DB2 10 for z/OS

Application Programming Guide and Reference for Java

IBM
Before using this information and the product it supports, be sure to read the general information under “Notices” at the end of this information.
## Contents

**About this information** ........................................... ix
Who should read this information ................................. ix
DB2 Utilities Suite for z/OS ........................................... ix
Terminology and citations ............................................ ix
Accessibility features for DB2 10 for z/OS ....................... x
How to send your comments ......................................... x
How to read syntax diagrams ....................................... x

**Chapter 1. Java application development for IBM data servers** .......................... 1

**Chapter 2. Supported drivers for JDBC and SQLJ.** ................................. 3
JDBC driver and database version compatibility .................. 4
How to find the IBM Data Server Driver for JDBC and SQLJ version ........................................... 5
DB2 for z/OS and IBM Data Server Driver for JDBC and SQLJ version ........................................... 6
DB2 and IBM Data Server Driver for JDBC and SQLJ levels ........ 6

**Chapter 3. JDBC application programming** ........................................ 9
Example of a simple JDBC application ............................... 9
How JDBC applications connect to a data source ................ 11
Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ ........................................... 13
Connecting to a data source using the DataSource interface 20
How to determine which type of IBM Data Server Driver for JDBC and SQLJ connectivity to use . 22
JDBC connection objects ................................................. 23
Creating and deploying DataSource objects ............................ 24
Java packages for JDBC support ........................................ 25
Learning about a data source using DatabaseMetaData methods ........................................... 26
DatabaseMetaData methods for identifying the type of data server ........................................... 27
Variables in JDBC applications ........................................... 28
Comments in a JDBC application ......................................... 29
JDBC interfaces for executing SQL ....................................... 29
Creating and modifying database objects using the Statement.executeUpdate method ............ 30
Updating data in tables using the PreparedStatement.executeUpdate method ........................................... 31
JDBC executeUpdate methods against a DB2 for z/OS server ........................................... 33
Making batch updates in JDBC applications ......................... 33
Learning about parameters in a PreparedStatement using ParameterMetaData methods ........................................... 37
Data retrieval in JDBC applications ..................................... 39
Calling stored procedures in JDBC applications ....................... 53
LOBs in JDBC applications with the IBM Data Server Driver for JDBC and SQLJ ........................................... 59
ROWIDs in JDBC with the IBM Data Server Driver for JDBC and SQLJ ........................................... 65
Update and retrieval of timestamps with time zone information in JDBC applications ........................................... 66
Distinct types in JDBC applications ..................................... 75
Savepoints in JDBC applications ........................................ 75
Retrieval of automatically generated keys in JDBC applications ........................................... 77
Named parameter markers in JDBC applications ..................... 80
Retrieving JSON documents from a ResultSet ......................... 84
Providing extended client information to the data source with IBM Data Server Driver for JDBC and SQLJ-only methods ........................................... 87
Providing extended client information to the data source with client info properties ............... 88
Extended parameter information with the IBM Data Server Driver for JDBC and SQLJ ........................................... 91
Using DB2PreparedStatement methods or constants to provide extended parameter information ........................................... 92
Using DB2ResultSet methods or DB2PreparedStatement constants to provide extended parameter information ........................................... 94
XML data in JDBC applications ........................................... 95
XML column updates in JDBC applications ......................... 96
XML data retrieval in JDBC applications ............................. 98
Chapter 4. SQLJ application programming ........................................ 125

Example of a simple SQLJ application ........................................ 125

Connecting to a data source using SQLJ ...................................... 127
  SQLJ connection technique 1: JDBC DriverManager interface ........ 127
  SQLJ connection technique 2: JDBC DriverManager interface ........ 129
  SQLJ connection technique 3: JDBC DataSource interface .......... 130
  SQLJ connection technique 4: JDBC DataSource interface .......... 132
  SQLJ connection technique 5: Use a previously created connection context 133
  SQLJ connection technique 6: Use the default connection .......... 134

Java packages for SQLJ support ............................................. 134

Variables in SQLJ applications .............................................. 135

Indicator variables in SQLJ applications .................................... 136

Comments in an SQLJ application .......................................... 140

SQL statement execution in SQLJ applications ............................ 140
  Creating and modifying database objects in an SQLJ application .. 141
  Performing positioned UPDATE and DELETE operations in an SQLJ application 141
  Data retrieval in SQLJ applications ..................................... 151
  Calling stored procedures in SQLJ applications ....................... 162
  LOBs in SQLJ applications with the IBM Data Server Driver for JDBC and SQLJ 164
  SQLJ and JDBC in the same application ................................ 166
  Controlling the execution of SQL statements in SQLJ ............... 170
  ROWIDs in SQLJ with the IBM Data Server Driver for JDBC and SQLJ 170
  TIMESTAMP WITH TIME ZONE values in SQLJ applications ....... 172
  Distinct types in SQLJ applications .................................... 173
  Savepoints in SQLJ applications ....................................... 174

XML data in SQLJ applications ............................................. 175
  XML column updates in SQLJ applications ............................ 176
  XML data retrieval in SQLJ applications .............................. 178
  XMLCAST in SQLJ applications ......................................... 179

Inserting data from file reference variables into tables in SQLJ applications 180

SQLJ utilization of SDK for Java Version 5 function ....................... 181

Transaction control in SQLJ applications .................................. 183
  Setting the isolation level for an SQLJ transaction ................. 184
  Committing or rolling back SQLJ transactions ....................... 184
  Handling SQL errors and warnings in SQLJ applications .......... 184
    Handling SQL errors in an SQLJ application ....................... 185
    Handling SQL warnings in an SQLJ application ................... 185
  Closing the connection to a data source in an SQLJ application . 186

Chapter 5. Java stored procedures and user-defined functions .......... 189

Setting up the environment for Java routines ............................. 189
  Setting up the WLM application environment for Java routines ... 190
  Runtime environment for Java routines ................................ 193
Chapter 6. Preparing and running JDBC and SQLJ programs. .............................. 209
  Program preparation for JDBC programs .......................................................... 209
  Program preparation for SQLJ programs .......................................................... 209
  Binding SQLJ applications to access multiple database servers ....................... 210
  Program preparation for Java routines ............................................................ 212
    Preparation of Java routines with no SQLJ clauses ....................................... 212
    Preparation of Java routines with SQLJ clauses .......................................... 214
  Creating JAR files for Java routines ................................................................ 217
  Running JDBC and SQLJ programs ................................................................. 218

Chapter 7. JDBC and SQLJ reference information. .................................................. 219
  Data types that map to database data types in Java applications ....................... 219
    Date, time, and timestamp values that can cause problems in JDBC and SQLJ applications .......................................................... 226
    Data loss for timestamp data in JDBC and SQLJ applications ....................... 230
    Retrieval of special values from DECIMAL columns in Java applications ........ 231
    Use of PreparedStatement.executeUpdate to set values in TIMESTAMP WITH TIME ZONE columns ....................................................... 233
  Properties for the IBM Data Server Driver for JDBC and SQLJ ....................... 234
    Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products ..................................................... 235
    Common IBM Data Server Driver for JDBC and SQLJ properties for DB2 servers. ............................................................................ 263
    Common IBM Data Server Driver for JDBC and SQLJ properties for DB2 for z/OS and IBM Informix ......................................................... 276
    Common IBM Data Server Driver for JDBC and SQLJ properties for IBM Informix and Db2 on Linux, UNIX, and Windows systems ............. 278
    IBM Data Server Driver for JDBC and SQLJ properties for Db2 on Linux, UNIX, and Windows systems .................................................... 278
    IBM Data Server Driver for JDBC and SQLJ properties for DB2 for z/OS ............ 282
    IBM Data Server Driver for JDBC and SQLJ properties for IBM Informix ......... 290
    IBM Data Server Driver for JDBC and SQLJ configuration properties ............. 295
  Driver support for JDBC APIs ............................................................................ 320
  IBM Data Server Driver for JDBC and SQLJ support for SQL escape syntax ....... 346
  SQLJ statement reference information ............................................................... 346
    SQLJ clause ........................................................................................................ 346
    SQLJ host-expression ......................................................................................... 346
    SQLJ implements-clause .................................................................................... 348
    SQLJ with-clause ............................................................................................... 348
    SQLJ connection-declaration-clause ................................................................ 350
    SQLJ iterator-declaration-clause ....................................................................... 351
    SQLJ executable-clause .................................................................................... 353
    SQLJ context-clause .......................................................................................... 353
    SQLJ statement-clause ...................................................................................... 354
    SQLJ SET-TRANSACTION-clause ...................................................................... 356
    SQLJ assignment-clause .................................................................................... 357
    SQLJ iterator-conversion-clause ...................................................................... 358
  Interfaces and classes in the sqlj.runtime package ......................................... 358
    sqlj.runtime.ConnectionContext interface .................................................... 359
    sqlj.runtime.ForUpdate interface .................................................................... 364
    sqlj.runtime.NamedIterator interface ............................................................ 364
    sqlj.runtime.PositionedIterator interface ...................................................... 365
    sqlj.runtime.ResultSetIterator interface ...................................................... 365
    sqlj.runtime.Scrollable interface ................................................................... 368
    sqlj.runtime.AsiiStream class .......................................................................... 370
    sqlj.runtime.BinaryStream class ..................................................................... 371
Failover support with IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS
Java client support for high availability for connections to IBM Informix servers
Java client support for high availability for connections to Db2 on Linux, UNIX, and Windows systems
Security for preparing SQLJ applications with the IBM Data Server Driver for JDBC and SQLJ
Chapter 9. Installing the IBM Data Server Driver for JDBC and SQLJ on Db2 on Linux, UNIX, and Windows systems
Special setup for accessing DB2 for z/OS servers from Java programs
Chapter 10. Security under the IBM Data Server Driver for JDBC and SQLJ
User ID and password security under the IBM Data Server Driver for JDBC and SQLJ
User ID-only security under the IBM Data Server Driver for JDBC and SQLJ
Encrypted password, user ID, or data security under the IBM Data Server Driver for JDBC and SQLJ
Kerberos security under the IBM Data Server Driver for JDBC and SQLJ
IBM Data Server Driver for JDBC and SQLJ trusted context support
IBM Data Server Driver for JDBC and SQLJ support for SSL
Configuring connections under the IBM Data Server Driver for JDBC and SQLJ to use SSL
Configuring the Java Runtime Environment to use SSL
IBM Data Server Driver for JDBC and SQLJ support for certificate authentication
Security for preparing SSL for connections to Db2 on Linux, UNIX, and Windows systems
Chapter 11. Java client support for high availability on IBM data servers
Java client support for high availability for connections to Db2 on Linux, UNIX, and Windows systems servers
Configuration of Db2 on Linux, UNIX, and Windows systems automatic client reroute support for Java clients
Example of enabling Db2 on Linux, UNIX, and Windows systems automatic client reroute support in Java applications
Configuration of Db2 on Linux, UNIX, and Windows systems workload balancing support for Java clients
Example of enabling Db2 on Linux, UNIX, and Windows systems workload balancing support in Java applications
Operation of automatic client reroute for connections to Db2 on Linux, UNIX, and Windows systems from Java clients
Operation of alternate group support for connections to Db2 on Linux, UNIX, and Windows systems
Operation of workload balancing for connections to Db2 on Linux, UNIX, and Windows systems
Application programming requirements for high availability for connections to Db2 on Linux, UNIX, and Windows systems servers
Client affinities for Db2 on Linux, UNIX, and Windows systems
Java client support for high availability for connections to IBM Informix servers
Configuration of IBM Informix high-availability support for Java clients
Example of enabling IBM Informix high availability support in Java applications
Operation of automatic client reroute for connections to IBM Informix from Java clients
Operation of workload balancing for connections to IBM Informix from Java clients
Application programming requirements for high availability for connections from Java clients to IBM Informix servers
Client affinities for connections to IBM Informix from Java clients
Java client direct connect support for high availability for connections to DB2 for z/OS servers
Configuration of Sysplex workload balancing and automatic client reroute for Java clients
Example of enabling DB2 for z/OS Sysplex workload balancing and automatic client reroute in Java applications
Operation of Sysplex workload balancing for connections from Java clients to DB2 for z/OS servers
Operation of automatic client reroute for connections from Java clients to DB2 for z/OS servers
Application programming requirements for high availability for connections from Java clients to DB2 for z/OS servers
Failover support with IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS
Chapter 12. JDBC and SQLJ connection pooling support ............... 617

Chapter 13. IBM Data Server Driver for JDBC and SQLJ statement caching ........ 619

Chapter 14. IBM Data Server Driver for JDBC and SQLJ type 4 connectivity JDBC and SQLJ distributed transaction support .......................................................... 621
Example of a distributed transaction that uses JTA methods ............... 621

Chapter 15. JDBC and SQLJ global transaction support .................. 627

Chapter 16. Problem diagnosis with the IBM Data Server Driver for JDBC and SQLJ 629
DB2Jcc - IBM Data Server Driver for JDBC and SQLJ diagnostic utility .................................................. 631
Examples of using configuration properties to start a JDBC trace .......... 632
Example of a trace program under the IBM Data Server Driver for JDBC and SQLJ .............. 634
Techniques for monitoring IBM Data Server Driver for JDBC and SQLJ Sysplex support ............... 637

Chapter 17. Tracing IBM Data Server Driver for JDBC and SQLJ C/C++ native driver code ............................................................. 641
db2jctrace - Format IBM Data Server Driver for JDBC and SQLJ trace data for C/C++ native driver code .... 641

Chapter 18. System monitoring for the IBM Data Server Driver for JDBC and SQLJ 643
Information resources for DB2 10 for z/OS and related products ............ 647

Notices .................................................. 649
Programming interface information ............................................. 650
Trademarks ............................................................................. 651
Privacy policy considerations .................................................... 651

Glossary .................................................................................. 653

Index ...................................................................................... 655
About this information

This information describes Db2® for z/OS® support for Java™. This support lets you access relational databases from Java application programs.

Who should read this information

This information is for the following users:

- DB2 for z/OS application developers who are familiar with Structured Query Language (SQL) and who know the Java programming language.
- DB2 for z/OS system programmers who are installing JDBC and SQLJ support.

DB2 Utilities Suite for z/OS

**Important:** In DB2® 10, the DB2 Utilities Suite for z/OS is available as an optional product. You must separately order and purchase a license to such utilities, and discussion of those utility functions in this publication is not intended to otherwise imply that you have a license to them.

