTransactionsOnly	Yes	Yes	Yes
supportsDifferentTableCorrelationNames	Yes[4]	Yes	Yes[4]

Table 14-18. Support for DatabaseMetaData methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
supportsExpressionsInOrderBy	Yes	Yes	Yes
supportsExtendedSQLGrammar	Yes	Yes	Yes
supportsFullOuterJoins	Yes[3]	Yes	Yes[3]
supportsGetGeneratedKeys	Yes	No	No
supportsGroupBy	Yes	Yes	Yes
supportsGroupByBeyondSelect	Yes	Yes	Yes
supportsGroupByUnrelated	Yes	Yes	Yes
supportsIntegrityEnhancementFacility	Yes	Yes	Yes
supportsLikeEscapeClause	Yes	Yes	Yes
supportsLimitedOuterJoins	Yes	Yes	Yes
supportsMinimumSQLGrammar	Yes	Yes	Yes
supportsMixedCaseIdentifiers	Yes	Yes	Yes
supportsMixedCaseQuotedIdentifiers	Yes[3]	Yes	Yes[3]
supportsMultipleOpenResults	Yes[5]	No	Yes[5]
supportsMultipleResultSets	Yes[5]	Yes	Yes[5]
supportsMultipleTransactions	Yes	Yes	Yes
supportsNamedParameters	Yes	No	No
supportsNonNullableColumns	Yes	Yes	Yes
supportsOpenCursorsAcrossCommit	Yes[3]	Yes	Yes[3]
supportsOpenCursorsAcrossRollback	Yes	Yes	Yes
supportsOpenStatementsAcrossCommit	Yes[3]	Yes	Yes[3]
supportsOpenStatementsAcrossRollback	Yes[3]	Yes	Yes[3]
supportsOrderByUnrelated	Yes	Yes	Yes
supportsOuterJoins	Yes	Yes	Yes
supportsPositionedDelete	Yes	Yes	Yes
supportsPositionedUpdate	Yes	Yes	Yes
supportsResultSetConcurrency	Yes	Yes	Yes
supportsResultSetHoldability	Yes	No	No
supportsResultSetType	Yes	Yes	Yes
supportsSavepoints	Yes	No	Yes
supportsSchemasInDataManipulation	Yes	Yes	Yes
supportsSchemasInIndexDefinitions	Yes	Yes	Yes
supportsSchemasInPrivilegeDefinitions	Yes	Yes	Yes
supportsSchemasInProcedureCalls	Yes	Yes	Yes
supportsSchemasInTableDefinitions	Yes	Yes	Yes
supportsSelectForUpdate	Yes	Yes	Yes

Table 14-18. Support for DatabaseMetaData methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
supportsStoredProcedures	Yes	Yes	Yes
supportsSubqueriesInComparisons	Yes	Yes	Yes
supportsSubqueriesInExists	Yes	Yes	Yes
supportsSubqueriesInIns	Yes	Yes	Yes
supportsSubqueriesInQuantifieds	Yes	Yes	Yes
supportsSuperTables	Yes	No	No
supportsSuperTypes	Yes	No	No
supportsTableCorrelationNames	Yes	Yes	Yes
supportsTransactionIsolationLevel	Yes	Yes	Yes
supportsTransactions	Yes	Yes	Yes
supportsUnion	Yes	Yes	Yes
supportsUnionAll	Yes	Yes	Yes
updatesAreDetected	Yes	Yes	Yes
usesLocalFilePerTable	Yes	Yes	Yes
usesLocalFiles	Yes	Yes	Yes

Notes:

1. DB2 data sources return `false` for this method. IBM Informix data sources return `true`.
2. This method is supported for connections to DB2 Database for Linux, UNIX, and Windows and IBM Informix only.
3. Under the IBM Data Server Driver for JDBC and SQLJ, DB2 data sources and IBM Informix data sources return `true` for this method. Under the IBM Informix JDBC Driver, IBM Informix data sources return `false`.
4. Under the IBM Data Server Driver for JDBC and SQLJ, DB2 data sources and IBM Informix data sources return `false` for this method. Under the IBM Informix JDBC Driver, IBM Informix data sources return `true`.
5. DB2 data sources return `true` for this method. IBM Informix data sources return `false`.
6. This is a JDBC 4.0 method.
7. This method returns the additional column that is described by the JDBC 4.0 specification.
8. JDBC 3.0 and earlier implementations of the IBM Data Server Driver for JDBC and SQLJ return "IBM DB2 JDBC Universal Driver Architecture."

   The JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ returns "IBM Data Server Driver for JDBC and SQLJ."
9. The JDBC 4.0 form and previous forms of this method are supported.
10. The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows does not support the JDBC 3.0 form of this method.
11. The method can be executed, but it returns an empty ResultSet.

Table 14-19. Support for DataSource methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
getConnection	Yes	Yes	Yes
getLoginTimeout	Yes	Yes[1]	Yes

Table 14-19. Support for DataSource methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
getLogWriter	Yes	Yes	Yes
setLoginTimeout	Yes[2]	Yes[1]	Yes
setLogWriter	Yes	Yes	Yes

Notes:

1. The DB2 JDBC Type 2 Driver does not use this setting.
2. This method is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

Table 14-20. Support for DataTruncation methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Throwable	Yes	Yes	Yes
Methods inherited from java.sql.SQLException	Yes	Yes	Yes
Methods inherited from java.sql.SQLWarning	Yes	Yes	Yes
getDataSize	Yes	Yes	Yes
getIndex	Yes	Yes	Yes
getParameter	Yes	Yes	Yes
getRead	Yes	Yes	Yes
getTransferSize	Yes	Yes	Yes

Table 14-21. Support for Driver methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
acceptsURL	Yes	Yes	Yes
connect	Yes	Yes	Yes
getMajorVersion	Yes	Yes	Yes
getMinorVersion	Yes	Yes	Yes
getPropertyInfo	Yes	Yes	Yes
jdbcCompliant	Yes	Yes	Yes

Table 14-22. Support for DriverManager methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
deregisterDriver	Yes	Yes	Yes
getConnection	Yes	Yes	Yes
getDriver	Yes	Yes	Yes

Table 14-22. Support for DriverManager methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
getDrivers	Yes	Yes	Yes
getLoginTimeout	Yes	Yes[1]	Yes
getLogStream	Yes	Yes	Yes
getLogWriter	Yes	Yes	Yes
println	Yes	Yes	Yes
registerDriver	Yes	Yes	Yes
setLoginTimeout	Yes[2]	Yes[1]	Yes
setLogStream	Yes	Yes	Yes
setLogWriter	Yes	Yes	Yes

Notes:

1. The DB2 JDBC Type 2 Driver does not use this setting.
2. This method is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

Table 14-23. Support for ParameterMetaData methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
getParameterClassName	No	No	No
getParameterCount	Yes	No	No
getParameterMode	Yes	No	No
getParameterType	Yes	No	No
getParameterTypeName	Yes	No	No
getPrecision	Yes	No	No
getScale	Yes	No	No
isNullable	Yes	No	No
isSigned	Yes	No	No

Table 14-24. Support for PooledConnection methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
addConnectionEventListener	Yes	Yes	Yes
addStatementEventListener[1]	Yes	No	No
close	Yes	Yes	Yes
getConnection	Yes	Yes	Yes
removeConnectionEventListener	Yes	Yes	Yes
removeStatementEventListener[1]	Yes	No	No

Notes:

1. This is a JDBC 4.0 method.

Table 14-25. Support for PreparedStatement methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.sql.Statement	Yes	Yes	Yes
addBatch	Yes	Yes	Yes
clearParameters	Yes	Yes	Yes
execute	Yes	Yes	Yes
executeQuery	Yes	Yes	Yes
executeUpdate	Yes	Yes	Yes
getMetaData	Yes	Yes	Yes
getParameterMetaData	Yes	Yes	Yes
setArray	No	No	No
setAsciiStream	Yes[1,2]	Yes	Yes
setBigDecimal	Yes	Yes	Yes
setBinaryStream	Yes[1,3]	Yes	Yes
setBlob	Yes[4]	Yes	Yes
setBoolean	Yes	Yes	Yes
setByte	Yes	Yes	Yes
setBytes	Yes	Yes	Yes
setCharacterStream	Yes[1,5]	Yes	Yes
setClob	Yes[6]	Yes	Yes
setDate	Yes[8]	Yes[8]	Yes[8]
setDouble	Yes	Yes	Yes
setFloat	Yes	Yes	Yes
setInt	Yes	Yes	Yes
setLong	Yes	Yes	Yes
setNull	Yes[9]	Yes[9]	Yes[9]
setObject	Yes	Yes	Yes
setRef	No	No	No
setRowId[7]	Yes	No	No
setShort	Yes	Yes	Yes
setString	Yes[10]	Yes[10]	Yes[10]
setTime	Yes[8]	Yes[8]	Yes[8]
setTimestamp	Yes[8]	Yes[8]	Yes[8]
setUnicodeStream	Yes	Yes	Yes
setURL	Yes	Yes	Yes

Table 14-25. Support for PreparedStatement methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support

Notes:

1. If the value of the *length* parameter is -1, all of the data from the InputStream or Reader is read and sent to the data source.
2. Supported forms of this method include the following JDBC 4.0 forms:

   ```
   setAsciiStream(int parameterIndex, InputStream x, long length)
   setAsciiStream(int parameterIndex, InputStream x)
   ```
3. Supported forms of this method include the following JDBC 4.0 forms:

   ```
   setBinaryStream(int parameterIndex, InputStream x, long length)
   setBinaryStream(int parameterIndex, InputStream x)
   ```
4. Supported forms of this method include the following JDBC 4.0 form:

   ```
   setBlob(int parameterIndex, InputStream inputStream, long length)
   ```
5. Supported forms of this method include the following JDBC 4.0 forms:

   ```
   setCharacterStream(int parameterIndex, Reader reader, long length)
   setCharacterStream(int parameterIndex, Reader reader)
   ```
6. Supported forms of this method include the following JDBC 4.0 form:

   ```
   setClob(int parameterIndex, Reader reader, long length)
   ```
7. This is a JDBC 4.0 method.
8. The database server does no timezone adjustment for datetime values. The JDBC driver adjusts a value for the local timezone before sending the value to the server if you specify a form of the setDate, setTime, or setTimestamp method that includes a java.util.Calendar parameter.
9. The following form of setNull is not supported:

   ```
   setNull(int parameterIndex, int jdbcType, String typeName)
   ```
10. setString is not supported if the column has the FOR BIT DATA attribute or the data type is BLOB.

Table 14-26. Support for Ref methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
get BaseTypeName	No	No	No

Table 14-27. Support for ResultSet methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
absolute	Yes	Yes	Yes
afterLast	Yes	Yes	Yes
beforeFirst	Yes	Yes	Yes
cancelRowUpdates	Yes	No	No
clearWarnings	Yes	Yes	Yes
close	Yes	Yes	Yes
deleteRow	Yes	No	No
findColumn	Yes	Yes	Yes
first	Yes	Yes	Yes
getArray	No	No	No

Table 14-27. Support for ResultSet methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
getAsciiStream	Yes	Yes	Yes
getBigDecimal	Yes	Yes	Yes
getBinaryStream	Yes[1]	Yes	Yes
getBlob	Yes	Yes	Yes
getBoolean	Yes	Yes	Yes
getByte	Yes	Yes	Yes
getBytes	Yes	Yes	Yes
getCharacterStream	Yes	Yes	Yes
getClob	Yes	Yes	Yes
getConcurrency	Yes	Yes	Yes
getCursorName	Yes	Yes	Yes
getDate	Yes[3]	Yes[3]	Yes[3]
getDouble	Yes	Yes	Yes
getFetchDirection	Yes	Yes	Yes
getFetchSize	Yes	Yes	Yes
getFloat	Yes	Yes	Yes
getInt	Yes	Yes	Yes
getLong	Yes	Yes	Yes
getMetaData	Yes	Yes	Yes
getObject	Yes[4]	Yes[4]	Yes[4]
getRef	No	No	No
getRow	Yes	Yes	Yes
getRowId[10]	Yes	No	No
getShort	Yes	Yes	Yes
getStatement	Yes	Yes	Yes
getString	Yes	Yes	Yes
getTime	Yes[3]	Yes[3]	Yes[3]
getTimestamp	Yes[3]	Yes[3]	Yes[3]
getType	Yes	Yes	Yes
getUnicodeStream	Yes	Yes	Yes
getURL	Yes	Yes	Yes
getWarnings	Yes	Yes	Yes
insertRow	Yes	No	No
isAfterLast	Yes	Yes	Yes
isBeforeFirst	Yes	Yes	Yes
isFirst	Yes	Yes	Yes
isLast	Yes	Yes	Yes
last	Yes	Yes	Yes

Table 14-27. Support for ResultSet methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
moveToCurrentRow	Yes	No	No
moveToInsertRow	Yes	No	No
next	Yes	Yes	Yes
previous	Yes	Yes	Yes
refreshRow	Yes	No	No
relative	Yes	Yes	Yes
rowDeleted	Yes	No	No
rowInserted	Yes	No	No
rowUpdated	Yes	No	No
setFetchDirection	Yes	Yes	Yes
setFetchSize	Yes	Yes	Yes
updateArray	No	No	No
updateAsciiStream	Yes[5]	No	No
updateBigDecimal	Yes	No	No
updateBinaryStream	Yes[6]	No	No
updateBlob	Yes[7]	No	No
updateBoolean	Yes	No	No
updateByte	Yes	No	No
updateBytes	Yes	No	No
updateCharacterStream	Yes[8]	No	No
updateClob	Yes[9]	No	No
updateDate	Yes	No	No
updateDouble	Yes	No	No
updateFloat	Yes	No	No
updateInt	Yes	No	No
updateLong	Yes	No	No
updateNull	Yes	No	No
updateObject	Yes	No	No
updateRef	No	No	No
updateRow	Yes	No	No
updateRowId[10]	Yes	No	No
updateShort	Yes	No	No
updateString	Yes	No	No
updateTime	Yes	No	No
updateTimestamp	Yes	No	No
wasNull	Yes	Yes	Yes

Table 14-27. Support for ResultSet methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support

Notes:

1. getBinaryStream is not supported for CLOB columns.
2. getMetaData pads the schema name, if the returned schema name is less than 8 characters, to fill 8 characters.
3. The database server does no timezone adjustment for datetime values. The JDBC driver adjusts a value for the local timezone after retrieving the value from the server if you specify a form of the getDate, getTime, or getTimestamp method that includes a java.util.Calendar parameter.
4. The following form of the getObject method is not supported:

   ```
   getObject(int parameterIndex, java.util.Map  map)
   ```

5. Supported forms of this method include the following JDBC 4.0 forms:

   ```
   updateAsciiStream(int columnIndex, InputStream x)
   updateAsciiStream(String columnLabel, InputStream x)
   updateAsciiStream(int columnIndex, InputStream x, long length)
   updateAsciiStream(String columnLabel, InputStream x, long length)
   ```

6. Supported forms of this method include the following JDBC 4.0 forms:

   ```
   updateBinaryStream(int columnIndex, InputStream x)
   updateBinaryStream(String columnLabel, InputStream x)
   updateBinaryStream(int columnIndex, InputStream x, long length)
   updateBinaryStream(String columnLabel, InputStream x, long length)
   ```

7. Supported forms of this method include the following JDBC 4.0 forms:

   ```
   updateBlob(int columnIndex, InputStream x)
   updateBlob(String columnLabel, InputStream x)
   updateBlob(int columnIndex, InputStream x, long length)
   updateBlob(String columnLabel, InputStream x, long length)
   ```

8. Supported forms of this method include the following JDBC 4.0 forms:

   ```
   updateCharacterStream(int columnIndex, Reader reader)
   updateCharacterStream(String columnLabel, Reader reader)
   updateCharacterStream(int columnIndex, Reader reader, long length)
   updateCharacterStream(String columnLabel, Reader reader, long length)
   ```

9. Supported forms of this method include the following JDBC 4.0 forms:

   ```
   updateClob(int columnIndex, Reader reader)
   updateClob(String columnLabel, Reader reader)
   updateClob(int columnIndex, Reader reader, long length)
   updateClob(String columnLabel, Reader reader, long length)
   ```

10. This is a JDBC 4.0 method.

Table 14-28. Support for ResultSetMetaData methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
getCatalogName	Yes	Yes	Yes
getColumnClassName	No	Yes	Yes
getColumnCount	Yes	Yes	Yes
getColumnDisplaySize	Yes	Yes	Yes
getColumnLabel	Yes	Yes	Yes
getColumnName	Yes	Yes	Yes
getColumnType	Yes	Yes	Yes
getColumnTypeName	Yes	Yes	Yes

Table 14-28. Support for ResultSetMetaData methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
getPrecision	Yes	Yes	Yes
getScale	Yes	Yes	Yes
getSchemaName	Yes	Yes	Yes
getTableName	Yes[1]	Yes	Yes
isAutoIncrement	Yes	Yes	Yes
isCaseSensitive	Yes	Yes	Yes
isCurrency	Yes	Yes	Yes
isDefinitelyWritable	Yes	Yes	Yes
isNullable	Yes	Yes	Yes
isReadOnly	Yes	Yes	Yes
isSearchable	Yes	Yes	Yes
isSigned	Yes	Yes	Yes
isWritable	Yes	Yes	Yes

Notes:

1. For IBM Informix data sources, getTableName does not return a value.
2. getSchemaName pads the schema name, if the returned schema name is less than 8 characters, to fill 8 characters.

Table 14-29. Support for RowId methods[1]

JDBC method	IBM Data Server Driver for JDBC and SQLJ support[2]	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
equals	Yes	No	No
getBytes	Yes	No	No
hashCode	No	No	No
toString	Yes	No	No

Notes:

1. These methods are JDBC 4.0 methods.
2. These methods are supported for connections to DB2 for z/OS, DB2 for i, and IBM Informix data sources.

Table 14-30. Support for SQLClientInfoException methods[1]

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No
getFailedProperties	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 14-31. Support for SQLData methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
getSQLTypeName	No	No	No
readSQL	No	No	No
writeSQL	No	No	No

Table 14-32. Support for SQLDataException methods[1]

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 14-33. Support for SQLDataException methods[1]

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 14-34. Support for SQLException methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Exception	Yes	Yes	Yes
getSQLState	Yes	Yes	Yes
getErrorCode	Yes	Yes	Yes
getNextException	Yes	Yes	Yes
setNextException	Yes	Yes	Yes

Table 14-35. Support for SQLFeatureNotSupported methods[1]

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 14-36. Support for SQLInput methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
readArray	No	No	No
readAsciiStream	No	No	No
readBigDecimal	No	No	No
readBinaryStream	No	No	No
readBlob	No	No	No
readBoolean	No	No	No
readByte	No	No	No
readBytes	No	No	No
readCharacterStream	No	No	No
readClob	No	No	No
readDate	No	No	No
readDouble	No	No	No
readFloat	No	No	No
readInt	No	No	No
readLong	No	No	No
readObject	No	No	No
readRef	No	No	No
readShort	No	No	No
readString	No	No	No
readTime	No	No	No
readTimestamp	No	No	No
wasNull	No	No	No

Table 14-37. Support for SQLIntegrityConstraintViolationException methods[1]

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 14-38. Support for SQLInvalidAuthorizationSpecException methods[1]

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 14-39. Support for SQLNonTransientConnectionException methods[1]

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 14-40. Support for SQLNonTransientException methods[1]

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No

Table 14-40. Support for SQLNonTransientException methods[1] (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support

Note:

1. This is a JDBC 4.0 class.

Table 14-41. Support for SQLOutput methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
writeArray	No	No	No
writeAsciiStream	No	No	No
writeBigDecimal	No	No	No
writeBinaryStream	No	No	No
writeBlob	No	No	No
writeBoolean	No	No	No
writeByte	No	No	No
writeBytes	No	No	No
writeCharacterStream	No	No	No
writeClob	No	No	No
writeDate	No	No	No
writeDouble	No	No	No
writeFloat	No	No	No
writeInt	No	No	No
writeLong	No	No	No
writeObject	No	No	No
writeRef	No	No	No
writeShort	No	No	No
writeString	No	No	No
writeStruct	No	No	No
writeTime	No	No	No
writeTimestamp	No	No	No

Table 14-42. Support for SQLRecoverableException methods[1]

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No

Table 14-42. Support for SQLRecoverableException methods[1] (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support

Note:

1. This is a JDBC 4.0 class.

Table 14-43. Support for SQLSyntaxErrorException methods[1]

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 14-44. Support for SQLTimeoutException methods[1]

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 14-45. Support for SQLTransientConnectionException methods[1]

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 14-46. Support for SQLTransientException methods[1]

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 14-47. Support for SQLTransientRollbackException methods[1]

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 14-48. Support for SQLXML methods[1]

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
free	Yes	No	No
getBinaryStream	Yes	No	No
getCharacterStream	Yes	No	No
getSource	Yes	No	No
getString	Yes	No	No
setBinaryStream	Yes	No	No
setCharacterStream	Yes	No	No
setResult	Yes	No	No
setString	Yes	No	No

Notes:

1. These are JDBC 4.0 methods. These methods are not supported for connections to IBM Informix servers.

Table 14-49. Support for Statement methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
addBatch	Yes	Yes	Yes

Table 14-49. Support for Statement methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
cancel	Yes[1]	Yes[2]	Yes
clearBatch	Yes	Yes	Yes
clearWarnings	Yes	Yes	Yes
close	Yes	Yes	Yes
execute	Yes	Yes[3]	Yes
executeBatch	Yes	Yes	Yes
executeQuery	Yes	Yes	Yes
executeUpdate	Yes	Yes[3]	Yes
getConnection	Yes	Yes	Yes
getFetchDirection	Yes	Yes	Yes
getFetchSize	Yes	Yes	Yes
getGeneratedKeys	Yes	No	No
getMaxFieldSize	Yes	Yes	Yes
getMaxRows	Yes	Yes	Yes
getMoreResults	Yes	Yes[3]	Yes
getQueryTimeout	Yes[7,6]	Yes	Yes
getResultSet	Yes	Yes	Yes
getResultSetConcurrency	Yes	Yes	Yes
getResultSetHoldability	Yes	No	No
getResultSetType	Yes	Yes	Yes
getUpdateCount[4]	Yes	Yes	Yes
getWarnings	Yes	Yes	Yes
isClosed[8]	Yes	No	No
isPoolable[8]	Yes	No	No
setCursorName	Yes	Yes	Yes
setEscapeProcessing	Yes	Yes	Yes
setFetchDirection	Yes	Yes	Yes
setFetchSize	Yes	Yes	Yes
setMaxFieldSize	Yes	Yes	Yes
setMaxRows	Yes	Yes	Yes
setPoolable[8]	Yes	No	No
setQueryTimeout	Yes[5,7,6]	Yes	Yes

Table 14-49. Support for Statement methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support

Notes:

1. For the IBM Data Server Driver for JDBC and SQLJ, Statement.cancel is supported only in the following environments:
   - Type 2 and type 4 connectivity from a Linux, UNIX, or Windows client to a DB2 Database for Linux, UNIX, and Windows server, Version 8 or later
   - Type 2 and type 4 connectivity from a Linux, UNIX, or Windows client to a DB2 for z/OS server, Version 9 or later
   - Type 4 connectivity from a z/OS client to a DB2 Database for Linux, UNIX, and Windows server, Version 8 or later
   - Type 4 connectivity from a z/OS client to a DB2 for z/OS server, Version 8 or later

   The action that the IBM Data Server Driver for JDBC and SQLJ takes when the application executes Statement.cancel is also dependent on the setting of the DB2BaseDataSource.interruptProcessingMode property.
2. For the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows, Statement.cancel is supported only in the following environments:
   - Connections to a DB2 Database for Linux, UNIX, and Windows server, Version 8 or later
   - Connections to a DB2 for z/OS server, Version 9 or later
3. The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows does not support the JDBC 3.0 form of this method.
4. Not supported for stored procedure ResultSets.
5. For DB2 for i, this method is supported only for a *seconds* value of 0.
6. For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, Statement.setQueryTimeout is supported only if Connection or DataSource property queryTimeoutInterruptProcessingMode is set to INTERRUPT_PROCESSING_MODE_CLOSE_SOCKET.
7. For the IBM Data Server Driver for JDBC and SQLJ Version 4.0 and later, Statement.setQueryTimeout is supported for the following methods:
   - Statement.execute
   - Statement.executeUpdate
   - Statement.executeQuery

   Statement.setQueryTimeout is not supported for the Statement.executeBatch method.
8. This is a JDBC 4.0 method.

Table 14-50. Support for Struct methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
getSQLTypeName	No	No	No
getAttributes	No	No	No

Table 14-51. Support for Wrapper methods

JDBC method[1]	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
isWrapperFor	Yes	No	No
unWrap	Yes	No	No

Notes:

1. These are JDBC 4.0 methods.

Table 14-52. Support for javax.sql.XAConnection methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support[1]	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
Methods inherited from javax.sql.PooledConnection	Yes	Yes	Yes
getXAResource	Yes	Yes	Yes

Notes:

1. These methods are supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to a DB2 Database for Linux, UNIX, and Windows server or IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Table 14-53. Support for XADataSource methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
getLoginTimeout	Yes	Yes	Yes
getLogWriter	Yes	Yes	Yes
getXAConnection	Yes	Yes	Yes
setLoginTimeout	Yes	Yes	Yes
setLogWriter	Yes	Yes	Yes

Table 14-54. Support for javax.transaction.xa.XAResource methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IBM Informix JDBC Driver support
commit	Yes[1]	Yes	Yes
end	Yes[1]	Yes	Yes
forget	Yes[1]	Yes	Yes
getTransactionTimeout	Yes[2]	Yes	Yes
isSameRM	Yes[1]	Yes	Yes
prepare	Yes[1]	Yes	Yes
recover	Yes[1]	Yes	Yes
rollback	Yes[1]	Yes	Yes
setTransactionTimeout	Yes[2]	Yes	Yes
start	Yes[1]	Yes	Yes

Notes:

1. This method is supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to a DB2 Database for Linux, UNIX, and Windows server or IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.
2. This method is supported for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 Database for Linux, UNIX, and Windows Version 9.1 or later.

IBM Data Server Driver for JDBC and SQLJ support for SQL escape syntax

The IBM Data Server Driver for JDBC and SQLJ supports SQL escape syntax, as described in the JDBC 1.0 specification.

This is the same syntax that is used in vendor escape clauses in ODBC and CLI applications.

SQL escape syntax is supported in JDBC and SQLJ applications.

SQLJ statement reference information

SQLJ statements are used for transaction control and SQL statement execution.

SQLJ clause

The SQL statements in an SQLJ program are in SQLJ clauses.

Syntax

Usage notes

Keywords in an SQLJ clause are case sensitive, unless those keywords are part of an SQL statement in an executable clause.

SQLJ host-expression

A host expression is a Java variable or expression that is referenced by SQLJ clauses in an SQLJ application program.

Syntax

Description

: Indicates that the variable or expression that follows is a host expression. The colon must immediately precede the variable or expression.

IN|OUT|INOUT
For a host expression that is used as a parameter in a stored procedure call, identifies whether the parameter provides data to the stored procedure (IN), retrieves data from the stored procedure (OUT), or does both (INOUT). The default is IN.

simple-variable
Specifies a Java unqualified identifier.

complex-expression
Specifies a Java expression that results in a single value.

Usage notes

- A complex expression must be enclosed in parentheses.
- ANSI/ISO rules govern where a host expression can appear in a static SQL statement.

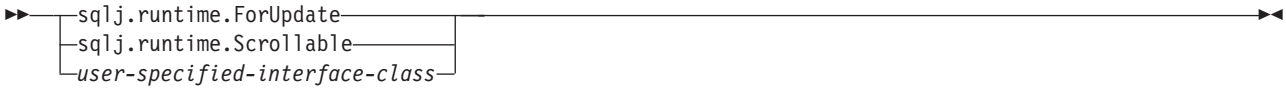
- , ... *variable-n*

SQLJ implements-clause

The implements clause derives one or more classes from a Java interface.

Syntax

interface-element:



Description

interface-element

Specifies a user-defined Java interface, the SQLJ interface sqlj.runtime.ForUpdate or the SQLJ interface sqlj.runtime.Scrollable.

You need to implement sqlj.runtime.ForUpdate when you declare an iterator for a positioned UPDATE or positioned DELETE operation. See "Perform positioned UPDATE and DELETE operations in an SQLJ application" for information on performing a positioned UPDATE or positioned DELETE operation in SQLJ.

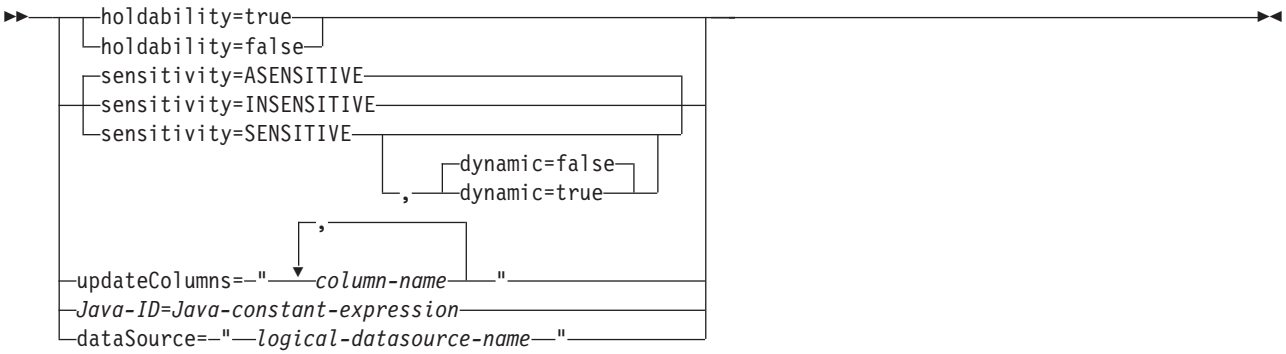
You need to implement sqlj.runtime.Scrollable when you declare a scrollable iterator. See "Use scrollable iterators in an SQLJ application" for information on scrollable iterators.

SQLJ with-clause

The with clause specifies a set of one or more attributes for an iterator or a connection context.

Syntax

with-element:



Description

holdability
For an iterator, specifies whether an iterator keeps its position in a table after a COMMIT is executed. The value for holdability must be `true` or `false`.

sensitivity
For an iterator, specifies whether changes that are made to the underlying table can be visible to the iterator after it is opened. The value must be INSENSITIVE, SENSITIVE, or ASENSITIVE. The default is ASENSITIVE.