DB2 Utilities Suite for z/OS can work with DB2 Sort for z/OS and the DFSORT program. You are licensed to use DFSORT in support of the DB2 utilities even if you do not otherwise license DFSORT for general use. If your primary sort product is not DFSORT, consider the following informational APARs mandatory reading:

- II14047/II14213: USE OF DFSORT BY DB2 UTILITIES
- II13495: HOW DFSORT TAKES ADVANTAGE OF 64-BIT REAL ARCHITECTURE

These informational APARs are periodically updated.

Related concepts:

- [DB2 utilities packaging (DB2 Utilities)](#)

Terminology and citations

When referring to a DB2 product other than DB2 for z/OS, this information uses the product's full name to avoid ambiguity.

The following terms are used as indicated:

- **DB2** Represents either the DB2 licensed program or a particular DB2 subsystem.
- **Tivoli® OMEGAMON® XE for DB2 Performance Expert on z/OS**
  - Refers to any of the following products:
    - IBM® Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS
    - IBM Tivoli OMEGAMON XE for DB2 Performance Monitor for z/OS
    - IBM DB2 Performance Expert for Multiplatforms and Workgroups
    - IBM DB2 Buffer Pool Analyzer for z/OS
- **C, C++, and C language**
  - Represent the C or C++ programming language.
- **CICS®** Represents CICS Transaction Server for z/OS.
- **IMS™** Represents the IMS Database Manager or IMS Transaction Manager.
MVS™ Represents the MVS element of the z/OS operating system, which is equivalent to the Base Control Program (BCP) component of the z/OS operating system.

RACF® Represents the functions that are provided by the RACF component of the z/OS Security Server.

Accessibility features for DB2 10 for z/OS

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 z/OS products, including DB2 10 for z/OS. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers and screen magnifiers.
- Customization of display attributes such as color, contrast, and font size

Tip: The IBM Knowledge Center (which includes information for DB2 for z/OS) and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features using the keyboard instead of the mouse.

Keyboard navigation

For information about navigating the DB2 for z/OS ISPF panels using TSO/E or ISPF, refer to the z/OS TSO/E Primer, the z/OS TSO/E User’s Guide, and the z/OS ISPF User’s Guide. These guides describe how to navigate each interface, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Related accessibility information

IBM and accessibility

See the IBM Accessibility Center at http://www.ibm.com/able for more information about the commitment that IBM has to accessibility.

How to send your comments

Your feedback helps IBM to provide quality information. Please send any comments that you have about this book or other DB2 for z/OS documentation.

Send your comments by email to db2zinfo@us.ibm.com and include the name of the product, the version number of the product, and the number of the book. If you are commenting on specific text, please list the location of the text (for example, a chapter and section title or a help topic title).

How to read syntax diagrams

Certain conventions apply to the syntax diagrams that are used in IBM documentation.
Apply the following rules when reading the syntax diagrams that are used in DB2 for z/OS documentation:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.
  The ►► symbol indicates the beginning of a statement.
  The ―► symbol indicates that the statement syntax is continued on the next line.
  The ◄► symbol indicates that a statement is continued from the previous line.
  The ◄◄ symbol indicates the end of a statement.
- Required items appear on the horizontal line (the main path).

![Required Item Diagram]

- Optional items appear below the main path.

![Optional Item Diagram]

If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

![Optional Item Diagram Above Main Path]

- If you can choose from two or more items, they appear vertically, in a stack.
  If you must choose one of the items, one item of the stack appears on the main path.

![Required Choice Stack Diagram]

If choosing one of the items is optional, the entire stack appears below the main path.

![Optional Choice Stack Diagram Below Main Path]

If one of the items is the default, it appears above the main path and the remaining choices are shown below.

![Default Choice Stack Diagram]

- An arrow returning to the left, above the main line, indicates an item that can be repeated.

![Repeatable Item Diagram]
If the repeat arrow contains a comma, you must separate repeated items with a comma.

A repeat arrow above a stack indicates that you can repeat the items in the stack.

• Sometimes a diagram must be split into fragments. The syntax fragment is shown separately from the main syntax diagram, but the contents of the fragment should be read as if they are on the main path of the diagram.

• With the exception of XPath keywords, keywords appear in uppercase (for example, FROM). Keywords must be spelled exactly as shown. XPath keywords are defined as lowercase names, and must be spelled exactly as shown. Variables appear in all lowercase letters (for example, column-name). They represent user-supplied names or values.

• If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Related concepts:
- Syntax rules for DB2 commands (DB2 Commands)
- DB2 online utilities (DB2 Utilities)
- DB2 stand-alone utilities (DB2 Utilities)

Related information:
- DB2 and related commands (DB2 Commands)
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 Data 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 pureQuery data access applications.

Related concepts:

- Chapter 2, “Supported drivers for JDBC and SQLJ,” on page 3

Related reference:
IBM Data Studio product documentation (IBM Data Studio, IBM Optim Database Administrator, IBM infoSphere Data Architect)

Developing high-performance data access applications with pureQuery
Chapter 2. Supported drivers for JDBC and SQLJ

The IBM Data Server Driver for JDBC and SQLJ product includes support for two types of JDBC driver architecture. However not all data servers support both types.

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.

**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.

**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.

DB2 for z/OS supports the IBM Data Server Driver for JDBC and SQLJ, which combines type 2 and type 4 JDBC implementations. The driver is packaged in the following way:
- db2jcc.jar and sqlj.zip for JDBC 3.0 and earlier support
- db2jcc4.jar and sqlj4.zip for JDBC 4.0 or later, and JDBC 3.0 or earlier support

You control the level of JDBC support that you want by specifying the appropriate set of files in the CLASSPATH.

**Important:** db2jcc.jar and sqlj.zip are now deprecated. After version 3.72 of the IBM Data Server Driver for JDBC and SQLJ, which is delivered with DB2 Version 11.1 for Linux, UNIX, and Windows Modification Pack 1 Fix Pack 1, db2jcc.jar and sqlj.zip will include no new features. However, fixes will continue to be delivered.

**IBM Data Server Driver for JDBC and SQLJ (type 2 and 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. When an application loads the IBM Data Server Driver for JDBC and SQLJ, a single driver instance is loaded for type 2 and type 4 implementations. The application can make type 2 and type 4 connections using this single driver instance. The type 2 and type 4 connections can be made concurrently. IBM Data Server Driver for JDBC and SQLJ type 2 driver behavior is referred to as **IBM Data Server Driver for JDBC and SQLJ type 2 connectivity**. 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.xx is JDBC 3.0-compliant. IBM Data Server Driver for JDBC and SQLJ version 4.xx is compliant with JDBC 4.0 or later.

The IBM Data Server Driver for JDBC and SQLJ supports these JDBC and SQLJ functions:

- Version 3.xx supports all of the methods that are described in the JDBC 3.0 specifications.
- Version 4.xx supports all of the methods that are described in the JDBC 4.0 or later specifications.
- SQLJ application programming interfaces, as defined by the SQLJ standards, for simplified data access from Java applications.
- Connections that are enabled for connection pooling. WebSphere® Application Server or another application server does the connection pooling.
- Connections to a data server from Java user-defined functions and stored procedures use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity only. Applications that call user-defined functions or stored procedures can use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity or IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to connect to a data server.
- 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 Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS environment, Version 7 or later, or to Db2 on Linux, UNIX, and Windows systems).

In general, you should use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity for Java programs that run on the same z/OS system or IBM Z logical partition (LPAR) as the target DB2 subsystem. Use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity for Java programs that run on a different z/OS system or LPAR from the target DB2 subsystem.

For z/OS systems or LPARs that do not have DB2 for z/OS, the z/OS Application Connectivity to DB2 for z/OS optional feature can be installed to provide IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to a Db2 on Linux, UNIX, and Windows systems data server.

To use the IBM Data Server Driver for JDBC and SQLJ, you need Java 2 Technology Edition, V5 or later.

Related concepts:

- “Environment variables for the z/OS Application Connectivity to DB2 for z/OS feature” on page 539

### JDBC driver and database version compatibility

The compatibility of a particular version of the IBM Data Server Driver for JDBC and SQLJ with a database version depends on the type of driver connectivity that you are using and the type of data source to which you are connecting.
Compatibility for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity

The IBM Data Server Driver for JDBC and SQLJ is always downward compatible with DB2 databases at the previous release level.

For example, IBM Data Server Driver for JDBC and SQLJ type 4 connectivity from the IBM Data Server Driver for JDBC and SQLJ version 3.61, which is shipped with Db2 on Linux, UNIX, and Windows systems Version 9.7 Fix Pack 3, to a Db2 on Linux, UNIX, and Windows systems Version 8 database is supported.

The IBM Data Server Driver for JDBC and SQLJ is upward compatible with the next version of a DB2 database if the applications under which the driver runs use no new features.

For example, IBM Data Server Driver for JDBC and SQLJ type 4 connectivity from the IBM Data Server Driver for JDBC and SQLJ version 2.x, which is shipped with DB2 for z/OS Version 8, to a z/OS Version 9.1 database is supported, if the applications under which the driver runs contain no z/OS Version 9.1 features.

IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to IBM Informix is supported only for IBM Informix Version 11 and later.

Compatibility for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity

In general, IBM Data Server Driver for JDBC and SQLJ type 2 connectivity is intended for connections to the local database system, using the driver version that is shipped with that database version. For example, version 3.6x of the IBM Data Server Driver for JDBC and SQLJ is shipped with Db2 on Linux, UNIX, and Windows systems Version 9.5 and Version 9.7, and DB2 for z/OS Version 8 and later.

However, for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to a local Db2 on Linux, UNIX, and Windows systems database, the database version can be one version earlier or one version later than the version with which the driver was shipped. For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to a local DB2 for z/OS subsystem, the subsystem version can be one version later than the z/OS version with which the driver was shipped.

If the database version to which your applications are connecting is later than the database version with which the driver was shipped, the applications cannot use features of the later database version.

Related concepts:
Chapter 2, “Supported drivers for JDBC and SQLJ,” on page 3

How to find the IBM Data Server Driver for JDBC and SQLJ version

To determine the version of the IBM Data Server Driver for JDBC and SQLJ, run the DB2Jcc utility with the -version parameter.

Enter this text on the client command line.
java com.ibm.db2.jcc.DB2Jcc -version

Related reference:
DB2 for z/OS and IBM Data Server Driver for JDBC and SQLJ levels

DB2 for z/OS APARs are used to ship the IBM Data Server Driver for JDBC and SQLJ. Each APAR contains a JDBC 3 version and a JDBC 4 version of the driver.

Important: The JDBC 3 version of the IBM Data Server Driver for JDBC and SQLJ is now deprecated. After version 3.72, the JDBC 3 version will include no new features. However, fixes will continue to be delivered.

IBM Data Server Driver for JDBC and SQLJ versions and DB2 for z/OS Version 10 APARs

The following table lists the major 4.xx versions of the IBM Data Server Driver for JDBC and SQLJ and the DB2 for z/OS Version 10 APAR that delivered the initial release of each version.

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ version</th>
<th>DB2 for z/OS Version 10 APAR/PTF</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.15</td>
<td>PM84363/UK94639</td>
</tr>
<tr>
<td>4.14</td>
<td>PM65003/UK83409</td>
</tr>
<tr>
<td>4.13</td>
<td>PM47801/UK76380</td>
</tr>
<tr>
<td>4.12</td>
<td>PM32361/UK66666</td>
</tr>
</tbody>
</table>

The following table lists the major 3.xx versions of the IBM Data Server Driver for JDBC and SQLJ and the DB2 for z/OS Version 10 APAR that delivered the initial release of each version.

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ version</th>
<th>DB2 for z/OS Version 10 APAR/PTF</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.65</td>
<td>PM84359/UK94635</td>
</tr>
<tr>
<td>3.64</td>
<td>PM65007/UK83411</td>
</tr>
<tr>
<td>3.63</td>
<td>PM47803/UK76374</td>
</tr>
<tr>
<td>3.62</td>
<td>PM32360/UK66662</td>
</tr>
</tbody>
</table>

Related information:
- IBM Data Server Driver for JDBC and SQLJ Versions and DB2 for z/OS APARs

DB2 and IBM Data Server Driver for JDBC and SQLJ levels

Each version of DB2 is shipped with a JDBC 3 version and a JDBC 4 version of the IBM Data Server Driver for JDBC and SQLJ.
Important: The JDBC 3 version of the IBM Data Server Driver for JDBC and SQLJ is now deprecated. After version 3.72, the JDBC 3 version will include no new features. However, fixes will continue to be delivered.

The following table lists the DB2 versions and corresponding IBM Data Server Driver for JDBC and SQLJ versions.

You can use this information to determine the level of DB2 or Db2 Connect that is associated with the IBM Data Server Driver for JDBC and SQLJ instance under which a client program is running.

Table 3. Db2 on Linux, UNIX, and Windows systems fix pack levels and versions of the IBM Data Server Driver for JDBC and SQLJ.

<table>
<thead>
<tr>
<th>DB2 version and fix pack level</th>
<th>IBM Data Server Driver for JDBC and SQLJ version</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2 Version 11.1 Modification 1 Fix Pack</td>
<td>3.72.xx, 4.22.xx</td>
</tr>
<tr>
<td>DB2 Version 11.1</td>
<td>3.71.xx, 4.21.xx</td>
</tr>
<tr>
<td>DB2 Version 10.5 Fix Pack 5</td>
<td>3.69.xx, 4.19.xx</td>
</tr>
<tr>
<td>Db2 Cancun Release 10.5.0.4</td>
<td>3.68.xx, 4.18.xx</td>
</tr>
<tr>
<td>DB2 Version 10.5 Fix Pack 2</td>
<td>3.67.xx, 4.17.xx</td>
</tr>
<tr>
<td>DB2 Version 10.5</td>
<td>3.66.xx, 4.16.xx</td>
</tr>
<tr>
<td>DB2 Version 10.1 Fix Pack 2</td>
<td>3.65.xx, 4.15.xx</td>
</tr>
<tr>
<td>DB2 Version 10.1 Fix Pack 1</td>
<td>3.64.xx, 4.14.xx</td>
</tr>
<tr>
<td>DB2 Version 10.1</td>
<td>3.63.xx, 4.13.xx</td>
</tr>
<tr>
<td>DB2 Version 9.7 Fix Pack 6</td>
<td>3.64.xx, 4.14.xx</td>
</tr>
<tr>
<td>DB2 Version 9.7 Fix Pack 5</td>
<td>3.63.xx, 4.13.xx</td>
</tr>
<tr>
<td>DB2 Version 9.7 Fix Pack 2</td>
<td>3.59.xx, 4.9.xx</td>
</tr>
<tr>
<td>DB2 Version 9.7 Fix Pack 1</td>
<td>3.58.xx, 4.8.xx</td>
</tr>
<tr>
<td>DB2 Version 9.7</td>
<td>3.57.xx, 4.7.xx</td>
</tr>
<tr>
<td>DB2 Version 9.5 Fix Pack 7</td>
<td>3.61.xx, 4.8.xx</td>
</tr>
<tr>
<td>DB2 Version 9.5 Fix Pack 6</td>
<td>3.58.xx, 4.8.xx</td>
</tr>
<tr>
<td>DB2 Version 9.5 Fix Pack 5</td>
<td>3.57.xx, 4.7.xx</td>
</tr>
<tr>
<td>DB2 Version 9.5 Fix Pack 3 and Fix Pack 4</td>
<td>3.53.xx, 4.3.xx</td>
</tr>
<tr>
<td>DB2 Version 9.5 Fix Pack 2</td>
<td>3.52.xx, 4.2.xx</td>
</tr>
<tr>
<td>DB2 Version 9.5 Fix Pack 1</td>
<td>3.51.xx, 4.1.xx</td>
</tr>
<tr>
<td>DB2 Version 9.5</td>
<td>3.50.xx, 4.0.xx</td>
</tr>
<tr>
<td>DB2 Version 9.1 Fix Pack 5 and later</td>
<td>3.7.xx</td>
</tr>
<tr>
<td>DB2 Version 9.1 Fix Pack 4</td>
<td>3.6.xx</td>
</tr>
<tr>
<td>DB2 Version 9.1 Fix Pack 3</td>
<td>3.4.xx</td>
</tr>
<tr>
<td>DB2 Version 9.1 Fix Pack 2</td>
<td>3.3.xx</td>
</tr>
<tr>
<td>DB2 Version 9.1 Fix Pack 1</td>
<td>3.2.xx</td>
</tr>
<tr>
<td>DB2 Version 9.1</td>
<td>3.1.xx</td>
</tr>
</tbody>
</table>
Table 3. Db2 on Linux, UNIX, and Windows systems fix pack levels and versions of the IBM Data Server Driver for JDBC and SQLJ (continued).

<table>
<thead>
<tr>
<th>DB2 version and fix pack level</th>
<th>IBM Data Server Driver for JDBC and SQLJ version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note:</td>
<td></td>
</tr>
<tr>
<td>1. All driver versions are of the form n.m.xx. n.m stays the same within a GA level or a fix pack level. xx changes when a new version of the IBM Data Server Driver for JDBC and SQLJ is introduced through an APAR fix.</td>
<td></td>
</tr>
</tbody>
</table>

You can find more detailed information about IBM Data Server Driver for JDBC and SQLJ and Db2 on Linux, UNIX, and Windows systems versions at the following URL:

http://www.ibm.com/support/docview.wss?uid=swg21363866
Chapter 3. 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 data server 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.

```
import java.sql.*;  

public class EzJava  
{
    public static void main(String[] args)  
    {  
        String urlPrefix = "jdbc:db2:";  
        String url;  
        String user;  
        String password;  
        String empNo;  
        Connection con;  
        Statement stmt;  
        ResultSet rs;  

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

        // Check that the first argument has the correct form for the portion  
        // of the URL that follows jdbc:db2:,  
        // 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 2 connectivity,  
        // args[0] might be MVS1DB2M. For  
        // type 4 connectivity, args[0] might  
        // be //stlmvs1:10110/MVS1DB2M.  

        if (args.length!=3)  
        {  
            System.err.println("Invalid value. First argument appended to "+  
            "jdbc:db2: 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.");  
        }
    }
}
```

Figure 1. Simple JDBC application
System.exit(1);
}
url = urlPrefix + args[0];
user = args[1];
password = args[2];
try
{
    // Load the driver
    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
    con = DriverManager.getConnection(url, user, password);
    // Commit changes manually
    con.setAutoCommit(false);
    System.out.println("**** Created a JDBC connection to the data source");

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

    // Execute a query and generate a ResultSet instance
    rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE");
    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();
    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)
{
    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
}
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 2 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, and native code

Figure 2. Java application flow for IBM Data Server Driver for JDBC and SQLJ type 2 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 3. 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.

Procedure

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 or later, 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`
   
   For compatibility with previous JDBC drivers, you can use the following argument instead:
   
   `COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver`
   
   The following code demonstrates loading the IBM Data Server Driver for JDBC and SQLJ:
   
   ```java
   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:
   
   ```java
   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.

Examples

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

```java
String url = "jdbc:db2://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:

```java
String url = "jdbc:db2://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:

```java
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:db2://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**

```
jdbc:db2://server://database
jdbc:db2j:net://database
jdbc:ids://database
jdbc:ibmdb://database
```

**connection-options:**

(1) property=value; special-registers query-acceleration
special-registers:

```
specialRegisters = special-register-name=special-register-value;
```

query-acceleration:

```
queryAcceleration = NONE
-ENABLE
-ENABLE WITH FAILBACK
-ELIGIBLE
-ALL
```

Notes:

1. property=value pairs and the special-registers string can be specified in any order.

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 on Linux, UNIX, and Windows systems.
  - 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:ibmdb:**

Indicates that the connection is to any IBM Data Server that the IBM Data Server Driver for JDBC and SQLJ supports.

- This includes Db2 on Linux, UNIX, and Windows systems, DB2 for z/OS, the Db2 for IBM i, IBM Informix, and IBM dashDB® servers.

**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. The default is 446.

**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:

```sql
SELECT CURRENT SERVER FROM SYIBM.SYSDUMMY1;
```

• If the connection is to a DB2 for z/OS server or a Db2 for IBM i server, all characters in *database* must be uppercase characters.

• If the connection is to a Db2 on Linux, UNIX, and Windows systems 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:db2://sysmvs1.stl.ibm.com:5021/STLEC1" + ":user=dbadm;password=dbadm;" + ":traceLevel=" + 
(com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL) + ";";
```

```java
Connection con = 
java.sql.DriverManager.getConnection(url);
```

### specialRegisters= specially register-name=special-register-value,... special-register-name=special-register-value

A list of special register settings for the JDBC connection. You can specify one or more special register name and value pairs. Special register name and value pairs must be delimited by commas (,). The last pair must end with a semicolon (;). For example:

```
String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/STLEC1" + ":user=dbadm;password=dbadm;" + ":specialRegisters=CURRENT_PATH=SYSIBM,CURRENT_CLIENT_USERID=test;";
```

```java
Connection con = 
java.sql.DriverManager.getConnection(url);
```

For special registers that can be set through IBM Data Server Driver for JDBC and SQLJ Connection properties, if you set a special register value in a URL string using specialRegisters properties, and you also set that value in a
java.util.Properties object using the following form of getConnection, the special register is set to the value from the URL string.

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

You can specify only one value for each special register using the `specialRegisters` parameter. For special registers that take multiple values, such as CURRENT PATH, CURRENT PACKAGE PATH, CURRENT PACKAGESET, you can specify multiple values for a special register by using the `DataSource` interface and the `DB2DataSource.setSpecialRegisters` method.

```
queryAcceleration=value;
```

Changes the value of the CURRENT QUERY ACCELERATION special register.

Possible values are:

**NONE**

Specifies that no query acceleration is done.

**ENABLE**

Specifies that queries are accelerated only if DB2 determines that it is advantageous to do so. If there is an accelerator failure while a query is running, or the accelerator returns an error, DB2 returns a negative SQLCODE to the application.

**ENABLE WITH FAILBACK**

Specifies that queries are accelerated only if DB2 determines that it is advantageous to do so. If the accelerator returns an error during the PREPARE or first OPEN for the query, DB2 executes the query without the accelerator. If the accelerator returns an error during a FETCH or a subsequent OPEN, DB2 returns the error to the user, and does not execute the query.

**ELIGIBLE**

Specifies that queries are accelerated if they are eligible for acceleration. DB2 does not use cost information to determine whether to accelerate the queries. Queries that are not eligible for acceleration are executed by DB2. If there is an accelerator failure while a query is running, or the accelerator returns an error, DB2 returns a negative SQLCODE to the application.

**ALL**

Specifies that queries are accelerated if they are eligible for acceleration. DB2 does not use cost information to determine whether to accelerate the queries. Queries that are not eligible for acceleration are not executed by DB2, and an SQL error is returned. If there is an accelerator failure while a query is running, or the accelerator returns an error, DB2 returns a negative SQLCODE to the application.

Related reference:

"DB2DataSource class" on page 424

**URL format for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity**

If you are using type 2 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 2 connectivity.
IBM Data Server Driver for JDBC and SQLJ type 2 connectivity URL syntax

connection-options:

(1) property=value; special-registers; query-acceleration

special-registers:

specialRegisters= special-register-name=special-register-value

query-acceleration:

queryAcceleration= NONE, ENABLE, ENABLE WITH FAILBACK, ELIGIBLE, ALL

Notes:

1 property=value pairs and the special-registers string can be specified in any order.

IBM Data Server Driver for JDBC and SQLJ type 2 connectivity URL options descriptions

The parts of the URL have the following meanings:

jdbc:db2: or jdbc:db2os390: or jdbc:db2os390sqlj: or jdbc:default:connection

The meanings of the initial portion of the URL are:

jdbc:db2: or jdbc:db2os390: or jdbc:db2os390sqlj:
Indicates that the connection is to a DB2 for z/OS or Db2 on Linux, UNIX, and Windows systems server. jdbc:db2os390: and jdbc:db2os390sqlj: are for compatibility of programs that were written for older drivers, and apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS only.

jdbc:default:connection
Indicates that the URL is for a connection to the local subsystem through a DB2 thread that is controlled by CICS, IMS, or the Java stored procedure environment.
database
A name for the database server.

- database is a location name that is defined in the SYSIBM.Locations
catalog table.

All characters in the DB2 location name must be uppercase characters.
However, for a connection to a DB2 for z/OS server, the IBM Data Server
Driver for JDBC and SQLJ converts lowercase characters in the database
value to uppercase.

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:

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

specialRegisters=special-register-name=special-register-value,...
special-register-name=special-register-value
A list of special register settings for the JDBC connection. You can specify one
or more special register name and value pairs. Special register name and value
pairs must be delimited by commas (,). The last pair must end with a
semicolon (;). For example:

```java
String url =
"jdbc:db2:STLEC1" +
":user=dbadm;password=dbadm;" +
"specialRegisters=CURRENT_PATH=SYSIBM,CURRENT_CLIENT_USERID=test" + ";";
Connection con =
java.sql.DriverManager.getConnection(url);
```

For special registers that can be set through IBM Data Server Driver for JDBC
and SQLJ Connection properties, if you set a special register value in a URL
string using specialRegisters, and you also set that value in a
java.util.Properties object using the following form of getConnection, the
special register is set to the value from the URL string.

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

If you specify a special register that is supported on the data server, but you
specify a value that is not supported on the data server, the IBM Data Server
Driver for JDBC and SQLJ returns an error. If you specify a special register that
is not supported on the data server, the driver returns a warning.

You can specify only one value for each special register using the
specialRegisters parameter. For special registers that take multiple values, such as
CURRENT PATH, CURRENT PACKAGE PATH, CURRENT PACKAGESET,
you can specify multiple values for a special register by using the DataSource interface and the DB2DataSource.setSpecialRegisters method.

```java
queryAcceleration=value;
```

Changes the value of the CURRENT QUERY ACCELERATION special register.

Possible values are:

**NONE**

Specifies that no query acceleration is done.

**ENABLE**

Specifies that queries are accelerated only if DB2 determines that it is advantageous to do so. If there is an accelerator failure while a query is running, or the accelerator returns an error, DB2 returns a negative SQLCODE to the application.

**ENABLE WITH FAILBACK**

Specifies that queries are accelerated only if DB2 determines that it is advantageous to do so. If the accelerator returns an error during the PREPARE or first OPEN for the query, DB2 executes the query without the accelerator. If the accelerator returns an error during a FETCH or a subsequent OPEN, DB2 returns the error to the user, and does not execute the query.

**ELIGIBLE**

Specifies that queries are accelerated if they are eligible for acceleration. DB2 does not use cost information to determine whether to accelerate the queries. Queries that are not eligible for acceleration are executed by DB2. If there is an accelerator failure while a query is running, or the accelerator returns an error, DB2 returns a negative SQLCODE to the application.

**ALL**

Specifies that queries are accelerated if they are eligible for acceleration. DB2 does not use cost information to determine whether to accelerate the queries. Queries that are not eligible for acceleration are not executed by DB2, and an SQL error is returned. If there is an accelerator failure while a query is running, or the accelerator returns an error, DB2 returns a negative SQLCODE to the application.

Related reference:

"DB2DataSource class" on page 424

### 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.

### About this task

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.