For connections to IBM Informix, only INSENSITIVE is supported.

dynamic
For an iterator that is defined with sensitivity=SENSITIVE, specifies whether the following cases are true:

- When the application executes positioned UPDATE and DELETE statements with the iterator, those changes are visible to the iterator.
- When the application executes INSERT, UPDATE, and DELETE statements within the application but outside the iterator, those changes are visible to the iterator.

The value for dynamic must be `true` or `false`. The default is `false`.

DB2 Database for Linux, UNIX, and Windows servers do not support dynamic scrollable cursors. Specify `true` only if your application accesses data on DB2 for z/OS servers, at Version 9 or later.

For connections to IBM Informix, only `false` is supported. IBM Informix does not support dynamic cursors.

updateColumns
For an iterator, specifies the columns that are to be modified when the iterator is used for a positioned UPDATE statement. The value for updateColumns must be a literal string that contains the column names, separated by commas.

column-name
For an iterator, specifies a column of the result table that is to be updated using the iterator.

Java-ID
For an iterator or connection context, specifies a Java variable that identifies a user-defined attribute of the iterator or connection context. The value of *Java-constant-expression* is also user-defined.

dataSource
For a connection context, specifies the logical name of a separately-created

DataSource object that represents the data source to which the application will connect. This option is available only for the IBM Data Server Driver for JDBC and SQLJ.

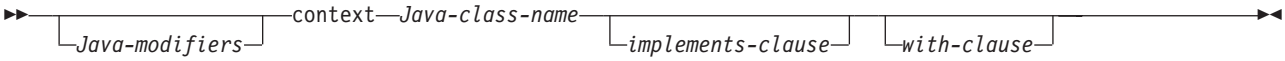
Usage notes

- The value on the left side of a with element must be unique within its with clause.
- If you specify updateColumns in a with element of an iterator declaration clause, the iterator declaration clause must also contain an implements clause that specifies the sqlj.runtime.ForUpdate interface.
- If you do not customize your SQLJ program, the JDBC driver ignores the value of holdability that is in the with clause. Instead, the driver uses the JDBC driver setting for holdability.

SQLJ connection-declaration-clause

The connection declaration clause declares a connection to a data source in an SQLJ application program.

Syntax



Description

Java-modifiers
Specifies modifiers that are valid for Java class declarations, such as static, public, private, or protected.

Java-class-name
Specifies a valid Java identifier. During the program preparation process, SQLJ generates a connection context class whose name is this identifier.

implements-clause
See "SQLJ implements-clause" for a description of this clause. In a connection declaration clause, the interface class to which the implements clause refers must be a user-defined interface class.

with-clause
See "SQLJ with-clause" for a description of this clause.

Usage notes

- SQLJ generates a connection class declaration for each connection declaration clause you specify. SQLJ data source connections are objects of those generated connection classes.
- You can specify a connection declaration clause anywhere that a Java class definition can appear in a Java program.

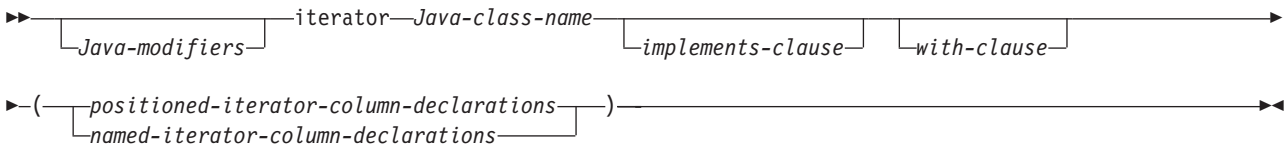
SQLJ iterator-declaration-clause

An iterator declaration clause declares a positioned iterator class or a named iterator class in an SQLJ application program.

An iterator contains the result table from a query. SQLJ generates an iterator class for each iterator declaration clause you specify. An iterator is an object of an iterator class.

An iterator declaration clause has a form for a positioned iterator and a form for a named iterator. The two kinds of iterators are distinct and incompatible Java types that are implemented with different interfaces.

Syntax



positioned-iterator-column declarations:

named-iterator-column-declarations:

Description

Java-modifiers
Any modifiers that are valid for Java class declarations, such as static, public, private, or protected.

Java-class-name
Any valid Java identifier. During the program preparation process, SQLJ generates an iterator class whose name is this identifier.

implements-clause
See "SQLJ implements-clause" for a description of this clause. For an iterator declaration clause that declares an iterator for a positioned UPDATE or positioned DELETE operation, the implements clause must specify interface sqlj.runtime.ForUpdate. For an iterator declaration clause that declares a scrollable iterator, the implements clause must specify interface sqlj.runtime.Scrollable.

with-clause
See "SQLJ with-clause" for a description of this clause.

positioned-iterator-column-declarations
Specifies a list of Java data types, which are the data types of the columns in the positioned iterator. The data types in the list must be separated by commas. The order of the data types in the positioned iterator declaration is the same as the order of the columns in the result table. For online checking during serialized profile customization to succeed, the data types of the columns in the iterator must be compatible with the data types of the columns in the result table. See "Java, JDBC, and SQL data types" for a list of compatible data types.

named-iterator-column-declarations
Specifies a list of Java data types and Java identifiers, which are the data types and names of the columns in the named iterator. Pairs of data types and names must be separated by commas. The name of a column in the iterator must match, except for case, the name of a column in the result table. For online checking during serialized profile customization to succeed, the data types of the columns in the iterator must be compatible with the data types of the columns in the result table. See "Java, JDBC, and SQL data types" for a list of compatible data types.

Usage notes

- An iterator declaration clause can appear anywhere in a Java program that a Java class declaration can appear.
- When a named iterator declaration contains more than one pair of Java data types and Java IDs, all Java IDs within the list must be unique. Two Java IDs are not unique if they differ only in case.

SQLJ executable-clause

An executable clause contains an SQL statement or an assignment statement. An assignment statement assigns the result of an SQL operation to a Java variable.

This topic describes the general form of an executable clause.

Syntax

Usage notes

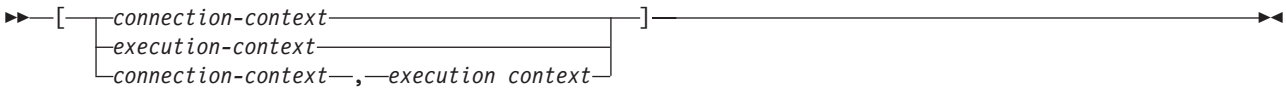
- An executable clause can appear anywhere in a Java program that a Java statement can appear.
- SQLJ reports negative SQL codes from executable clauses through class java.sql.SQLException.

  If SQLJ raises a run-time exception during the execution of an executable clause, the value of any host expression of type OUT or INOUT is undefined.

SQLJ context-clause

A context clause specifies a connection context, an execution context, or both. You use a connection context to connect to a data source. You use an execution context to monitor and modify SQL statement execution.

Syntax



Description

connection-context
Specifies a valid Java identifier that is declared earlier in the SQLJ program.

That identifier must be declared as an instance of the connection context class that SQLJ generates for a connection declaration clause.

execution-context
Specifies a valid Java identifier that is declared earlier in the SQLJ program. That identifier must be declared as an instance of class sqlj.runtime.ExecutionContext.

Usage notes

- If you do not specify a connection context in an executable clause, SQLJ uses the default connection context.
- If you do not specify an execution context, SQLJ obtains the execution context from the connection context of the statement.

SQLJ statement-clause

A statement clause contains an SQL statement or a SET TRANSACTION clause.

Syntax

Description

SQL-statement
You can include SQL statements in Table 14-55 in a statement clause.

SET-TRANSACTION-clause
Sets the isolation level for SQL statements in the program and the access mode for the connection. The SET TRANSACTION clause is equivalent to the SET TRANSACTION statement, which is described in the ANSI/ISO SQL standard of 1992 and is supported in some implementations of SQL.

Table 14-55. Valid SQL statements in an SQLJ statement clause

Statement	Applicable data sources
ALTER DATABASE	1 on page 14-104, 2 on page 14-104
ALTER FUNCTION	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
ALTER INDEX	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
ALTER PROCEDURE	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
ALTER STOGROUP	1 on page 14-104, 2 on page 14-104
ALTER TABLE	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
ALTER TABLESPACE	1 on page 14-104, 2 on page 14-104
CALL	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
COMMENT ON	1 on page 14-104, 2 on page 14-104
COMMIT	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
Compound SQL (BEGIN ATOMIC...END)	2 on page 14-104
CREATE ALIAS	1 on page 14-104, 2 on page 14-104
CREATE DATABASE	1 on page 14-104, 2 on page 14-104, 3a on page 14-104
CREATE DISTINCT TYPE	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
CREATE FUNCTION	1 on page 14-104, 2 on page 14-104, 3 on page 14-104

Table 14-55. Valid SQL statements in an SQLJ statement clause (continued)

Statement	Applicable data sources
CREATE GLOBAL TEMPORARY TABLE	1 on page 14-104, 2 on page 14-104
CREATE TEMP TABLE	3 on page 14-104
CREATE INDEX	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
CREATE PROCEDURE	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
CREATE STOGROUP	1 on page 14-104, 2 on page 14-104
CREATE SYNONYM	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
CREATE TABLE	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
CREATE TABLESPACE	1 on page 14-104, 2 on page 14-104
CREATE TYPE (cursor)	2 on page 14-104
CREATE TRIGGER	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
CREATE VIEW	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
DECLARE GLOBAL TEMPORARY TABLE	1 on page 14-104, 2 on page 14-104
DELETE	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
DROP ALIAS	1 on page 14-104, 2 on page 14-104
DROP DATABASE	1 on page 14-104, 2 on page 14-104, 3a on page 14-104
DROP DISTINCT TYPE	1 on page 14-104, 2 on page 14-104
DROP TYPE	3 on page 14-104
DROP FUNCTION	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
DROP INDEX	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
DROP PACKAGE	1 on page 14-104, 2 on page 14-104
DROP PROCEDURE	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
DROP STOGROUP	1 on page 14-104, 2 on page 14-104
DROP SYNONYM	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
DROP TABLE	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
DROP TABLESPACE	1 on page 14-104, 2 on page 14-104
DROP TRIGGER	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
DROP VIEW	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
FETCH	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
GRANT	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
INSERT	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
LOCK TABLE	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
MERGE	1 on page 14-104, 2 on page 14-104
REVOKE	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
ROLLBACK	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
SAVEPOINT	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
SELECT INTO	1 on page 14-104, 2 on page 14-104, 3 on page 14-104
SET CURRENT APPLICATION ENCODING SCHEME	1 on page 14-104
SET CURRENT DEBUG MODE	1 on page 14-104
SET CURRENT DEFAULT TRANSFORM GROUP	2 on page 14-104
SET CURRENT DEGREE	1 on page 14-104, 2 on page 14-104

Table 14-55. Valid SQL statements in an SQLJ statement clause (continued)

Statement	Applicable data sources
SET CURRENT EXPLAIN MODE	2
SET CURRENT EXPLAIN SNAPSHOT	2
SET CURRENT ISOLATION	1, 2
SET CURRENT LOCALE LC_CTYPE	1
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	1, 2
SET CURRENT OPTIMIZATION HINT	1, 2
SET CURRENT PACKAGE PATH	1
SET CURRENT PACKAGESET (USER is not supported)	1, 2
SET CURRENT PRECISION	1, 2
SET CURRENT QUERY OPTIMIZATION	2
SET CURRENT REFRESH AGE	1, 2
SET CURRENT ROUTINE VERSION	1
SET CURRENT RULES	1
SET CURRENT SCHEMA	2
SET CURRENT SQLID	1
SET PATH	1, 2
TRUNCATE	1
UPDATE	1, 2, 3

Note: The SQL statement applies to connections to the following data sources:

1. DB2 for z/OS
2. DB2 Database for Linux, UNIX, and Windows
3. IBM Informix
   a. IBM Informix, for the SYSMASTER database only.

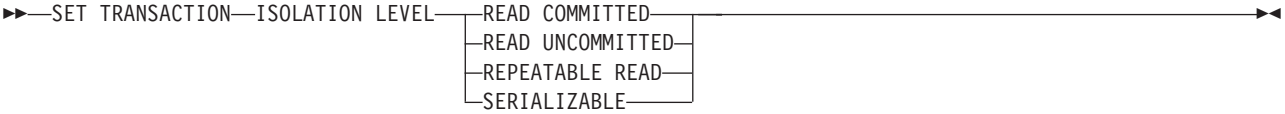
Usage notes

- SQLJ supports both positioned and searched DELETE and UPDATE operations.
- For a FETCH statement, a positioned DELETE statement, or a positioned UPDATE statement, you must use an iterator to refer to rows in a result table.

SQLJ SET-TRANSACTION-clause

The SET TRANSACTION clause sets the isolation level for the current unit of work.

Syntax



Description

ISOLATION LEVEL
Specifies one of the following isolation levels:

READ COMMITTED
Specifies that the current IDS isolation level is cursor stability.

READ UNCOMMITTED
Specifies that the current IDS isolation level is uncommitted read.

REPEATABLE READ
Specifies that the current IDS isolation level is read stability.

SERIALIZABLE
Specifies that the current IDS isolation level is repeatable read.

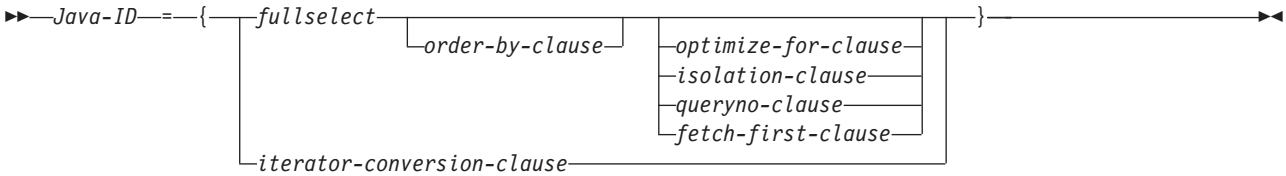
Usage notes

You can execute SET TRANSACTION only at the beginning of a transaction.

SQLJ assignment-clause

The assignment clause assigns the result of an SQL operation to a Java variable.

Syntax



Description

Java-ID
Identifies an iterator that was declared previously as an instance of an iterator class.

fullselect
Generates a result table.

iterator-conversion-clause
See "SQLJ iterator-conversion-clause" for a description of this clause.

Usage notes

- If the object that is identified by *Java-ID* is a positioned iterator, the number of columns in the result set must match the number of columns in the iterator. In addition, the data type of each column in the result set must be compatible with the data type of the corresponding column in the iterator. See "Java, JDBC, and SQL data types" for a list of compatible Java and SQL data types.
- If the object that is identified by *Java-ID* is a named iterator, the name of each accessor method must match, except for case, the name of a column in the result set. In addition, the data type of the object that an accessor method returns must be compatible with the data type of the corresponding column in the result set.
- You can put an assignment clause anywhere in a Java program that a Java assignment statement can appear. However, you cannot put an assignment

clause where a Java assignment expression can appear. For example, you cannot specify an assignment clause in the control list of a for statement.

SQLJ iterator-conversion-clause

The iterator conversion clause converts a JDBC ResultSet to an iterator.

Syntax

Description

host-expression
Identifies the JDBC ResultSet that is to be converted to an SQLJ iterator.

Usage notes

- If the iterator to which the JDBC ResultSet is to be converted is a positioned iterator, the number of columns in the ResultSet must match the number of columns in the iterator. In addition, the data type of each column in the ResultSet must be compatible with the data type of the corresponding column in the iterator.
- If the iterator is a named iterator, the name of each accessor method must match, except for case, the name of a column in the ResultSet. In addition, the data type of the object that an accessor method returns must be compatible with the data type of the corresponding column in the ResultSet.
- When an iterator that is generated through the iterator conversion clause is closed, the ResultSet from which the iterator is generated is also closed.

Interfaces and classes in the sqlj.runtime package

The sqlj.runtime package defines the run-time classes and interfaces that are used directly or indirectly by the SQLJ programmer.

Classes such as AsciiStream are used directly by the SQLJ programmer. Interfaces such as ResultSetIterator are implemented as part of generated class declarations.

sqlj.runtime interfaces

The following table summarizes the interfaces in sqlj.runtime.

Table 14-56. Summary of sqlj.runtime interfaces

Interface name	Purpose
ConnectionContext	Manages the SQL operations that are performed during a connection to a data source.
ForUpdate	Implemented by iterators that are used in a positioned UPDATE or DELETE statement.
NamedIterator	Implemented by iterators that are declared as named iterators.
PositionedIterator	Implemented by iterators that are declared as positioned iterators.
ResultSetIterator	Implemented by all iterators to allow query results to be processed using a JDBC ResultSet.
Scrollable	Provides a set of methods for manipulating scrollable iterators.

sqlj.runtime classes

The following table summarizes the classes in sqlj.runtime.

Table 14-57. Summary of sqlj.runtime classes

Class name	Purpose
AsciiStream	A class for handling an input stream whose bytes should be interpreted as ASCII.
BinaryStream	A class for handling an input stream whose bytes should be interpreted as binary.
CharacterStream	A class for handling an input stream whose bytes should be interpreted as Character.
DefaultRuntime	Implemented by SQLJ to satisfy the expected runtime behavior of SQLJ for most JVM environments. This class is for internal use only and is not described in this documentation.
ExecutionContext	Implemented when an SQLJ execution context is declared, to control the execution of SQL operations.
RuntimeContext	Defines system-specific services that are provided by the runtime environment. This class is for internal use only and is not described in this documentation.
SQLNullException	Derived from the java.sql.SQLException class. An sqlj.runtime.SQLNullException is thrown when an SQL NULL value is fetched into a host identifier with a Java primitive type.
StreamWrapper	Wraps a java.io.InputStream instance.
UnicodeStream	A class for handling an input stream whose bytes should be interpreted as Unicode.

sqlj.runtime.ConnectionContext interface

The sqlj.runtime.ConnectionContext interface provides a set of methods that manage SQL operations that are performed during a session with a specific data source.

Translation of an SQLJ connection declaration clause causes SQLJ to create a connection context class. A connection context object maintains a JDBC Connection object on which dynamic SQL operations can be performed. A connection context object also maintains a default ExecutionContext object.

Variables

CLOSE_CONNECTION

Format:

```
public static final boolean CLOSE_CONNECTION=true;
```

A constant that can be passed to the close method. It indicates that the underlying JDBC Connection object should be closed.

KEEP_CONNECTION

Format:

```
public static final boolean KEEP_CONNECTION=false;
```

A constant that can be passed to the close method. It indicates that the underlying JDBC Connection object should not be closed.

Methods

close()

Format:

```
public abstract void close() throws SQLException
```

Performs the following functions:
- Releases all resources that are used by the given connection context object
- Closes any open ConnectedProfile objects
- Closes the underlying JDBC Connection object

close() is equivalent to close(CLOSE_CONNECTION).

close(boolean)

Format:

```
public abstract void close (boolean close-connection)
 throws SQLException
```

Performs the following functions:
- Releases all resources that are used by the given connection context object
- Closes any open ConnectedProfile objects
- Closes the underlying JDBC Connection object, depending on the value of the *close-connection* parameter

Parameters:

close-connection
Specifies whether the underlying JDBC Connection object is closed when a connection context object is closed:

CLOSE_CONNECTION
Closes the underlying JDBC Connection object.

KEEP_CONNECTION
Does not close the underlying JDBC Connection object.

getConnectedProfile

Format:

```
public abstract ConnectedProfile getConnectedProfile(Object profileKey)
 throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

getConnection

Format:

```
public abstract Connection getConnection()
```

Returns the underlying JDBC Connection object for the given connection context object.

getExecutionContext

Format:

```
public abstract ExecutionContext getExecutionContect()
```

Returns the default ExecutionContext object that is associated with the given connection context object.

isClosed

Format:

```
public abstract boolean isClosed()
```

Returns `true` if the given connection context object has been closed. Returns `false` if the connection context object has not been closed.

Constructors

The following constructors are defined in a concrete implementation of the ConnectionContext interface that results from translation of the statement `#sql context Ctx;`:

Ctx(String, boolean)

Format:

```
public Ctx(String url, boolean autocommit)
 throws SQLException
```

Parameters:

url
The representation of a data source, as specified in the JDBC getConnection method.

autocommit
Whether autocommit is enabled for the connection. A value of `true` means that autocommit is enabled. A value of `false` means that autocommit is disabled.

Ctx(String, String, String, boolean)

Format:

```
public Ctx(String url, String user, String password,
 boolean autocommit)
 throws SQLException
```

Parameters:

url
The representation of a data source, as specified in the JDBC getConnection method.

user
The user ID under which the connection to the data source is made.

password
The password for the user ID under which the connection to the data source is made.

autocommit
Whether autocommit is enabled for the connection. A value of `true` means that autocommit is enabled. A value of `false` means that autocommit is disabled.

Ctx(String, Properties, boolean)

Format:

```
public Ctx(String url, Properties info, boolean autocommit)
 throws SQLException
```

Parameters:

url
The representation of a data source, as specified in the JDBC getConnection method.

info
An object that contains a set of driver properties for the connection. Any of the IBM Data Server Driver for JDBC and SQLJ properties can be specified.

autocommit
Whether autocommit is enabled for the connection. A value of `true` means that autocommit is enabled. A value of `false` means that autocommit is disabled.

Ctx(Connection)

Format:

```
public Ctx(java.sql.Connection JDBC-connection-object)
 throws SQLException
```

Parameters:

JDBC-connection-object
A previously created JDBC Connection object.

If the constructor call throws an SQLException, the JDBC Connection object remains open.

Ctx(ConnectionContext)

Format:

```
public Ctx(sqlj.runtime.ConnectionContext SQLJ-connection-context-object)
 throws SQLException
```

Parameters:

SQLJ-connection-context-object
A previously created SQLJ ConnectionContext object.

The following constructors are defined in a concrete implementation of the ConnectionContext interface that results from translation of the statement `#sql context Ctx with (dataSource ="jdbc/TestDS");`:

Ctx()

Format:

```
public Ctx()
 throws SQLException
```

Ctx(String, String)

Format:

```
public Ctx(String user, String password,
)
 throws SQLException
```

Parameters:

user
The user ID under which the connection to the data source is made.

password
The password for the user ID under which the connection to the data source is made.

Ctx(Connection)

Format:

```
public Ctx(java.sql.Connection JDBC-connection-object)
 throws SQLException
```

Parameters:

JDBC-connection-object
A previously created JDBC Connection object.

If the constructor call throws an SQLException, the JDBC Connection object remains open.

Ctx(ConnectionContext)

Format:

```
public Ctx(sqlj.runtime.ConnectionContext SQLJ-connection-context-object)
 throws SQLException
```

Parameters:

SQLJ-connection-context-object
A previously created SQLJ ConnectionContext object.

Methods

The following additional methods are generated in a concrete implementation of the ConnectionContext interface that results from translation of the statement `#sql context Ctx;`:

getDefaultContext

Format:

```
public static Ctx getDefaultContext()
```

Returns the default connection context object for the Ctx class.

getProfileKey

Format:

```
public static Object getProfileKey(sqlj.runtime.profile.Loader loader,
String profileName) throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

getProfile

Format:

```
public static sqlj.runtime.profile.Profile getProfile(Object key)
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

getTypeMap

Format:

```
public static java.util.Map getTypeMap()
```

Returns an instance of a class that implements java.util.Map, which is the user-defined type map that is associated with the ConnectionContext. If there is no associated type map, Java null is returned.

This method is used by code that is generated by the SQLJ translator for executable clauses and iterator declaration clauses, but it can also be invoked in an SQLJ application for direct use in JDBC statements.

SetDefaultContext

Format:

```
public static void Ctx setDefaultContext(Ctx default-context)
```

Sets the default connection context object for the Ctx class.

Recommendation: Do not use this method for multithreaded applications. Instead, use explicit contexts.

sqlj.runtime.ForUpdate interface

SQLJ implements the sqlj.runtime.ForUpdate interface in SQLJ programs that contain an iterator declaration clause with `implements sqlj.runtime.ForUpdate`.

An SQLJ program that does positioned UPDATE or DELETE operations (UPDATE...WHERE CURRENT OF or DELETE...WHERE CURRENT OF) must include an iterator declaration clause with `implements sqlj.runtime.ForUpdate`.

Methods

getCursorName

Format:

```
public abstract String getCursorName() throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

sqlj.runtime.NamedIterator interface

The sqlj.runtime.NamedIterator interface is implemented when an SQLJ application executes an iterator declaration clause for a named iterator.

A named iterator includes result table column names, and the order of the columns in the iterator is not important.

An implementation of the sqlj.runtime.NamedIterator interface includes an accessor method for each column in the result table. An accessor method returns the data from its column of the result table. The name of an accessor method matches the name of the corresponding column in the named iterator.

Methods (inherited from the ResultSetIterator interface)

close

Format:

```
public abstract void close() throws SQLException
```

Releases database resources that the iterator uses.

isClosed

Format:

```
public abstract boolean isClosed() throws SQLException
```

Returns a value of `true` if the close method has been invoked. Returns `false` if the close method has not been invoked.

next

Format:

```
public abstract boolean next() throws SQLException
```

Advances the iterator to the next row. Before an instance of the next method is invoked for the first time, the iterator is positioned before the first row of the result table. next returns a value of `true` when a next row is available and `false` when all rows have been retrieved.

sqlj.runtime.PositionedIterator interface

The sqlj.runtime.PositionedIterator interface is implemented when an SQLJ application executes an iterator declaration clause for a positioned iterator.

The order of columns in a positioned iterator must be the same as the order of columns in the result table, and a positioned iterator does not include result table column names.

Methods

sqlj.runtime.PositionedIterator inherits all **ResultSetIterator** methods, and includes the following additional method:

endFetch

Format:

```
public abstract boolean endFetch() throws SQLException
```

Returns a value of `true` if the iterator is not positioned on a row. Returns a value of `false` if the iterator is positioned on a row.

sqlj.runtime.ResultSetIterator interface

The sqlj.runtime.ResultSetIterator interface is implemented by SQLJ for all iterator declaration clauses.

An untyped iterator can be generated by declaring an instance of the sqlj.runtime.ResultSetIterator interface directly. In general, use of untyped iterators is not recommended.

Variables

ASENSITIVE

Format:

```
public static final int ASENSITIVE
```

A constant that can be returned by the getSensitivity method. It indicates that the iterator is defined as ASENSITIVE.

This value is not returned by IBM Informix.

FETCH_FORWARD

Format:

```
public static final int FETCH_FORWARD
```

A constant that can be used by the following methods:

- Set by sqlj.runtime.Scrollable.setFetchDirection and sqlj.runtime.ExecutionContext.setFetchDirection
- Returned by sqlj.runtime.ExecutionContext.getFetchDirection

It indicates that the iterator fetches rows in a result table in the forward direction, from first to last.

FETCH_REVERSE

Format:

```
public static final int FETCH_REVERSE
```

A constant that can be used by the following methods:

- Set by sqlj.runtime.Scrollable.setFetchDirection and sqlj.runtime.ExecutionContext.setFetchDirection
- Returned by sqlj.runtime.ExecutionContext.getFetchDirection

It indicates that the iterator fetches rows in a result table in the backward direction, from last to first.

This value is not returned by IBM Informix.

FETCH_UNKNOWN

Format:

```
public static final int FETCH_UNKNOWN
```

A constant that can be used by the following methods:
- Set by sqlj.runtime.Scrollable.setFetchDirection and sqlj.runtime.ExecutionContext.setFetchDirection
- Returned by sqlj.runtime.ExecutionContext.getFetchDirection

It indicates that the iterator fetches rows in a result table in an unknown order.

This value is not returned by IBM Informix.

INSENSITIVE

Format:

```
public static final int INSENSITIVE
```

A constant that can be returned by the getSensitivity method. It indicates that the iterator is defined as INSENSITIVE.

SENSITIVE

Format:

```
public static final int SENSITIVE
```

A constant that can be returned by the getSensitivity method. It indicates that the iterator is defined as SENSITIVE.

This value is not returned by IBM Informix.

Methods

clearWarnings

Format:

```
public abstract void clearWarnings() throws SQLException
```

After clearWarnings is called, getWarnings returns null until a new warning is reported for the iterator.

close

Format:

```
public abstract void close() throws SQLException
```

Closes the iterator and releases underlying database resources.

getFetchSize

Format:

```
synchronized public int getFetchSize() throws SQLException
```

Returns the number of rows that should be fetched by SQLJ when more rows are needed. The returned value is the value that was set by the setFetchSize method, or 0 if no value was set by setFetchSize.

getResultSet

Format:

```
public abstract ResultSet getResultSet() throws SQLException
```

Returns the JDBC ResultSet object that is associated with the iterator.

getRow

Format:

```
synchronized public int getRow() throws SQLException
```

Returns the current row number. The first row is number 1, the second is number 2, and so on. If the iterator is not positioned on a row, 0 is returned.

getSensitivity

Format:

```
synchronized public int getSensitivity() throws SQLException
```

Returns the sensitivity of the iterator. The sensitivity is determined by the sensitivity value that was specified or defaulted in the with clause of the iterator declaration clause.

getWarnings

Format:

```
public abstract SQLWarning getWarnings() throws SQLException
```

Returns the first warning that is reported by calls on the iterator. Subsequent iterator warnings are be chained to this SQLWarning. The warning chain is automatically cleared each time the iterator moves to a new row.

isClosed

Format:

```
public abstract boolean isClosed() throws SQLException
```

Returns a value of `true` if the iterator is closed. Returns `false` otherwise.

next

Format:

```
public abstract boolean next() throws SQLException
```

Advances the iterator to the next row. Before next is invoked for the first time, the iterator is positioned before the first row of the result table. next returns a value of `true` when a next row is available and `false` when all rows have been retrieved.

setFetchSize

Format:

```
synchronized public void setFetchSize(int number-of-rows) throws SQLException
```

Gives SQLJ a hint as to the number of rows that should be fetched when more rows are needed.

Parameters:

number-of-rows
: The expected number of rows that SQLJ should fetch for the iterator that is associated with the given execution context.

If *number-of-rows* is less than 0 or greater than the maximum number of rows that can be fetched, an SQLException is thrown.

sqlj.runtime.Scrollable interface

sqlj.runtime.Scrollable provides methods to move around in the result table and to check the position in the result table.

sqlj.runtime.Scrollable is implemented when a scrollable iterator is declared.