**Procedure**

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:

   ```java
   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.

**Examples**

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

```java
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jcc.*;
import java.sql.Connection;

DB2SimpleDataSource dbds=new DB2SimpleDataSource();
dbds.setDatabaseName("dbloc1");
// 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();
// Create a Connection object
```

Figure 4. Obtaining a connection using a DataSource object

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

Note  Description
1 Import the package that contains the implementation of the DataSource interface.
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 the DataSource implementations of DB2.
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.

Related tasks:
"Connecting to a data source using SQLJ” on page 127

How to determine which type of IBM Data Server Driver for JDBC and SQLJ connectivity to use

The IBM Data Server Driver for JDBC and SQLJ supports two types of connectivity: type 2 connectivity and type 4 connectivity.

For the DriverManager interface, you specify the type of connectivity through the URL in the DriverManager.getConnection method. For the DataSource interface, you specify the type of connectivity through the driverType property.

The following table summarizes the differences between type 2 connectivity and type 4 connectivity:
Table 4. Comparison of IBM Data Server Driver for JDBC and SQLJ type 2 connectivity and IBM Data Server Driver for JDBC and SQLJ type 4 connectivity

<table>
<thead>
<tr>
<th>Function</th>
<th>IBM Data Server Driver for JDBC and SQLJ type 2 connectivity support</th>
<th>IBM Data Server Driver for JDBC and SQLJ type 4 connectivity support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>Better for accessing a local data server</td>
<td>Better for accessing a remote data server</td>
</tr>
<tr>
<td>Installation</td>
<td>Requires installation of native libraries in addition to Java classes</td>
<td>Requires installation of Java classes only</td>
</tr>
<tr>
<td>Stored procedures</td>
<td>Can be used to call or execute stored procedures</td>
<td>Can be used only to call stored procedures</td>
</tr>
<tr>
<td>Distributed transaction processing (XA)</td>
<td>Not supported</td>
<td>Supported</td>
</tr>
<tr>
<td>J2EE 1.4 compliance</td>
<td>Compliant</td>
<td>Compliant</td>
</tr>
<tr>
<td>CICS environment</td>
<td>Supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>IMS environment</td>
<td>Supported</td>
<td>Not supported</td>
</tr>
</tbody>
</table>

The following points can help you determine which type of connectivity to use.

Use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity under these circumstances:
- Your JDBC or SQLJ application runs locally most of the time.
  Local applications have better performance with type 2 connectivity.
- You are running a Java stored procedure.
  A stored procedure environment consists of two parts: a client program, from which you call a stored procedure, and a server program, which is the stored procedure. You can call a stored procedure in a JDBC or SQLJ program that uses type 2 or type 4 connectivity, but you must run a Java stored procedure using type 2 connectivity.
- Your application runs in the CICS environment or IMS environment.

Use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity under these circumstances:
- Your JDBC or SQLJ application runs remotely most of the time.
  Remote applications have better performance with type 4 connectivity.
- You are using IBM Data Server Driver for JDBC and SQLJ connection concentrator and Sysplex workload balancing support.

**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”.

Related concepts:
[JDBC interfaces for executing SQL” on page 29

Related tasks:
“Learning about a data source using DatabaseMetaData methods” on page 26
“Committing or rolling back JDBC transactions” on page 112
“Disconnecting from data sources in JDBC applications” on page 123

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.

About this task
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. You can use this implementation with IBM Data Server Driver for JDBC and SQLJ type 2 connectivity or IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.
• com.ibm.db2.jcc.DB2ConnectionPoolDataSource, which supports connection pooling. You can use this implementation with IBM Data Server Driver for JDBC and SQLJ type 2 connectivity or IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.
• 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. You can use this implementation only with IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Procedure
To 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.
Example

The following example shows how to perform these tasks.

```java
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();
Ctx.bind("jdbc/sampledb",dbds); 3
```

Figure 6. Example of creating and deploying a DataSource object

**Note**

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

**Related tasks:**

- “Connecting to a data source using the DataSource interface” on page 20

**Related reference:**

- “Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 234

---

## 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.

**Related concepts:**
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.

About this task

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

Procedure

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.

Examples

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.

Connection con;
DatabaseMetaData dbmtadta;
ResultSet rs;
int mdadtaint;
String procSchema;
String procName;
String dtfnList;
DatabaseMetaData methods for identifying the type of data server

Use the DatabaseMetaData.getDatabaseProductVersion method to identify the data server to which an application is connected. Use the DatabaseMetaData.getDatabaseProductName method to obtain additional information about the data server.

Important: DatabaseMetaData.getDatabaseProductVersion is the preferred method for determining the data server and version because the returned value is in the standard product identifier (PRDID) format.

The format for data servers other than Db2 on Linux, UNIX, and Windows systems is:

```
ppvvrrrm
```

The format for Db2 on Linux, UNIX, and Windows systems data servers is:

```
ppvvrrrmf
```

The parts of the product identifier are

- **ppp**: The 3-byte product code:
  - **ARI**: Db2 Server for VM and VSE
  - **DSN**: DB2 for z/OS
  - **QSO**: Db2 for IBM i
  - **SQL**: Db2 on Linux, UNIX, and Windows systems

- **vv**: The 2-byte version identifier.
- **rr**: The 2-byte release identifier.
- **m**: The 1-byte modification identifier.
- **f**: The 1-byte fix pack identifier. The fix pack identifier is included only for Db2 on Linux, UNIX, and Windows systems data servers.
The following table shows examples of values that are returned by `DatabaseMetaData.getDatabaseProductVersion`.

<table>
<thead>
<tr>
<th>Returned value</th>
<th>Database product version</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSN11015</td>
<td>DB2 for z/OS Version 11 in new-function mode</td>
</tr>
<tr>
<td>SQL11010</td>
<td>Db2 on Linux, UNIX, and Windows systems Version 11.1 GA</td>
</tr>
<tr>
<td>SQL110122</td>
<td>Db2 on Linux, UNIX, and Windows systems Version 11.1 mod pack 2 fix pack 2</td>
</tr>
<tr>
<td>IFX11000</td>
<td>IBM Informix Version 11.10</td>
</tr>
</tbody>
</table>

`DatabaseMetaData.getDatabaseProductName` is a method for obtaining additional information about the data server and its operating system. `DatabaseMetaData.getDatabaseProductName` returns a free-form string.

**Important:** Do not use `DatabaseMetaData.getDatabaseProductName` as the only method for determining the data server type. The contents of the returned string might change between data server versions and between driver versions.

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

<table>
<thead>
<tr>
<th>Returned value</th>
<th>Database product</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2</td>
<td>DB2 for z/OS</td>
</tr>
<tr>
<td>DB2 for z/OS VUE</td>
<td>DB2 for z/OS Value Unit Edition</td>
</tr>
<tr>
<td>DB2/LINUXX8664</td>
<td>DB2 on Linux on x86</td>
</tr>
<tr>
<td>IBM Informix/UNIX64</td>
<td>IBM Informix on UNIX</td>
</tr>
</tbody>
</table>

**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.

```java
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 the data server picks better access paths when the data types of your Java variables map closely to the data server data types.

**Related concepts:**

"Example of a simple JDBC application" on page 9
Comments in a JDBC application

To document your JDBC program, you need to include comments. You can use Java comments outside of JDBC methods and Java or SQL comments in SQL statement strings.

You can include Java comments outside JDBC methods, wherever the Java language permits them. Within an SQL statement string in a JDBC method call, you can use comments in the following places:

- For connections to Db2 on Linux, UNIX, and Windows systems data servers or Informix data servers, comments can be:
  - Anywhere in the SQL statement string, and enclosed in /* and */ pairs. /* and */ pairs can be nested.
  - At the end of the SQL statement string, and preceded by two hyphens (--).
- For connections to Informix data servers only, comments can be enclosed in left curly bracket ({) and right curly bracket (}) pairs.

**Restriction:** A comment that is enclosed in a { and } pair is not valid if either of the following conditions is true:

- The SQL statement string is not a stored procedure call, the SQL statement string is preceded and followed by comments that are enclosed in { and } pairs, and the comment at the beginning of the SQL statement string begins with the word call.
- The SQL statement string is a stored procedure call, and the comment {call} is at the beginning of the escape syntax for the stored procedure call.
- The comment contains any of the following characters:
  - Single quotation mark (’)
  - Double quotation mark ("")
  - Left curly bracket ({)
  - Right curly bracket (})
  - /*

- The comment can be interpreted as SQL escape syntax. Comments that begin with the following characters can be interpreted as SQL escape syntax:
  - d followed by a space
  - t followed by a space
  - ts followed by a space
  - escape followed by a space
  - oj followed by a space
  - fn followed by a space

For example, the following SQL statement strings are not valid:

```
"{call comment at beginning} select * from systables {ending comment}"
"{{{call} call mysp(? , ?)}}"
```

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.

Related tasks:

- Creating and modifying database objects using the Statement.executeUpdate method
- Retrieving data from tables using the Statement.executeQuery method” on page 39
- “Updating data in tables using the PreparedStatement.executeUpdate method” on page 31
- “Retrieving data from tables using the PreparedStatement.executeQuery method” on page 40

Related reference:

- “Driver support for JDBC APIs” on page 320

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.

About this task

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.

Procedure

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.

**Example**

Suppose that you want to execute this SQL statement:

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

The following code creates a `Statement` object, 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.

```java
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'"); // Perform the update 2
stmt.close(); // Close Statement object 3
```

*Figure 8. Using Statement.executeUpdate*

**Related reference:**

“Driver support for JDBC APIs” on page 320

### Updating data in tables using the `PreparedStatement.executeUpdate` method

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

**About this task**

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.

DB2 for z/OS does not support dynamic execution of the CALL statement. For calls to stored procedures that are on DB2 for z/OS data sources, the parameters can be parameter markers or literals, but not expressions. The following types of literals are supported:

- Integer
- Double
- Decimal
- Character
- Hexadecimal
- Graphic

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:

```sql
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:

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

Variables of this form are called parameter markers.

**Procedure**

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.

**Example**

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.

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

*Figure 9. 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 32. The following example demonstrates these steps.

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

Figure 10. Using PreparedStatement.executeUpdate for an SQL statement without parameter markers

Related tasks:
- "Using named parameter markers with PreparedStatement objects" on page 81

Related reference:
- "Driver support for JDBC APIs" on page 320

**JDBC `executeUpdate` methods against a DB2 for z/OS server**

The JDBC standard states that the `executeUpdate` method returns a row count or 0. However, if the `executeUpdate` method is executed against a DB2 for z/OS server, it can return a value of -1.

For `executeUpdate` statements against a DB2 for z/OS server, the value that is returned depends on the type of SQL statement that is being executed:

- For an SQL statement that can have an update count, such as an INSERT, UPDATE, DELETE, or MERGE statement, the returned value is the number of affected rows. It can be:
  - A positive number, if a positive number of rows are affected by the operation, and the operation is not a mass delete on a segmented table space.
  - 0, if no rows are affected by the operation.
  - -1, if the operation is a mass delete on a segmented table space.
- For an SQL CALL statement, a value of -1 is returned, because the data source cannot determine the number of affected rows. Calls to `getUpdateCount` or `getMoreResults` for a CALL statement also return -1.
- For any other SQL statement, a value of -1 is returned.

Related tasks:
- "Creating and modifying database objects using the `Statement.executeUpdate` method" on page 30

**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.

**About this task**

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.

Procedure

To make batch updates, follow one of the following sets of steps.
- 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. If the batched statement is a searched UPDATE, searched DELETE, or
     MERGE statement, set the autocommit mode for the connection to false.
  2. Invoke the prepareStatement method to create a PreparedStatement object.
  3. 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.
  4. Invoke the executeBatch method to execute the statements with all sets of
     parameters.
5. 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.
   
6. If errors occurred, process the BatchUpdateException.

Example

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.

```java
try {
    ... 
    PreparedStatement preps = conn.prepareStatement(
        "UPDATE DEPT SET MGRNO=? WHERE DEPTNO=?");  // 2
    ps.setString(1,mgrnum1);  // 3a
    ps.setString(2,deptnum1);  // 3b
    ps.addBatch();
    ps.setString(1,mgrnum2);
    ps.setString(2,deptnum2);
    ps.addBatch();
    int [] numUpdates=ps.executeBatch();  // 4
    for (int i=0; i < numUpdates.length; i++) {  // 5a
        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");
    }
```
In the following code fragment, a batched INSERT statement returns automatically generated keys.

```java
conn.commit();
}
```

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

```java
import java.sql.*;
import com.ibm.db2.jcc.*;
...
Connection conn;
...
try {
    PreparedStatement ps = conn.prepareStatement(  
        "INSERT INTO DEPT (DEPTNO, DEPTNAME, ADMRDEPT) " +  
        "VALUES (?, ?, ?)",  
        Statement.RETURN_GENERATED_KEYS);
    ps.setString(1, "X01");  
    ps.setString(2, "Finance");
    ps.setString(3, "A00");
    ps.addBatch();
    ps.setString(1, "Y01");
    ps.setString(2, "Accounting");
    ps.setString(3, "A00");
    ps.addBatch();

    int[] numUpdates = ps.executeBatch();
    for (int i = 0; i < numUpdates.length; i++) {
        if (numUpdates[i] == org.db2.jcc.BatchUpdateException.SUCCEED_NO_INFO) {
            System.out.println("Execution " + i + " successful: " + numUpdates[i] + " rows updated");
        } else {
            System.out.println("Execution " + i + " unknown number of rows updated");
        }
    }
    conn.commit();
    ResultSet[] resultList = ((DB2PreparedStatement)ps).getDBGeneratedKeys();
    if (resultList.length != 0) {
        for (int 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) {
    // process BatchUpdateException
}
```

In the following code fragment, a batched UPDATE statement returns automatically generated keys. The code names the DEPTNO column as an automatically generated key, updates two rows in the DEPT table in a batch, and retrieves the values of DEPTNO for the updated rows. The numbers to the right of selected statements correspond to the previously described steps.

```java
In the following code fragment, a batched UPDATE statement returns automatically generated keys. The code names the DEPTNO column as an automatically generated key, updates two rows in the DEPT table in a batch, and retrieves the values of DEPTNO for the updated rows. The numbers to the right of selected statements correspond to the previously described steps.
```
import java.sql.*;
import com.ibm.db2.jcc.*;
...
Connection conn;
...
String[] agkNames = \{"DEPTNO"\};
try {
...
conn.setAutoCommit(false);
PreparedStatement ps = conn.prepareStatement("UPDATE DEPT SET DEPTNAME=? " +
    "WHERE DEPTNO=?",agkNames);
ps.setString(1,"X01");
ps.setString(2,"Planning");
ps.addBatch();
ps.setString(1,"Y01");
ps.setString(2,"Bookkeeping");
ps.addBatch();
int[] numUpdates=ps.executeBatch();
for (int i=0; i < numUpdates.length; i++) {
    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();
ResultSet[] resultList = ((DB2PreparedStatement)ps).getDBGeneratedKeys();
if (resultList.length != 0) {
    for (i = 0; i < resultList.length; i++) {
        while (resultList[i].next()) {
            String deptNoKey = rs.getString(1);
            // Get automatically generated key
            // value
            System.out.println("Automatically generated key value = " + deptNoKey);
        }
    }
} else {
    System.out.println("Error retrieving automatically generated keys");
}
} catch(BatchUpdateException b) {
    // process BatchUpdateException
}

Related tasks:
"Making batch updates in SQLJ applications" on page 146
"Retrieving information from a BatchUpdateException" on page 120
"Making batch queries in JDBC applications" on page 41
"Committing or rolling back JDBC transactions" on page 112

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.
About this task

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.

Procedure

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.

Example

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.

```java
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();
   // Create a ParameterMetaData object
2 mtadtacnt = pmtadta.getParameterCount();
   // Determine the number of parameters
3 System.out.println("Number of statement parameters: " + mtadtacnt);
   for (int i = 1; i <= mtadtacnt; i++) {
5    sqlType = pmtadta.getParameterTypeName(i);
       // Get SQL type for each parameter
4     System.out.println("SQL type of parameter " + i + " is " + sqlType);
   }
...
pstmt.close();
   // Close the PreparedStatement
```

Figure 11. Using ParameterMetaData methods to get information about a PreparedStatement

Related reference:

"Driver support for JDBC APIs“ on page 320
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.

About this task

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.

Procedure

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.

Example

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
rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE"); // Get the result table from the query
while (rs.next()) {
    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
stmt.close(); // Close the Statement

Figure 12. Using Statement.executeQuery

Related reference:
"Driver support for JDBC APIs" on page 320

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.

About this task

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.

Procedure

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.

Restriction: For a PreparedStatement that contains an IN predicate, the expression that is the argument of the IN predicate cannot have more than 32767 parameters if the target data server is a Db2 on Linux, UNIX, and Windows systems system. Otherwise, the IBM Data Server Driver for JDBC and SQLJ throws an SQLException with error code -4499.
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.

Example

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.

```java
String empnum, phonenum;
Connection con;
PreparedStatement pstmt;
ResultSet rs;
...
pstmt = con.prepareStatement("SELECT EMPNO, PHONENO FROM EMPLOYEE WHERE EMPNO=?");
  // Create a PreparedStatement object
pstmt.setString(1,"000010");
  // Assign value to input parameter
rs = pstmt.executeQuery();
  // Get the result table from the query
while (rs.next()) {
  // Position the cursor
  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
}
r.close();
  // Close the ResultSet
pstmt.close();
  // Close the PreparedStatement
```

Figure 13. Example of using PreparedStatement.executeQuery

Related reference:

"Driver support for JDBC APIs" on page 320

Making batch queries in JDBC applications

The IBM Data Server Driver for JDBC and SQLJ provides a IBM Data Server Driver for JDBC and SQLJ-only DB2PreparedStatement interface that lets you perform batch queries on a homogeneous batch.

Procedure

To make batch queries using a single statement with several sets of input parameters, follow these basic steps:

1. Invoke the prepareStatement method to create a PreparedStatement object for the SQL statement with input parameters.

2. For each set of input parameter values:
   a. Execute PreparedStatement.setXXX methods to assign values to the input parameters.
   b. Invoke the PreparedStatement.addBatch method to add the set of input parameters to the batch.

3. Cast the PreparedStatement object to a DB2PreparedStatement object, and invoke the DB2PreparedStatement.executeQueryBatch method to execute the statement with all sets of parameters.
4. In a loop, retrieve the ResultSet objects:
   a. Retrieve each ResultSet object.
   b. Retrieve all the rows from each ResultSet object.

**Example**

In the following code fragment, two sets of parameters are batched. A SELECT statement that takes one input parameter is then executed twice, once with each parameter value. The numbers to the right of selected statements correspond to the previously described steps.

```java
java.sql.Connection con = java.sql.DriverManager.getConnection(url, properties);
java.sql.Statement s = con.createStatement();
// Clean up from previous executions
try {
    s.executeUpdate("drop table TestQBatch");
}
catch (Exception e) {

// Create and populate a test table
s.executeUpdate("create table TestQBatch (col1 int, col2 char(10))");
s.executeUpdate("insert into TestQBatch values (1, 'test1')");
s.executeUpdate("insert into TestQBatch values (2, 'test2')");
s.executeUpdate("insert into TestQBatch values (3, 'test3')");
s.executeUpdate("insert into TestQBatch values (4, 'test4')");
s.executeUpdate("insert into TestQBatch values (1, 'test5')");
s.executeUpdate("insert into TestQBatch values (2, 'test6')");

try {
    PreparedStatement pstmt =
        con.prepareStatement("Select * from TestQBatch where col1 = ?");
pstmt.setInt(1,1);
pstmt.addBatch();
    // Add some more values to the batch
    pstmt.setInt(1,2);
pstmt.addBatch();
pstmt.setInt(1,3);
pstmt.addBatch();
pstmt.setInt(1,4);
pstmt.addBatch();
((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).executeDB2QueryBatch();
}
catch (BatchUpdateException b) {
    // process BatchUpdateException
}
ResultSet rs;
while(pstmt.getMoreResults()) {
    rs = pstmt.getResultSet();
    while (rs.next()) {
        System.out.print (rs.getInt (1) + " ");
        System.out.println (rs.getString (2));
    }
    System.out.println();
    rs.close ();
}
// Clean up
s.close ();
pstmt.close ();
con.close ();

**Related tasks:**

"Making batch updates in JDBC applications" on page 33
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.

About this task

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

Procedure

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.getColumnCount call reflects the column name information that is stored in the catalog for that data server.

Example

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 rsmtdata;
int colCount;
int mtdataint;
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
rsmtdata = rs.getMetaData(); // Create a ResultSetMetaData object
colCount = rsmtdata.getColumnCount(); // Find number of columns in EMP
for (i=1; i<= colCount; i++) {
    colName = rsmtdata.getColumnName(); // Get column name
    colType = rsmtdata.getColumnTypeName(); // Get column data type
    System.out.println("Column = " + colName + " is data type " + colType); // Print the column value
}

Figure 14. Using ResultSetMetaData methods to get information about a ResultSet

Related tasks:
“Retrieving data from tables using the Statement.executeQuery method” on page 39
“Retrieving multiple result sets from a stored procedure in a JDBC application” on page 56
“Calling stored procedures in JDBC applications” on page 53

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>
<thead>
<tr>
<th>JDBC setting</th>
<th>DB2 cursor setting</th>
<th>IBM Informix cursor setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCUR_READ_ONLY</td>
<td>FOR READ ONLY</td>
<td>FOR READ ONLY</td>
</tr>
<tr>
<td>CONCUR_UPDATABLE</td>
<td>FOR UPDATE</td>
<td>FOR UPDATE</td>
</tr>
<tr>
<td>HOLD_CURSORS_OVER_COMMIT</td>
<td>WITH HOLD</td>
<td>WITH HOLD</td>
</tr>
<tr>
<td>TYPE_FORWARD_ONLY</td>
<td>SCROLL not specified</td>
<td>SCROLL not specified</td>
</tr>
<tr>
<td>TYPE_SCROLL_INSENSITIVE</td>
<td>INSENSITIVE SCROLL</td>
<td>SCROLL</td>
</tr>
<tr>
<td>TYPE_SCROLL_SENSITIVE</td>
<td>SENSITIVE STATIC, SENSITIVE DYNAMIC, or ASENSITIVE, depending on the cursorSensitivity Connection and DataSource property</td>
<td>Not supported</td>
</tr>
</tbody>
</table>

**Important:** Like static scrollable cursors in any other language, JDBC static scrollable ResultSet objects use declared temporary tables for their internal processing. This means that before you can execute any applications that contain JDBC static scrollable ResultSet objects, your database administrator needs to create a temporary database and temporary table spaces for those declared temporary tables.

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.

**Related information:**

**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.

**Before you begin**

If you plan to update ResultSet objects, ensure that the `enableExtendedDescribe` property is not set, or is set to `DB2BaseDataSource.YES` (2). Updates of ResultSet objects do not work correctly unless extended describe capability is enabled.

**About this task**

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. These steps change the scrollability, updatability, and holdability attributes for a ResultSet.

Procedure

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:

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

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

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

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

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

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

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

**Important:** In a prepareStatement method invocation in which the first parameter is a CALL statement, you cannot specify the scrollability, updatability, or holdability of result sets that are returned from a stored procedure. Those characteristics are determined by the stored procedure code, when it declares the cursors for the result sets that are returned. If you use the prepareStatement method to prepare a CALL statement, and the prepareStatement call includes the scrollability, updatability, or holdability parameters, the IBM Data Server Driver for JDBC and SQLJ does not use those parameter values. A prepareStatement method with scrollability, updatability, or holdability parameters applies only to preparation of SQL statements other than the CALL statement.

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

<table>
<thead>
<tr>
<th>resultSetType Value</th>
<th>resultSetConcurrency Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE_FORWARD_ONLY</td>
<td>CONCUR_READ_ONLY</td>
</tr>
<tr>
<td>TYPE_FORWARD_ONLY</td>
<td>CONCUR_UPDATABLE</td>
</tr>
<tr>
<td>TYPE_SCROLL_INSENSITIVE</td>
<td>CONCUR_READ_ONLY</td>
</tr>
<tr>
<td>TYPE_SCROLL_SENSITIVE</td>
<td>CONCUR_READ_ONLY</td>
</tr>
<tr>
<td>TYPE_SCROLL_SENSITIVE</td>
<td>CONCUR_READ_ONLY</td>
</tr>
<tr>
<td></td>
<td>CONCUR_UPDATABLE</td>
</tr>
</tbody>
</table>

**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 resultSetConcurr
ency 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 on Linux, UNIX, and Windows systems server, the SELECT list of the SELECT statement that defines the ResultSet cannot include columns with the following data types:

- LONG VARCHAR
- LONG VARCHAR
- 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.

   **Restriction:** If resultSetType is TYPE_FORWARD_ONLY, only ResultSet.next is valid.

<table>
<thead>
<tr>
<th>Method</th>
<th>Positions the cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>On the first row of the ResultSet</td>
</tr>
<tr>
<td>last</td>
<td>On the last row of the ResultSet</td>
</tr>
<tr>
<td>next</td>
<td>On the next row of the ResultSet</td>
</tr>
<tr>
<td>previous</td>
<td>On the previous row of the ResultSet</td>
</tr>
<tr>
<td>absolute(int n)</td>
<td>If n&gt;0, on row n of the ResultSet. If n&lt;0, and m is the number of rows in the ResultSet, on row m+n+1 of the ResultSet.</td>
</tr>
<tr>
<td>relative(int n)</td>
<td>If n&gt;0, on the row that is n rows after the current row. If n&lt;0, on the row that is n rows before the current row. If n=0, on the current row.</td>
</tr>
<tr>
<td>afterLast</td>
<td>After the last row in the ResultSet</td>
</tr>
<tr>
<td>beforeFirst</td>
<td>Before the first row in the ResultSet</td>
</tr>
</tbody>
</table>

Chapter 3. JDBC application programming 47
Table 9. ResultSet methods for positioning a scrollable cursor (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Positions the cursor</th>
</tr>
</thead>
</table>

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 \( x+n \geq m \), 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 46 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.

Example

The following code demonstrates how to retrieve all rows from the employee table in reverse order, and update the phone number for employee number "000010". The numbers to the right of selected statements correspond to the previously-described steps.
String s;
String stmtsrc;
Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                            ResultSet.CONCUR_UPDATABLE);
        // Create a Statement object
        // for a scrollable, updatable
        // ResultSet
stmtsrc = "SELECT EMPNO, PHONENO FROM EMPLOYEE " +
                    "FOR UPDATE OF PHONENO";
rs = stmt.executeQuery(stmtsrc);  // Create the ResultSet
        // Position the cursor at the end of
        // the ResultSet
rs.afterLast();
        // Position the cursor backward
while (rs.previous()) {
        // Retrieve the employee number
        // (column 1 in the result
        // table)
        s = rs.getString("EMPNO");
        // Print the column value
        if (s.compareTo("000010") == 0) {
                        // Look for employee 000010
                rs.updateString("PHONENO","4657");
                        // Update their phone number
                rs.updateRow();
                        // Update the row
        }
}
rs.close();  // Close the ResultSet
stmt.close();  // Close the Statement

Figure 15. Using a scrollable cursor

Related tasks:
"Retrieving data from tables using the Statement.executeQuery method" on page 39

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 setting of the enableRowsetSupport property
- The setting of the useRowsetCursor property, for connections to DB2 for z/OS
- 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 true.
  - 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.

**Testing whether the current row of a ResultSet is a delete hole or update hole in a JDBC application:**

If a ResultSet has the TYPE_SCROLL_SENSITIVE attribute, and the underlying cursor is SENSITIVE STATIC, you need to test for delete holes or update holes before you attempt to retrieve rows of the ResultSet.

**About this task**

After a SENSITIVE STATIC ResultSet is opened, it does not change size. This means that deleted rows are replaced by placeholders, which are also called holes. If updated rows no longer fit the criteria for the ResultSet, those rows also become holes. You cannot retrieve rows that are holes.

**Procedure**

To test whether the current row in a ResultSet is a delete hole or update hole, follow these steps:

1. Call the DatabaseMetaData.deletesAreDetected or DatabaseMetaData.updatesAreDetected method with the TYPE_SCROLL SENSITIVE argument to determine whether the data source creates holes for a TYPE_SCROLL SENSITIVE ResultSet.
2. If DatabaseMetaData.deletesAreDetected or DatabaseMetaData.updatesAreDetected returns true, which means that the data source can create holes, call the ResultSet.rowDeleted or ResultSet.rowUpdated
method to determine whether the current row is a delete or update hole. If the method returns true, the current row is a hole.

Example

The following code tests whether the current row is a delete hole.

```java
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                        ResultSet.CONCUR_UPDATABLE); // Create a Statement object
                                        // for a scrollable, updatable
                                        // ResultSet
ResultSet rs =
    stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE FOR UPDATE OF PHONENO"); // Create the ResultSet
DatabaseMetaData dbmd = con.getMetaData(); // Create the DatabaseMetaData object
boolean dbSeesDeletes =
    dbmd.deletesAreDetected(ResultSet.TYPESCROLL_SENSITIVE); // Can the database see delete holes?
rs.afterLast(); // Position the cursor at the end of // the ResultSet
while (rs.previous()) { // Position the cursor backward
    if (dbSeesDeletes) { // If delete holes can be detected
        if (!rs.rowDeleted()) { // If this row is not a delete hole
            s = rs.getString("EMPNO"); // Retrieve the employee number
            System.out.println("Employee number = "+s); // Print the column value
        }
    }
}
rs.close(); // Close the ResultSet
stmt.close(); // Close the Statement
```

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.

Before you begin

Ensure that the enableExtendedDescribe property is not set, or is set to
DB2BaseDataSource.YES (2). Insertion of a row into a ResultSet does not work unless extended describe capability is enabled.