Methods

absolute(int)

Format:

```
public abstract boolean absolute (int n) throws SQLException
```

Moves the iterator to a specified row.

If *n*>0, positions the iterator on row *n* of the result table. If *n*<0, and *m* is the number of rows in the result table, positions the iterator on row *m+n+1* of the result table.

If the absolute value of *n* is greater than the number of rows in the result table, positions the cursor after the last row if *n* is positive, or before the first row if *n* is negative.

absolute(0) is the same as beforeFirst(). absolute(1) is the same as first(). absolute(-1) is the same as last().

Returns `true` if the iterator is on a row. Otherwise, returns `false`.

afterLast()

Format:

```
public abstract void afterLast() throws SQLException
```

Moves the iterator after the last row of the result table.

beforeFirst()

Format:

```
public abstract void beforeFirst() throws SQLException
```

Moves the iterator before the first row of the result table.

first()

Format:

```
public abstract boolean first() throws SQLException
```

Moves the iterator to the first row of the result table.

Returns `true` if the iterator is on a row. Otherwise, returns `false`.

getFetchDirection()

Format:

```
public abstract int getFetchDirection() throws SQLException
```

Returns the fetch direction of the iterator. Possible values are:

sqlj.runtime.ResultSetIterator.FETCH_FORWARD
Rows are processed in a forward direction, from first to last.

sqlj.runtime.ResultSetIterator.FETCH_REVERSE
Rows are processed in a backward direction, from last to first.

sqlj.runtime.ResultSetIterator.FETCH_UNKNOWN
The order of processing is not known.

isAfterLast()

Format:

```
public abstract boolean isAfterLast() throws SQLException
```

Returns `true` if the iterator is positioned after the last row of the result table. Otherwise, returns `false`.

`isBeforeFirst()`

Format:

```
public abstract boolean isBeforeFirst() throws SQLException
```

Returns `true` if the iterator is positioned before the first row of the result table. Otherwise, returns `false`.

`isFirst()`

Format:

```
public abstract boolean isFirst() throws SQLException
```

Returns `true` if the iterator is positioned on the first row of the result table. Otherwise, returns `false`.

`isLast()`

Format:

```
public abstract boolean isLast() throws SQLException
```

Returns `true` if the iterator is positioned on the last row of the result table. Otherwise, returns `false`.

`last()`

Format:

```
public abstract boolean last() throws SQLException
```

Moves the iterator to the last row of the result table.

Returns `true` if the iterator is on a row. Otherwise, returns `false`.

`previous()`

Format:

```
public abstract boolean previous() throws SQLException
```

Moves the iterator to the previous row of the result table.

Returns `true` if the iterator is on a row. Otherwise, returns `false`.

`relative(int)`

Format:

```
public abstract boolean relative(int n) throws SQLException
```

If $n>0$, positions the iterator on the row that is n rows after the current row. If $n<0$, positions the iterator on the row that is n rows before the current row. If $n=0$, positions the iterator on the current row.

The cursor must be on a valid row of the result table before you can use this method. If the cursor is before the first row or after the last throw, the method throws an SQLException.

Suppose that m is the number of rows in the result table and x is the current row number in the result table. If $n>0$ and $x+n>m$, the iterator is positioned after the last row. If $n<0$ and $x+n<1$, the iterator is positioned before the first row.

Returns `true` if the iterator is on a row. Otherwise, returns `false`.

`setFetchDirection(int)`

Format:

```
public abstract void setFetchDirection (int) throws SQLException
```

Gives the SQLJ runtime environment a hint as to the direction in which rows of this iterator object are processed. Possible values are:

sqlj.runtime.ResultSetIterator.FETCH_FORWARD
Rows are processed in a forward direction, from first to last.

sqlj.runtime.ResultSetIterator.FETCH_REVERSE
Rows are processed in a backward direction, from last to first.

sqlj.runtime.ResultSetIterator.FETCH_UNKNOWN
The order of processing is not known.

sqlj.runtime.AsciiStream class

The sqlj.runtime.AsciiStream class is for an input stream of ASCII data with a specified length.

The sqlj.runtime.AsciiStream class is derived from the java.io.InputStream class, and extends the sqlj.runtime.StreamWrapper class. SQLJ interprets the bytes in an sqlj.runtime.AsciiStream object as ASCII characters. An InputStream object with ASCII characters needs to be passed as a sqlj.runtime.AsciiStream object.

Constructors

AsciiStream(InputStream)

Format:

```
public AsciiStream(java.io.InputStream input-stream)
```

Creates an ASCII java.io.InputStream object with an unspecified length.

Parameters:

input-stream
The InputStream object that SQLJ interprets as an AsciiStream object.

AsciiStream(InputStream, int)

Format:

```
public AsciiStream(java.io.InputStream input-stream, int length)
```

Creates an ASCII java.io.InputStream object with a specified length.

Parameters:

input-stream
The InputStream object that SQLJ interprets as an AsciiStream object.

length
The length of the InputStream object that SQLJ interprets as an AsciiStream object.

sqlj.runtime.BinaryStream class

The sqlj.runtime.BinaryStream class is for an input stream of binary data with a specified length.

The sqlj.runtime.BinaryStream class is derived from the java.io.InputStream class, and extends the sqlj.runtime.StreamWrapper class. SQLJ interprets the bytes in an sqlj.runtime.BinaryStream object are interpreted as Binary characters. An InputStream object with Binary characters needs to be passed as a sqlj.runtime.BinaryStream object.

Constructors

BinaryStream(InputStream)

Format:

```
public BinaryStream(java.io.InputStream input-stream)
```

Creates an Binary java.io.InputStream object with an unspecified length.

Parameters:

input-stream
The InputStream object that SQLJ interprets as an BinaryStream object.

BinaryStream(InputStream, int)

Format:

```
public BinaryStream(java.io.InputStream input-stream, int length)
```

Creates an Binary java.io.InputStream object with a specified length.

Parameters:

input-stream
The InputStream object that SQLJ interprets as an BinaryStream object.

length
The length of the InputStream object that SQLJ interprets as an BinaryStream object.

sqlj.runtime.CharacterStream class

The sqlj.runtime.CharacterStream class is for an input stream of character data with a specified length.

The sqlj.runtime.CharacterStream class is derived from the java.io.Reader class, and extends the java.io.FilterReader class. SQLJ interprets the bytes in an sqlj.runtime.CharacterStream object are interpreted as Unicode data. A Reader object with Unicode data needs to be passed as a sqlj.runtime.CharacterStream object.

Constructors

CharacterStream(InputStream)

Format:

```
public CharacterStream(java.io.Reader input-stream)
```

Creates a character java.io.Reader object with an unspecified length.

Parameters:

input-stream
The Reader object that SQLJ interprets as an CharacterStream object.

CharacterStream(InputStream, int)

Format:

```
public CharacterStream(java.io.Reader input-stream, int length)
```

Creates a character java.io.Reader object with a specified length.

Parameters:

input-stream
The Reader object that SQLJ interprets as an CharacterStream object.

length
The length of the Reader object that SQLJ interprets as an CharacterStream object.

Methods

getReader
Format:

```
public Reader getReader()
```

Returns the underlying Reader object that is wrapped by the CharacterStream object.

getLength
Format:

```
public void getLength()
```

Returns the length in characters of the wrapped Reader object, as specified by the constructor or in the last call to setLength.

setLength
Format:

```
public void setLength (int length)
```

Sets the number of characters that are read from the Reader object when the object is passed as an input argument to an SQL operation.

Parameters:

length
The number of characters that are read from the Reader object.

sqlj.runtime.ExecutionContext class

The sqlj.runtime.ExecutionContext class is defined for execution contexts. An execution context is used to control the execution of SQL statements.

Variables

ADD_BATCH_COUNT
Format:

```
public static final int ADD_BATCH_COUNT
```

A constant that can be returned by the getUpdateCount method. It indicates that the previous statement was not executed but was added to the existing statement batch.

AUTO_BATCH
Format:

```
public static final int AUTO_BATCH
```

A constant that can be passed to the setBatchLimit method. It indicates that implicit batch execution should be performed, and that SQLJ should determine the batch size.

DBDefault
Format:

```
public static final short DBDefault=-5;
```

A constant that can be assigned to an indicator variable. It specifies that the corresponding host variable value that is passed to the data server is the default value.

DBNonNull

Format:

```
public static final short DBNonNull=0;
```

A constant that can be assigned to an indicator variable. It specifies that the corresponding host variable value that is passed to the data server is a non-null value.

DBNull

Format:

```
public static final short DBNull=-1;
```

A constant that can be assigned to an indicator variable. It specifies that the corresponding host variable value that is passed to the data server is the SQL NULL value.

DBUnassigned

Format:

```
public static final short DBUnassigned=-7;
```

A constant that can be assigned to an indicator variable. It specifies that no value for the corresponding host variable is passed to the data server.

EXEC_BATCH_COUNT

Format:

```
public static final int EXEC_BATCH_COUNT
```

A constant that can be returned from the getUpdateCount method. It indicates that a statement batch was just executed.

EXCEPTION_COUNT

Format:

```
public static final int EXCEPTION_COUNT
```

A constant that can be returned from the getUpdateCount method. It indicates that an exception was thrown before the previous execution completed, or that no operation has been performed on the execution context object.

NEW_BATCH_COUNT

Format:

```
public static final int NEW_BATCH_COUNT
```

A constant that can be returned from the getUpdateCount method. It indicates that the previous statement was not executed, but was added to a new statement batch.

QUERY_COUNT

Format:

```
public static final int QUERY_COUNT
```

A constant that can be passed to the setBatchLimit method. It indicates that the previous execution produced a result set.

UNLIMITED_BATCH

Format:

```
public static final int UNLIMITED_BATCH
```

A constant that can be returned from the getUpdateCount method. It indicates that statements should continue to be added to a statement batch, regardless of the batch size.

Constructors:

ExecutionContext

Format:

```
public ExecutionContext()
```

Creates an ExecutionContext instance.

Methods

cancel

Format:

```
public void cancel() throws SQLException
```

Cancels an SQL operation that is currently being executed by a thread that uses the execution context object. If there is a pending statement batch on the execution context object, the statement batch is canceled and cleared.

The cancel method throws an SQLException if the statement cannot be canceled.

execute

Format:

```
public boolean execute ( ) throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

executeBatch

Format:

```
public synchronized int[] executeBatch() throws SQLException
```

Executes the pending statement batch and returns an array of update counts. If no pending statement batch exists, null is returned. When this method is called, the statement batch is cleared, even if the call results in an exception.

Each element in the returned array can be one of the following values:

-2 This value indicates that the SQL statement executed successfully, but the number of rows that were updated could not be determined.

-3 This value indicates that the SQL statement failed.

Other integer
This value is the number of rows that were updated by the statement.

The executeBatch method throws an SQLException if a database error occurs while the statement batch executes.

executeQuery

Format:

```
public RTResultSet executeQuery ( ) throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

executeUpdate

Format:

```
public int executeUpdate() throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

getBatchLimit

Format:

```
synchronized public int getBatchLimit()
```

Returns the number of statements that are added to a batch before the batch is implicitly executed.

The returned value is one of the following values:

UNLIMITED_BATCH
This value indicates that the batch size is unlimited.

AUTO_BATCH
This value indicates that the batch size is finite but unknown.

Other integer
The current batch limit.

getBatchUpdateCounts

Format:

```
public synchronized int[] getBatchUpdateCounts()
```

Returns an array that contains the number of rows that were updated by each statement that successfully executed in a batch. The order of elements in the array corresponds to the order in which statements were inserted into the batch. Returns null if no statements in the batch completed successfully.

Each element in the returned array can be one of the following values:

-2 This value indicates that the SQL statement executed successfully, but the number of rows that were updated could not be determined.

-3 This value indicates that the SQL statement failed.

Other integer
This value is the number of rows that were updated by the statement.

getFetchDirection

Format:

```
synchronized public int getFetchDirection() throws SQLException
```

Returns the current fetch direction for scrollable iterator objects that were generated from the given execution context. If a fetch direction was not set for the execution context, sqlj.runtime.ResultSetIterator.FETCH_FORWARD is returned.

getFetchSize

Format:

```
synchronized public int getFetchSize() throws SQLException
```

Returns the number of rows that should be fetched by SQLJ when more rows are needed. This value applies only to iterator objects that were generated from the given execution context. The returned value is the value that was set by the setFetchSize method, or 0 if no value was set by setFetchSize.

getMaxFieldSize

Format:

```
public synchronized int getMaxFieldSize()
```

Returns the maximum number of bytes that are returned for any string (character, graphic, or varying-length binary) column in queries that use the given execution context. If this limit is exceeded, SQLJ discards the remaining bytes. A value of 0 means that the maximum number of bytes is unlimited.

getMaxRows

Format:

```
public synchronized int getMaxRows()
```

Returns the maximum number of rows that are returned for any query that uses the given execution context. If this limit is exceeded, SQLJ discards the remaining rows. A value of 0 means that the maximum number of rows is unlimited.

getNextResultSet()

Format:

```
public ResultSet getNextResultSet() throws SQLException
```

After a stored procedure call, returns a result set from the stored procedure.

A null value is returned if any of the following conditions are true:

- There are no more result sets to be returned.
- The stored procedure call did not produce any result sets.
- A stored procedure call has not been executed under the execution context.

When you invoke getNextResultSet(), SQLJ closes the currently-open result set and advances to the next result set.

If an error occurs during a call to getNextResultSet, resources for the current JDBC ResultSet object are released, and an SQLException is thrown. Subsequent calls to getNextResultSet return null.

getNextResultSet(int)

Formats:

```
public ResultSet getNextResultSet(int current)
```

After a stored procedure call, returns a result set from the stored procedure.

A null value is returned if any of the following conditions are true:

- There are no more result sets to be returned.
- The stored procedure call did not produce any result sets.
- A stored procedure call has not been executed under the execution context.

If an error occurs during a call to getNextResultSet, resources for the current JDBC ResultSet object are released, and an SQLException is thrown. Subsequent calls to getNextResultSet return null.

Parameters:

current
: Indicates what SQLJ does with the currently open result set before it advances to the next result set:

java.sql.Statement.CLOSE_CURRENT_RESULT
Specifies that the current ResultSet object is closed when the next ResultSet object is returned.

java.sql.Statement.KEEP_CURRENT_RESULT
Specifies that the current ResultSet object stays open when the next ResultSet object is returned.

java.sql.Statement.CLOSE_ALL_RESULTS
Specifies that all open ResultSet objects are closed when the next ResultSet object is returned.

getQueryTimeout

Format:

```
public synchronized int getQueryTimeout()
```

Returns the maximum number of seconds that SQL operations that use the given execution context object can execute. If an SQL operation exceeds the limit, an SQLException is thrown. The returned value is the value that was set by the setQueryTimeout method, or 0 if no value was set by setQueryTimeout. 0 means that execution time is unlimited.

getUpdateCount

Format:

```
public abstract int getUpdateCount() throws SQLException
```

Returns:

ExecutionContext.ADD_BATCH_COUNT
If the statement was added to an existing batch.

ExecutionContext.NEW_BATCH_COUNT
If the statement was the first statement in a new batch.

ExecutionContext.EXCEPTION_COUNT
If the previous statement generated an SQLException, or no previous statement was executed.

ExecutionContext.EXEC_BATCH_COUNT
If the statement was part of a batch, and the batch was executed.

ExecutionContext.QUERY_COUNT
If the previous statement created an iterator object or JDBC ResultSet.

Other integer
If the statement was executed rather than added to a batch. This value is the number of rows that were updated by the statement.

getWarnings

Format:

```
public synchronized SQLWarning getWarnings()
```

Returns the first warning that was reported by the last SQL operation that was executed using the given execution context. Subsequent warnings are chained to the first warning. If no warnings occurred, null is returned.

getWarnings is used to retrieve positive SQLCODEs.

isBatching

Format:

```
public synchronized boolean isBatching()
```

Returns `true` if batching is enabled for the execution context. Returns `false` if batching is disabled.

registerStatement

Format:

```
public RTStatement registerStatement(ConnectionContext connCtx,
 Object profileKey, int stmtNdx)
 throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

releaseStatement

Format:

```
public void releaseStatement() throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

setBatching

Format:

```
public synchronized void setBatching(boolean batching)
```

Parameters:

batching
Indicates whether batchable statements that are registered with the given execution context can be added to a statement batch:

true
Statements can be added to a statement batch.

false
Statements are executed individually.

setBatching affects only statements that occur in the program after setBatching is called. It does not affect previous statements or an existing statement batch.

setBatchLimit

Format:

```
public synchronized void setBatchLimit(int batch-size)
```

Sets the maximum number of statements that are added to a batch before the batch is implicitly executed.

Parameters:

batch-size
One of the following values:

ExecutionContext.UNLIMITED_BATCH
Indicates that implicit execution occurs only when SQLJ encounters a statement that is batchable but incompatible, or not batchable. Setting this value is the same as not invoking setBatchLimit.

ExecutionContext.AUTO_BATCH
Indicates that implicit execution occurs when the number of statements in the batch reaches a number that is set by SQLJ.

Positive integer
The number of statements that are added to the batch before SQLJ executes the batch implicitly. The batch might be executed before this

many statements have been added if SQLJ encounters a statement that is batchable but incompatible, or not batchable.

setBatchLimit affects only statements that occur in the program after setBatchLimit is called. It does not affect an existing statement batch.

setFetchDirection

Format:

```
public synchronized void setFetchDirection(int direction) throws SQLException
```

Gives SQLJ a hint as to the current fetch direction for scrollable iterator objects that were generated from the given execution context.

Parameters:

direction

One of the following values:

sqlj.runtime.ResultSetIterator.FETCH_FORWARD
Rows are fetched in a forward direction. This is the default.

sqlj.runtime.ResultSetIterator.FETCH_REVERSE
Rows are fetched in a backward direction.

sqlj.runtime.ResultSetIterator.FETCH_UNKNOWN
The order of fetching is unknown.

Any other input value results in an SQLException.

setFetchSize

Format:

```
synchronized public void setFetchSize(int number-of-rows) throws SQLException
```

Gives SQLJ a hint as to the number of rows that should be fetched when more rows are needed.

Parameters:

number-of-rows

The expected number of rows that SQLJ should fetch for the iterator that is associated with the given execution context.

If *number-of-rows* is less than 0 or greater than the maximum number of rows that can be fetched, an SQLException is thrown.

setMaxFieldSize

Format:

```
public void setMaxFieldSize(int max-bytes)
```

Specifies the maximum number of bytes that are returned for any string (character, graphic, or varying-length binary) column in queries that use the given execution context. If this limit is exceeded, SQLJ discards the remaining bytes.

Parameters:

max-bytes

The maximum number of bytes that SQLJ should return from a BINARY, VARBINARY, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC column. A value of 0 means that the number of bytes is unlimited. 0 is the default.

setMaxRows

Format:

```
public synchronized void setMaxRows(int max-rows)
```

Specifies the maximum number of rows that are returned for any query that uses the given execution context. If this limit is exceeded, SQLJ discards the remaining rows.

Parameters:

max-rows
The maximum number of rows that SQLJ should return for a query that uses the given execution context. A value of 0 means that the number of rows is unlimited. 0 is the default.

setQueryTimeout

Format:

```
public synchronized void setQueryTimeout(int timeout-value)
```

Specifies the maximum number of seconds that SQL operations that use the given execution context object can execute. If an SQL operation exceeds the limit, an SQLException is thrown.

Parameters:

timeout-value
The maximum number of seconds that SQL operations that use the given execution context object can execute. 0 means that execution time is unlimited. 0 is the default.

sqlj.runtime.SQLNullException class

The sqlj.runtime.SQLNullException class is derived from the java.sql.SQLException class.

An sqlj.runtime.SQLNullException is thrown when an SQL NULL value is fetched into a host identifier with a Java primitive type. The SQLSTATE value for an instance of SQLNullException is '22002'.

sqlj.runtime.StreamWrapper class

The sqlj.runtime.StreamWrapper class wraps a java.io.InputStream instance and extends the java.io.InputStream class.

The sqlj.runtime.AsciiStream, sqlj.runtime.BinaryStream, and sqlj.runtime.UnicodeStream classes extend sqlj.runtime.StreamWrapper. `sqlj.runtime.StreamWrapper` supports methods for specifying the length of sqlj.runtime.AsciiStream, sqlj.runtime.BinaryStream, and sqlj.runtime.UnicodeStream objects.

Constructors

StreamWrapper(InputStream)

Format:

```
protected StreamWrapper(InputStream input-stream)
```

Creates an sqlj.runtime.StreamWrapper object with an unspecified length.

Parameters:

input-stream
The InputStream object that the sqlj.runtime.StreamWrapper object wraps.

StreamWrapper(InputStream, int)
Format:

```
protected StreamWrapper(java.io.InputStream input-stream, int length)
```

Creates an sqlj.runtime.StreamWrapper object with a specified length.

Parameters:

input-stream
The InputStream object that the sqlj.runtime.StreamWrapper object wraps.

length
The length of the InputStream object in bytes.

Methods

getInputStream
Format:

```
public InputStream getInputStream()
```

Returns the underlying InputStream object that is wrapped by the StreamWrapper object.

getLength
Format:

```
public void getLength()
```

Returns the length in bytes of the wrapped InputStream object, as specified by the constructor or in the last call to setLength.

setLength
Format:

```
public void setLength (int length)
```

Sets the number of bytes that are read from the wrapped InputStream object when the object is passed as an input argument to an SQL operation.

Parameters:

length
The number of bytes that are read from the wrapped InputStream object.

sqlj.runtime.UnicodeStream class

The sqlj.runtime.UnicodeStream class is for an input stream of Unicode data with a specified length.

The sqlj.runtime.UnicodeStream class is derived from the java.io.InputStream class, and extends the sqlj.runtime.StreamWrapper class. SQLJ interprets the bytes in an sqlj.runtime.UnicodeStream object as Unicode characters. An InputStream object with Unicode characters needs to be passed as a sqlj.runtime.UnicodeStream object.

Constructors

UnicodeStream(InputStream)
Format:

```
public UnicodeStream(java.io.InputStream input-stream)
```

Creates a Unicode java.io.InputStream object with an unspecified length.

Parameters:

input-stream
The InputStream object that SQLJ interprets as an UnicodeStream object.

UnicodeStream(InputStream, int)
Format:

```
public UnicodeStream(java.io.InputStream input-stream, int length)
```

Creates a Unicode java.io.InputStream object with a specified length.

Parameters:

input-stream
The InputStream object that SQLJ interprets as an UnicodeStream object.

length
The length of the InputStream object that SQLJ interprets as an UnicodeStream object.

IBM Data Server Driver for JDBC and SQLJ extensions to JDBC

The IBM Data Server Driver for JDBC and SQLJ provides a set of extensions to the support that is provided by the JDBC specification.

To use IBM Data Server Driver for JDBC and SQLJ-only methods in classes that have corresponding, standard classes, cast an instance of the related, standard JDBC class to an instance of the IBM Data Server Driver for JDBC and SQLJ-only class. For example:

```
javax.sql.DataSource ds =
 new com.ibm.db2.jcc.DB2SimpleDataSource();
((com.ibm.db2.jcc.DB2BaseDataSource) ds).setServerName("sysmvs1.stl.ibm.com");
```

Table 14-58 summarizes the IBM Data Server Driver for JDBC and SQLJ-only interfaces.

Table 14-58. Summary of IBM Data Server Driver for JDBC and SQLJ-only interfaces provided by the IBM Data Server Driver for JDBC and SQLJ

Interface name	Applicable data sources	Purpose
DB2CallableStatement	1 on page 14-131, 2 on page 14-131	Extends the java.sql.CallableStatement and the com.ibm.db2.jcc.DB2PreparedStatement interfaces.
DB2Connection	1 on page 14-131, 2 on page 14-131, 3 on page 14-131	Extends the java.sql.Connection interface.
DB2DatabaseMetaData	1 on page 14-131, 2 on page 14-131, 3 on page 14-131	Extends the java.sql.DatabaseMetaData interface.
DB2Diagnosable	1 on page 14-131, 2 on page 14-131, 3 on page 14-131	Provides a mechanism for getting DB2 diagnostics from a DB2 SQLException.
DB2ParameterMetaData	2 on page 14-131	Extends the java.sql.ParameterMetaData interface.
DB2PreparedStatement	1 on page 14-131, 2 on page 14-131, 3 on page 14-131	Extends the com.ibm.db2.jcc.DB2Statement and java.sql.PreparedStatement interfaces.
DB2ResultSet	1 on page 14-131, 2 on page 14-131, 3 on page 14-131	Extends the java.sql.ResultSet interface.
DB2RowID	1 on page 14-131, 2 on page 14-131	Used for declaring Java objects for use with the ROWID data type.

Table 14-58. Summary of IBM Data Server Driver for JDBC and SQLJ-only interfaces provided by the IBM Data Server Driver for JDBC and SQLJ (continued)

Interface name	Applicable data sources	Purpose
DB2Statement	1, 2, 3	Extends the java.sql.Statement interface.
DB2Struct	2	Provides methods for working with java.sql.Struct objects.
DB2SystemMonitor	1, 2, 3	Used for collecting system monitoring data for a connection.
DB2TraceManagerMXBean	1, 2, 3	Provides the MBean interface for the remote trace controller.
DB2Xml	1, 2	Used for updating data in XML columns and retrieving data from XML columns.
DBBatchUpdateException	1, 2, 3	Used for retrieving error information about batch execution of statements that return automatically generated keys.

Note: The interface applies to connections to the following data sources:

1. DB2 for z/OS
2. DB2 Database for Linux, UNIX, and Windows
3. IBM Informix

Table 14-59 summarizes the IBM Data Server Driver for JDBC and SQLJ-only classes.

Table 14-59. Summary of IBM Data Server Driver for JDBC and SQLJ-only classes provided by the IBM Data Server Driver for JDBC and SQLJ

Class name	Applicable data sources	Purpose
DB2Administrator (DB2 Database for Linux, UNIX, and Windows only)	2 on page 14-132	Instances of the DB2Administrator class are used to retrieve DB2CataloguedDatabase objects.
DB2BaseDataSource	1 on page 14-132, 2 on page 14-132, 3 on page 14-132	The abstract data source parent class for all IBM Data Server Driver for JDBC and SQLJ-specific implementations of javax.sql.DataSource, javax.sql.ConnectionPoolDataSource, and javax.sql.XADataSource.
DB2CataloguedDatabase	2 on page 14-132	Contains methods that retrieve information about a local DB2 Database for Linux, UNIX, and Windows database.
DB2ClientRerouteServerList	1 on page 14-132, 2 on page 14-132	Implements the java.io.Serializable and javax.naming.Referenceable interfaces.
DB2ConnectionPoolDataSource	1 on page 14-132, 2 on page 14-132, 3 on page 14-132	A factory for PooledConnection objects.
DB2ExceptionFormatter	1 on page 14-132, 2 on page 14-132, 3 on page 14-132	Contains methods for printing diagnostic information to a stream.
DB2JCCPlugin	2 on page 14-132	The abstract class for implementation of JDBC security plug-ins.
DB2PooledConnection	1 on page 14-132, 2 on page 14-132, 3 on page 14-132	Provides methods that an application server can use to switch users on a preexisting trusted connection.

Table 14-59. Summary of IBM Data Server Driver for JDBC and SQLJ-only classes provided by the IBM Data Server Driver for JDBC and SQLJ (continued)

Class name	Applicable data sources	Purpose
DB2PoolMonitor	1, 2	Provides methods for monitoring the global transport objects pool for the connection concentrator and Sysplex workload balancing.
DB2SimpleDataSource	1, 2, 3	Extends the DataBaseDataSource class. Does not support connection pooling or distributed transactions.
DB2Sqlca	1, 2, 3	An encapsulation of the DB2 SQLCA.
DB2TraceManager	1, 2, 3	Controls the global log writer.
DB2Types	1 on page 14-131	Defines data type constants.
DB2XADataSource	1, 2, 3	A factory for XADataSource objects. An object that implements this interface is registered with a naming service that is based on the Java Naming and Directory Interface (JNDI).
DBTimestamp	1, 2, 3	A subclass of Timestamp that handles timestamp values with extra precision or time zone information.

Note: The class applies to connections to the following data sources:

1. DB2 for z/OS
2. DB2 Database for Linux, UNIX, and Windows
3. IBM Informix

DBBatchUpdateException interface

The com.ibm.db2.jcc.DBBatchUpdateException interface is used for retrieving error information about batch execution of statements that return automatically generated keys.

DBBatchUpdateException methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getDBGeneratedKeys

Format:

```
public java.sql.ResultSet[] getDBGeneratedKeys()
  throws java.sql.SQLException
```

Retrieves automatically generated keys that were created when INSERT statements were executed in a batch. Each ResultSet object that is returned contains the automatically generated keys for a single statement in the batch. ResultSet objects that are null correspond to failed statements.

DB2BaseDataSource class

The com.ibm.db2.jcc.DB2BaseDataSource class is the abstract data source parent class for all IBM Data Server Driver for JDBC and SQLJ-specific implementations of javax.sql.DataSource, javax.sql.ConnectionPoolDataSource, and javax.sql.XADataSource.

DB2BaseDataSource implements the java.sql.Wrapper interface.

DB2BaseDataSource properties

The following properties are defined only for the IBM Data Server Driver for JDBC and SQLJ.

You can set all properties on a DataSource or in the *url* parameter in a DriverManager.getConnection call.

All properties **except** the following properties have a setXXX method to set the value of the property and a getXXX method to retrieve the value:
- dumpPool
- dumpPoolStatisticsOnSchedule
- dumpPoolStatisticsOnScheduleFile
- maxRefreshInterval
- maxTransportObjectIdleTime
- maxTransportObjectWaitTime
- minTransportObjects

A setXXX method has this form:

```
void setProperty-name(data-type property-value)
```

A getXXX method has this form:

```
data-type getProperty-name()
```

Property-name is the unqualified property name. For properties that are not specific to IBM Informix, the first character of the property name is capitalized. For properties that are used only by IBM Informix, all characters of the property name are capitalized.

The following table lists the IBM Data Server Driver for JDBC and SQLJ properties and their data types.

Table 14-60. DB2BaseDataSource properties and their data types

Property name	Applicable data sources	Data type
com.ibm.db2.jcc.DB2BaseDataSource.accountingInterval	1	String
com.ibm.db2.jcc.DB2BaseDataSource.affinityFailbackInterval	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.allowNextOnExhaustedResultSet	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.allowNullResultSetForExecuteQuery	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.atomicMultiRowInsert	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.blockingReadConnectionTimeout	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.charOutputSize	1	short
com.ibm.db2.jcc.DB2BaseDataSource.clientAccountingInformation	1, 2	String
com.ibm.db2.jcc.DB2BaseDataSource.clientApplicationInformation	1, 2	String
com.ibm.db2.jcc.DB2BaseDataSource.clientDebugInfo (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	1, 2	String
com.ibm.db2.jcc.DB2BaseDataSource.clientProgramId	1, 2	String
com.ibm.db2.jcc.DB2BaseDataSource.clientProgramName (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	1, 2	String
com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteAlternateServerName	1, 2, 3	String
com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteAlternatePortNumber	1, 2, 3	String
com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteServerListJNDIContext	1, 2, 3	javax.naming.Context
com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteServerListJNDIName	1, 2, 3	String

Table 14-60. DB2BaseDataSource properties and their data types (continued)

Property name	Applicable data sources	Data type
com.ibm.db2.jcc.DB2BaseDataSource.clientUser (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS only)	1	String
com.ibm.db2.jcc.DB2BaseDataSource.clientWorkstation (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS only)	1	String
com.ibm.db2.jcc.DB2BaseDataSource.connectionCloseWithInFlightTransaction	1, 2, 3	String
com.ibm.db2.jcc.DB2BaseDataSource.concurrentAccessResolution	1, 2	int
com.ibm.db2.jcc.DB2BaseDataSource.connectNode	2	int
com.ibm.db2.jcc.DB2BaseDataSource.currentDegree	1, 2	String
com.ibm.db2.jcc.DB2BaseDataSource.currentExplainMode	1, 2	String
com.ibm.db2.jcc.DB2BaseDataSource.currentExplainSnapshot	2	String
com.ibm.db2.jcc.DB2BaseDataSource.currentFunctionPath	1, 2	String
com.ibm.db2.jcc.DB2BaseDataSource.currentLockTimeout	2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.currentMaintainedTableTypesForOptimization	1, 2	String
com.ibm.db2.jcc.DB2BaseDataSource.currentPackagePath	1, 2	String
com.ibm.db2.jcc.DB2BaseDataSource.currentPackageSet	1, 2	String
com.ibm.db2.jcc.DB2BaseDataSource.currentQueryOptimization	2	int
com.ibm.db2.jcc.DB2BaseDataSource.currentRefreshAge	1, 2	long
com.ibm.db2.jcc.DB2BaseDataSource.currentSchema	1, 2	String
com.ibm.db2.jcc.DB2BaseDataSource.cursorSensitivity	1, 2	int
com.ibm.db2.jcc.DB2BaseDataSource.currentSQLID	1	String
com.ibm.db2.jcc.DB2BaseDataSource.databaseName	1, 2, 3	String
com.ibm.db2.jcc.DB2BaseDataSource.dateFormat	1, 2	int
com.ibm.db2.jcc.DB2BaseDataSource.decimalSeparator	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.decimalStringFormat	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.defaultIsolationLevel	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.deferPrepares	1, 2, 3	boolean
com.ibm.db2.jcc.DB2BaseDataSource.description	1, 2, 3	String
com.ibm.db2.jcc.DB2BaseDataSource.downgradeHoldCursorsUnderXa	1, 2,3	boolean
com.ibm.db2.jcc.DB2BaseDataSource.driverType	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.dumpPool	3	int
com.ibm.db2.jcc.DB2BaseDataSource.dumpPoolStatisticsOnSchedule	3	int
com.ibm.db2.jcc.DB2BaseDataSource.dumpPoolStatisticsOnScheduleFile	3	String
com.ibm.db2.jcc.DB2BaseDataSource.enableClientAffinitiesList	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.enableExtendedIndicators	1, 2	int
com.ibm.db2.jcc.DB2BaseDataSource.enableNamedParameterMarkers	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.enableConnectionConcentrator (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	1, 3	boolean
com.ibm.db2.jcc.DB2BaseDataSource.enableMultiRowInsertSupport	1	boolean
com.ibm.db2.jcc.DB2BaseDataSource.enableRowsetSupport	1, 2	int
com.ibm.db2.jcc.DB2BaseDataSource.enableSeamlessFailover	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.enableSysplexWLB (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	1, 2, 3	boolean
com.ibm.db2.jcc.DB2BaseDataSource.encryptionAlgorithm	1, 2	int
com.ibm.db2.jcc.DB2BaseDataSource.fetchSize	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.floatingPointStringFormat	1, 2, 3	int

Table 14-60. DB2BaseDataSource properties and their data types (continued)

Property name	Applicable data sources	Data type
com.ibm.db2.jcc.DB2BaseDataSource.fullyMaterializeInputStreams	1, 2	boolean
com.ibm.db2.jcc.DB2BaseDataSource.fullyMaterializeLobData	1, 2, 3	boolean
com.ibm.db2.jcc.DB2BaseDataSource.gssCredential	1, 2	Object
com.ibm.db2.jcc.DB2BaseDataSource.interruptProcessingMode (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.jdbcCollection	1	String
com.ibm.db2.jcc.DB2BaseDataSource.keepDynamic	1, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.kerberosServerPrincipal	1, 2	String
com.ibm.db2.jcc.DB2BaseDataSource.loginTimeout (not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS)	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.logWriter	1, 2, 3	PrintWriter
com.ibm.db2.jcc.DB2BaseDataSource.maxRetriesForClientReroute	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.maxRowsetSize (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS only)	1	int
com.ibm.db2.jcc.DB2BaseDataSource.maxTransportObjectIdleTime	3	int
com.ibm.db2.jcc.DB2BaseDataSource.maxTransportObjectWaitTime	3	int
com.ibm.db2.jcc.DB2BaseDataSource.maxTransportObjects	1, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.minTransportObjects	3	int
com.ibm.db2.jcc.DB2BaseDataSource.optimizationProfile	2	String
com.ibm.db2.jcc.DB2BaseDataSource.optimizationProfileToFlush	2	String
com.ibm.db2.jcc.DB2BaseDataSource.password	1, 2, 3	String
com.ibm.db2.jcc.DB2BaseDataSource.pdqProperties	1, 2	String
com.ibm.db2.jcc.DB2BaseDataSource.pkList (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity only)	1	String
com.ibm.db2.jcc.DB2BaseDataSource.planName (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity only)	1	String
com.ibm.db2.jcc.DB2BaseDataSource.plugin	2	Object
com.ibm.db2.jcc.DB2BaseDataSource.pluginName	2	String
com.ibm.db2.jcc.DB2BaseDataSource.portNumber	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.progressiveStreaming	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.queryCloseImplicit	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.queryDataSize	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.queryTimeoutProcessingMode	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.readOnly	1, 2	boolean
com.ibm.db2.jcc.DB2BaseDataSource.reportLongTypes	1	short
com.ibm.db2.jcc.DB2BaseDataSource.resultSetHoldability	1, 2,3	int
com.ibm.db2.jcc.DB2BaseDataSource.resultSetHoldabilityForCatalogQueries	1, 2	int
com.ibm.db2.jcc.DB2BaseDataSource.retrieveMessagesFromServerOnGetMessage	1, 2, 3	boolean
com.ibm.db2.jcc.DB2BaseDataSource.retryIntervalForClientReroute	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.retryWithAlternativeSecurityMechanism (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	2	int
com.ibm.db2.jcc.DB2BaseDataSource.returnAlias	1, 2	short
com.ibm.db2.jcc.DB2BaseDataSource.securityMechanism	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.sendCharInputsUTF8	1	int
com.ibm.db2.jcc.DB2BaseDataSource.sendDataAsIs	1, 2, 3	boolean
com.ibm.db2.jcc.DB2BaseDataSource.serverName	1, 2, 3	String

Table 14-60. DB2BaseDataSource properties and their data types (continued)

Property name	Applicable data sources	Data type
com.ibm.db2.jcc.DB2BaseDataSource.sessionTimeZone	1	String
com.ibm.db2.jcc.DB2BaseDataSource.sqljEnableClassLoaderSpecificProfiles	1	boolean
com.ibm.db2.jcc.DB2BaseDataSource.ssid (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS only)	1	String
com.ibm.db2.jcc.DB2BaseDataSource.sslConnection (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	1, 2, 3	boolean
com.ibm.db2.jcc.DB2BaseDataSource.sslTrustStoreLocation (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	1, 2, 3	String
com.ibm.db2.jcc.DB2BaseDataSource.sslTrustStorePassword (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	1, 2, 3	String
com.ibm.db2.jcc.DB2BaseDataSource.statementConcentrator	1, 2	int
com.ibm.db2.jcc.DB2BaseDataSource.streamBufferSize	1, 2	int
com.ibm.db2.jcc.DB2BaseDataSource.stripTrailingZerosForDecimalNumbers	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.supportsAsynchronousXARollback	1, 2	int
com.ibm.db2.jcc.DB2BaseDataSource.sysSchema	1, 2	String
com.ibm.db2.jcc.DB2BaseDataSource.timeFormat	1, 2	int
com.ibm.db2.jcc.DB2BaseDataSource.timestampFormat	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.timestampPrecisionReporting	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.traceDirectory	1, 2, 3	String
com.ibm.db2.jcc.DB2BaseDataSource.traceFile	1, 2, 3	String
com.ibm.db2.jcc.DB2BaseDataSource.traceFileAppend	1, 2, 3	boolean
com.ibm.db2.jcc.DB2BaseDataSource.traceLevel	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.useCachedCursor	1, 2	boolean
com.ibm.db2.jcc.DB2BaseDataSource.useJDBC4ColumnNameAndLabelSemantics	1, 2	int
com.ibm.db2.jcc.DB2BaseDataSource.user	1, 2, 3	String
com.ibm.db2.jcc.DB2BaseDataSource.useIdentityValLocalForAutoGeneratedKeys	1	boolean
com.ibm.db2.jcc.DB2BaseDataSource.useRowsetCursor	1	boolean
com.ibm.db2.jcc.DB2BaseDataSource.useTransactionRedirect	2	boolean
com.ibm.db2.jcc.DB2BaseDataSource.xaNetworkOptimization	1, 2, 3	boolean
com.ibm.db2.jcc.DB2BaseDataSource.xmlFormat	1	int
com.ibm.db2.jcc.DB2BaseDataSource.DBANSIWARN	3	boolean
com.ibm.db2.jcc.DB2BaseDataSource.DBDATE	3	String
com.ibm.db2.jcc.DB2BaseDataSource.DBPATH	3	String
com.ibm.db2.jcc.DB2BaseDataSource.DBSPACETEMP	3	String
com.ibm.db2.jcc.DB2BaseDataSource.DBTEMP	3	String
com.ibm.db2.jcc.DB2BaseDataSource.DBUPSPACE	3	String
com.ibm.db2.jcc.DB2BaseDataSource.DELIMIDENT	3	boolean
com.ibm.db2.jcc.DB2BaseDataSource.IFX_DIRECTIVES	3	String
com.ibm.db2.jcc.DB2BaseDataSource.IFX_EXTDIRECTIVES	3	String
com.ibm.db2.jcc.DB2BaseDataSource.IFX_UPDDESC	3	String
com.ibm.db2.jcc.DB2BaseDataSource.IFX_XASTDCOMPLIANCE_XAEND	3	String
com.ibm.db2.jcc.DB2BaseDataSource.INFORMIXOPCACHE	3	String
com.ibm.db2.jcc.DB2BaseDataSource.INFORMIXSTACKSIZE	3	String
com.ibm.db2.jcc.DB2BaseDataSource.NODEFDAC	3	String
com.ibm.db2.jcc.DB2BaseDataSource.OPTCOMPIND	3	String

Table 14-60. DB2BaseDataSource properties and their data types (continued)

Property name	Applicable data sources	Data type
com.ibm.db2.jcc.DB2BaseDataSource.OPTOFC	3	String
com.ibm.db2.jcc.DB2BaseDataSource.PDQPRIORITY	3	String
com.ibm.db2.jcc.DB2BaseDataSource.PSORT_DBTEMP	3	String
com.ibm.db2.jcc.DB2BaseDataSource.PSORT_NPROCS	3	String
com.ibm.db2.jcc.DB2BaseDataSource.STMT_CACHE	3	String

Note: The property applies to connections to the following data sources:

1. DB2 for z/OS
2. DB2 Database for Linux, UNIX, and Windows
3. IBM Informix

DB2BaseDataSource fields

The following constants are defined only for the IBM Data Server Driver for JDBC and SQLJ.

public final static int INTERRUPT_PROCESSING_MODE_DISABLED = 0
A constant for the interruptProcessingMode property. This value indicates that interrupt processing is disabled.

public final static int INTERRUPT_PROCESSING_MODE_STATEMENT_CANCEL = 1
A constant for the interruptProcessingMode property. This value indicates that the IBM Data Server Driver for JDBC and SQLJ cancels the currently executing statement when an application executes Statement.cancel, if the data server supports interrupt processing.

public final static int INTERRUPT_PROCESSING_MODE_CLOSE_SOCKET = 2
A constant for the interruptProcessingMode property. This value indicates that the IBM Data Server Driver for JDBC and SQLJ drops the underlying socket and closes the connection when an application executes Statement.cancel.

public final static int NO = 2
The NO value for properties.

public final static int NOT_SET = 0
The default value for properties.

public final static int YES = 1
The YES value for properties.

DB2BaseDataSource methods

In addition to the getXXX and setXXX methods for the DB2BaseDataSource properties, the following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getReference
Format:

```
public javax.naming.Reference getReference()
  throws javax.naming.NamingException
```

Retrieves the Reference of a DataSource object. For an explanation of a Reference, see the description of javax.naming.Referenceable in the Java Platform Standard Edition documentation.

DB2ClientRerouteServerList class

The com.ibm.db2.jcc.DB2ClientRerouteServerList class implements the java.io.Serializable and javax.naming.Referenceable interfaces.

DB2ClientRerouteServerList methods

getAlternatePortNumber

Format:

```
public int[] getAlternatePortNumber()
```

Retrieves the port numbers that are associated with the alternate servers.

getAlternateServerName

Format:

```
public String[] getAlternateServerName()
```

Retrieves an array that contains the names of the alternate servers. These values are IP addresses or DNS server names.

getPrimaryPortNumber

Format:

```
public int getPrimaryPortNumber()
```

Retrieves the port number that is associated with the primary server.

getPrimaryServerName

Format:

```
public String[] getPrimaryServerName()
```

Retrieves the name of the primary server. This value is an IP address or a DNS server name.

setAlternatePortNumber

Format:

```
public void setAlternatePortNumber(int[] alternatePortNumberList)
```

Sets the port numbers that are associated with the alternate servers.

setAlternateServerName

Format:

```
public void setAlternateServerName(String[] alternateServer)
```

Sets the alternate server names for servers. These values are IP addresses or DNS server names.

setPrimaryPortNumber

Format:

```
public void setPrimaryPortNumber(int primaryPortNumber)
```

Sets the port number that is associated with the primary server.

setPrimaryServerName

Format:

```
public void setPrimaryServerName(String primaryServer)
```

Sets the primary server name for a server. This value is an IP address or a DNS server name.

DB2Connection interface

The com.ibm.db2.jcc.DB2Connection interface extends the java.sql.Connection interface.

DB2Connection implements the java.sql.Wrapper interface.

DB2Connection fields

The following fields are defined only for the IBM Data Server Driver for JDBC and SQLJ, and apply only to connections to IBM Informix databases.

public int TRANSACTION_IDS_CURSOR_STABILITY (-1)
Specifies a level of transaction isolation for a connection. With this level of transaction isolation, an application acquires a shared lock on a fetched row. That lock is released when the application fetches another row or closes the cursor. Other applications can take a shared lock on the same row, but no process can acquire an exclusive lock to modify data in the row.

Use TRANSACTION_IDS_CURSOR_STABILITY for programs that fetch a row from a table and update another table with the contents of that row.

TRANSACTION_IDS_CURSOR_STABILITY has the same behavior as java.sql.Connection.TRANSACTION_READ_COMMITTED when an application is not performing database updates.

public int TRANSACTION_IDS_LAST_COMMITTED (-2)
Specifies a level of transaction isolation for a connection. With this level of transaction isolation, when an application attempts to read a row of a table, and another application has an exclusive lock on the table, the data source returns the most recently committed version of the row to the first application.

Use TRANSACTION_IDS_LAST_COMMITTED in programs for which currency of the data is less important than avoidance of deadlocks or timeouts.

DB2Connection methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

alternateWasUsedOnConnect
Format:

```
public boolean alternateWasUsedOnConnect()
  throws java.sql.SQLException
```

Returns `true` if the driver used alternate server information to obtain the connection. The alternate server information is available in the transient clientRerouteServerList information on the DB2BaseDataSource, which the database server updates as primary and alternate servers change.

changeDB2Password
Format:

```
public abstract void changeDB2Password(String oldPassword,
  String newPassword)
  throws java.sql.SQLException
```

Changes the password for accessing the data source, for the user of the Connection object.

Parameter descriptions:

oldPassword
The original password for the Connection.

newPassword
The new password for the Connection.

createArrayOf

Format:

```
Array createArrayOf(String typeName,
  Object[] elements)
  throws SQLException;
```

Creates a java.sql.Array object.

Parameter descriptions:

typeName
The SQL data type of the elements of the array map to. *typeName* can be a built-in data type or a distinct type.

elements
The elements that populate the Array object.

getDB2ClientAccountingInformation

Format:

```
public String getDB2ClientAccountingInformation()
  throws SQLException
```

Returns accounting information for the current client.

Important: getDB2ClientAccountingInformation is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use java.sql.Connection.getClientInfo instead.

getDB2ClientApplicationInformation

Format:

```
public String getDB2ClientApplicationInformation()
  throws java.sql.SQLException
```

Returns application information for the current client.

Important: getDB2ClientApplicationInformation is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use java.sql.Connection.getClientInfo instead.

getDB2ClientProgramId

Format:

```
public String getDB2ClientProgramId()
  throws java.sql.SQLException
```

Returns the user-defined program identifier for the client. The program identifier can be used to identify the application at the data source.

getDB2ClientProgramId does not apply to DB2 Database for Linux, UNIX, and Windows data servers.

getDB2ClientUser

Format:

```
public String getDB2ClientUser()
  throws java.sql.SQLException
```

Returns the current client user name for the connection. This name is not the user value for the JDBC connection.

Important: getDB2ClientUser is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use java.sql.Connection.getClientInfo instead.

getDB2ClientWorkstation

Format:

```
public String getDB2ClientWorkstation()
  throws java.sql.SQLException
```

Returns current client workstation name for the current client.

Important: getDB2ClientWorkstation is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use java.sql.Connection.getClientInfo instead.

getDB2Correlator

Format:

```
String getDB2Correlator()
  throws java.sql.SQLException
```

Returns the value of the crrtkn (correlation token) instance variable that DRDA sends with the ACCRDB command. The correlation token uniquely identifies a logical connection to a server.

getDB2CurrentPackagePath

Format:

```
public String getDB2CurrentPackagePath()
  throws java.sql.SQLException
```

Returns the list of DB2 package collections that are searched for JDBC and SQLJ packages.

The getDB2CurrentPackagePath method applies only to connections to DB2 database systems.

getDB2CurrentPackageSet

Format:

```
public String getDB2CurrentPackageSet()
  throws java.sql.SQLException
```

Returns the collection ID for the connection.

The getDB2CurrentPackageSet method applies only to connections to DB2 database systems.

getDB2SecurityMechanism

Format:

```
public int getDB2SecurityMechanism()
  throws java.sql.SQLException
```

Returns the security mechanism that is in effect for the connection:

3 Clear text password security

4 User ID-only security

7 Encrypted password security

9 Encrypted user ID and password security

11 Kerberos security

12 Encrypted user ID and data security

13 Encrypted user ID, password, and data security

15 Plugin security

16 Encrypted user ID-only security

getDB2SystemMonitor

Format:

```
public abstract DB2SystemMonitor getDB2SystemMonitor()
  throws java.sql.SQLException
```

Returns the system monitor object for the connection. Each IBM Data Server Driver for JDBC and SQLJ connection can have a single system monitor.

getDBConcurrentAccessResolution

Format:

```
public int getDBConcurrentAccessResolution()
  throws java.sql.SQLException
```

Returns the concurrent access setting for the connection. The concurrent access setting is set by the setDBConcurrentAccessResolution method or by the concurrentAccessResolution property.

getDBConcurrentAccessResolution applies only to connections to DB2 for z/OS and DB2 Database for Linux, UNIX, and Windows.

getDBStatementConcentrator

Format:

```
public int getDBStatementConcentrator()
  throws java.sql.SQLException
```

Returns the statement concentrator use setting for the connection. The statement concentrator use setting is set by the setDBStatementConcentrator method or by the statementConcentrator property.

getJccLogWriter

Format:

```
public PrintWriter getJccLogWriter()
  throws java.sql.SQLException
```

Returns the current trace destination for the IBM Data Server Driver for JDBC and SQLJ trace.

getJccSpecialRegisterProperties

Format:

```
public java.util.Properties getJccSpecialRegisterProperties()
  throws java.sql.SQLException
```

Returns a java.util.Properties object, in which the keys are the special registers that are supported at the target data source, and the key values are the current values of those special registers.

This method does not apply to connections to IBM Informix data sources.

getSavePointUniqueOption

Format:

```
public boolean getSavePointUniqueOption()
  throws java.sql.SQLException
```

Returns `true` if setSavePointUniqueOption was most recently called with a value of `true`. Returns `false` otherwise.

isDB2Alive

Format:

```
public boolean DB2Connection.isDB2Alive()
  throws java.sql.SQLException
```

Returns `true` if the socket for a connection to the data source is still active.

Important: isDB2Alive is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use Connection.isDBValid instead.

reconfigureDB2Connection

Format:

```
public void reconfigureDB2Connection(java.util.Properties properties)
  throws SQLException
```

Reconfigures a connection with new settings. The connection does not need to be returned to a connection pool before it is reconfigured. This method can be called while a transaction is in progress, and can be used for trusted or untrusted connections.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to:
  - DB2 Database for Linux, UNIX, and Windows Version 9.5 or later
  - DB2 for z/OS Version 9.1 or later
  - IBM Informix Version 11.70 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 9.1 or later

Parameter descriptions:

properties
New properties for the connection. These properties override any properties that are already defined on the DB2Connection instance.

setDBConcurrentAccessResolution

Format:

```
public void setDBConcurrentAccessResolution(int concurrentAccessResolution)
  throws java.sql.SQLException
```

Specifies whether the IBM Data Server Driver for JDBC and SQLJ requests that a read transaction can access a committed and consistent image of rows that are incompatibly locked by write transactions, if the data source supports accessing currently committed data, and the application isolation level is cursor stability (CS) or read stability (RS). This option has the same effect as the DB2 CONCURRENTACCESSRESOLUTION bind option.
setDBConcurrentAccessResolution affects only statements that are created after setDBConcurrentAccessResolution is executed.

setDBConcurrentAccessResolution applies only to connections to DB2 for z/OS and DB2 Database for Linux, UNIX, and Windows.

Parameter descriptions:

concurrentAccessResolution
One of the following integer values:

**DB2BaseDataSource.-
CONCURRENTACCESS_USE_CURRENTLY_COMMITTED (1)**
The IBM Data Server Driver for JDBC and SQLJ requests that:

- Read transactions access the currently committed data when the data is being updated or deleted.
- Read transactions skip rows that are being inserted.

DB2BaseDataSource.CONCURRENTACCESS_WAIT_FOR_OUTCOME (2)
The IBM Data Server Driver for JDBC and SQLJ requests that:

- Read transactions wait for a commit or rollback operation when they encounter data that is being updated or deleted.
- Read transactions do not skip rows that are being inserted.

DB2BaseDataSource.CONCURRENTACCESS_NOT_SET (0)
Enables the data server's default behavior for read transactions when lock contention occurs. This is the default value.

setDBStatementConcentrator

Format:

```
public void setDBStatementConcentrator(int statementConcentratorUse)
  throws java.sql.SQLException
```

Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses the data source's statement concentrator functionality. The statement concentrator is the ability to bypass preparation of a statement when it is the same as a statement in the dynamic statement cache, except for literal values. Statement concentrator functionality applies only to SQL statements that have literals but no parameter markers. setDBStatementConcentrator overrides the setting of the statementConcentrator Connection or DataSource property. setDBStatementConcentrator affects only statements that are created after setDBStatementConcentrator is executed.

Parameter descriptions:

statementConcentratorUse
One of the following integer values:

DB2BaseDataSource.STATEMENT_CONCENTRATOR_OFF (1)
The IBM Data Server Driver for JDBC and SQLJ does not use the data source's statement concentrator functionality.

DB2BaseDataSource.STATEMENT_CONCENTRATOR_WITH_LITERALS (2)
The IBM Data Server Driver for JDBC and SQLJ uses the data source's statement concentrator functionality.

DB2BaseDataSource.STATEMENT_CONCENTRATOR_NOT_SET (0)
Enables the data server's default behavior for statement concentrator functionality. This is the default value.

For DB2 Database for Linux, UNIX, and Windows data sources that support statement concentrator functionality, the functionality is used if the STMT_CONC configuration parameter is set to ON at the data source. Otherwise, statement concentrator functionality is not used.

For DB2 for z/OS data sources that support statement concentrator functionality, the functionality is not used if statementConcentrator is not set.

reuseDB2Connection (trusted connection reuse)

Formats:

```
public void reuseDB2Connection(byte[] cookie,
  String user,
  String password,
  String usernameRegistry,
  byte[] userSecToken,
  String originalUser,
  java.util.Properties properties)
  throws java.sql.SQLException
public void reuseDB2Connection(byte[] cookie,
  org.ietf.GSSCredential gssCredential,
  String usernameRegistry,
  byte[] userSecToken,
  String originalUser,
  java.util.Properties properties)
  throws java.sql.SQLException
```

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to:
  - DB2 Database for Linux, UNIX, and Windows Version 9.5 or later
  - DB2 for z/OS Version 9.1 or later
  - IBM Informix Version 11.70 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 9.1 or later

The second of these forms of reuseDB2Connection does not apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

These forms of reuseDB2Connection are used by a trusted application server to reuse a preexisting trusted connection on behalf of a new user. Properties that can be reset are passed, including the new user ID. The database server resets the associated physical connection. If reuseDB2Connection executes successfully, the connection becomes available for immediate use, with different properties, by the new user.

Parameter descriptions:

cookie
: A unique cookie that the JDBC driver generates for the Connection instance. The cookie is known only to the application server and the underlying JDBC driver that established the initial trusted connection. The application server passes the cookie that was created by the driver when the pooled connection instance was created. The JDBC driver checks that the supplied cookie matches the cookie of the underlying trusted physical connection to ensure that the request originated from the application server that established the trusted physical connection. If the cookies match, the connection becomes available for immediate use, with different properties, by the new user .

user
: The client ID that the database system uses to establish the database authorization ID. If the user was not authenticated by the application server, the application server needs to pass a client ID that represents an unauthenticated user.

password
: The password for *user*.

gssCredential
If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

userNameRegistry
A name that identifies a mapping service that maps a workstation user ID to a z/OS RACF® ID. An example of a mapping service is the Integrated Security Services Enterprise Identity Mapping (EIM). The mapping service is defined by a plugin. Valid values for *userNameRegistry* are defined by the plugin providers. If *userNameRegistry* is null, no mapping of *user* is done.

userSecToken
The client's security tokens. This value is traced as part of DB2 for z/OS accounting data. The content of *userSecToken* is described by the application server and is referred to by the database system as an application server security token.

originalUser
The original user ID that was used by the application server.

properties
Properties for the reused connection.

reuseDB2Connection (untrusted reuse with reauthentication)

Formats:

```
public void reuseDB2Connection(String user,
  String password,
  java.util.Properties properties)
  throws java.sql.SQLException
public void reuseDB2Connection(
  org.ietf.jgss.GSSCredential gssCredential,
  java.util.Properties properties)
  throws java.sql.SQLException
```

The first of these forms of reuseDB2Connection is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

The second of these forms of reuseDB2Connection does not apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

In a heterogeneous pooling environment, these forms of reuseDB2Connection reuse an existing Connection instance after reauthentication.

Parameter description:

user
The authorization ID that is used to establish the connection.

password
The password for the authorization ID that is used to establish the connection.

gssCredential
If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

properties
Properties for the reused connection. These properties override any properties that are already defined on the DB2Connection instance.

reuseDB2Connection (untrusted or trusted reuse without reauthentication)

Formats:

```
public void reuseDB2Connection(java.util.Properties properties)
  throws java.sql.SQLException
```

Reuses an existing Connection instance without reauthentication. This method is intended for reuse of a Connection instance when the properties do not change.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to:
  - DB2 Database for Linux, UNIX, and Windows Version 9.5 or later
  - DB2 for z/OS Version 9.1 or later
  - IBM Informix Version 11.70 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 9.1 or later

This method is for *dirty reuse* of a connection. This means that the connection state is not reset when the object is reused from the pool. Special register settings and property settings remain in effect unless they are overridden by passed properties. Global temporary tables are not deleted. Properties that are not specified are not re-initialized. All JDBC standard transient properties, such as the isolation level, autocommit mode, and read-only mode are reset to their JDBC defaults. Certain properties, such as `user`, `password`, `databaseName`, `serverName`, `portNumber`, `planName`, and `pkList` remain unchanged.

Parameter description:

properties
: Properties for the reused connection. These properties override any properties that are already defined on the DB2Connection instance.

setDB2ClientAccountingInformation

Format:

```
public void setDB2ClientAccountingInformation(String info)
  throws java.sql.SQLException
```

Specifies accounting information for the connection. This information is for client accounting purposes. This value can change during a connection.

Parameter description:

info
: User-specified accounting information. The maximum length depends on the server. For a DB2 Database for Linux, UNIX, and Windows server, the maximum length is 255 bytes. For a DB2 for z/OS server, the maximum length is 22 bytes. A Java empty string ("") is valid for this parameter value, but a Java `null` value is not valid.

Important: setDB2ClientAccountingInformation is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use java.sql.Connection.setClientInfo instead.

setDB2ClientApplicationInformation

Format:

```
public String setDB2ClientApplicationInformation(String info)
  throws java.sql.SQLException
```

Specifies application information for the current client.

Important: setDB2ClientApplicationInformation is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use java.sql.Connection.setClientInfo instead.

Parameter description:

info
User-specified application information. The maximum length depends on the server. For a DB2 Database for Linux, UNIX, and Windows server, the maximum length is 255 bytes. For a DB2 for z/OS server, the maximum length is 32 bytes. A Java empty string ("") is valid for this parameter value, but a Java `null` value is not valid.

setDB2ClientDebugInfo

Formats:

```
public void setDB2ClientDebugInfo(String debugInfo)
  throws java.sql.SQLException
public void setDB2ClientDebugInfo(String mgrInfo,
  String traceInfo)
  throws java.sql.SQLException
```

Sets a value for the CLIENT DEBUGINFO connection attribute, to notify the database system that stored procedures and user-defined functions that are using the connection are running in debug mode. CLIENT DEBUGINFO is used by the DB2 Unified Debugger. Use the first form to set the entire CLIENT DEBUGINFO string. Use the second form to modify only the session manager and trace information in the CLIENT DEBUGINFO string.

Setting the CLIENT DEBUGINFO attribute to a string of length greater than zero requires one of the following privileges:
- The DEBUGSESSION privilege
- SYSADM authority

Parameter description:

debugInfo
A string of up to 254 bytes, in the following form:

```
Mip:port,Iip,Ppid,Ttid,Cid,Llvl
```

The parts of the string are:

M*ip:port*
Session manager IP address and port number

I*ip* Client IP address

P*pid* Client process ID

T*tid* Client thread ID (optional)

C*id* Data connection generated ID

L*lvl* Debug library diagnostic trace level

For example:

```
M9.72.133.89:8355,I9.72.133.89,P4552,T123,C1,L0
```

See the description of SET CLIENT DEBUGINFO for a detailed description of this string.

mgrInfo
A string of the following form, which specifies the IP address and port number for the Unified Debugger session manager.

M*ip:port*

For example:

```
M9.72.133.89:8355
```

See the description of SET CLIENT DEBUGINFO for a detailed description of this string.

trcInfo
A string of the following form, which specifies the debug library diagnostics trace level.