Procedure

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 an 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.

```java
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");
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.

Procedure

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 or PreparedStatement class.
**Procedure**

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.

   **Restriction:** The parameter types that are permitted depend on whether the data source supports dynamic execution of the `CALL` statement. DB2 for z/OS does not support dynamic execution of the `CALL` statement. For a call to a stored procedure that is on a DB2 for z/OS database server, the parameters can be parameter markers or literals, but not expressions. Even if all parameters are literals, you cannot use `Statement` methods to execute `CALL` statements. You must use `PreparedStatement` methods or `CallableStatement` methods. The following table lists the types of literals that are supported, and the JDBC types to which they map.

<table>
<thead>
<tr>
<th>Literal parameter type</th>
<th>JDBC type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td><code>java.sql.Types.INTEGER</code></td>
<td>-122, 40022, +27</td>
</tr>
<tr>
<td>Floating-point decimal</td>
<td><code>java.sql.Types.DOUBLE</code></td>
<td>23E12, 40022E-4, +2723E+15, 1E+23, 0E0</td>
</tr>
<tr>
<td>Fixed-point decimal</td>
<td><code>java.sql.Types.DECIMAL</code></td>
<td>-23.12, 40022.4295, 0.0, +2723.23, 10000000000</td>
</tr>
<tr>
<td>Character</td>
<td><code>java.sql.Types.VARCHAR</code></td>
<td>'Grantham Lutz', 'O''Conner', 'ABcde?z?'</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td><code>java.sql.Types.VARBINARY</code></td>
<td>X'C1C30427', X'00CF18E0'</td>
</tr>
<tr>
<td>Unicode string</td>
<td><code>java.sql.Types.VARCHAR</code></td>
<td>UX'0041', UX'0054006500730074'</td>
</tr>
</tbody>
</table>

   **Important:** In a `prepareCall` method invocation, you cannot specify the scrollability, updatability, or holdability of result sets that are returned from a stored procedure. Those characteristics are determined by the stored procedure code, when it declares the cursors for the result sets that are returned. If you specify any of the forms of `prepareCall` that include scrollability, updatability, or holdability parameters, the IBM Data Server Driver for JDBC and SQLJ does not use those parameter values. A `prepareCall` method with scrollability, updatability, or holdability parameters applies only to preparation of SQL statements other than the `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.
**Restriction:** Invoking `CallableStatement.setXXX` methods to pass values to the OUT parameters is not supported. There is no need to set values for the OUT parameters of a stored procedure because the stored procedure does not use those values.

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 ResultSet. The numbers to the right of selected statements correspond to the previously-described steps.

```java
int ifcaret;
int ifcareas;
int xsbytes;
String errbuff;
Connection con;
CallableStatement cstmt;
ResultSet rs;
...

cstmt = con.prepareCall("CALL DSN8.DSNBED2(?,?,?,?,?)");  // 1
    // Create a CallableStatement object

cstmt.setString (1, "DISPLAY THREAD(*)");  // 2
```
Related tasks:

“Using named parameter markers with CallableStatement objects” on page 82

Related reference:

“Driver support for JDBC APIs” on page 320

Retrieving multiple result sets from a stored procedure in a JDBC application

If you call a stored procedure that returns result sets, you need to include code to retrieve the result sets.

About this task

The steps that you take depend on whether you know how many result sets are returned, and whether you know the contents of those result sets.

Related tasks:

“Retrieving data from tables using the Statement.executeQuery method” on page 39

“Retrieving data from tables using the PreparedStatement.executeQuery method” on page 40

“Calling stored procedures in JDBC applications” on page 53

Retrieving a known number of result sets from a stored procedure in a JDBC application:

Retrieving a known number of result sets from a stored procedure is a simpler procedure than retrieving an unknown number of result sets.

Procedure

To retrieve result sets when you know the number of result sets and their contents, follow these steps:

1. Invoke the Statement.execute method, the PreparedStatement.execute method, or the CallableStatement.execute method to call the stored procedure.
   Use PreparedStatement.execute if the stored procedure has input parameters.
2. Invoke the getResultSet method to obtain the first result set, which is 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.
4. If there are n result sets, repeat the following steps n-1 times:
   a. Invoke the getMoreResults method to close the current result set and point to the next result set.
b. Invoke the getResultSet method to obtain the next result set, which is in a ResultSet object.

c. 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.

Example

The following code illustrates retrieving two result sets. The first result set contains an INTEGER column, and the second result set contains a CHAR column. The numbers to the right of selected statements correspond to the previously described steps.

```java
CallableStatement cstmt;
ResultSet rs;
int i;
String s;
...
cstmt.execute(); // Call the stored procedure
rs = cstmt.getResultSet(); // Get the first result set
while (rs.next()) {
    i = rs.getInt(1); // Position the cursor
    System.out.println("Value from first result set = " + i);
    // Print the value
}
cstmt.getMoreResults(); // Point to the second result set
rs = cstmt.getResultSet(); // Get the second result set
while (rs.next()) {
    s = rs.getString(1); // Position the cursor
    System.out.println("Value from second result set = " + s);
    // Print the value
}
rs.close(); // Close the result set
cstmt.close(); // Close the statement
```

Retrieving an unknown number of result sets from a stored procedure in a JDBC application:

Retrieving an unknown number of result sets from a stored procedure is a more complicated procedure than retrieving a known number of result sets.

About this task

To retrieve result sets when you do not know the number of result sets or their contents, you need to retrieve ResultSets, until no more ResultSets are returned. For each ResultSet, use ResultSetMetaData methods to determine its contents.

After you call a stored procedure, follow these basic steps to retrieve the contents of an unknown number of result sets.

Procedure

1. Check the value that was returned from the execute statement that called the stored procedure.
   
   If the returned value is true, there is at least one result set, so you need to go to the next step.

2. Repeat the following steps in a loop:
   
   a. Invoke the getResultSet method to obtain a result set, which is in a ResultSet object. Invoking this method closes the previous result set.
b. Use ResultSetMetaData methods to determine the contents of the ResultSet, and retrieve data from the ResultSet.

c. Invoke the getMoreResults method to determine whether there is another result set. If getMoreResults returns true, go to step 1 on page 57 to get the next result set.

Example

The following code illustrates retrieving result sets when you do not know the number of result sets or their contents. The numbers to the right of selected statements correspond to the previously described steps.

```java
CallableStatement cstmt;
ResultSet rs;
...
boolean resultsAvailable = cstmt.execute(); // Call the stored procedure
while (resultsAvailable) {
    // Test for result sets
    ResultSet rs = cstmt.getResultSet(); // Get a result set
    ...
    // Process the ResultSet
    // as you would process
    // a ResultSet from a table
    resultsAvailable = cstmt.getMoreResults(); // Check for next result set
    // (Also closes the
    // previous result set)
}
```

Related tasks:

- “Learning about a ResultSet using ResultSetMetaData methods” on page 43

Keeping result sets open when retrieving multiple result sets from a stored procedure in a JDBC application:

The getMoreResults method has a form that lets you leave the current ResultSet open when you open the next ResultSet.

Procedure

To specify whether result sets stay open, follow this process:

When you call getMoreResults to check for the next ResultSet, use this form:

```java
CallableStatement.getMoreResults(int current);
```

- To keep the current ResultSet open when you check for the next ResultSet, specify a value of Statement.KEEP_CURRENT_RESULT for current.
- To close the current ResultSet when you check for the next ResultSet, specify a value of Statement.CLOSE_CURRENT_RESULT for current.
- To close all ResultSet objects, specify a value of Statement.CLOSE_ALL_RESULTS for current.

Example

The following code keeps all ResultSets open until the final ResultSet has been retrieved, and then closes all ResultSets.

```java
CallableStatement cstmt;
...
boolean resultsAvailable = cstmt.execute(); // Call the stored procedure
if (resultsAvailable==true) {
    // Test for result set
    ResultSet rs1 = cstmt.getResultSet(); // Get a result set
    ...
    resultsAvailable = cstmt.getMoreResults(Statement.KEEP_CURRENT_RESULT);
```
// Check for next result set
// but do not close
// previous result set
if (resultsAvailable==true) {
    // Test for another result set
    ResultSet rs2 = cstmt.getResultSet();
    // Get next result set
    // Process either ResultSet
}
}
resultsAvailable = cstmt.getMoreResults(Statement.CLOSE_ALL_RESULTS);
// Close the result sets

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.

Related reference:
“Driver support for JDBC APIs” on page 320
“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 234

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 on Linux, UNIX, and Windows systems Version 9.5 and later, IBM Informix Version 11.50 and later, and Db2 for IBM 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:

<table>
<thead>
<tr>
<th>Minimum IBM Data Server Driver for JDBC and SQLJ version</th>
<th>Minimum data server version</th>
<th>Types of objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.53</td>
<td>Db2 for IBM i V6R1</td>
<td>LOB, XML</td>
</tr>
<tr>
<td>3.50</td>
<td>Db2 on Linux, UNIX, and Windows systems Version 9.5</td>
<td>LOB</td>
</tr>
<tr>
<td>3.50</td>
<td>IBM Informix Version 11.50</td>
<td>LOB</td>
</tr>
<tr>
<td>3.2</td>
<td>DB2 for z/OS Version 9</td>
<td>LOB, XML</td>
</tr>
</tbody>
</table>

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

For DB2 for z/OS Version 9.1 and later data sources, or Db2 on Linux, UNIX, and Windows systems Version 9.5 and later data sources, you can set the progressive streaming behavior for existing connections with the DB2Connection.setDBProgressiveStreaming(DB2BaseDataSource.YES) method. If
you call `DB2Connection.setDBProgressiveStreaming(DB2BaseDataSource.YES)`, all ResultSet objects that are created on the connection use progressive streaming behavior.

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.

A LOB object is also materialized when an application program performs any of the following actions, regardless of the progressiveStreaming or streamBufferSize settings:

- Invokes a user-defined function with a LOB value as an argument
- Calls a stored procedure with a LOB value as an input or output argument
- Assigns a LOB host variable to a LOB locator host variable

**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:

```java
... 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:
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.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS, when the JDBC driver processes a CallableStatement.setXXX call for a stored procedure input parameter, or a CallableStatement.registerOutParameter call for a stored procedure output parameter, the driver cannot determine the parameter data types.

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

• Use a CallableStatement.setObject call that specifies that the target data type is BLOB:
  ```java
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
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:
  ```java
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:
  ```java
  cstmt.setClob(parmIndex, clobData);
  ```
- Use a CallableStatement.setObject call that specifies that the target data type is CLOB:
  ```java
  String charData = "CharacterString";
  cstmt.setObject(parmIndex, 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
    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:
    ```java
    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:
  ```java
  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:
  ```java
  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:
  ```java
  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:
  ```java
  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.

**Related concepts:**

"LOBs in JDBC applications with the IBM Data Server Driver for JDBC and SQLJ" on page 59

**Related reference:**

"Data types that map to database data types in Java applications” on page 219

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

DB2 for z/OS and Db2 for IBM 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)
- `getBytes`
- `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>
<thead>
<tr>
<th>Database server</th>
<th>DatabaseMetaData.getRowIdLifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2 for z/OS</td>
<td>ROWID_VALID_TRANSACTION</td>
</tr>
<tr>
<td>Db2 on Linux, UNIX, and Windows systems</td>
<td>ROWID_UNSUPPORTED</td>
</tr>
<tr>
<td>Db2 for IBM i</td>
<td>ROWID_VALID_FOREVER</td>
</tr>
<tr>
<td>IBM Informix</td>
<td>ROWID_VALID_FOREVER</td>
</tr>
</tbody>
</table>

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

```java
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 rid, use this form of the ResultSet.getRowId method:

```java
java.sql.RowId rid = 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:

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

Related reference:

“Data types that map to database data types in Java applications” on page 219

**Update and retrieval of timestamps with time zone information in JDBC applications**

The JDBC methods and data types that you use and the information that the IBM Data Server Driver for JDBC and SQLJ has about the column data types determine the timestamp values that are sent to and received from TIMESTAMP WITH TIME ZONE or TIMESTAMP columns.

**Updates of values in TIMESTAMP or TIMESTAMP WITH TIME ZONE columns**

You can use the following standard JDBC methods to update a TIMESTAMP WITH TIME ZONE or TIMESTAMP column:

- PreparedStatement.setObject
- PreparedStatement.setTimestamp
- PreparedStatement.setString

For a PreparedStatement.setTimestamp call in which the second parameter is a DBTimestamp object and the third parameter is a Calendar object, the value that is
passed to a TIMESTAMP WITH TIME ZONE or TIMESTAMP column contains the
time zone value in the Calendar parameter, and not the time zone value in the
DBTimestamp object. For a PreparedStatement.setTimestamp in which the second
parameter is a DBTimestamp object and there is no Calendar parameter, the IBM
Data Server Driver for JDBC and SQLJ value that is passed to a TIMESTAMP
WITH TIME ZONE or TIMESTAMP column has the default time zone, which is
that of the Java virtual machine in which the application is running.

If you want the value that is passed to a TIMESTAMP WITH TIME ZONE or
TIMESTAMP column to use the time zone that is in the
DBTimestamp object, you
need to use PreparedStatement.setObject.

Example: Suppose that table TSTABLE is defined like this:
CREATE TABLE TSTABLE (TSCOL TIMESTAMP,
TSTZCOL TIMESTAMP WITH TIME ZONE)

Also suppose that the default time zone of the Java Virtual Machine (JVM) is
UTC-08:00 (Pacific Standard Time). The following code assigns timestamp values to
the column.

```java
java.util.TimeZone esttz = java.util.TimeZone.getTimeZone("EST");
java.util.Calendar estcal = java.util.Calendar.getInstance(esttz);
// Construct a Calendar object with the
// UTC-05:00 (Eastern Standard Time) time zone.
java.util.Calendar defcal = java.util.Calendar.getInstance();
// Construct a Calendar object
// with the default time zone.
java.sql.Timestamp ts =
java.sql.Timestamp.valueOf("2010-10-27 21:22:33.123456");
// Assign a timestamp to a Timestamp object.
DBTimestamp dbts = new DBTimestamp(ts, estcal);
// Construct a DBTimestamp object that
// has the UTC-05:00 time zone.
...
PreparedStatement ps = con.prepareStatement(
"INSERT INTO TSTABLE (TSCOL,TSTZCOL) VALUES (?,?)");
//
// Use setTimestamp methods to assign a timestamp value to a
// TIMESTAMP WITH TIME ZONE or TIMESTAMP column
//
ps.setTimestamp(1, ts); // Assign a timestamp value in a Timestamp
// object to a TIMESTAMP column.
ps.setTimestamp(2, ts); // Assign the same timestamp value to
// a TIMESTAMP WITH TIME ZONE column.
ps.execute(); // 2010-10-27 21.22.33.123456 is assigned to TSCOL
// if the driver has information that the column
// has the TIMESTAMP data type.
// 2010-10-27-21.22.33.123456-08:00 is assigned to TSCOL
// if the driver has no information about the column
// data type.
// 2010-10-27-21.22.33.123456-08:00 is assigned
// to TSTZCOL regardless of whether the driver
// has information that the the column has
// the TIMESTAMP WITH TIME ZONE data type.
ps.setTimestamp(1, dbts);
// Assign a timestamp value in a DBTimestamp
// object to a TIMESTAMP column.
ps.setTimestamp(2, dbts);
// Assign the same timestamp value to
// a TIMESTAMP WITH TIME ZONE column.
ps.execute(); // 2010-10-27-21.22.33.123456 is assigned to TSCOL
// if the driver has information that the column
```
ps.setTimestamp(1, ts, estcal);
  // Assign a timestamp value in a Timestamp
  // object to a TIMESTAMP column. Include
  // a Calendar parameter that specifies
  // the UTC-05:00 time zone.
ps.setTimestamp(2, ts, estcal);
  // Assign the same timestamp value to
  // a TIMESTAMP WITH TIME ZONE column. Include
  // a Calendar parameter that specifies the
  // UTC-05:00 time zone.
ps.execute();  // 2010-10-28-00.22.33.123456 is assigned to TSCOL
  // if the driver has information that the column
  // has the TIMESTAMP data type.
  // The value is adjusted for the difference
  // between the time zone in the Calendar parameter and
  // the default time zone.
  // 2010-10-28-00.22.33.123456-05:00 is assigned to TSCOL
  // if the driver has no information about the column
  // data type. The value is adjusted for the difference
  // between the time zone in the Calendar parameter and
  // the default time zone.
  // 2010-10-28-00.22.33.123456-05:00 is assigned
  // to TSTZCOL regardless of whether the driver
  // has information that the the column has
  // the TIMESTAMP WITH TIME ZONE data type. The
  // default time zone of UTC-08:00 is sent to
  // the column.
ps.setTimestamp(1, dbts, estcal);
  // Assign a timestamp value in a DBTimestamp
  // object to a TIMESTAMP column. Include
  // a Calendar parameter that specifies the
  // UTC-05:00 time zone.
ps.setTimestamp(2, dbts, estcal);
  // Assign the same timestamp value to
  // a TIMESTAMP WITH TIME ZONE column.
ps.execute();  // 2010-10-28-00.22.33.123456 is assigned to TSCOL
  // if the driver has information that the column
  // has the TIMESTAMP data type.
  // The value is adjusted for the difference
  // between the time zone in the Calendar parameter and
  // the default time zone.
  // 2010-10-28-00.22.33.123456-05:00 is assigned to TSCOL
  // if the driver has no information about the column
  // data type. The value is adjusted for the difference
  // between the time zone in the Calendar parameter and
  // the default time zone.
ps.setTimestamp(1, ts, defcal);
    // Assign a timestamp value in a Timestamp
    // object to a TIMESTAMP column. Include
    // a Calendar parameter that specifies
    // the default time zone (UTC-08:00).

ps.setTimestamp(2, ts, defcal);
    // Assign the same timestamp value to
    // a TIMESTAMP WITH TIME ZONE column. Include
    // a Calendar parameter that specifies the
    // default (UTC-08:00) time zone.

ps.execute(); // 2010-10-27-21.22.33.123456 is assigned to TSCOL
    // if the driver has information that the column
    // has the TIMESTAMP data type.
    // 2010-10-27-21.22.33.123456-08:00 is assigned to TSCOL
    // if the driver has no information about the column
    // data type.
    // 2010-10-27-21.22.33.123456-08:00 is assigned
    // to TSTZCOL regardless of whether the driver
    // has information that the the column has
    // the TIMESTAMP WITH TIME ZONE data type.

ps.setTimestamp(1, dbts, defcal);
    // Assign a timestamp value in a DBTimestamp
    // object to a TIMESTAMP column

ps.setTimestamp(2, dbts, defcal);
    // Assign the same timestamp value to
    // a TIMESTAMP WITH TIME ZONE column

ps.execute(); // 2010-10-27-21.22.33.123456 is assigned to TSCOL
    // if the driver has information that the column
    // has the TIMESTAMP data type.
    // 2010-10-27-21.22.33.123456-08:00 is assigned to TSCOL
    // if the driver has no information about the column
    // data type.
    // 2010-10-27-21.22.33.123456-08:00 is assigned
    // to TSTZCOL regardless of whether the driver
    // has information that the the column has
    // the TIMESTAMP WITH TIME ZONE data type. The
    // default time zone in the Calendar parameter,
    // UTC-08:00, is sent to the column.

    // Use setObject methods to assign a timestamp value to a
    // TIMESTAMP WITH TIME ZONE or TIMESTAMP column

ps.setObject(1, ts); // Assign a timestamp value in a Timestamp
    // object to a TIMESTAMP column.

ps.setObject(2, ts);
    // Assign the same timestamp value to
    // a TIMESTAMP WITH TIME ZONE column.

ps.execute(); // 2010-10-27-21.22.33.123456 is assigned to TSCOL
    // if the driver has information that the column
    // has the TIMESTAMP data type.
    // 2010-10-27-21.22.33.123456-08:00 is assigned to TSCOL
    // if the driver has no information about the column
    // data type. The time zone is the default time zone.
    // 2010-10-27-21.22.33.123456-08:00 is assigned
    // to TSTZCOL regardless of whether the driver
    // has information that the the column has
    // the TIMESTAMP WITH TIME ZONE data type. The
    // time zone is the default time zone.

ps.setObject(1, dbts);
    // Assign a timestamp value in a DBTimestamp
    // object to a TIMESTAMP column.

ps.setObject(2, dbts);
    // Assign the same timestamp value to
    // a TIMESTAMP WITH TIME ZONE column.

ps.execute(); // 2010-10-28-00.22.33.123456 is assigned to TSCOL
    // if the driver has information that the column
    // has the TIMESTAMP data type.
Alternatively, if you want to assign data that has a time zone or has a precision of greater than nine to a TIMESTAMP WITH TIME ZONE column, you can construct a DBTimestamp object, and use the IBM Data Server Driver for JDBC and SQLJ-only method DB2PreparedStatement.setDBTimestamp to update a TIMESTAMP WITH TIME ZONE column.

Example: Suppose that table TSTABLE is defined like this:

```java
CREATE TABLE TSTABLE (  TSCOL TIMESTAMP,  TSTZCOL TIMESTAMP(12) WITH TIME ZONE)
```

The following code assigns a timestamp value with a time zone and a precision of 10 to each column.

```java
DBTimestamp tstz = DBTimestamp.valueOfDBString("2010-10-28-00.22.33.1234567890-05:00");  // Create a DBTimestamp object from the input value PreparedStatement ps = con.prepareStatement(
```
"INSERT INTO TSTABLE (TSCOL, TSTXCOL) VALUES (?,?)");

DB2PreparedStatement dbps = (DB2PreparedStatement)ps;
dbps.setDBTimestamp(1, tstz);
dbps.setDBTimestamp(2, tstz);
dbps.execute(); // 2010-10-28-00.22.33.123456 is assigned to TSCOL if // the driver has information that the column data type is // TIMESTAMP. // 2010-10-28-00.22.33.1234567890-05:00 is assigned to TSCOL // if the driver has no information about the column // data type. // 2010-10-28-00.22.33.1234567890-05:00 is assigned to TSTZCOL // regardless of whether the driver has information that // the column data type is TIMESTAMP(12) WITH TIME ZONE.

Retrieval of values from TIMESTAMP or TIMESTAMP WITH TIME ZONE columns

You can use the following standard JDBC methods to retrieve data from a TIMESTAMP WITH TIME ZONE or TIMESTAMP column or output parameter:

- ResultSet.getTimestamp
- CallableStatement.getTimestamp
- ResultSet.getObject
- CallableStatement.getObject
- ResultSet.getString
- CallableStatement.getString

For a ResultSet.getTimestamp, CallableStatement.getTimestamp, ResultSet.getObject, or CallableStatement.getObject call, you can specify the type of object that you want the IBM Data Server Driver for JDBC and SQLJ to return by setting the DB2BaseDataSource.timestampOutputType property:

- If you set the property to DB2BaseDataSource.JDBC_TIMESTAMP (1), the driver returns a java.sql.Timestamp object.
- If you set the property to DB2BaseDataSource.JCC_DBTIMESTAMP (2), the driver returns a com.ibm.db2.jcc.DBTimestamp object.

For a ResultSet.getTimestamp or CallableStatement.getTimestamp call, if the ResultSet.getTimestamp or CallableStatement.getTimestamp call has a Calendar parameter with a non-null value, the IBM Data Server Driver for JDBC and SQLJ uses the Calendar object when it constructs the returned object. If the ResultSet.getTimestamp or CallableStatement.getTimestamp call has no Calendar parameter, or the Calendar parameter value is null, the IBM Data Server Driver for JDBC and SQLJ uses the default time zone when it constructs the returned object.

If you want to retrieve a timestamp with the time zone value that is in a TIMESTAMP WITH TIME ZONE column, call ResultSet.getObject or CallableStatement.getObject, and then call DBTimestamp.toDBString(true) to retrieve the timestamp with the time zone.

getString retrieves the timestamp value in the standard JDBC format: without the time zone, and with a precision of up to nine. The returned value is adjusted for the difference between the time zone of the column value and the default time zone.

Example: Suppose that table TSTABLE is defined like this:

```
CREATE TABLE TSTABLE (
    TSCOL TIMESTAMP,
    TSTZCOL TIMESTAMP WITH TIME ZONE)
```
Also suppose that the default time zone is UTC-08:00 (Pacific Standard Time). The following code retrieves timestamp values from the TIMESTAMP column.

```java
java.util.TimeZone esttz = java.util.TimeZone.getTimeZone("EST");
java.util.Calendar estcal = java.util.Calendar.getInstance(esttz);
java.util.Calendar defcal = java.util.Calendar.getInstance();
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT TSCOL, TSTZCOL FROM TSTABLE");
com.ibm.db2.jcc.DB2ResultSet dbrs = (com.ibm.db2.jcc.DB2ResultSet)rs;
Timestamp ts;
DBTimestamp dbts;
...
rs.next();
// Suppose that the TSCOL column value is 2010-10-27-21.22.33.123456

    ts=rs.getTimestamp(1);  // Retrieve the TIMESTAMP column value
    ts.toString();          // Format the Timestamp object as a String.
    ((DBTimestamp)ts).toDBString(false); // Cast the retrieved object to a
                                        // DBTimestamp object, and format the
                                        // value as a String, without the time
                                        // zone information.
                                        // 2010-10-27-21.22.33.123456 is returned.
    ((DBTimestamp)ts).toDBString(true); // Cast the retrieved object to a
                                         // DBTimestamp object, and format the value
                                         // as a String, with the time zone
                                         // information.
                                         // 2009-02-27-21.22.33.123456-08:00 is returned. The time zone is the default
                                         // time zone.

    ts=rs.getTimestamp(1,estcal); // Retrieve the TIMESTAMP column value
                                  // into a Timestamp object. Specify a
                                  // calendar parameter that says that the
                                  // time zone is UTC-05:00.
    ts.toString();                // Format the value as a String, using the
                                  // default time zone of UTC-08:00.
    ((DBTimestamp)ts).toDBString(false); // Cast the retrieved object to a
                                          // DBTimestamp object, and format the value
                                          // as a String, without the time zone
                                          // information.
                                          // 2010-10-27-21.22.33.123456 is returned.
    ((DBTimestamp)ts).toDBString(true); // Cast the retrieved object to a
                                         // DBTimestamp object, and format the value
                                         // as a String, with the time zone
                                         // information.
                                         // 2010-10-27-21.22.33.123456-05:00 is returned. The time zone is the time zone
                                         // in the Calendar parameter.

    ts=rs.getObject(1);          // Retrieve the TIMESTAMP column value
                               // into an Object.
    ts.toString();              // Format the value as a String.
    ((DBTimestamp)ts).toDBString(false); // Cast the retrieved object to a
                                         // DBTimestamp object, and format the value
                                         // as a String, without the time zone
                                         // information.
                                         // 2010-10-27-21.22.33.123456 is returned.
    ((DBTimestamp)ts).toDBString(true); // Cast the retrieved object to a
                                         // DBTimestamp object, and format the value
                                         // as a String, with the time zone
                                         // information.
                                         // 2010-10-27-21.22.33.123456-08:00 is returned.
```
Alternatively, you can use DB2ResultSet methods to retrieve the TIMESTAMP or TIMESTAMP WITH TIME ZONE column values.

Example: Suppose that table TSTABLE is defined like this:

```java
CREATE TABLE TSTABLE (
TSCOL TIMESTAMP,
TSTZCOL TIMESTAMP(12) WITH TIME ZONE)
```

Also suppose that the default time zone is UTC-08:00 (Pacific Standard Time). The following code retrieves timestamp values from the TIMESTAMP and TIMESTAMP WITH TIME ZONE columns.