```
Llvl
```

For example:

```
L0
```

See the description of SET CLIENT DEBUGINFO for a detailed description of this string.

setDB2ClientProgramId
Format:

```
public abstract void setDB2ClientProgramId(String program-ID)
  throws java.sql.SQLException
```

Sets a user-defined program identifier for the connection, on DB2 for z/OS servers. That program identifier is an 80-byte string that is used to identify the caller.

setDB2ClientProgramId does not apply to DB2 Database for Linux, UNIX, and Windows or IBM Informix data servers.

The DB2 for z/OS server places the string in IFCID 316 trace records along with other statistics, so that you can identify which program is associated with a particular SQL statement.

setDB2ClientUser
Format:

```
public void setDB2ClientUser(String user)
  throws java.sql.SQLException
```

Specifies the current client user name for the connection. This name is for client accounting purposes, and is not the user value for the JDBC connection. Unlike the user for the JDBC connection, the current client user name can change during a connection.

Parameter description:

user
The user ID for the current client. The maximum length depends on the server. For a DB2 Database for Linux, UNIX, and Windows server, the maximum length is 255 bytes. For a DB2 for z/OS server, the maximum length is 16 bytes. A Java empty string ("") is valid for this parameter value, but a Java `null` value is not valid.

Important: setDB2ClientUser is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use java.sql.Connection.setClientInfo instead.

setDB2ClientWorkstation
Format:

```
public void setDB2ClientWorkstation(String name)
  throws java.sql.SQLException
```

Specifies the current client workstation name for the connection. This name is for client accounting purposes. The current client workstation name can change during a connection.

Parameter description:

name
The workstation name for the current client. The maximum length depends on the server. For a DB2 Database for Linux, UNIX, and Windows server, the maximum length is 255 bytes. For a DB2 for z/OS server, the maximum length is 18 bytes. A Java empty string ("") is valid for this parameter value, but a Java `null` value is not valid.

Important: getDB2ClientWorkstation is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use java.sql.Connection.getClientInfo instead.

setDB2CurrentPackagePath

Format:

```
public void setDB2CurrentPackagePath(String packagePath)
  throws java.sql.SQLException
```

Specifies a list of collection IDs that the database system searches for JDBC and SQLJ packages.

The setDB2CurrentPackagePath method applies only to connections to DB2 database systems.

Parameter description:

packagePath
A comma-separated list of collection IDs.

setDB2CurrentPackageSet

Format:

```
public void setDB2CurrentPackageSet(String packageSet)
  throws java.sql.SQLException
```

Specifies the collection ID for the connection. When you set this value, you also set the collection ID of the IBM Data Server Driver for JDBC and SQLJ instance that is used for the connection.

The setDB2CurrentPackageSet method applies only to connections to DB2 database systems.

Parameter description:

packageSet
The collection ID for the connection. The maximum length for the *packageSet* value is 18 bytes. You can invoke this method as an alternative to executing the SQL SET CURRENT PACKAGESET statement in your program.

setJccLogWriter

Formats:

```
public void setJccLogWriter(PrintWriter logWriter)
  throws java.sql.SQLException

public void setJccLogWriter(PrintWriter logWriter, int traceLevel)
  throws java.sql.SQLException
```

Enables or disables the IBM Data Server Driver for JDBC and SQLJ trace, or changes the trace destination during an active connection.

Parameter descriptions:

logWriter
An object of type java.io.PrintWriter to which the IBM Data Server Driver for JDBC and SQLJ writes trace output. To turn off the trace, set the value of *logWriter* to `null`.

traceLevel
Specifies the types of traces to collect. See the description of the *traceLevel* property in "Properties for the IBM Data Server Driver for JDBC and SQLJ" for valid values.

setSavePointUniqueOption

Format:

```
public void setSavePointUniqueOption(boolean flag)
  throws java.sql.SQLException
```

Specifies whether an application can reuse a savepoint name within a unit of recovery. Possible values are:

true A Connection.setSavepoint(savepoint-name) method cannot specify the same value for *savepoint-name* more than once within the same unit of recovery.

false A Connection.setSavepoint(savepoint-name) method can specify the same value for *savepoint-name* more than once within the same unit of recovery.

When `false` is specified, if the Connection.setSavepoint(savepoint-name) method is executed, and a savepoint with the name *savepoint-name* already exists within the unit of recovery, the database manager destroys the existing savepoint, and creates a new savepoint with the name *savepoint-name*.

Reuse of a savepoint is not the same as executing Connection.releaseSavepoint(savepoint-name). Connection.releaseSavepoint(savepoint-name) releases *savepoint-name*, and any savepoints that were subsequently set.

DB2ConnectionPoolDataSource class

DB2ConnectionPoolDataSource is a factory for PooledConnection objects. An object that implements this interface is registered with a naming service that is based on the Java Naming and Directory Interface (JNDI).

The com.ibm.db2.jcc.DB2ConnectionPoolDataSource class extends the com.ibm.db2.jcc.DB2BaseDataSource class, and implements the javax.sql.ConnectionPoolDataSource, java.io.Serializable, and javax.naming.Referenceable interfaces.

DB2ConnectionPoolDataSource properties

These properties are defined only for the IBM Data Server Driver for JDBC and SQLJ. "Properties for the IBM Data Server Driver for JDBC and SQLJ" for explanations of these properties.

These properties have a setXXX method to set the value of the property and a getXXX method to retrieve the value. A setXXX method has this form:

```
void setProperty-name(data-type property-value)
```

A getXXX method has this form:

```
data-type getProperty-name()
```

Property-name is the unqualified property name, with the first character capitalized.

The following table lists the IBM Data Server Driver for JDBC and SQLJ properties and their data types.

Table 14-61. DB2ConnectionPoolDataSource properties and their data types

Property name	Data type
com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements	int

DB2ConnectionPoolDataSource methods

getDB2PooledConnection

Formats:

```
public DB2PooledConnection getDB2PooledConnection(String user,
  String password,
  java.util.Properties properties)
  throws java.sql.SQLException
public DB2PooledConnection getDB2PooledConnection(
  org.ietf.jgss.GSSCredential gssCredential,
  java.util.Properties properties)
  throws java.sql.SQLException
```

Establishes the initial untrusted connection in a heterogeneous pooling environment.

The first form getDB2PooledConnection provides a user ID and password. The second form of getDB2PooledConnection is for connections that use Kerberos security.

Parameter descriptions:

user
The authorization ID that is used to establish the connection.

password
The password for the authorization ID that is used to establish the connection.

gssCredential
If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

properties
Properties for the connection.

getDB2TrustedPooledConnection

Formats:

```
public Object[] getDB2TrustedPooledConnection(String user,
  String password,
  java.util.Properties properties)
  throws java.sql.SQLException
public Object[] getDB2TrustedPooledConnection(
  java.util.Properties properties)
```

```
  throws java.sql.SQLException
public Object[] getDB2TrustedPooledConnection(
  org.ietf.jgss.GSSCredential gssCredential,
  java.util.Properties properties)
  throws java.sql.SQLException
```

An application server using a system authorization ID uses this method to establish a trusted connection.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to:
  - DB2 Database for Linux, UNIX, and Windows Version 9.5 or later
  - DB2 for z/OS Version 9.1 or later
  - IBM Informix Version 11.70 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 9.1 or later

The following elements are returned in Object[]:

- The first element is a trusted DB2PooledConnection instance.
- The second element is a unique cookie for the generated pooled connection instance.

The first form getDB2TrustedPooledConnection provides a user ID and password, while the second form of getDB2TrustedPooledConnection uses the user ID and password of the DB2ConnectionPoolDataSource object. The third form of getDB2TrustedPooledConnection is for connections that use Kerberos security.

Parameter descriptions:

user
The IDS authorization ID that is used to establish the trusted connection to the database server.

password
The password for the authorization ID that is used to establish the trusted connection.

gssCredential
If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

properties
Properties for the connection.

DB2DatabaseMetaData interface

The com.ibm.db2.jcc.DB2DatabaseMetaData interface extends the java.sql.DatabaseMetaData interface.

DB2DatabaseMetaData methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

isIDSDatabaseAnsiCompliant
Format:

```
public boolean isIDSDatabaseAnsiCompliant();
```

Returns true if the current active IBM Informix database is ANSI-compliant. Returns false otherwise.

An ANSI-compliant database is a database that was created with the WITH LOG MODE ANSI option.

This method applies to connections to IBM Informix data sources only. An SQLException is thrown if the data source is not an IBM Informix data source.

isIDSDatabaseLogging

Format:

```
public boolean isIDSDatabaseLogging();
```

Returns true if the current active IBM Informix database supports logging. Returns false otherwise.

An IBM Informix database that supports logging is a database that was created with the WITH LOG MODE ANSI option, the WITH BUFFERED LOG, or the WITH LOG option.

This method applies to connections to IBM Informix data sources only. An SQLException is thrown if the data source is not an IBM Informix data source.

isResetRequiredForDB2eWLM

Format:

```
public boolean isResetRequiredForDB2eWLM();
```

Returns true if the target database server requires clean reuse to support eWLM. Returns false otherwise.

supportsDB2ProgressiveStreaming

Format:

```
public boolean supportsDB2ProgressiveStreaming();
```

Returns true if the target data source supports progressive streaming. Returns false otherwise.

DB2Diagnosable interface

The com.ibm.db2.jcc.DB2Diagnosable interface provides a mechanism for getting IDS diagnostics from an SQLException.

DB2Diagnosable methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getSqlca

Format:

```
public DB2Sqlca getSqlca();
```

Returns a DB2Sqlca object from a java.sql.Exception that is produced under a IBM Data Server Driver for JDBC and SQLJ.

getThrowable

Format:

```
public Throwable getThrowable();
```

Returns a java.lang.Throwable object from a java.sql.Exception that is produced under a IBM Data Server Driver for JDBC and SQLJ.

printTrace

Format:

```
static public void printTrace(java.io.PrintWriter printWriter,
  String header);
```

Prints diagnostic information after a java.sql.Exception is thrown under a IBM Data Server Driver for JDBC and SQLJ.

Parameter descriptions:

printWriter
The destination for the diagnostic information.

header
User-defined information that is printed at the beginning of the output.

DB2ExceptionFormatter class

The com.ibm.db2.jcc.DB2ExceptionFormatter class contains methods for printing diagnostic information to a stream.

DB2ExceptionFormatter methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

printTrace

Formats:

```
static public void printTrace(java.sql.SQLException sqlException,
  java.io.PrintWriter printWriter, String header)

static public void printTrace(DB2Sqlca sqlca,
  java.io.PrintWriter printWriter, String header)

static public void printTrace(java.lang.Throwable throwable,
  java.io.PrintWriter printWriter, String header)
```

Prints diagnostic information after an exception is thrown.

Parameter descriptions:

sqlException|sqlca|throwable
The exception that was thrown during a previous JDBC or Java operation.

printWriter
The destination for the diagnostic information.

header
User-defined information that is printed at the beginning of the output.

DB2JCCPlugin class

The com.ibm.db2.jcc.DB2JCCPlugin class is an abstract class that defines methods that can be implemented to provide DB2 Database for Linux, UNIX, and Windows plug-in support. This class applies only to DB2 Database for Linux, UNIX, and Windows.

DB2JCCPlugin methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getTicket

Format:

```
public abstract byte[] getTicket(String user,
  String password,
  byte[] returnedToken)
  throws org.ietf.jgss.GSSException
```

Retrieves a Kerberos ticket for a user.

Parameter descriptions:

user
The user ID for which the Kerberos ticket is to be retrieved.

password
The password for *user*.

returnedToken

DB2ParameterMetaData interface

The com.ibm.db2.jcc.DB2ParameterMetaData interface extends the java.sql.ParameterMetaData interface.

DB2ParameterMetaData methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getParameterMarkerNames

Format:

```
public String[] getParameterMarkerNames()
  throws java.sql.SQLException
```

Returns a list of the parameter marker names that are used in an SQL statement.

This method returns null if the enableNamedParameterMarkers property is set DB2BaseDataSource.NOT_SET or DB2BaseDataSource.NO, or if there are no named parameter markers in the SQL statement.

getProcedureParameterName

Format:

```
public String getProcedureParameterName(int param)
  throws java.sql.SQLException
```

Returns the name in the CREATE PROCEDURE statement of a parameter in an SQL CALL statement. If the parameter has no name in the CREATE PROCEDURE statement, the ordinal position of the parameter in the CREATE PROCEDURE statement is returned.

Parameter descriptions:

param
The ordinal position of the parameter in the CALL statement.

This method applies to connections to DB2 Database for Linux, UNIX, and Windows 9.7 or later data servers only.

DB2PooledConnection class

The com.ibm.db2.jcc.DB2PooledConnection class provides methods that an application server can use to switch users on a preexisting trusted connection.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to:
  - DB2 Database for Linux, UNIX, and Windows Version 9.5 or later
  - DB2 for z/OS Version 9.1 or later
  - IBM Informix Version 11.70 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 9.1 or later

DB2PooledConnection methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getConnection (untrusted or trusted reuse without reauthentication)

Format:

```
public DB2Connection getConnection()
  throws java.sql.SQLException
```

This method is for *dirty reuse* of a connection. This means that the connection state is not reset when the object is reused from the pool. Special register settings and property settings remain in effect unless they are overridden by passed properties. Global temporary tables are not deleted. Properties that are not specified are not re-initialized. All JDBC standard transient properties, such as the isolation level, autocommit mode, and read-only mode are reset to their JDBC defaults. Certain properties, such as `user`, `password`, `databaseName`, `serverName`, `portNumber`, `planName`, and `pkList` remain unchanged.

getDB2Connection (trusted reuse)

Formats:

```
public DB2Connection getDB2Connection(byte[] cookie,
  String user,
  String password,
  String userRegistry,
  byte[] userSecToken,
  String originalUser,
  java.util.Properties properties)
  throws java.sql.SQLException
public Connection getDB2Connection(byte[] cookie,
  org.ietf.GSSCredential gssCredential,
  String usernameRegistry,
  byte[] userSecToken,
  String originalUser,
  java.util.Properties properties)
  throws java.sql.SQLException
```

Switches the user that is associated with a trusted connection without authentication.

The second form of getDB2Connection is supported only for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Parameter descriptions:

cookie

A unique cookie that the JDBC driver generates for the Connection

instance. The cookie is known only to the application server and the underlying JDBC driver that established the initial trusted connection. The application server passes the cookie that was created by the driver when the pooled connection instance was created. The JDBC driver checks that the supplied cookie matches the cookie of the underlying trusted physical connection to ensure that the request originated from the application server that established the trusted physical connection. If the cookies match, the connection can become available, with different properties, for immediate use by a new user .

user
The client identity that is used by the data source to establish the authorization ID for the database server. If the user was not authenticated by the application server, the application server must pass a user identity that represents an unauthenticated user.

password
The password for *user*.

gssCredential
If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

userNameRegistry
A name that identifies a mapping service that maps a workstation user ID to a z/OS RACF ID. An example of a mapping service is the Integrated Security Services Enterprise Identity Mapping (EIM). The mapping service is defined by a plugin. Valid values for *userNameRegistry* are defined by the plugin providers. If *userNameRegistry* is null, the connection does not use a mapping service.

userSecToken
The client's security tokens. This value is traced as part of DB2 for z/OS accounting data. The content of *userSecToken* is described by the application server and is referred to by the data source as an application server security token.

originalUser
The client identity that sends the original request to the application server. *originalUser* is included in DB2 for z/OS accounting data as the original user ID that was used by the application server.

properties
Properties for the reused connection. These properties override any properties that are already defined on the DB2PooledConnection instance.

getDB2Connection (untrusted reuse with reauthentication)
Formats:

```
public DB2Connection getDB2Connection(
  String user,
  String password,
  java.util.Properties properties)
  throws java.sql.SQLException
public DB2Connection getDB2Connection(org.ietf.jgss.GSSCredential gssCredential,
  java.util.Properties properties)
  throws java.sql.SQLException
```

Switches the user that is associated with a untrusted connection, with authentication.

The first form getDB2Connection provides a user ID and password. The second form of getDB2Connection is for connections that use Kerberos security.

Parameter descriptions:

user
The user ID that is used by the data source to establish the authorization ID for the database server.

password
The password for *user*.

properties
Properties for the reused connection. These properties override any properties that are already defined on the DB2PooledConnection instance.

getDB2Connection (untrusted or trusted reuse without reauthentication)
Formats:

```
public java.sql.Connection getDB2Connection(
  java.util.Properties properties)
  throws java.sql.SQLException
```

Reuses an untrusted connection, without reauthentication.

This method is for *dirty reuse* of a connection. This means that the connection state is not reset when the object is reused from the pool. Special register settings and property settings remain in effect unless they are overridden by passed properties. Global temporary tables are not deleted. Properties that are not specified are not re-initialized. All JDBC standard transient properties, such as the isolation level, autocommit mode, and read-only mode are reset to their JDBC defaults. Certain properties, such as `user`, `password`, `databaseName`, `serverName`, `portNumber`, `planName`, and `pkList` remain unchanged.

Parameter descriptions:

properties
Properties for the reused connection. These properties override any properties that are already defined on the DB2PooledConnection instance.

DB2PoolMonitor class

The com.ibm.db2.jcc.DB2PoolMonitor class provides methods for monitoring the global transport objects pool that is used for the connection concentrator and Sysplex workload balancing.

DB2PoolMonitor fields

The following fields are defined only for the IBM Data Server Driver for JDBC and SQLJ.

public static final int TRANSPORT_OBJECT = 1
This value is a parameter for the DB2PoolMonitor.getPoolMonitor method.

DB2PoolMonitor methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

agedOutObjectCount
Format:

```
public abstract int agedOutObjectCount()
```

Retrieves the number of objects that exceeded the idle time that was specified by db2.jcc.maxTransportObjectIdleTime and were deleted from the pool.

createdObjectCount

Format:

```
public abstract int createdObjectCount()
```

Retrieves the number of objects that the IBM Data Server Driver for JDBC and SQLJ created since the pool was created.

getMonitorVersion

Format:

```
public int getMonitorVersion()
```

Retrieves the version of the DB2PoolMonitor class that is shipped with the IBM Data Server Driver for JDBC and SQLJ.

getPoolMonitor

Format:

```
public static DB2PoolMonitor getPoolMonitor(int monitorType)
```

Retrieves an instance of the DB2PoolMonitor class.

Parameter descriptions:

monitorType
The monitor type. This value must be DB2PoolMonitor.TRANSPORT_OBJECT.

heavyWeightReusedObjectCount

Format:

```
public abstract int heavyWeightReusedObjectCount()
```

Retrieves the number of objects that were reused from the pool.

lightWeightReusedObjectCount

Format:

```
public abstract int lightWeightReusedObjectCount()
```

Retrieves the number of objects that were reused but were not in the pool. This can happen if a Connection object releases a transport object at a transaction boundary. If the Connection object needs a transport object later, and the original transport object has not been used by any other Connection object, the Connection object can use that transport object.

longestBlockedRequestTime

Format:

```
public abstract long longestBlockedRequestTime()
```

Retrieves the longest amount of time that a request was blocked, in milliseconds.

numberOfConnectionReleaseRefused

Format:

```
public abstract int numberOfConnectionReleaseRefused()
```

Retrieves the number of times that the release of a connection was refused.

numberOfRequestsBlocked

Format:

```
public abstract int numberOfRequestsBlocked()
```

Retrieves the number of requests that the IBM Data Server Driver for JDBC and SQLJ made to the pool that the pool blocked because the pool reached its maximum capacity. A blocked request might be successful if an object is returned to the pool before the db2.jcc.maxTransportObjectWaitTime is exceeded and an exception is thrown.

numberOfRequestsBlockedDataSourceMax

Format:

```
public abstract int numberOfRequestsBlockedDataSourceMax()
```

Retrieves the number of requests that the IBM Data Server Driver for JDBC and SQLJ made to the pool that the pool blocked because the pool reached the maximum for the DataSource object.

numberOfRequestsBlockedPoolMax

Format:

```
public abstract int numberOfRequestsBlockedPoolMax()
```

Retrieves the number of requests that the IBM Data Server Driver for JDBC and SQLJ made to the pool that the pool blocked because the maximum number for the pool was reached.

removedObjectCount

Format:

```
public abstract int removedObjectCount()
```

Retrieves the number of objects that have been deleted from the pool since the pool was created.

shortestBlockedRequestTime

Format:

```
public abstract long shortestBlockedRequestTime()
```

Retrieves the shortest amount of time that a request was blocked, in milliseconds.

successfullRequestsFromPool

Format:

```
public abstract int successfullRequestsFromPool()
```

Retrieves the number of successful requests that the IBM Data Server Driver for JDBC and SQLJ has made to the pool since the pool was created. A successful request means that the pool returned an object.

totalPoolObjects

Format:

```
public abstract int totalPoolObjects()
```

Retrieves the number of objects that are currently in the pool.

totalRequestsToPool

Format:

```
public abstract int totalRequestsToPool()
```

Retrieves the total number of requests that the IBM Data Server Driver for JDBC and SQLJ has made to the pool since the pool was created.

totalTimeBlocked

Format:

```
public abstract long totalTimeBlocked()
```

Retrieves the total time in milliseconds for requests that were blocked by the pool. This time can be much larger than the elapsed execution time of the application if the application uses multiple threads.

DB2PreparedStatement interface

The com.ibm.db2.jcc.DB2PreparedStatement interface extends the com.ibm.db2.jcc.DB2Statement and java.sql.PreparedStatement interfaces.

DB2PreparedStatement fields

The following constants are defined only for the IBM Data Server Driver for JDBC and SQLJ.

public static DBIndicatorDefault DB_PARAMETER_DEFAULT

This constant can be used with standard interfaces, such as PreparedStatement.setObject or ResultSet.updateObject to indicate that the default value is assigned to the associated parameter.

public static DBIndicatorUnassigned DB_PARAMETER_UNASSIGNED

This constant can be used with standard interfaces, such as PreparedStatement.setObject or ResultSet.updateObject to indicate that the associated parameter is unaassigned.

DB2PreparedStatement methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

executeDB2QueryBatch

Format:

```
public void executeDB2QueryBatch()
  throws java.sql.SQLException
```

Executes a statement batch that contains queries with parameters.

This method is not supported for connections to IBM Informix data sources.

getDBGeneratedKeys

Format:

```
public java.sql.ResultSet[] getDBGeneratedKeys()
  throws java.sql.SQLException
```

Retrieves automatically generated keys that were created when INSERT statements were executed in a batch. Each ResultSet object that is returned contains the automatically generated keys for a single statement in the batch.

getDBGeneratedKeys returns an array of length 0 under the following conditions:

- getDBGeneratedKeys is called out of sequence. For example, if getDBGeneratedKeys is called before executeBatch, an array of length 0 is returned.
- The PreparedStatement that is executed in a batch was not created using one of the following methods:

```
Connection.prepareStatement(String sql, int[] autoGeneratedKeys)
Connection.prepareStatement(String sql, String[] autoGeneratedColumnNames)
Connection.prepareStatement(String sql, Statement.RETURN_GENERATED_KEYS)
```

If getDBGeneratedKeys is called against a PreparedStatement that was created using one of the previously listed methods, and the PreparedStatement is not in a batch, a single ResultSet is returned.

getEstimateCost

Format:

```
public int getEstimateCost()
  throws java.sql.SQLException
```

Returns the estimated cost of an SQL statement from the data server after the data server dynamically prepares the statement successfully. This value is the same as the fourth element in the sqlerrd array of the SQLCA.

If the deferPrepares property is set to `true`, calling getEstimateCost causes the data server to execute a dynamic prepare operation.

If the SQL statement cannot be prepared, or the data server does not return estimated cost information at prepare time, getEstimateCost returns -1.

getEstimateRowCount

Format:

```
public int getEstimateRowCount()
  throws java.sql.SQLException
```

Returns the estimated row count for an SQL statement from the data server after the data server dynamically prepares the statement successfully. This value is the same as the third element in the sqlerrd array of the SQLCA.

If the deferPrepares property is set to `true`, calling getEstimateRowCount causes the data server to execute a dynamic prepare operation.

If the SQL statement cannot be prepared, or the data server does not return estimated row count information at prepare time, getEstimateRowCount returns -1.

setDBTimestamp

Format:

```
public void setDBTimestamp(int parameterIndex,
  DBTimestamp timestamp)
  throws java.sql.SQLException
```

Assigns a DBTimestamp value to a parameter.

Parameters:

parameterIndex
The index of the parameter marker to which a DBTimestamp variable value is assigned.

timestamp
The DBTimestamp value that is assigned to the parameter marker.

This method is not supported for connections to IBM Informix data sources.

setJccArrayAtName

Format:

```
public void setJccArrayAtName(String parameterMarkerName,
  java.sql.Array x)
  throws java.sql.SQLException
```

Assigns a java.sql.Array value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The java.sql.Array value that is assigned to the named parameter marker.

setJccAsciiStreamAtName
Formats:

Supported by the IBM Data Server Driver for JDBC and SQLJ version 3.57 and later:

```
public void setJccAsciiStreamAtName(String parameterMarkerName,
  java.io.InputStream x, int length)
  throws java.sql.SQLException
```

Supported by the IBM Data Server Driver for JDBC and SQLJ version 4.7 and later:

```
public void setJccAsciiStreamAtName(String parameterMarkerName,
  java.io.InputStream x)
  throws java.sql.SQLException
public void setJccAsciiStreamAtName(String parameterMarkerName,
  java.io.InputStream x, long length)
  throws java.sql.SQLException
```

Assigns an ASCII value in a java.io.InputStream to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The ASCII java.io.InputStream value that is assigned to the parameter marker.

length
The length in bytes of the java.io.InputStream value that is assigned to the named parameter marker.

setJccBigDecimalAtName
Format:

```
public void setJccBigDecimalAtName(String parameterMarkerName,
  java.math.BigDecimal x)
  throws java.sql.SQLException
```

Assigns a java.math.BigDecimal value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The java.math.BigDecimal value that is assigned to the named parameter marker.

setJccBinaryStreamAtName

Formats:

Supported by the IBM Data Server Driver for JDBC and SQLJ version 3.57 and later:

```
public void setJccBinaryStreamAtName(String parameterMarkerName,
  java.io.InputStream x, int length)
  throws java.sql.SQLException
```

Supported by the IBM Data Server Driver for JDBC and SQLJ version 4.7 and later:

```
public void setJccBinaryStreamAtName(String parameterMarkerName,
  java.io.InputStream x)
  throws java.sql.SQLException
public void setJccBinaryStreamAtName(String parameterMarkerName,
  java.io.InputStream x, long length)
  throws java.sql.SQLException
```

Assigns a binary value in a java.io.InputStream to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The binary java.io.InputStream value that is assigned to the parameter marker.

length
The number of bytes of the java.io.InputStream value that are assigned to the named parameter marker.

setJccBlobAtName

Formats:

Supported by the IBM Data Server Driver for JDBC and SQLJ version 3.57 and later:

```
public void setJccBlobAtName(String parameterMarkerName,
  java.sql.Blob x)
  throws java.sql.SQLException
```

Supported by the IBM Data Server Driver for JDBC and SQLJ version 4.7 and later:

```
public void setJccBlobAtName(String parameterMarkerName,
  java.io.InputStream x)
  throws java.sql.SQLException
public void setJccBlobAtName(String parameterMarkerName,
  java.io.InputStream x, long length)
  throws java.sql.SQLException
```

Assigns a BLOB value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The java.sql.Blob value or java.io.InputStream value that is assigned to the parameter marker.

length
The number of bytes of the java.io.InputStream value that are assigned to the named parameter marker.

setJccBooleanAtName
Format:

```
public void setJccBooleanAtName(String parameterMarkerName,
  boolean x)
  throws java.sql.SQLException
```

Assigns a boolean value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The boolean value that is assigned to the named parameter marker.

setJccByteAtName
Format:

```
public void setJccByteAtName(String parameterMarkerName,
  byte x)
  throws java.sql.SQLException
```

Assigns a byte value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The byte value that is assigned to the named parameter marker.

setJccBytesAtName
Format:

```
public void setJccBytesAtName(String parameterMarkerName,
  byte[] x)
  throws java.sql.SQLException
```

Assigns an array of byte values to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The byte array that is assigned to the named parameter marker.

setJccCharacterStreamAtName
Formats:

Supported by the IBM Data Server Driver for JDBC and SQLJ version 3.57 and later:

```
public void setJccCharacterStreamAtName(String parameterMarkerName,
  java.io.Reader x, int length)
  throws java.sql.SQLException
```

Supported by the IBM Data Server Driver for JDBC and SQLJ version 4.7 and later:

```
public void setJccCharacterStreamAtName(String parameterMarkerName,
  java.io.Reader x)
  throws java.sql.SQLException
public void setJccCharacterStreamAtName(String parameterMarkerName,
  java.io.Reader x, long length)
  throws java.sql.SQLException
```

Assigns a Unicode value in a java.io.Reader to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The Unicode java.io.Reader value that is assigned to the named parameter marker.

length
The number of characters of the java.io.InputStream value that are assigned to the named parameter marker.

setJccClobAtName

Formats:

Supported by the IBM Data Server Driver for JDBC and SQLJ version 3.57 and later:

```
public void setJccClobAtName(String parameterMarkerName,
  java.sql.Clob x)
  throws java.sql.SQLException
```

Supported by the IBM Data Server Driver for JDBC and SQLJ version 4.7 and later:

```
public void setJccClobAtName(String parameterMarkerName,
  java.io.Reader x)
  throws java.sql.SQLException
public void setJccClobAtName(String parameterMarkerName,
  java.io.Reader x, long length)
  throws java.sql.SQLException
```

Assigns a CLOB value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The java.sql.Clob value or java.io.Reader value that is assigned to the named parameter marker.

length
The number of bytes of the java.io.InputStream value that are assigned to the named parameter marker.

setJccDateAtName
Formats:

```
public void setJccDateAtName(String parameterMarkerName,
  java.sql.Date x)
  throws java.sql.SQLException
public void setJccDateAtName(String parameterMarkerName,
  java.sql.Date x,
  java.util.Calendar cal)
  throws java.sql.SQLException
```

Assigns a java.sql.Date value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The java.sql.Date value that is assigned to the named parameter marker.

cal
The java.util.Calendar object that the IBM Data Server Driver for JDBC and SQLJ uses to construct the date.

setJccDBTimestampAtName
Format:

```
public void setJccDBTimestampAtName(String parameterMarkerName,
  DBTimestamp timestamp)
  throws java.sql.SQLException
```

Assigns a DBTimestamp value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a DBTimestamp variable value is assigned.

timestamp
The DBTimestamp value that is assigned to the named parameter marker.

This method is not supported for connections to IBM Informix data sources.

setJccDBDefaultAtName
Formats:

```
public void setJccDBDefaultAtName(String parameterMarkerName)
  throws SQLException
```

Assigns the default value to a named parameter marker. Execution of setJccDBDefaultAtName produces the same results as using the literal DEFAULT in the SQL string, instead of the parameter marker name.

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

This method is not supported for connections to IBM Informix data sources.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

setJccDBUnassignedAtName

Formats:

```
public void setJccDBUnassignedAtName(String parameterMarkerName)
  throws SQLException
```

Does not assign a value to the specified named parameter. Execution of setJccDBUnassignedAtName produces the same result as if the specified parameter marker name had not appeared in the SQL string.

Parameters:

parameterMarkerName
The name of the parameter marker whose value is to be unassigned.

This method is not supported for connections to IBM Informix data sources.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

setJccDoubleAtName

Format:

```
public void setJccDoubleAtName(String parameterMarkerName,
  double x)
  throws java.sql.SQLException
```

Assigns a value of type double to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The value of type double that is assigned to the parameter marker.

setJccFloatAtName

Format:

```
public void setJccFloatAtName(String parameterMarkerName,
  float x)
  throws java.sql.SQLException
```

Assigns a value of type float to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The value of type float that is assigned to the parameter marker.

setJccIntAtName

Format:

```
public void setJccIntAtName(String parameterMarkerName,
  int x)
  throws java.sql.SQLException
```

Assigns a value of type int to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The value of type int that is assigned to the parameter marker.

setJccLongAtName

Format:

```
public void setJccLongAtName(String parameterMarkerName,
  long x)
  throws java.sql.SQLException
```

Assigns a value of type long to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The value of type long that is assigned to the parameter marker.

setJccNullAtName

Format:

```
public void setJccNullAtName(String parameterMarkerName,
  int jdbcType)
  throws java.sql.SQLException
public void setJccNullAtName(String parameterMarkerName,
  int jdbcType,
  String typeName)
  throws java.sql.SQLException
```

Assigns the SQL NULL value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

jdbcType
The JDBC type code of the NULL value that is assigned to the parameter marker, as defined in java.sql.Types.

typeName
If *jdbcType* is java.sql.Types.DISTINCT or java.sql.Types.REF, the fully-qualified name of the SQL user-defined type of the NULL value that is assigned to the parameter marker.

setJccObjectAtName

Formats:

```
public void setJccObjectAtName(String parameterMarkerName,
  java.sql.Object x)
  throws java.sql.SQLException
public void setJccObjectAtName(String parameterMarkerName,
  java.sql.Object x,
  int targetJdbcType)
  throws java.sql.SQLException
public void setJccObjectAtName(String parameterMarkerName,
  java.sql.Object x,
  int targetJdbcType,
  int scale)
  throws java.sql.SQLException
```

Assigns a value with type java.lang.Object to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to `DB2BaseDataSource.YES` (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The value with type Object that is assigned to the parameter marker.

targetJdbcType
The data type, as defined in java.sql.Types, that is assigned to the input value when it is sent to the data source.

scale
The scale of the value that is assigned to the parameter marker. This parameter applies only to these cases:

- If *targetJdbcType* is java.sql.Types.DECIMAL or java.sql.Types.NUMERIC, *scale* is the number of digits to the right of the decimal point.
- If *x* has type java.io.InputStream or java.io.Reader, *scale* is the this is the length of the data in the Stream or Reader object.

setJccShortAtName

Format:

```
public void setJccShortAtName(String parameterMarkerName,
  short x)
  throws java.sql.SQLException
```

Assigns a value of type short to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to `DB2BaseDataSource.YES` (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The value of type short that is assigned to the parameter marker.

setJccSQLXMLAtName

Format:

```
public void setJccSQLXMLAtName(String parameterMarkerName,
  java.sql.SQLXML x)
  throws java.sql.SQLException
```

Assigns a value of type java.sql.SQLXML to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

This method is supported only for connections to DB2 Database for Linux, UNIX, and Windows Version 9.1 or later or DB2 for z/OS Version 9 or later.

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The value of type java.sql.SQLXML that is assigned to the parameter marker.

setJccStringAtName
Format:

```
public void setJccStringAtName(String parameterMarkerName,
  String x)
  throws java.sql.SQLException
```

Assigns a value of type String to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The value of type String that is assigned to the parameter marker.

setJccTimeAtName
Formats:

```
public void setJccTimeAtName(String parameterMarkerName,
  java.sql.Time x)
  throws java.sql.SQLException
public void setJccTimeAtName(String parameterMarkerName,
  java.sql.Time x,
  java.util.Calendar cal)
  throws java.sql.SQLException
```

Assigns a java.sql.Time value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The java.sql.Time value that is assigned to the parameter marker.

cal
The java.util.Calendar object that the IBM Data Server Driver for JDBC and SQLJ uses to construct the time.

setJccTimestampAtName
Formats:

```
public void setJccTimestampAtName(String parameterMarkerName,
  java.sql.Timestamp x)
  throws java.sql.SQLException
```

```
public void setJccTimestampAtName(String parameterMarkerName,
  java.sql.Timestamp x,
  java.util.Calendar cal)
  throws java.sql.SQLException
```

Assigns a java.sql.Timestamp value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The java.sql.Timestamp value that is assigned to the parameter marker.

cal
The java.util.Calendar object that the IBM Data Server Driver for JDBC and SQLJ uses to construct the timestamp.

setJccUnicodeStreamAtName

Format:

```
public void setJccUnicodeStreamAtName(String parameterMarkerName,
  java.io.InputStream x, int length)
  throws java.sql.SQLException
```

Assigns a Unicode value in a java.io.InputStream to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The Unicode java.io.InputStream value that is assigned to the parameter marker.

length
The number of bytes of the java.io.InputStream value that are assigned to the parameter marker.

setDBDefault

Formats:

```
public void setDBDefault(int parameterIndex)
  throws SQLException
```

Assigns the default value to the specified parameter. Execution of setDBDefault produces the same results as using the literal DEFAULT in the SQL string, instead of the parameter.

Parameters:

parameterIndex
The number of the parameter whose value is being updated.

This method is not supported for connections to IBM Informix data sources.

setDBUnassigned

Formats:

```
public void setDBUnassigned(int parameterIndex)
  throws SQLException
```

Does not assign a value to the specified parameter. Execution of setDBUnassigned produces the same result as if the specified parameter had not appeared in the SQL string.

Parameters:

parameterIndex
The number of the parameter whose value is to be unassigned.

This method is not supported for connections to IBM Informix data sources.

DB2ResultSet interface

The com.ibm.db2.jcc.DB2ResultSet interface is used to create objects from which IBM Data Server Driver for JDBC and SQLJ-only query information can be obtained.

DB2ResultSet implements the java.sql.Wrapper interface.

DB2ResultSet methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getDB2RowChangeToken

Format:

```
public long DB2ResultSet.getDB2RowChangeToken()
  throws java.sql.SQLException
```

Returns the row change token for the current row, if it is available. Returns 0 if optimistic locking columns were not requested or are not available.

This method applies only to connections to DB2 Database for Linux, UNIX, and Windows.

getDB2RID

Format:

```
public Object DB2ResultSet.getDB2RID()
  throws java.sql.SQLException
```

Returns the RID for the current row, if it is available. The RID is available if optimistic locking columns were requested and are available. Returns null if optimistic locking columns were not requested or are not available.

This method applies only to connections to DB2 Database for Linux, UNIX, and Windows.

getDB2RIDType

Format:

```
public int DB2ResultSet.getDB2RIDType()
  throws java.sql.SQLException
```

Returns the data type of the RID column in a DB2ResultSet. The returned value maps to a `java.sql.Types` constant. If the DB2ResultSet does not contain a RID column, `java.sql.Types.NULL` is returned.

This method applies only to connections to DB2 Database for Linux, UNIX, and Windows.

getDBTimestamp

Formats:

```
public DBTimestamp getDBTimestamp(int parameterIndex)
  throws SQLException
public DBTimestamp getDBTimestamp(String parameterName)
  throws SQLException
```

Returns the value in the current row of a TIMESTAMP or TIMESTAMP WITH TIME ZONE column that is in a DB2ResultSet object as a DBTimestamp object. For a TIMESTAMP column, the returned value has the local time zone. If the value of the DB2ResultSet column is NULL, the returned value is null.

Parameters:

parameterIndex
The number of the column in the DB2ResultSet whose value is being retrieved.

parameterName
The name of the column in the DB2ResultSet whose value is being retrieved.

updateDBDefault

Formats:

```
public void updateDBDefault(int parameterIndex)
  throws SQLException
public void updateDBDefault(String columnName)
  throws SQLException
```

Assigns the default value to the specified column in a DB2ResultSet object. This method does not update the underlying table.

Parameters:

parameterIndex
The number of the column in the DB2ResultSet whose value is being updated.

columnName
The name of the column in the DB2ResultSet whose value is being updated.

This method is not supported for connections to IBM Informix data sources.

DB2ResultSetMetaData interface

The com.ibm.db2.jcc.DB2ResultSetMetaData interface provides methods that provide information about a ResultSet object.

Before a com.ibm.db2.jcc.DB2ResultSetMetaData method can be used, a java.sql.ResultSetMetaData object that is returned from a java.sql.ResultSet.getMetaData call needs to be cast to com.ibm.db2.jcc.DB2ResultSetMetaData.

DB2ResultSetMetaData methods:

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

isDB2ColumnNameDerived

Format:

```
public boolean isDB2ColumnNameDerived (int column)
  throws java.sql.SQLException
```

Returns `true` if the name of a ResultSet column is in the SQL SELECT list that generated the ResultSet.

For example, suppose that a ResultSet is generated from the SQL statement SELECT EMPNAME, SUM(SALARY) FROM EMP. Column name EMPNAME is derived from the SQL SELECT list, but the name of the column in the ResultSet that corresponds to SUM(SALARY) is not derived from the SELECT list.

Parameter descriptions:

`column`
The ordinal position of a column in the ResultSet.

DB2RowID interface

The com.ibm.db2.jcc.DB2RowID interface is used for declaring Java objects for use with the SQL ROWID data type.

The com.ibm.db2.jcc.DB2RowID interface does not apply to connection to IBM Informix.

DB2RowID methods

The following method is defined only for the IBM Data Server Driver for JDBC and SQLJ.

`getBytes`
Format:

```
public byte[] getBytes()
```

Converts a com.ibm.jcc.DB2RowID object to bytes.

DB2SimpleDataSource class

The com.ibm.db2.jcc.DB2SimpleDataSource class extends the DB2BaseDataSource class.

A DB2BaseDataSource object does not support connection pooling or distributed transactions. It contains all of the properties and methods that the DB2BaseDataSource class contains. In addition, DB2SimpleDataSource contains the following IBM Data Server Driver for JDBC and SQLJ-only properties.

DB2SimpleDataSource implements the java.sql.Wrapper interface.

DB2SimpleDataSource properties

The following properties are defined only for the IBM Data Server Driver for JDBC and SQLJ.

String com.ibm.db2.jcc.DB2SimpleDataSource.password

DB2SimpleDataSource methods

The following method is defined only for the IBM Data Server Driver for JDBC and SQLJ.

`setPassword`
Format:

```
public synchronized void setPassword(String password)
```

Sets the password for the DB2SimpleDataSource object. There is no corresponding getPassword method. Therefore, the password cannot be encrypted because there is no way to retrieve the password so that you can decrypt it.

DB2Sqlca class

The com.ibm.db2.jcc.DB2Sqlca class is an encapsulation of the SQLCA.

DB2Sqlca methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getMessage

Format:

```
public abstract String getMessage()
```

Returns error message text.

getSqlCode

Format:

```
public abstract int getSqlCode()
```

Returns an SQL error code value.

getSqlErrd

Format:

```
public abstract int[] getSqlErrd()
```

Returns an array, each element of which contains an SQLCA SQLERRD.

getSqlErrmc

Format:

```
public abstract String getSqlErrmc()
```

Returns a string that contains the SQLCA SQLERRMC values, delimited with spaces.

getSqlErrmcTokens

Format:

```
public abstract String[] getSqlErrmcTokens()
```

Returns an array, each element of which contains an SQLCA SQLERRMC token.

getSqlErrp

Format:

```
public abstract String getSqlErrp()
```

Returns the SQLCA SQLERRP value.

getSqlState

Format:

```
public abstract String getSqlState()
```

Returns the SQLCA SQLSTATE value.

getSqlWarn

Format:

```
public abstract char[] getSqlWarn()
```

Returns an array, each element of which contains an SQLCA SQLWARN value.

DB2Statement interface

The com.ibm.db2.jcc.DB2Statement interface extends the java.sql.Statement interface.

DB2Statement implements the java.sql.Wrapper interface.

DB2Statement methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getDB2ClientProgramId

Format:

```
public String getDB2ClientProgramId()
  throws java.sql.SQLException
```

Returns the user-defined client program identifier for the connection, which is stored on the data source.

getDB2ClientProgramId does not apply to DB2 Database for Linux, UNIX, and Windows data servers.

setDB2ClientProgramId

Format:

```
public abstract void setDB2ClientProgramId(String program-ID)
  throws java.sql.SQLException
```

Sets a user-defined program identifier for the connection on a data server. That program identifier is an 80-byte string that is used to identify the caller.

setDB2ClientProgramId does not apply to DB2 Database for Linux, UNIX, and Windows data servers.

The DB2 for z/OS server places the string in IFCID 316 trace records along with other statistics, so that you can identify which program is associated with a particular SQL statement.

getIDSBigSerial

Format:

```
public int getIDSBigSerial()
  throws java.sql.SQLException
```

Retrieves an automatically generated key from a BIGSERIAL column after the automatically generated key was inserted by a previously executed INSERT statement.

The following conditions must be true for getIDSBigSerial to execute successfully:

- The INSERT statement is the last SQL statement that is executed before this method is called.
- The table into which the row is inserted contains a BIGSERIAL column.

- The form of the JDBC Connection.prepareStatement method or Statement.executeUpdate method that prepares or executes the INSERT statement does not have parameters that request automatically generated keys.

This method applies only to connections to IBM Informix databases.

getIDSSerial

Format:

```
public int getIDSSerial()
  throws java.sql.SQLException
```

Retrieves an automatically generated key from a SERIAL column after the automatically generated key was inserted by a previously executed INSERT statement.

The following conditions must be true for getIDSSerial to execute successfully:

- The INSERT statement is the last SQL statement that is executed before this method is called.
- The table into which the row is inserted contains a SERIAL column.
- The form of the JDBC Connection.prepareStatement method or Statement.executeUpdate method that prepares or executes the INSERT statement does not have parameters that request automatically generated keys.

This method applies only to connections to IBM Informix databases.

getIDSSerial8

Format:

```
public long getIDSSerial8()
  throws java.sql.SQLException
```

Retrieves an automatically generated key from a SERIAL8 column after the automatically generated key was inserted by a previously executed INSERT statement.

The following conditions must be true for getIDSSerial8 to execute successfully:

- The INSERT statement is the last SQL statement that is executed before this method is called.
- The table into which the row is inserted contains a SERIAL8 column.
- The form of the JDBC Connection.prepareStatement method or Statement.executeUpdate method that prepares or executes the INSERT statement does not have parameters that request automatically generated keys.

This method applies only to connections to IBM Informix data sources.

getIDSSQLStatementOffSet

Format:

```
public int getIDSSQLStatementOffSet()
  throws java.sql.SQLException
```

After an SQL statement executes on an IBM Informix data source, if the statement has a syntax error, getIDSSQLStatementOffSet returns the offset into the statement text of the syntax error.

getIDSSQLStatementOffSet returns:

- 0, if the statement does not have a syntax error.

- -1, if the data source is not IBM Informix.

This method applies only to connections to IBM Informix data sources.

DB2SystemMonitor interface

The com.ibm.db2.jcc.DB2SystemMonitor interface is used for collecting system monitoring data for a connection. Each connection can have one DB2SystemMonitor instance.

DB2SystemMonitor fields

The following fields are defined only for the IBM Data Server Driver for JDBC and SQLJ.

public final static int RESET_TIMES
public final static int ACCUMULATE_TIMES
These values are arguments for the DB2SystemMonitor.start method. RESET_TIMES sets time counters to zero before monitoring starts. ACCUMULATE_TIMES does not set time counters to zero.

DB2SystemMonitor methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

enable
Format:

```
public void enable(boolean on)
  throws java.sql.SQLException
```

Enables the system monitor that is associated with a connection. This method cannot be called during monitoring. All times are reset when enable is invoked.

getApplicationTimeMillis
Format:

```
public long getApplicationTimeMillis()
  throws java.sql.SQLException
```

Returns the sum of the application, JDBC driver, network I/O, and database server elapsed times. The time is in milliseconds.

A monitored elapsed time interval is the difference, in milliseconds, between these points in the JDBC driver processing:

Interval beginning
When start is called.

Interval end
When stop is called.

getApplicationTimeMillis returns 0 if system monitoring is disabled. Calling this method without first calling the stop method results in an SQLException.

getCoreDriverTimeMicros
Format:

```
public long getCoreDriverTimeMicros()
  throws java.sql.SQLException
```

Returns the sum of elapsed monitored API times that were collected while system monitoring was enabled. The time is in microseconds.

A monitored API is a JDBC driver method for which processing time is collected. In general, elapsed times are monitored only for APIs that might result in network I/O or database server interaction. For example, PreparedStatement.setXXX methods and ResultSet.getXXX methods are not monitored.

Monitored API elapsed time includes the total time that is spent in the driver for a method call. This time includes any network I/O time and database server elapsed time.

A monitored API elapsed time interval is the difference, in microseconds, between these points in the JDBC driver processing:

Interval beginning
When a monitored API is called by the application.

Interval end
Immediately before the monitored API returns control to the application.

getCoreDriverTimeMicros returns 0 if system monitoring is disabled. Calling this method without first calling the stop method, or calling this method when the underlying JVM does not support reporting times in microseconds results in an SQLException.

getNetworkIOTimeMicros

Format:

```
public long getNetworkIOTimeMicros()
  throws java.sql.SQLException
```

Returns the sum of elapsed network I/O times that were collected while system monitoring was enabled. The time is in microseconds.

Elapsed network I/O time includes the time to write and read DRDA data from network I/O streams. A network I/O elapsed time interval is the time interval to perform the following operations in the JDBC driver:

- Issue a TCP/IP command to send a DRDA message to the database server. This time interval is the difference, in microseconds, between points immediately before and after a write and flush to the network I/O stream is performed.
- Issue a TCP/IP command to receive DRDA reply messages from the database server. This time interval is the difference, in microseconds, between points immediately before and after a read on the network I/O stream is performed.

Network I/O time intervals are captured for all send and receive operations, including the sending of messages for commits and rollbacks.

The time spent waiting for network I/O might be impacted by delays in CPU dispatching at the database server for low-priority SQL requests.

getNetworkIOTimeMicros returns 0 if system monitoring is disabled. Calling this method without first calling the stop method, or calling this method when the underlying JVM does not support reporting times in microseconds results in an SQLException.

getServerTimeMicros

Format:

```
public long getServerTimeMicros()
  throws java.sql.SQLException
```

Returns the sum of all reported database server elapsed times that were collected while system monitoring was enabled. The time is in microseconds.

The database server reports elapsed times under these conditions:

- The database server supports returning elapsed time data to the client.

  DB2 Database for Linux, UNIX, and Windows Version 9.5 and later and DB2 for z/OS support this function.
- The database server performs operations that can be monitored. For example, database server elapsed time is not returned for commits or rollbacks.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows, and IBM Data Server Driver for JDBC and SQLJ type 4 connectivity: The database server elapsed time is defined as the elapsed time to parse the request data stream, process the command, and generate the reply data stream at the database server. Network time to receive or send the data stream is not included. The database server elapsed time interval is the difference, in microseconds, between these points in the database server processing:

Interval beginning
When the operating system dispatches the database server to process a TCP/IP message that is received from the JDBC driver.

Interval end
When the database server is ready to issue the TCP/IP command to return the reply message to the client.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS: The database server elapsed time interval is the difference, in microseconds, between these points in the JDBC driver native processing:

Interval beginning
The z/OS Store Clock (STCK) value when a JDBC driver native method calls the RRS attachment facility to process an SQL request.

Interval end
The z/OS Store Clock (STCK) value when control returns to the JDBC driver native method following an RRS attachment facility call to process an SQL request.

getServerTimeMicros returns 0 if system monitoring is disabled. Calling this method without first calling the stop method results in an SQLException.

start

Format:

```
public void start (int lapMode)
  throws java.sql.SQLException
```

If the system monitor is enabled, start begins the collection of system monitoring data for a connection. Valid values for *lapMode* are RESET_TIMES or ACCUMULATE_TIMES.

Calling this method with system monitoring disabled does nothing. Calling this method more than once without an intervening stop call results in an SQLException.

stop

Format:

```
public void stop()
  throws java.sql.SQLException
```

If the system monitor is enabled, stop ends the collection of system monitoring data for a connection. After monitoring is stopped, monitored times can be obtained with the getXXX methods of DB2SystemMonitor.

Calling this method with system monitoring disabled does nothing. Calling this method without first calling start, or calling this method more than once without an intervening start call results in an SQLException.

DB2TraceManager class

The com.ibm.db2.jcc.DB2TraceManager class controls the global log writer.

The global log writer is driver-wide, and applies to all connections. The global log writer overrides any other JDBC log writers. In addition to starting the global log writer, the DB2TraceManager class provides the ability to suspend and resume tracing of any type of log writer. That is, the suspend and resume methods of the DB2TraceManager class apply to all current and future DriverManager log writers, DataSource log writers, or IBM Data Server Driver for JDBC and SQLJ-only connection-level log writers.

DB2TraceManager methods

getTraceManager

Format:

```
static public DB2TraceManager getTraceManager()
  throws java.sql.SQLException
```

Gets an instance of the global log writer.

setLogWriter

Formats:

```
public abstract void setLogWriter(String traceDirectory,
  String baseTraceFileName, int traceLevel)
  throws java.sql.SQLException
public abstract void setLogWriter(String traceFile,
  boolean fileAppend, int traceLevel)
  throws java.sql.SQLException
public abstract void setLogWriter(java.io.PrintWriter logWriter,
  int traceLevel)
  throws java.sql.SQLException
```

Enables a global trace. After setLogWriter is called, all calls for DataSource or Connection traces are discarded until DB2TraceManager.unsetLogWriter is called.

When setLogWriter is called, all future Connection or DataSource traces are redirected to a trace file or PrintWriter, depending on the form of setLogWriter that you use. If the global trace is suspended when setLogWriter is called, the specified settings take effect when the trace is resumed.

Parameter descriptions:

traceDirectory

Specifies a directory into which global trace information is written. This setting overrides the settings of the traceDirectory and logWriter properties for a DataSource or DriverManager connection.

When the form of setLogWriter with the traceDirectory parameter is used, the JDBC driver sets the traceFileAppend property to `false` when setLogWriter is called, which means that the existing log files are overwritten. Each JDBC driver connection is traced to a different file in the specified directory. The naming convention for the files in that directory depends on whether a non-null value is specified for baseTraceFileName:

- If a null value is specified for baseTraceFileName, a connection is traced to a file named traceFile_global_*n*.

  n is the *n*th JDBC driver connection.
- If a non-null value is specified for baseTraceFileName, a connection is traced to a file named *baseTraceFileName*_global_*n*.

  baseTraceFileName is the value of the baseTraceFileName parameter.

  n is the *n*th JDBC driver connection.

baseTraceFileName
Specifies the stem for the names of the files into which global trace information is written. The combination of baseTraceFileName and traceDirectory determines the full path name for the global trace log files.

traceFileName
Specifies the file into which global trace information is written. This setting overrides the settings of the traceFile and logWriter properties for a DataSource or DriverManager connection.

When the form of setLogWriter with the traceFileName parameter is used, only one log file is written.

traceFileName can include a directory path.

logWriter
Specifies a character output stream to which all global log records are written.

This value overrides the logWriter property on a DataSource or DriverManager connection.

traceLevel
Specifies what to trace.

You can specify one or more of the following traces with the traceLevel parameter:

- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_NONE` (X'00')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTION_CALLS` (X'01')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_STATEMENT_CALLS` (X'02')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_CALLS` (X'04')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRIVER_CONFIGURATION` (X'10')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS` (X'20')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS` (X'40')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_META_DATA` (X'80')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_PARAMETER_META_DATA` (X'100')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DIAGNOSTICS` (X'200')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SQLJ` (X'400')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_XA_CALLS (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity for DB2 Database for Linux, UNIX, and Windows only)` (X'800')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_META_CALLS` (X'2000')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DATASOURCE_CALLS` (X'4000')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_LARGE_OBJECT_CALLS` (X'8000')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SYSTEM_MONITOR` (X'20000')

- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_TRACEPOINTS () (X'40000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL (X'FFFFFFFF')

To specify more than one trace, use one of these techniques:

- Use bitwise OR (|) operators with two or more trace values. For example, to trace DRDA flows and connection calls, specify this value for traceLevel:

  ```
  TRACE_DRDA_FLOWS|TRACE_CONNECTION_CALLS
  ```

- Use a bitwise complement (tilde (~)) operator with a trace value to specify all except a certain trace. For example, to trace everything except DRDA flows, specify this value for traceLevel:

  ```
  ~TRACE_DRDA_FLOWS
  ```

fileAppend
Specifies whether to append to or overwrite the file that is specified by the traceFile parameter. `true` means that the existing file is not overwritten.

unsetLogWriter
Format:

```
public abstract void unsetLogWriter()
  throws java.sql.SQLException
```

Disables the global log writer override for future connections.

suspendTrace
Format:

```
public void suspendTrace()
  throws java.sql.SQLException
```

Suspends all global, Connection-level, or DataSource-level traces for current and future connections. suspendTrace can be called when the global log writer is enabled or disabled.

resumeTrace
Format:

```
public void resumeTrace()
  throws java.sql.SQLException
```

Resumes all global, Connection-level, or DataSource-level traces for current and future connections. resumeTrace can be called when the global log writer is enabled or disabled. If the global log writer is disabled, resumeTrace resumes Connection-level or DataSource-level traces. If the global log writer is enabled, resumeTrace resumes the global trace.

getLogWriter
Format:

```
public abstract java.io.PrintWriter getLogWriter()
  throws java.sql.SQLException
```

Returns the PrintWriter for the global log writer, if it is set. Otherwise, getLogWriter returns null.

getTraceFile
Format:

```
public abstract String getTraceFile()
  throws java.sql.SQLException
```

Returns the name of the destination file for the global log writer, if it is set. Otherwise, getTraceFile returns null.

getTraceDirectory

Format:

```
public abstract String getTraceDirectory()
  throws java.sql.SQLException
```

Returns the name of the destination directory for global log writer files, if it is set. Otherwise, getTraceDirectory returns null.

getTraceLevel

Format:

```
public abstract int getTraceLevel()
  throws java.sql.SQLException
```

Returns the trace level for the global trace, if it is set. Otherwise, getTraceLevel returns -1 (TRACE_ALL).

getTraceFileAppend

Format:

```
public abstract boolean getTraceFileAppend()
  throws java.sql.SQLException
```

Returns true if the global trace records are appended to the trace file. Otherwise, getTraceFileAppend returns false.

DB2TraceManagerMXBean interface

The com.ibm.db2.jcc.mx.DB2TraceManagerMXBean interface is the means by which an application makes DB2TraceManager available as an MXBean for the remote trace controller.

DB2TraceManagerMXBean methods

setTraceFile

Format:

```
public void setTraceFile(String traceFile,
  boolean fileAppend, int traceLevel)
  throws java.sql.SQLException
```

Specifies the name of the file into which the remote trace manager writes trace information, and the type of information that is to be traced.

Parameter descriptions:

traceFileName
Specifies the file into which global trace information is written. This setting overrides the settings of the traceFile and logWriter properties for a DataSource or DriverManager connection.

When the form of setLogWriter with the traceFileName parameter is used, only one log file is written.

traceFileName can include a directory path.

fileAppend
Specifies whether to append to or overwrite the file that is specified by the traceFile parameter. true means that the existing file is not overwritten.

traceLevel
Specifies what to trace.

You can specify one or more of the following traces with the traceLevel parameter:

- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_NONE (X'00')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTION_CALLS (X'01')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_STATEMENT_CALLS (X'02')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_CALLS (X'04')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRIVER_CONFIGURATION (X'10')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS (X'20')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS (X'40')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_META_DATA (X'80')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_PARAMETER_META_DATA (X'100')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DIAGNOSTICS (X'200')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SQLJ (X'400')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_META_CALLS (X'2000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DATASOURCE_CALLS (X'4000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_LARGE_OBJECT_CALLS (X'8000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SYSTEM_MONITOR (X'20000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_TRACEPOINTS () (X'40000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL (X'FFFFFFFF')

To specify more than one trace, use one of these techniques:

- Use bitwise OR (|) operators with two or more trace values. For example, to trace DRDA flows and connection calls, specify this value for traceLevel:

  ```
  TRACE_DRDA_FLOWS|TRACE_CONNECTION_CALLS
  ```

- Use a bitwise complement (tilde (~)) operator with a trace value to specify all except a certain trace. For example, to trace everything except DRDA flows, specify this value for traceLevel:

  ```
  ~TRACE_DRDA_FLOWS
  ```

getTraceFile

Format:

```
public void getTraceFile()
  throws java.sql.SQLException
```

Returns the name of the destination file for the remote trace controller, if it is set. Otherwise, getTraceFile returns null.

setTraceDirectory

Format:

```
public void setTraceDirectory(String traceDirectory,
  String baseTraceFileName,
  int traceLevel) throws java.sql.SQLException
```

Specifies the name of the directory into which the remote trace controller writes trace information, and the type of information that is to be traced.

Parameter descriptions:

traceDirectory

Specifies a directory into which trace information is written. This setting overrides the settings of the traceDirectory and logWriter properties for a DataSource or DriverManager connection.

Each JDBC driver connection is traced to a different file in the specified directory. The naming convention for the files in that directory depends on whether a non-null value is specified for baseTraceFileName:

- If a null value is specified for baseTraceFileName, a connection is traced to a file named traceFile_global_*n*.

  n is the *n*th JDBC driver connection.

- If a non-null value is specified for baseTraceFileName, a connection is traced to a file named *baseTraceFileName*_global_*n*.

  baseTraceFileName is the value of the baseTraceFileName parameter.

  n is the *n*th JDBC driver connection.

baseTraceFileName
Specifies the stem for the names of the files into which global trace information is written. The combination of baseTraceFileName and traceDirectory determines the full path name for the global trace log files.

traceLevel
Specifies what to trace.

You can specify one or more of the following traces with the traceLevel parameter:

- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_NONE (X'00')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTION_CALLS (X'01')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_STATEMENT_CALLS (X'02')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_CALLS (X'04')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRIVER_CONFIGURATION (X'10')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS (X'20')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS (X'40')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_META_DATA (X'80')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_PARAMETER_META_DATA (X'100')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DIAGNOSTICS (X'200')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SQLJ (X'400')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_META_CALLS (X'2000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DATASOURCE_CALLS (X'4000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_LARGE_OBJECT_CALLS (X'8000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SYSTEM_MONITOR (X'20000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_TRACEPOINTS () (X'40000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL (X'FFFFFFFF')

To specify more than one trace, use one of these techniques:

- Use bitwise OR (|) operators with two or more trace values. For example, to trace DRDA flows and connection calls, specify this value for traceLevel:

  ```
  TRACE_DRDA_FLOWS|TRACE_CONNECTION_CALLS
  ```

- Use a bitwise complement (tilde (~)) operator with a trace value to specify all except a certain trace. For example, to trace everything except DRDA flows, specify this value for traceLevel:

  ```
  ~TRACE_DRDA_FLOWS
  ```

getTraceFileAppend
Format:

```
public abstract boolean getTraceFileAppend()
  throws java.sql.SQLException
```

Returns `true` if trace records that are generated by the trace controller are appended to the trace file. Otherwise, getTraceFileAppend returns `false`.

getTraceDirectory
Format:

```
public void getTraceDirectory()
  throws java.sql.SQLException
```

Returns the name of the destination directory for trace records that are generated by the trace controller, if it is set. Otherwise, getTraceDirectory returns null.

getTraceLevel

Format:

```
public void getTraceLevel()
  throws java.sql.SQLException
```

Returns the trace level for the trace records that are generated by the trace controller, if it is set. Otherwise, getTraceLevel returns -1 (TRACE_ALL).

unsetLogWriter

Format:

```
public abstract void unsetLogWriter()
  throws java.sql.SQLException
```

Disables the global log writer override for future connections.

suspendTrace

Format:

```
public void suspendTrace()
  throws java.sql.SQLException
```

Suspends all global, Connection-level, or DataSource-level traces for current and future connections. suspendTrace can be called when the global log writer is enabled or disabled.

resumeTrace

Format:

```
public void resumeTrace()
  throws java.sql.SQLException
```

Resumes all global, Connection-level, or DataSource-level traces for current and future connections. resumeTrace can be called when the global log writer is enabled or disabled. If the global log writer is disabled, resumeTrace resumes Connection-level or DataSource-level traces. If the global log writer is enabled, resumeTrace resumes the global trace.

DB2Types class

The com.ibm.db2.jcc.DB2Types class provides fields that define IBM Data Server Driver for JDBC and SQLJ-only data types.

DB2Types fields

The following constants define types codes only for the IBM Data Server Driver for JDBC and SQLJ.

- public final static int BLOB_FILE = -100002
- public final static int CLOB_FILE = -100003
- public final static int CURSOR = -100008
- public final static int DECFLOAT = -100001
- public final static int XML_AS_BLOB_FILE = -100004
- public final static int XML_AS_CLOB_FILE = -100005
- public final static int TIMESTAMPTZ =-100010

DB2XADataSource class

DB2XADataSource is a factory for XADataSource objects. An object that implements this interface is registered with a naming service that is based on the Java Naming and Directory Interface (JNDI).

The com.ibm.db2.jcc.DB2XADataSource class extends the com.ibm.db2.jcc.DB2BaseDataSource class, and implements the javax.sql.XADataSource, java.io.Serializable, and javax.naming.Referenceable interfaces.

DB2XADataSource methods

getDB2TrustedXAConnection

Formats:

```
public Object[] getDB2TrustedXAConnection(String user,
  String password,
  java.util.Properties properties)
  throws java.sql.SQLException
public Object[] getDB2TrustedXAConnection(
  java.util.Properties properties)
  throws java.sql.SQLException
public Object[] getDB2TrustedXAConnection(
  org.ietf.jgss.GSSCredential gssCredential,
  java.util.Properties properties)
  throws java.sql.SQLException
```

An application server using a system authorization ID uses this method to establish a trusted connection.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to:
  - DB2 Database for Linux, UNIX, and Windows Version 9.5 or later
  - DB2 for z/OS Version 9.1 or later
  - IBM Informix Version 11.70 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 9.1 or later

The following elements are returned in Object[]:

- The first element is a DB2TrustedXAConnection instance.
- The second element is a unique cookie for the generated XA connection instance.

The first form getDB2TrustedXAConnection provides a user ID and password. The second form of getDB2TrustedXAConnection uses the user ID and password of the DB2XADataSource object. The third form of getDB2TrustedXAConnection is for connections that use Kerberos security.

Parameter descriptions:

user
The authorization ID that is used to establish the trusted connection.

password
The password for the authorization ID that is used to establish the trusted connection.

gssCredential
If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

properties
Properties for the connection.

getDB2TrustedPooledConnection
Format:

```
public Object[] getDB2TrustedPooledConnection(java.util.Properties properties)
  throws java.sql.SQLException
```

An application server using a system authorization ID uses this method to establish a trusted connection, using the user ID and password for the DB2XADataSource object.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to:
  - DB2 Database for Linux, UNIX, and Windows Version 9.5 or later
  - DB2 for z/OS Version 9.1 or later
  - IBM Informix Version 11.70 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 9.1 or later

The following elements are returned in Object[]:

- The first element is a trusted DB2TrustedPooledConnection instance.
- The second element is a unique cookie for the generated pooled connection instance.

Parameter descriptions:

properties
Properties for the connection.

getDB2XAConnection
Formats:

```
public DB2XAConnection getDB2XAConnection(String user,
  String password,
  java.util.Properties properties)
  throws java.sql.SQLException
public DB2XAConnection getDB2XAConnection(
   org.ietf.jgss.GSSCredential gssCredential,
  java.util.Properties properties)
  throws java.sql.SQLException
```

Establishes the initial untrusted connection in a heterogeneous pooling environment.

The first form getDB2PooledConnection provides a user ID and password. The second form of getDB2XAConnection is for connections that use Kerberos security.

Parameter descriptions:

user
The authorization ID that is used to establish the connection.

password
The password for the authorization ID that is used to establish the connection.

gssCredential
If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

properties
Properties for the connection.

DBTimestamp class

The com.ibm.db2.jcc.DBTimestamp class can be used to create timestamp objects with a precision of up to picoseconds and time zone information. This class is primarily for support of the SQL TIMESTAMP WITH TIME ZONE data type, which is supported only by DB2 for z/OS.

The com.ibm.db2.jcc.DBTimestamp class is a subclass of the java.sql.Timestamp class. Therefore, a com.ibm.db2.jcc.DBTimestamp object can be used with any methods that normally operate on a java.sql.Timestamp object, or take a java.sql.Timestamp object as an argument.

The IBM Data Server Driver for JDBC and SQLJ returns a DBTimestamp object for all JDBC methods that return timestamp information, such as ResultSet.getTimestamp or CallableStatement.getTimestamp.

DBTimestamp constructor

The following constructor is defined only for the IBM Data Server Driver for JDBC and SQLJ.

DBTimestamp
Formats:

```
public DBTimestamp(long time,
  java.util.Calendar calendar)
  throws java.sql.SQLException
public DBTimestamp(long time, )
  throws java.sql.SQLException
public DBTimestamp(java.sql.Timestamp timestamp)
  throws java.sql.SQLException
public DBTimestamp(java.sql.Timestamp timestamp,
  java.util.Calendar calendar)
  throws java.sql.SQLException
```

Constructs a DBTimestamp object.

Parameter descriptions:

time
The number of milliseconds since January 1, 1970.

timestamp
A Timestamp value with a precision of up to picoseconds.

calendar
The Calendar value that provides the time zone.

DBTimestamp methods

getPicos
Formats:

```
public long getPicos()
```

Returns the fractional seconds component of a DBTimestamp value.

getTimeZone
Formats:

```
public java.util.TimeZone getTimeZone()
```

Returns the time zone component of a DBTimestamp value.

setPicos

Format:

```
public void setPicos(long p)
  throws SQLException
```

Assigns the given value to the fractional seconds component of a DBTimestamp value.

Parameter descriptions:

p A value between 0 and 999999999, inclusive, which is the fractional sections component of a DBTimestamp value.

setTimeZone

Format:

```
public void setTimeZone(java.util.TimeZone timeZone)
  throws SQLException
```

Assigns the given value to the time zone component of a DBTimestamp value.

Parameter descriptions:

timeZone
The time zone component of a DBTimestamp value.

valueOfDBString

Format:

```
public static DBTimestamp valueOfDBString(String s)
  throws java.lang.IllegalArgumentException
```

Constructs a DBTimestamp value from the string representation of a timestamp value.

Parameter descriptions:

s The string representation of a timestamp value. The value must be in one of the following formats:

yyyy-mm-dd.hh.mm.ss[.fffffffffff]-th:tm
yyyy-mm-dd hh:mm:ss[.fffffffffff]-th:tm
yyyy-mm-dd.hh.mm.ss[.fffffffffff]
yyyy-mm-dd hh:mm:ss[.fffffffffff]

- *yyyy* is a year.
- *mm* is a month.
- *dd* is a day.
- *hh* is hours.
- *mm* is minutes.
- *ss* is seconds.
- *[.fffffffffff]* is one to 12 optional fractions of seconds.
- *th* is the hours component of a time zone.
- *tm* is the minutes component of a time zone.

toDBString

Format:

```
public String toDBString(boolean includeTimeZone)
```

Returns the string representation of a DBTimestamp object.

The returned value has one of the following formats:

yyyy-mm-dd.hh.mm.ss[.fffffffffff]-th:tm
yyyy-mm-dd.hh.mm.ss[.fffffffffff]

Parameter description:

includeTimeZone
Specifies whether to include the time zone (*-th*:*tm*) in the returned string.

JDBC differences between versions of the IBM Data Server Driver for JDBC and SQLJ

Before you can upgrade your JDBC applications from older to newer versions of the IBM Data Server Driver for JDBC and SQLJ, you need to understand the differences between those drivers.

Supported methods

For a list of methods that the IBM Data Server Driver for JDBC and SQLJ supports, see "Driver support for JDBC APIs".

Use of progressive streaming by the JDBC drivers

For IBM Data Server Driver for JDBC and SQLJ, Version 3.50 and later, progressive streaming, which is also known as dynamic data format, behavior is the default for LOB retrieval, for connections to DB2 Database for Linux, UNIX, and Windows Version 9.5 and later.

Progressive streaming is supported in the IBM Data Server Driver for JDBC and SQLJ Version 3.1 and later, but for IBM Data Server Driver for JDBC and SQLJ version 3.2 and later, progressive streaming behavior is the default for LOB and XML retrieval, for connections to DB2 for z/OS Version 9.1 and later.

Previous versions of the IBM Data Server Driver for JDBC and SQLJ did not support progressive streaming.

Important: With progressive streaming, when you retrieve a LOB or XML value from a ResultSet into an application variable, you can manipulate the contents of that application variable until you move the cursor or close the cursor on the ResultSet. After that, the contents of the application variable are no longer available to you. If you perform any actions on the LOB in the application variable, you receive an SQLException. For example, suppose that progressive streaming is enabled, and you execute statements like this:

```
...
ResultSet rs = stmt.executeQuery("SELECT CLOBCOL FROM MY_TABLE");
rs.next();                // Retrieve the first row of the ResultSet
Clob clobFromRow1  = rs.getClob(1);
                          // Put the CLOB from the first column of
                          // the first row in an application variable
String substr1Clob = clobFromRow1.getSubString(1,50);
                          // Retrieve the first 50 bytes of the CLOB
rs.next();                // Move the cursor to the next row.
                          // clobFromRow1 is no longer available.
// String substr2Clob = clobFromRow1.getSubString(51,100);
                          // This statement would yield an SQLException
Clob clobFromRow2  = rs.getClob(1);
                          // Put the CLOB from the first column of
                          // the second row in an application variable
rs.close();               // Close the ResultSet.
                          // clobFromRow2 is also no longer available.
```

After you execute rs.next() to position the cursor at the second row of the ResultSet, the CLOB value in clobFromRow1 is no longer available to you. Similarly, after you execute rs.close() to close the ResultSet, the values in clobFromRow1 and clobFromRow2 are no longer available.

To avoid errors that are due to this changed behavior, you need to take one of the following actions:

- Modify your applications.

  Applications that retrieve LOB data into application variables can manipulate the data in those application variables only until the cursors that were used to retrieve the data are moved or closed.
- Disable progressive streaming by setting the `progressiveStreaming` property to `DB2BaseDataSource.NO` (2).

ResultSetMetaData values for IBM Data Server Driver for JDBC and SQLJ version 4.0 and later

For the IBM Data Server Driver for JDBC and SQLJ version 4.0 and later, the default behavior of ResultSetMetaData.getColumnName and `ResultSetMetaData.getColumnLabel` differs from the default behavior for earlier JDBC drivers.

If you need to use IBM Data Server Driver for JDBC and SQLJ version 4.0 or later, but your applications need to return the ResultSetMetaData.getColumnName and `ResultSetMetaData.getColumnLabel` values that were returned with older JDBC drivers, you can set the useJDBC4ColumnNameAndLabelSemantics Connection and `DataSource` property to `DB2BaseDataSource.NO` (2).

Batch updates with automatically generated keys have different results in different driver versions

With the IBM Data Server Driver for JDBC and SQLJ version 3.52 or later, preparing an SQL statement for retrieval of automatically generated keys is supported.

With the IBM Data Server Driver for JDBC and SQLJ version 3.50 or version 3.51, preparing an SQL statement for retrieval of automatically generated keys and using the PreparedStatement object for batch updates causes an SQLException.

Versions of the IBM Data Server Driver for JDBC and SQLJ before Version 3.50 do not throw an SQLException when an application calls the addBatch or executeBatch method on a PreparedStatement object that is prepared to return automatically generated keys. However, the PreparedStatement object does not return automatically generated keys.

Batch updates of data on DB2 for z/OS servers have different results in different driver versions

After you successfully invoke an executeBatch statement, the IBM Data Server Driver for JDBC and SQLJ returns an array. The purpose of the array is to indicate the number of rows that are affected by each SQL statement that is executed in the batch.

If the following conditions are true, the IBM Data Server Driver for JDBC and SQLJ returns `Statement.SUCCESS_NO_INFO` (-2) in the array elements:

- The application is connected to a subsystem that is in DB2 for z/OS Version 8 new-function mode, or later.
- The application is using Version 3.1 or later of the IBM Data Server Driver for JDBC and SQLJ.
- The IBM Data Server Driver for JDBC and SQLJ uses multi-row INSERT operations to execute batch updates.

This occurs because with multi-row INSERT, the database server executes the entire batch as a single operation, so it does not return results for individual SQL statements.

If you are using an earlier version of the IBM Data Server Driver for JDBC and SQLJ, or you are connected to a data source other than DB2 for z/OS Version 8 or later, the array elements contain the number of rows that are affected by each SQL statement.

Batch updates and deletes of data on DB2 for z/OS servers have different size limitations in different driver versions

Before IBM Data Server Driver for JDBC and SQLJ version 3.59 or 4.9, a DisconnectException with error code -4499 was thrown for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS if the size of an update or delete batch was greater than 32KB. Starting with version 3.59 or 4.9, this restriction no longer exists, and the exception is no longer thrown.

Initial value of the CURRENT CLIENT_ACCTNG special register

For a JDBC or SQLJ application that runs under the IBM Data Server Driver for JDBC and SQLJ version 2.6 or later, using type 4 connectivity, the initial value for the DB2 for z/OS CURRENT CLIENT_ACCTNG special register is the concatenation of the DB2 for z/OS version and the value of the clientWorkStation property. For any other JDBC driver, version, and connectivity, the initial value is not set.

Properties that control the use of multi-row FETCH

Before version 3.7 and version 3.51 of the IBM Data Server Driver for JDBC and SQLJ, multi-row FETCH support was enabled and disabled through the useRowsetCursor property, and was available only for scrollable cursors, and for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS. Starting with version 3.7 and 3.51:

- For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, the IBM Data Server Driver for JDBC and SQLJ uses only the enableRowsetSupport property to determine whether to use multi-row FETCH for scrollable or forward-only cursors.
- For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows, or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 Database for Linux, UNIX, and Windows, the IBM Data Server Driver for JDBC and SQLJ uses the enableRowsetSupport property to determine whether to use multi-row FETCH for scrollable cursors, if enableRowsetSupport is set. If enableRowsetSupport is not set, the driver uses the useRowsetCursor property to determine whether to use multi-row FETCH.

JDBC 1 positioned updates and deletes and multi-row FETCH

Before version 3.7 and version 3.51 of the IBM Data Server Driver for JDBC and SQLJ, multi-row FETCH from DB2 for z/OS tables was controlled by the useRowsetCursor property. If an application contained JDBC 1 positioned update or delete operations, and multi-row FETCH support was enabled, the IBM Data Server Driver for JDBC and SQLJ permitted the update or delete operations, but unexpected updates or deletes might occur.

Starting with version 3.7 and 3.51 of the IBM Data Server Driver for JDBC and SQLJ, the enableRowsetSupport property enables or disables multi-row FETCH from DB2 for z/OS tables or DB2 Database for Linux, UNIX, and Windows tables. The enableRowsetSupport property overrides the useRowsetCursor property. If multi-row FETCH is enabled through the enableRowsetSupport property, and an application contains a JDBC 1 positioned update or delete operation, the IBM Data Server Driver for JDBC and SQLJ throws an SQLException.

Valid forms of prepareStatement for retrieval of automatically generated keys from a DB2 for z/OS view

Starting with version 3.57 or version 4.7 of the IBM Data Server Driver for JDBC and SQLJ, if you are inserting data into a view on a DB2 for z/OS data server, and you want to retrieve automatically generated keys, you need to use one of the following methods to prepare the SQL statement that inserts rows into the view:

```
Connection.prepareStatement(sql-statement, String [] columnNames);
Connection.prepareStatement(sql-statement, int [] columnIndexes);
Statement.executeUpdate(sql-statement, String [] columnNames);
Statement.executeUpdate(sql-statement, int [] columnIndexes);
```

Examples of ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnLabel values

For the IBM Data Server Driver for JDBC and SQLJ version 4.0 and later, the default behavior of ResultSetMetaData.getColumnName and `ResultSetMetaData.getColumnLabel` differs from the default behavior for earlier JDBC drivers. You can use the useJDBC4ColumnNameAndLabelSemantics property to change this behavior.

The following examples show the values that are returned for IBM Data Server Driver for JDBC and SQLJ Version 4.0, and for previous JDBC drivers, when the useJDBC4ColumnNameAndLabelSemantics property is not set.

All queries use a table that is defined like this:

```
CREATE TABLE MYTABLE(INTCOL INT)
```

Example: The following query contains an AS CLAUSE, which defines a label for a column in the result set:

```
SELECT MYCOL AS MYLABEL FROM MYTABLE
```

The following table lists the ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnName values that are returned for the query:

Table 14-62. ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnName before and after IBM Data Server Driver for JDBC and SQLJ Version 4.0 for a query with an AS CLAUSE

	Behavior before IBM Data Server Driver for JDBC and SQLJ Version 4.0		Behavior for IBM Data Server Driver for JDBC and SQLJ Version 4.0 and later	
Target data source	**getColumnName value**	**getColumnLabel value**	**getColumnName value**	**getColumnLabel value**
DB2 Database for Linux, UNIX, and Windows	MYLABEL	MYLABEL	MYCOL	MYLABEL
IBM Informix	MYLABEL	MYLABEL	MYCOL	MYLABEL
DB2 for z/OS Version 8 or later, and DB2 UDB for iSeries V5R3 and later	MYLABEL	MYLABEL	MYCOL	MYLABEL
DB2 for z/OS Version 7, and DB2 UDB for iSeries V5R2	MYLABEL	MYLABEL	MYLABEL	MYLABEL

Example: The following query contains no AS clause:

```
SELECT MYCOL FROM MYTABLE
```

The ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnLabel methods on the query return MYCOL, regardless of the target data source.

Example: On a DB2 for z/OS or DB2 for i data source, a LABEL ON statement is used to define a label for a column:

```
LABEL ON COLUMN MYTABLE.MYCOL IS 'LABELONCOL'
```

The following query contains an AS CLAUSE, which defines a label for a column in the ResultSet:

```
SELECT MYCOL AS MYLABEL FROM MYTABLE
```

The following table lists the ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnName values that are returned for the query.

Table 14-63. ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnName before and after IBM Data Server Driver for JDBC and SQLJ Version 4.0 for a table column with a LABEL ON statement in a query with an AS CLAUSE

	Behavior before IBM Data Server Driver for JDBC and SQLJ Version 4.0		Behavior for IBM Data Server Driver for JDBC and SQLJ Version 4.0 and later	
Target data source	**getColumnName value**	**getColumnLabel value**	**getColumnName value**	**getColumnLabel value**
DB2 for z/OS Version 8 or later, and DB2 UDB for iSeries V5R3 and later	MYLABEL	LABELONCOL	MYCOL	MYLABEL
DB2 for z/OS Version 7, and DB2 UDB for iSeries V5R2	MYLABEL	LABELONCOL	MYCOL	LABELONCOL

Example: On a DB2 for z/OS or DB2 for i data source, a LABEL ON statement is used to define a label for a column:

```
LABEL ON COLUMN MYTABLE.MYCOL IS 'LABELONCOL'
```

The following query contains no AS CLAUSE:

```
SELECT MYCOL FROM MYTABLE
```

The following table lists the ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnName values that are returned for the query.

Table 14-64. ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnName before and after IBM Data Server Driver for JDBC and SQLJ Version 4.0 for a table column with a LABEL ON statement in a query with no AS CLAUSE

	Behavior before IBM Data Server Driver for JDBC and SQLJ Version 4.0		**Behavior for IBM Data Server Driver for JDBC and SQLJ Version 4.0**	
Target data source	**getColumnName value**	**getColumnLabel value**	**getColumnName value**	**getColumnLabel value**
DB2 for z/OS Version 8 or later, and DB2 UDB for i5/OS® V5R3 and later	MYCOL	LABELONCOL	MYCOL	MYCOL
DB2 for z/OS Version 7, and DB2 UDB for i5/OS V5R2	MYCOL	LABELONCOL	MYLABEL	LABELONCOL

Differences between the IBM Data Server Driver for JDBC and SQLJ and the IBM Informix JDBC Driver

Before you can migrate your JDBC applications from older drivers to the IBM Data Server Driver for JDBC and SQLJ, you need to understand the differences between those drivers.

The differences in support for JDBC methods are described in topic "Driver support for JDBC APIs" on page 14-68.

The following sections describe differences between the IBM Data Server Driver for JDBC and SQLJ and the Informix JDBC Driver:

• "Error codes" on page 14-206

IBM Informix environment variables not supported

The following IBM Informix environment variables that were supported for the Informix JDBC driver are not supported for the IBM Data Server Driver for JDBC and SQLJ.

- ALLOWREGISTEROUTFORINPARAM
- BIG_FET_BUF_SIZE
- CSM
- ENABLE_HDRSWITCH
- IFX_BATCHUPDATE_PER_SPEC
- IFX_CODESETLOB
- IFX_SET_FLOAT_AS_SMFLOAT
- IFX_TRIMTRAILINGSPACES
- IFXHOST_SECONDARY
- IFXHOST INFORMIXCONRETRY
- IFXPORTNO_SECONDARY
- INFORMIXCONTIME
- INFORMIXOPCACHE
- INFORMIXSERVER_SECONDARY
- INFORMIXSTACKSIZE
- JDBCTEMP
- LDAP_IFXBASE
- LDAP_PASSWD
- LDAP_URL
- LDAP_USER
- LOBCACHE
- NEWCODESET
- NEWLOCALE
- NEWNLSMAP
- PROXY
- SECURITY
- SQLH_FILE
- SQLH_LOC
- SQLH_TYPE
- SQLIDEBUG

SQL commands

The following SQL commands are not supported when changing frequently used connection attributes:

- CREATE DATABASE
- DROP DATABASE
- DATABASE
- SET ISOLATION LEVEL
- SET TRANSACTION LEVEL

Closing connections with active transactions

When closing a connection during an active transaction, IBM Data Server Driver for JDBC and SQLJ prevents the connection from closing and throws the following exception:

```
[ibm][db2][jcc][t4][10251] [10308][driver version] "java.sql.Connection.close()
requested while a transaction is in progress on the connection.
The transaction remains active, and the connection cannot be
closed. ERRORCODE=-4471, SQLSTATE=sqlstate"
```

Whereas, the Informix JDBC driver closes the connection and automatically rolls back the active transaction.

Environment variable mapping between Informix JDBC driver and IBM Data Server Driver for JDBC and SQLJ

This section describes the variables for IBM Data Server Driver for JDBC and SQLJ that replace those from Informix JDBC driver that are not supported.

Table 14-65. Replacement IBM Data Server Driver for JDBC and SQLJ environment variables

Environment variables used by the Informix JDBC driver	Equivalent environment variables for IBM Data Server Driver for JDBC and SQLJ
TRACEFILE, PROTOCOLTRACEFILE	traceFile
TRACE, PROTOCOLTRACE	traceLevel
FET_BUF_SIZE	queryBlockSize
IFX_AUTOFREE	queryCloseImplicit
INFORMIXCONTIME	loginTimeout, blockingReadConnectionTimeout
IFX_LOCK_MODE_WAIT	currentLockTimeout

Security mechanisms

This section describes differences in security mechanisms between Informix JDBC Driver and IBM Data Server Driver for JDBC and SQLJ.

Table 14-66. Security mechanisms differences

Functionality	Informix JDBC Driver	IBM Data Server Driver for JDBC and SQLJ
Requirement for a user ID and password	A user ID and password are not required to establish a connection.	At least a user ID must be supplied before a connection can be established. The user ID or password requirement depends on the value of the securityMechanism Connection and DataSource property.
Types of user ID and password security	Password encryption is supported, using environment variable SECURITY=PASSWORD. Userid encryption not supported.	Multiple types of user ID and password security are supported: • User ID only security • Encrypted user ID and password security • Clear text password security • Encrypted password security
Data encryption	Uses the communication support module (CSM), environment variable: `csm=(classname=com.informix.jdbc.Crypto, config=test.cfg)`	Not supported.
Authentication	Pluggable authentication modules; applications use com.informix.jdbc.IfmxPAM for Challenge-Response.	Not supported.

Namespaces

This section describes differences in namespace functionality between Informix JDBC Driver and IBM Data Server Driver for JDBC and SQLJ.

Table 14-67. Namespace differences

Functionality	Informix JDBC driver	IBM Data Server Driver for JDBC and SQLJ
Initial Class Loader	("com.informix.jdbc.IfxDriver")	("com.ibm.db2.jcc.DB2Driver") Change the initial class name. For example: `Class.forName("com.ibm.db2.jcc.DB2Driver")`
Package namespace	com.informix.*xxx*	com.ibm.db2.jcc.*xxx*
DataSource namespace	com.informix.jdbcx.IfxDataSource com.informix.jdbcx.IfxXA DataSource	com.ibm.db2.jcc.DB2SimpleDataSource com.ibm.db2.jcc.DB2XADataSource
DataSource method names	Environment variables are prepended with “getIfx” or “"setIfx”. For example: setIfxINFORMIXSTACKSIZE	Environment variables are prepended with “get” or “set”. For example: setINFORMIXSTACKSIZE

Cursors

This section describes differences in cursor functionality between Informix JDBC Driver and IBM Data Server Driver for JDBC and SQLJ.

Table 14-68. Cursor differences

Functionality	Informix JDBC Driver	IBM Data Server Driver for JDBC and SQLJ
Forward-only ResultSet	The Forward-only ResultSet stays open after the last row is accessed; it remains open until an explicit call to ResultSet.close();	The Forward-only ResultSet automatically closes after the last row is accessed. Throws an error if an application attempts to retrieve ResultSetMetaData after reading all rows. *
ResultSet.wasNull(): ResultSet has no data	• wasNull() returns FALSE if query returns no data (ResultSet not empty but does not contain data). • wasNull() returns FALSE if resultset is moved beyond ResetSet.last().	Throws an SQLException (client side) with client-specific error numbers.
Product name and version strings have semantic changes	dmd.getDatabaseProduct Version()="11.50.UC1"	dmd.getDatabaseProductVersion()= "IFX11500"
	dmd.getDatabaseProductName() ="Informix Dynamic Server"	dmd.getDatabaseProductName()= "IBM Informix Dynamic Server/UNIX32"
ResultSet.relative(): For illegal operations	Throws SQLException -79739 error "No current row".	Throws SQLException -4476 error "Cursor is not on a valid row."
ResultSet.next(): IFX does not have any data	Returns FALSE.	Throws an SQLException with following SELECT failed error: com.ibm.db2.jcc.am.SqlException: [ibm][db2][jcc][10120][10898] Invalid operation: result set is closed.
ResultSet.next(): if getByte() and getShort() on DECIMAL column result in overflow	Returns -1 for both methods to indicate an error.	Throws SQLException [ibm][db2][jcc][10177][11611] Invalid data conversion: Requested conversion would result in a loss of precision of *nnnnnn*.

Table 14-68. Cursor differences (continued)

Functionality	Informix JDBC Driver	IBM Data Server Driver for JDBC and SQLJ
ResultSet.updateTime()	ResultSet.updateTime() can be called on the DATETIME YEAR TO FRACTION column.	Not supported; use updateTimestamp().
Statement.setCursorName(""): Setting cursor name to an empty string	Checks for this condition and throws an illegal cursor name SQLException.	Passes the empty string to the server without returning an error.
Updatable Scroll cursors	Supported for files on the server.	Use Forward-Only, Updatable cursor

* For SELECT statements, the statement is complete when the associated result set is closed. The result set is closed as soon as one of the following conditions occurs:

- All of the rows have been retrieved.
- The associated Statement object is re-executed.
- Another Statement object is executed on the same connection.

Data types

This section describes differences in data type functionality between Informix JDBC Driver and IBM Data Server Driver for JDBC and SQLJ.

Table 14-69. Data type differences

Data Type	Informix JDBC Driver	IBM Data Server Driver for JDBC and SQLJ
BYTE, BLOB, TEXT, and CLOB need casting to resolve ambiguity when passed as parameters to functions or procedures	JDBC identifies BYTE as BINARY and smart large objects as BLOB.	Not supported.
DECIMAL	Maximum precision 32 digits.	Maximum precision 31 digits.
Floating point conversion to integer/byte	JDBC returns incorrect value.	Throws an SQLException if conversion is not possible.
getString() on BYTE column semantics	JDBC converts BYTE to STRING	Not supported.
Informix BOOLEAN is identified as SMALLINT	JDBC driver identifies BOOLEAN as BOOLEAN.	Does not differentiate BOOLEAN from SMALLINT.
Informix collection types: LIST, SET, MULTISET, and ROW	Supported.	Not supported.
Informix JDBC extensions classes for smart large objects (BLOB and CLOB) and simple large objects (BYTE and TEXT) and Statement type	Extension classes as supported by the Informix JDBC driver: • IfxBblob • IfxCblob • IfxSmartBlob • IfxStatementTypes • IfxTypes • IfxLoStat IfxLobDescriptor	Not Supported.
INTERVAL data type	Supported.	Not supported.
MONEY	Maximum precision 32 digits.	Maximum precision 31 digits.
Semantics of BLOB updates on client side being automatically reflected on the server	Changes to BLOB data on client side can be reflected on the server.	Does not automatically update BLOB unless the user issues an update.

Table 14-69. Data type differences (continued)

Data Type	Informix JDBC Driver	IBM Data Server Driver for JDBC and SQLJ
Smart large object functions like filetoblob, filetoclob, lotofile and locopy support	Supported.	Not supported.

Large object behavior

This section describes differences in large object behavior between Informix JDBC Driver and IBM Data Server Driver for JDBC and SQLJ.

Table 14-70. Large object behavior differences

Functionality	Informix JDBC Driver	IBM Data Server Driver for JDBC and SQLJ
getString on a BYTE column	Supported.	Returns a HEX representation of the inserted data inserted.
getBlob on a CLOB column	Supported.	Not supported.
Inserting LOB data using filetoblob or filetoclob	Supported.	Not supported.
BLOB and CLOB object updates with setBinaryStream() or setCharacterStream()	Objects updated automatically.	Objects not automatically updated. After calling setXXXStream(), you must issue an SQL UPDATE.

Data conversions

This section describes differences in data conversions between Informix JDBC Driver and IBM Data Server Driver for JDBC and SQLJ.

Table 14-71. Data conversion differences

Functionality	Informix JDBC Driver	IBM Data Server Driver for JDBC and SQLJ
Data Conversion from DECIMAL to SHORT/BYTE with out-of-range results	• getByte conversion: (-1) • getShort conversion: (-1) • getInt conversion: (-323855360)	• getByte conversion: (0) • getShort conversion: (23552) • getInt conversion: (-323855360)
DECIMAL PRECISION with PreparedStatement.setDouble()	A packed decimal can be up to 32.	A packed decimal can be up to 31.
REAL to DECIMAL conversion	Returns a REAL to DECIMAL conversion. For example: 12345.67871 is returned as 12345.67871.	Limits the scale to give consistent results for DECIMAL irrespective of the precision and scale. For example: 12345.67871 is returned as 12345.67800.
DECIMAL to BYTE/SHORT conversion	Not supported.	Throws an SQLException.

Prepared statements

This section describes differences in the prepared statement functionality between Informix JDBC Driver and IBM Data Server Driver for JDBC and SQLJ.

Table 14-72. Prepared statement differences

Functionality	Informix JDBC Driver	IBM Data Server Driver for JDBC and SQLJ
PreparedStatement.executeBatch() error handling	Throws an SQLException, only if there is an error in the batch execution.	Throws the SQLException:com.ibm. db2.jcc.am.BatchUpdateException'; applications that do not handle the exception fail at runtime.
PreparedStatement.setCharacter Stream(): characters in reader given do not match the length	Reads *n* characters as given in length from the start; characters exceeding the length are ignored.	Throws an SQLException.

Behavior of getIDSSerial() and getIDSSerial8()

This section describes differences for getIDSSerial() and getIDSSerial8() between Informix JDBC Driver and IBM Data Server Driver for JDBC and SQLJ.

Table 14-73. Differences for getIDSSerial() and getIDSSerial8()

Functionality	Informix JDBC Driver	IBM Data Server Driver for JDBC and SQLJ
When called after a rollback	Supported.	Not supported; returns 0.
Use with JDBC API getGeneratedKeys() method	Supported.	Not supported; returns 0.

Behavior of afterLast() followed by getRow()

When afterLast() is followed by getRow(), the IBM Data Server Driver for JDBC and SQLJ returns "0", resetting the cursor to the beginning. Following afterLast(), there are no valid rows.

The Informix JDBC driver returns the row number *last+1* if you call getRow() immediately after afterLast() call.

Parameter order

There are some differences in parameter order between Informix JDBC Driver and IBM Data Server Driver for JDBC and SQLJ.

According to the JDBC specifications, when the stored procedure returns a result parameter, a form of OUT parameter, it is treated just like any other OUT parameter. Its data type must be registered with the method registerOutParameter, and its value is retrieved with the appropriate getXXX method. Because a result parameter comes first in a call to a stored procedure, its ordinal position is always 1.

When the stored procedure returns the result parameter, it returns a result set and its value is retrieved with the appropriate result set getXXX method.

For IBM Data Server Driver for JDBC and SQLJ, the result of a stored procedure is treated as an OUT parameter. Its data type must be registered (like other OUT parameters) with the method **registeroutparameter** and its value is retrieved with the appropriate callable statement getXXX method.

To migrate your applications, remove the question mark (?) = from the {? = call (.....} statement.

Error codes

This section describes differences in error codes between Informix JDBC Driver and IBM Data Server Driver for JDBC and SQLJ.

Table 14-74. Error code differences

Functionality	Informix JDBC Driver	IBM Data Server Driver for JDBC and SQLJ
SQLException.getNextException and getErrorCode	Return an ISAM code and detailed message on SQLException.getNextException and getErrorCode.	Some IBM Informix errors have an SQL error code and an RSAM error code. The driver does not support RSAM error codes returned as nextException or as SQLException.getCause().
Client error messages	Uses -79xxx.	Some messages come from the server and have a different format and description. Client error messages are in the format: -4*nnn*.
Access to ISAM error codes	Gives ISAM error codes to applications.	Applications need to use diagnostics extensions to get ISAM error code. Applications can get detailed error message for ISAM error codes.

Case conversion of literal arguments in methods that return metadata

The Informix JDBC Driver converts arguments that represent database objects in DatabaseMetaData, ParameterMetaData, and ResultSetMetaData method calls to lowercase. The IBM Data Server Driver for JDBC and SQLJ does no case conversion.

For example, suppose that you want to retrieve information about the columns in a table named mytable. With the Informix JDBC Driver, calls of both of these forms return the information:

```
dbms.getColumns(null, null, "mytable", null);
dbms.getColumns(null, null, "MyTable", null);
```

With the IBM Data Server Driver for JDBC and SQLJ, only a call of the following form returns the information. The case of the table name in the getColumns call must be the same as the case of the table as it is defined in the database.

```
dbms.getColumns(null, null, "mytable", null);
```

Error codes issued by the IBM Data Server Driver for JDBC and SQLJ

Error codes in the ranges +4200 to +4299, +4450 to +4499, -4200 to -4299, and -4450 to -4499 are reserved for the IBM Data Server Driver for JDBC and SQLJ.

When you call the SQLException.getMessage method after a IBM Data Server Driver for JDBC and SQLJ error occurs, a string is returned that includes:

- Whether the connection is a type 2 or type 4 connection
- Diagnostic information for IBM Software Support
- The level of the driver

- An explanatory message
- The error code
- The SQLSTATE

For example:

```
[jcc][t4][20128][12071][3.50.54] Invalid queryBlockSize specified: 1,048,576,012.
Using default query block size of 32,767.  ERRORCODE=0, SQLSTATE=
```

Currently, the IBM Data Server Driver for JDBC and SQLJ issues the following error codes:

Table 14-75. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ

Error Code	Message text and explanation	SQLSTATE
+4204	Errors were encountered and tolerated as specified by the RETURN DATA UNTIL clause. **Explanation:** Tolerated errors include federated connection, authentication, and authorization errors. This warning applies only to connections to DB2 Database for Linux, UNIX, and Windows servers. It is issued only when a cursor operation, such as a ResultSet.next or ResultSet.previous call, returns `false`.	02506
+4222	*text-from-getMessage* **Explanation:** A warning condition occurred during connection to the data source. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
+4223	*text-from-getMessage* **Explanation:** A warning condition occurred during initialization. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
+4225	*text-from-getMessage* **Explanation:** A warning condition occurred when data was sent to a server or received from a server. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
+4226	*text-from-getMessage* **Explanation:** A warning condition occurred during customization or bind. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
+4228	*text-from-getMessage* **Explanation:** An warning condition occurred that does not fit in another category. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
+4450	Feature not supported: *feature-name*	

Table 14-75. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ (continued)

Error Code	Message text and explanation	SQLSTATE
+4460	*text-from-getMessage* **Explanation:** The specified value is not a valid option. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
+4461	*text-from-getMessage* **Explanation:** The specified value is invalid or out of range. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
+4462	*text-from-getMessage* **Explanation:** A required value is missing. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
+4470	*text-from-getMessage* **Explanation:** The requested operation cannot be performed because the target resource is closed. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
+4471	*text-from-getMessage* **Explanation:** The requested operation cannot be performed because the target resource is in use. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
+4472	*text-from-getMessage* **Explanation:** The requested operation cannot be performed because the target resource is because the target resource is unavailable. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
+4474	*text-from-getMessage* **Explanation:** The requested operation cannot be performed because the target resource cannot be changed. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4200	Invalid operation: An invalid COMMIT or ROLLBACK has been called in an XA environment during a Global Transaction. **Explanation:** An application that was in a global transaction in an XA environment issued a commit or rollback. A commit or rollback operation in a global transaction is invalid.	2D521

Table 14-75. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ (continued)

Error Code	Message text and explanation	SQLSTATE
-4201	Invalid operation: setAutoCommit(true) is not allowed during Global Transaction. **Explanation:** An application that was in a global transaction in an XA environment executed the setAutoCommit(true) statement. Issuing setAutoCommit(true) in a global transaction is invalid.	2D521
-4203	Error executing *function*. Server returned *rc*. **:** An error occurred on an XA connection during execution of an SQL statement. For network optimization, the IBM Data Server Driver for JDBC and SQLJ delays some XA flows until the next SQL statement is executed. If an error occurs in a delayed XA flow, that error is reported as part of the SQLException that is thrown by the current SQL statement.	
-4210	Timeout getting a transport object from pool.	57033
-4211	Timeout getting an object from pool.	57033
-4212	Sysplex member unavailable.	
-4213	Timeout.	57033
-4214	*text-from-getMessage* **Explanation:** Authorization failed. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	28000
-4220	*text-from-getMessage* **Explanation:** An error occurred during character conversion. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4221	*text-from-getMessage* **Explanation:** An error occurred during encryption or decryption. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4222	*text-from-getMessage* **Explanation:** An error occurred during connection to the data source. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4223	*text-from-getMessage* **Explanation:** An error occurred during initialization. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	

Table 14-75. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ (continued)

Error Code	Message text and explanation	SQLSTATE
-4224	*text-from-getMessage* **Explanation:** An error occurred during resource cleanup. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4225	*text-from-getMessage* **Explanation:** An error occurred when data was sent to a server or received from a server. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4226	*text-from-getMessage* **Explanation:** An error occurred during customization or bind. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4227	*text-from-getMessage* **Explanation:** An error occurred during reset. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4228	*text-from-getMessage* **Explanation:** An error occurred that does not fit in another category. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4229	*text-from-getMessage* **Explanation:** An error occurred during a batch execution. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4231	An error occurred during the conversion of column *column-number* of type *sql-data-type* with value *value* to a value of type java.math.BigDecimal.	
-4450	Feature not supported: *feature-name*	0A504
-4460	*text-from-getMessage* **Explanation:** The specified value is not a valid option. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	

Table 14-75. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ (continued)

Error Code	Message text and explanation	SQLSTATE
-4461	*text-from-getMessage* **Explanation:** The specified value is invalid or out of range. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	42815
-4462	*text-from-getMessage* **Explanation:** A required value is missing. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4463	*text-from-getMessage* **Explanation:** The specified value has a syntax error. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	42601
-4470	*text-from-getMessage* **Explanation:** The requested operation cannot be performed because the target resource is closed. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4471	*text-from-getMessage* **Explanation:** The requested operation cannot be performed because the target resource is in use. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4472	*text-from-getMessage* **Explanation:** The requested operation cannot be performed because the target resource is unavailable. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4473	*text-from-getMessage* **Explanation:** The requested operation cannot be performed because the target resource is no longer available. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4474	*text-from-getMessage* **Explanation:** The requested operation cannot be performed because the target resource cannot be changed. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	

Table 14-75. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ (continued)

Error Code	Message text and explanation	SQLSTATE
-4475	*text-from-getMessage* **Explanation:** The requested operation cannot be performed because access to the target resource is restricted. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4476	*text-from-getMessage* **Explanation:** The requested operation cannot be performed because the operation is not allowed on the target resource. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	
-4496	An SQL OPEN for a held cursor was issued on an XA connection. The JDBC driver does not allow a held cursor to be opened on the database server for an XA connection.	
-4497	The application must issue a rollback. The unit of work has already been rolled back in the DB2 server, but other resource managers involved in the unit of work might not have rolled back their changes. To ensure integrity of the application, all SQL requests are rejected until the application issues a rollback.	
-4498	A connection failed but has been reestablished. Host name or IP address: *host-name*, service name or port number: *port*, special register modification indicator: *rc*. **Explanation:** *host-name* and *port* indicate the data source at which the connection is reestablished. *rc* indicates whether SQL statements that set special register values were executed again: **1** SQL statements that set special register values were executed again. **2** SQL statements that set special register values might not have been executed again. For client reroute against DB2 for z/OS servers, special register values that were set after the last commit point are not re-established. The application is rolled back to the previous commit point. The connection state and global resources such as global temporary tables and open held cursors might not be maintained.	
-4499	*text-from-getMessage* **Explanation:** A fatal error occurred that resulted in a disconnect from the data source. The existing connection has become unusable. **User response:** Call SQLException.getMessage to retrieve specific information about the problem.	08001 or 58009
-30108	Client reroute exception for the Sysplex.	08506

Table 14-75. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ (continued)

Error Code	Message text and explanation	SQLSTATE
-99999	The IBM Data Server Driver for JDBC and SQLJ issued an error that does not yet have an error code.	

SQLSTATEs issued by the IBM Data Server Driver for JDBC and SQLJ

SQLSTATEs in the range 46600 to 466ZZ are reserved for the IBM Data Server Driver for JDBC and SQLJ.

The following table lists the SQLSTATEs that are generated or used by the IBM Data Server Driver for JDBC and SQLJ.

Table 14-76. SQLSTATEs returned by the IBM Data Server Driver for JDBC and SQLJ

SQLSTATE class	SQLSTATE	Description
01*xxx*		Warning
02*xxx*		No data
	02501	The cursor position is not valid for a FETCH of the current row.
	02506	Tolerable error
08*xxx*		Connection exception
	08001	The application requester is unable to establish the connection.
	08003	A connection does not exist
	08004	The application server rejected establishment of the connection
	08506	Client reroute exception
0A*xxx*		Feature not supported
	0A502	The action or operation is not enabled for this database instance
	0A504	The feature is not supported by the driver
22*xxx*		Data exception
	22007	The string representation of a datetime value is invalid
	22021	A character is not in the coded character set
23*xxx*		Constraint violation
	23502	A value that is inserted into a column or updates a column is null, but the column cannot contain null values.
24*xxx*		Invalid cursor state
	24501	The identified cursor is not open
28*xxx*		Authorization exception
	28000	Authorization name is invalid.
2D*xxx*		Invalid transaction termination
	2D521	SQL COMMIT or ROLLBACK are invalid in the current operating environment.

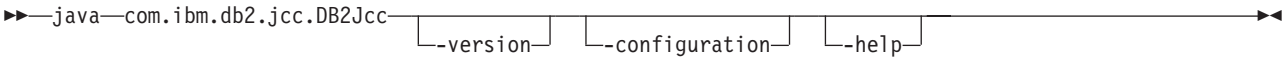
Table 14-76. SQLSTATEs returned by the IBM Data Server Driver for JDBC and SQLJ (continued)

SQLSTATE class	SQLSTATE	Description
34*xxx*		Invalid cursor name
	34000	Cursor name is invalid.
3B*xxx*		Invalid savepoint
	3B503	A SAVEPOINT, RELEASE SAVEPOINT, or ROLLBACK TO SAVEPOINT statement is not allowed in a trigger or global transaction.
40*xxx*		Transaction rollback
42*xxx*		Syntax error or access rule violation
	42601	A character, token, or clause is invalid or missing
	42734	A duplicate parameter name, SQL variable name, cursor name, condition name, or label was detected.
	42807	The INSERT, UPDATE, or DELETE is not permitted on this object
	42808	A column identified in the insert or update operation is not updateable
	42815	The data type, length, scale, value, or CCSID is invalid
	42820	A numeric constant is too long, or it has a value that is not within the range of its data type
	42968	The connection failed because there is no current software license.
57*xxx*		Resource not available or operator intervention
	57033	A deadlock or timeout occurred without automatic rollback
58*xxx*		System error
	58008	Execution failed due to a distribution protocol error that will not affect the successful execution of subsequent DDM commands or SQL statements
	58009	Execution failed due to a distribution protocol error that caused deallocation of the conversation
	58012	The bind process with the specified package name and consistency token is not active
	58014	The DDM command is not supported
	58015	The DDM object is not supported
	58016	The DDM parameter is not supported
	58017	The DDM parameter value is not supported

How to find IBM Data Server Driver for JDBC and SQLJ version and environment information

To determine the version of the IBM Data Server Driver for JDBC and SQLJ, as well as information about the environment in which the driver is running, run the DB2Jcc utility on the command line.

DB2Jcc syntax



DB2Jcc option descriptions

-version
Specifies that the IBM Data Server Driver for JDBC and SQLJ displays its name and version.

-configuration
Specifies that the IBM Data Server Driver for JDBC and SQLJ displays its name and version, and information about its environment, such as information about the Java runtime environment, operating system, path information, and license restrictions.

-help
Specifies that the DB2Jcc utility describes each of the options that it supports. If any other options are specified with -help, they are ignored.

Commands for SQLJ program preparation

To prepare SQLJ programs for execution, you use commands to translate SQLJ source code into Java source code and compile the Java source code.

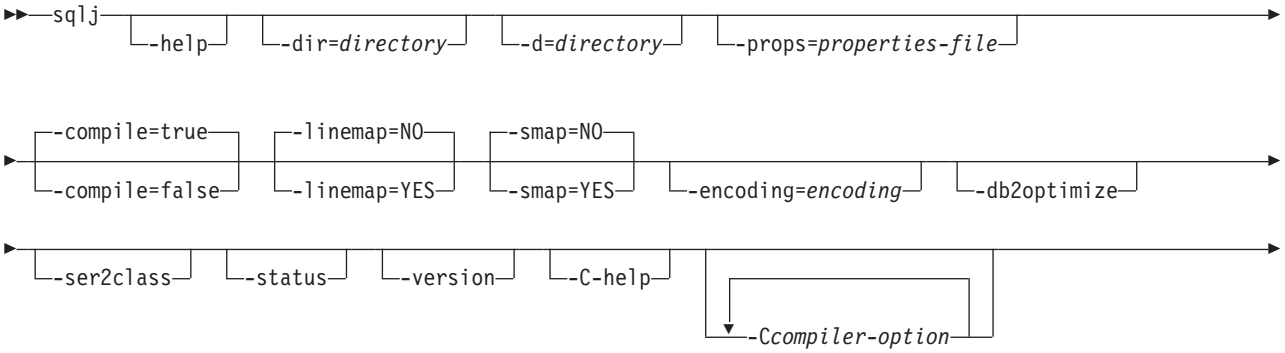
sqlj - SQLJ translator

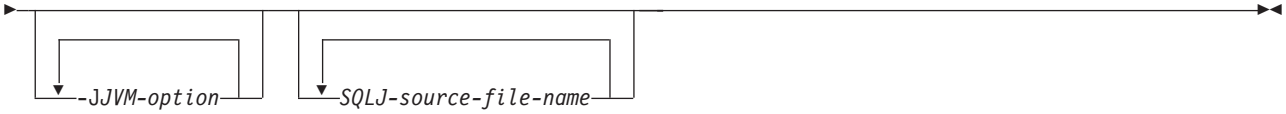
The sqlj command translates an SQLJ source file into a Java source file and zero or more SQLJ serialized profiles. By default, the sqlj command also compiles the Java source file.

Authorization

None

Command syntax





Command parameters

-help
Specifies that the SQLJ translator describes each of the options that the translator supports. If any other options are specified with -help, they are ignored.

-dir=*directory*
Specifies the name of the directory into which SQLJ puts .java files that are generated by the translator and .class files that are generated by the compiler. The default is the directory that contains the SQLJ source files.

The translator uses the directory structure of the SQLJ source files when it puts the generated files in directories. For example, suppose that you want the translator to process two files:
- file1.sqlj, which is not in a Java package
- file2.sqlj, which is in Java package sqlj.test

Also suppose that you specify the parameter -dir=/src when you invoke the translator. The translator puts the Java source file for file1.sqlj in directory /src and puts the Java source file for file2.sqlj in directory /src/sqlj/test.

-d=*directory*
Specifies the name of the directory into which SQLJ puts the binary files that are generated by the translator and compiler. These files include the .ser files, the *name*_SJProfileKeys.class files, and the .class files that are generated by the compiler.

The default is the directory that contains the SQLJ source files.

The translator uses the directory structure of the SQLJ source files when it puts the generated files in directories. For example, suppose that you want the translator to process two files:
- file1.sqlj, which is not in a Java package
- file2.sqlj, which is in Java package sqlj.test

Also suppose that you specify the parameter -d=/src when you invoke the translator. The translator puts the serialized profiles for file1.sqlj in directory /src and puts the serialized profiles for file2.sqlj in directory /src/sqlj/test.

-compile=true|false
Specifies whether the SQLJ translator compiles the generated Java source into bytecodes.

true
The translator compiles the generated Java source code. This is the default.

false
The translator does not compile the generated Java source code.

-linemap=no|yes
Specifies whether line numbers in Java exceptions match line numbers in the SQLJ source file (the .sqlj file), or line numbers in the Java source file that is generated by the SQLJ translator (the .java file).

no Line numbers in Java exceptions match line numbers in the Java source file. This is the default.

yes
Line numbers in Java exceptions match line numbers in the SQLJ source file.

-smap=no|yes
Specifies whether the SQLJ translator generates a source map (SMAP) file for each SQLJ source file. An SMAP file is used by some Java language debug tools. This file maps lines in the SQLJ source file to lines in the Java source file that is generated by the SQLJ translator. The file is in the Unicode UTF-8 encoding scheme. Its format is described by Original Java Specification Request (JSR) 45, which is available from this web site:

```
http://www.jcp.org
```

no Do not generated SMAP files. This is the default.

yes
Generate SMAP files. An SMAP file name is *SQLJ-source-file-name*.java.smap. The SQLJ translator places the SMAP file in the same directory as the generated Java source file.

-encoding=*encoding-name*
Specifies the encoding of the source file. Examples are JIS or EUC. If this option is not specified, the default converter for the operating system is used.

-db2optimize
Specifies that the SQLJ translator generates code for a connection context class that is optimized for IDS. -db2optimize optimizes the code for the user-defined context but not the default context.

When you run the SQLJ translator with the -db2optimize option, if your applications use JDBC 3.0 or earlier functions, the IBM Data Server Driver for JDBC and SQLJ file db2jcc.jar must be in the CLASSPATH for compiling the generated Java application. If your applications use JDBC 4.0 or earlier functions, the IBM Data Server Driver for JDBC and SQLJ file db2jcc4.jar must be in the CLASSPATH for compiling the generated Java application.

-ser2class
Specifies that the SQLJ translator converts .ser files to .class files.

-status
Specifies that the SQLJ translator displays status messages as it runs.

-version
Specifies that the SQLJ translator displays the version of the IBM Data Server Driver for JDBC and SQLJ. The information is in this form:

```
IBM SQLJ xxxx.xxxx.xx
```

-C-help
Specifies that the SQLJ translator displays help information for the Java compiler.

-C*compiler-option*
Specifies a valid Java compiler option that begins with a dash (-). Do not include spaces between -C and the compiler option. If you need to specify multiple compiler options, precede each compiler option with -C. For example:

```
-C-g -C-verbose
```

All options are passed to the Java compiler and are not used by the SQLJ translator, **except** for the following options:

-classpath
Specifies the user class path that is to be used by the SQLJ translator and the Java compiler. This value overrides the CLASSPATH environment variable.

-sourcepath
Specifies the source code path that the SQLJ translator and the Java compiler search for class or interface definitions. The SQLJ translator searches for .sqlj and .java files only in directories, not in JAR or zip files.

-J*JVM-option*
Specifies an option that is to be passed to the Java virtual machine (JVM) in which the sqlj command runs. The option must be a valid JVM option that begins with a dash (-). Do not include spaces between -J and the JVM option. If you need to specify multiple JVM options, precede each compiler option with -J. For example:

```
-J-Xmx128m -J-Xmine2M
```

SQLJ-source-file-name
Specifies a list of SQLJ source files to be translated. This is a required parameter. All SQLJ source file names must have the extension .sqlj.

Output

For each source file, *program-name*.sqlj, the SQLJ translator produces the following files:

- The generated source program

  The generated source file is named *program-name*.java.
- A serialized profile file for each connection context class that is used in an SQLJ executable clause

  A serialized profile name is of the following form:

  *program-name*_SJProfile*IDNumber*.ser
- If the SQLJ translator invokes the Java compiler, the class files that the compiler generates.

Examples

```
sqlj -encoding=UTF8 -C-O MyApp.sqlj
```

Appendix. Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

Accessibility features for IBM Informix products

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in IBM Informix products. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- The attachment of alternative input and output devices.

Tip: The information center and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features by using the keyboard instead of the mouse.

Keyboard navigation

This product uses standard Microsoft Windows navigation keys.

Related accessibility information

IBM is committed to making our documentation accessible to persons with disabilities. Our publications are available in HTML format so that they can be accessed with assistive technology such as screen reader software.

You can view the publications in Adobe Portable Document Format (PDF) by using the Adobe Acrobat Reader.

IBM and accessibility

See the *IBM Accessibility Center* at http://www.ibm.com/able for more information about the IBM commitment to accessibility.

Dotted decimal syntax diagrams

The syntax diagrams in our publications are available in dotted decimal format, which is an accessible format that is available only if you are using a screen reader.

In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), the elements can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read punctuation. All syntax elements that have the same dotted decimal number (for example, all syntax elements that have the number 3.1) are mutually exclusive

alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, the word or symbol is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is read as 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol that provides information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, that element is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 refers to a separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

? Specifies an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element (for example, 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! Specifies a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used.

However, if you hear the lines `2? FILE`, `2.1`, `2.1.1! (KEEP)`, and `2.1.1 (DELETE)`, the default option KEEP only applies to the next higher dotted decimal number, `2.1` (which does not have an associated keyword), and does not apply to `2? FILE`. Nothing is used if the keyword `FILE` is omitted.

* Specifies a syntax element that can be repeated zero or more times. A dotted decimal number followed by the `*` symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line `5.1* data-area`, you know that you can include more than one data area or you can include none. If you hear the lines `3*, 3 HOST`, and `3 STATE`, you know that you can include `HOST`, `STATE`, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write `HOST STATE`, but you cannot write `HOST HOST`.
3. The `*` symbol is equivalent to a loop-back line in a railroad syntax diagram.

\+ Specifies a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times. For example, if you hear the line `6.1+ data-area`, you must include at least one data area. If you hear the lines `2+`, `2 HOST`, and `2 STATE`, you know that you must include `HOST`, `STATE`, or both. As for the * symbol, you can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the `*` symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy,

modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

A

B

C

D

E

G

H

I

J

L

M

N

P

R

S

T

U

V

W

Printed in USA

SC27-3850-00

Spine information:

Informix Product Family Data Server Driver for JDBC and SQLJ **Version 9.7**

IBM Data Server Driver for JDBC and SQLJ for Informix