```java
java.util.TimeZone esttz = java.util.TimeZone.getTimeZone("EST");
java.util.Calendar estcal = java.util.Calendar.getInstance(esttz);
java.util.Calendar defcal = java.util.Calendar.getInstance();
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT TSCOL, TSTZCOL FROM TSTABLE");
com.ibm.db2.jcc.DB2ResultSet dbrs = (com.ibm.db2.jcc.DB2ResultSet)rs;
Timestamp ts;
DBTimestamp dbts;
...
rs.next();
// Suppose that the TSTZCOL column value is 2010-10-28-00.22.33.123456-05:00, and
// the TSCOL column value is 2010-10-27-21.22.33.123456.
ts=dbrs.getDBTimestamp(1);
// Retrieve the TIMESTAMP column value into
// a Timestamp object.
ts.toString();
// Format the Timestamp object as a String.
// 2010-10-27-21:22:33.123456 is
// returned.
((DBTimestamp)ts).toDBString(false);
// Format the value as a String, without
// the time zone information.
// 2010-10-27-21:22.33.123456 is returned.
((DBTimestamp)ts).toDBString(true);
// Format the value as a String, with the
// time zone information.
// 2009-02-27-21:22.33.123456-08:00 is
// returned. The time zone is the default
// time zone.

ts=dbrs.getDBTimestamp(2);
// Retrieve the TIMESTAMP WITH TIME ZONE
// column value into a Timestamp object.
ts.toString();
// Format the Timestamp object as a String.
// 2010-10-27-21:22.33.123456 is
// returned. The returned value differs
// from the original value because toString
// uses the default time zone in its
// calculations.
((DBTimestamp)ts).toDBString(false);
// Format the value as a String, without
// the time zone information.
// 2010-10-28-00.22.33.123456 is returned.
((DBTimestamp)ts).toDBString(true);
// Format the value as a String, with the
// time zone information.
// 2010-10-28-00.22.33.123456-05:00 is
// returned. The time zone is the time zone
// from the retrieved column value.

dbts = (DBTimestamp)rs.getTimestamp(2);
// Retrieve the TIMESTAMP WITH TIME ZONE
// column value into a DBTimestamp object.
dbts.toString();
// Format the DBTimestamp object as a String.
// 2010-10-27-21:22.33.123456 is
// returned. The value is adjusted for the
// difference between the time zone in the
// column value and the default time zone.
```
dbts.setDBTimestamp // Format the value as a String, without the time zone information. The value is adjusted for
// the difference between the time zone in the column value and the default time zone.
// 2010-10-27-21.22.33.123456 is returned.

dbts.setDBTimestamp(true); // Format the value as a String, with the time zone information.
// 2009-02-27-21.22.33.123456-08:00 is returned. The time zone is the default time zone. The value is adjusted for
// the difference between the time zone in the column value and the default time zone.

dbts.toDBString(true); // Format the value as a String, with the time zone information.
// 2009-02-27-21.22.33.123456-08:00 is returned. The time zone is the default time zone. The value is adjusted for
// the difference between the time zone in the column value and the default time zone.

dbts.toDBString(false); // Format the value as a String, without the time zone information.
// 2010-10-27-21.22.33.123456 is returned. The value is adjusted for
// the difference between the time zone in the column value and the time zone in the Calendar parameter.

dbts.toString(); // Format the DBTimestamp object as a String.
// 2010-10-27-21:22:33.123456 is returned. The value is adjusted for
// the difference between the time zone in the column value and the time zone in the Calendar parameter.

dbts.toDBString(true); // Format the value as a String, with the time zone information.
// 2009-02-27-21.22.33.123456-08:00 is returned. The value is adjusted for
// the difference between the time zone in the column value and the time zone in the Calendar parameter.

dbts.toDBString(false); // Format the value as a String, without the time zone information.
// 2010-10-27-21.22.33.123456 is returned. The value is adjusted for
// the difference between the time zone in the column value and the time zone in the Calendar parameter.

dbts.toString(); // Format the DBTimestamp object as a String.
// 2010-10-27-21:22:33.123456 is returned. The returned value differs from
// the original value because toString uses the default time zone in its calculations.

dbts.toDBString(true); // Format the value as a String, with the time zone information.
// 2009-02-27-21.22.33.123456-08:00 is returned. The time zone is the default time zone.

Recommendation: Use rs.getTimestamp, followed by rs.setObject or rs.setDBTimestamp when you need to preserve the
original timestamp with time zone information when you retrieve data from one table and insert it into another table.

Related reference:
- "DBTimestamp class" on page 476
- "Common IBM Data Server Driver for JDBC and SQLJ properties for DB2 servers" on page 263
Distinct types in JDBC applications

A distinct type is a user-defined data type that is internally represented as a built-in SQL data type. You create a distinct type by executing the SQL statement `CREATE DISTINCT TYPE`.

In a JDBC program, you can create a distinct type using the `executeUpdate` method to execute the `CREATE DISTINCT TYPE` statement. You can also use `executeUpdate` to create a table that includes a column of that type. When you retrieve data from a column of that type, or update a column of that type, you use Java identifiers with data types that correspond to the built-in types on which the distinct types are based.

The following example creates a distinct type that is based on an INTEGER type, creates a table with a column of that type, inserts a row into the table, and retrieves the row from the table:

```java
Connection con;
Statement stmt;
ResultSet rs;
String empNumVar;
int shoeSizeVar;
...
stmt = con.createStatement(); // Create a Statement object
stmt.executeUpdate("CREATE DISTINCT TYPE SHOESIZE AS INTEGER"); // Create distinct type
stmt.executeUpdate("CREATE TABLE EMP_SHOE (EMPNO CHAR(6), EMP_SHOE_SIZE SHOESIZE)"); // Create table with distinct type
stmt.executeUpdate("INSERT INTO EMP_SHOE " + "VALUES ('000000', 6)"); // Insert a row
rs = stmt.executeQuery("SELECT EMPNO, EMP_SHOE_SIZE FROM EMP_SHOE"); // Create ResultSet for query
while (rs.next()) {
    empNumVar = rs.getString(1); // Get employee number
    shoeSizeVar = rs.getInt(2); // Get shoe size (use int because underlying type of SHOESIZE is INTEGER)
    System.out.println("Employee number = " + empNumVar + " Shoe size = " + shoeSizeVar);
}
rs.close(); // Close ResultSet
stmt.close(); // Close Statement
```

Figure 16. Creating and using a distinct type

Related reference:

"Data types that map to database data types in Java applications" on page 219

CREATE TYPE (distinct) (DB2 SQL)

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, the data server 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.

```java
Connection con;
Statement stmt;
ResultSet rs;
String empNumVar;
int shoeSizeVar;
...
con.setAutoCommit(false); // set autocommit OFF
stmt = con.createStatement(); // Create a Statement object
...
con.commit(); // Commit the transaction
stmt.executeUpdate("INSERT INTO EMP_SHOE " +
"VALUES ('0000010', 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 ('0000020', 10)"); // Insert another row
conn.rollback(savept); // Roll back work to the point after the first insert
...
conn.releaseSavepoint(savept); // Release the savepoint
stmt.close(); // Close the Statement
conn.commit(); // Commit the transaction
```

Figure 17. Setting, rolling back to, and releasing a savepoint in a JDBC application

Related tasks:
“Committing or rolling back JDBC transactions” on page 112

Related reference:
“Data types that map to database data types in Java applications” on page 219
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.

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 an identity column. An identity column is a table column that provides a way for the data source to automatically generate a numeric value for each row. You define an identity column in a CREATE TABLE or ALTER TABLE statement by specifying the AS IDENTITY clause when you define a column that has an exact numeric type with a scale of 0 (SMALLINT, INTEGER, BIGINT, DECIMAL with a scale of zero, or a distinct type based on one of these types).

For connections to DB2 for z/OS or Db2 on Linux, UNIX, and Windows systems, the IBM Data Server Driver for JDBC and SQLJ supports the return of automatically generated keys for INSERT statements, for searched UPDATE or searched DELETE statements, or for MERGE statements. For UPDATE, DELETE, or MERGE statements, you can identify any columns as automatically generated keys, even if they are not generated by the data server. In this case, the column values that are returned are the column values for the rows that are modified by the UPDATE, DELETE, or MERGE statement.

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 for INSERT statements, for searched UPDATE or searched DELETE statements, or for MERGE statements. 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.

Related tasks:
- “Creating and modifying database objects using the Statement.executeUpdate method” on page 30
- “Updating data in tables using the PreparedStatement.executeUpdate method” on page 31

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.

Procedure

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:
The following form is valid for a table on any data source that supports identity columns.

**Restriction:** For IBM Data Server Driver for JDBC and SQLJ version 3.57 or later, the following form is not valid for inserting rows into a view on a DB2 for z/OS data server.

```java
Connection.prepareStatement(sql-statement,
    Statement.RETURN_GENERATED_KEYS);
```

If the data server is DB2 for z/OS, the following forms are valid only if the data server supports SELECT FROM INSERT statements. With the first form, you specify the names of the columns for which you want automatically generated keys. With the second form, you specify the positions in the table of the columns for which you want automatically generated keys.

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

- If you use the `Statement.executeUpdate` method to insert rows, invoke one of these forms of the `Statement.executeUpdate` method:

  The following form is valid for a table on any data source that supports identity columns.

  **Restriction:** For IBM Data Server Driver for JDBC and SQLJ version 3.57 or later, the following form is not valid for inserting rows into a view on a DB2 for z/OS data server.

  ```java
  Statement.executeUpdate(sql-statement, Statement.RETURN_GENERATED_KEYS);
  ```

  If the data server is DB2 for z/OS, the following forms are valid only if the data server supports SELECT FROM INSERT statements. With the first form, you specify the names of the columns for which you want automatically generated keys. With the second form, you specify the positions in the table of the columns for which you want automatically generated keys.

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

2. Invoke the `PreparedStatement.getGeneratedKeys` method or the `Statement.getGeneratedKeys` method to retrieve a `ResultSet` object that contains the automatically generated key values.

   If you include the `Statement.RETURN_GENERATED_KEYS` parameter, the data type of the automatically generated keys in the `ResultSet` is DECIMAL, regardless of the data type of the corresponding column.

**Example**

The following code creates a table with an identity 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.

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

Connection con;
Statement stmt;
ResultSet rs;
java.math.BigDecimal iDColVar;
...
stmt = con.createStatement();       // Create a Statement object
stmt.executeUpdate(
```
CREATE TABLE EMP_PHONE (EMPNO CHAR(6), PHONENO CHAR(4), IDENTCOL INTEGER GENERATED ALWAYS AS IDENTITY);

// Create table with identity column
stmt.executeUpdate("INSERT INTO EMP_PHONE (EMPNO, PHONENO) VALUES ('000010', '5555')", // Insert a row
Statement.RETURN_GENERATED_KEYS); // Indicate you want automatically
// generated keys
rs = stmt.getGeneratedKeys(); // Retrieve the automatically
// generated key value in a ResultSet. // Only one row is returned.
// Create ResultSet for query
while (rs.next()) {
    java.math.BigDecimal idColVar = rs.getBigDecimal(1); // Get automatically generated key
    System.out.println("automatically generated key value = " + idColVar);
}
rs.close(); // Close ResultSet
stmt.close(); // Close Statement

With any version of the IBM Data Server Driver for JDBC and SQLJ, you can retrieve the most recently assigned value of an identity column by explicitly executing the IDENTITY_VAL_LOCAL built-in function. Execute code similar to this:

String idntVal;
Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement(); // Create a Statement object
rs = stmt.executeQuery("SELECT IDENTITY_VAL_LOCAL() FROM SYSIBM.SYSDUMMY1"); // Get the result table from the query. // This is a single row with the most // recent identity column value.
while (rs.next()) {
    idntVal = rs.getString(1); // Retrieve column value
    System.out.println("Identity column value = " + idntVal); // Print the column value
}
rs.close(); // Close the ResultSet
stmt.close(); // Close the Statement

Retrieving auto-generated keys for an UPDATE, DELETE, or MERGE 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 a searched UPDATE, searched DELETE, or MERGE statement.

Procedure

To retrieve automatically generated keys that are generated by an UPDATE, DELETE, or MERGE statement, you need to perform these steps:

1. Construct a String array that contains the names of the columns from which you want to return automatically generated keys.
   The array must be an array of column names, and not column indexes.
2. Set the autocommit mode for the connection to false.
3. 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 update, delete, or merge rows, invoke this form of the Connection.prepareStatement method to create a PreparedStatement object:

    Connection.prepareStatement(sql-statement, String [] columnNames);

If you use the Statement.executeUpdate method to update, delete, or merge rows, invoke this form of the Statement.executeUpdate method:

    Statement.executeUpdate(sql-statement, String [] columnNames);

4. Invoke the PreparedStatement.getGeneratedKeys method or the Statement.getGeneratedKeys method to retrieve a ResultSet object that contains the automatically generated key values.

Example

Suppose that a table is defined like this and has thirty rows:

```sql
CREATE TABLE EMP_BONUS
    (EMPNO CHAR(6),
     BONUS DECIMAL(9,2))
```

The following code names the EMPNO column as an automatically generated key, updates the thirty rows in the EMP_BONUS table, and retrieves the values of EMPNO for the updated rows. The numbers to the right of selected statements correspond to the previously described steps.

```java
import java.sql.*;
...
Connection conn;
...
String[] agkNames = {"EMPNO"};  // 1
int updateCount = 0;
conn.setAutoCommit(false);
PreparedStatement ps =
    conn.prepareStatement("UPDATE EMP_BONUS SET BONUS = " +  // 2
                           " BONUS + 300.00",agkNames);
updateCount = ps.executeUpdate();
ResultSet rs = ps.getGeneratedKeys();   // 3
while (rs.next()) {
    String agkEmpNo = rs.getString(1);
    // Get automatically generated key value
    System.out.println("Automatically generated key value = " + agkEmpNo);
}
ps.close();
conn.close();
```

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
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.

Related reference:
“Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products” on page 235

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.

Before you begin

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.

About this task

Procedure

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.
Example

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.

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

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

Related reference:

"DB2PreparedStatement interface" on page 435

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.

Before you begin

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.
About this task

Procedure

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.
Example

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.

```java
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();
```

Related reference:

- "DB2CallableStatement interface" on page 393
- "DB2PreparedStatement interface" on page 435

Retrieving JSON documents from a ResultSet

You can query a relational table, and then convert the ResultSet that you retrieve to a com.ibm.db2.jcc.json.DB2JSONResultSet object, which holds the data in JSON format. After you create the DB2JSONResultSet object, you can use methods on that object to retrieve the JSON documents or JSON snippets.

Procedure

To retrieve JSON documents or snippets from ResultSet objects, follow this procedure.

1. Execute a query against a relational table whose data you want to convert to JSON format. Retrieve the contents of the table into a ResultSet object.
2. Create a DB2JSONResultSet object from the ResultSet object in one of the following ways:
   - For a stand-alone application, cast the ResultSet object to a DB2ResultSet object, and execute the DB2ResultSet.toJSONResultSet method to obtain a DB2JSONResultSet object. For example:
     ```java
     DB2JSONResultSet jsonrs = ((DB2ResultSet) rs).toJSONResultSet();
     ```
   - For a web application, unwrap the ResultSet object to a DB2ResultSet object. Then execute the DB2ResultSet.toJSONResultSet method to obtain a DB2JSONResultSet object. For example:
     ```java
     DB2ResultSet db2rs = rs.unwrap (DB2ResultSet.class);
     DB2JSONResultSet jsonrs = db2rs.toJSONResultSet();
     ```

The data type mappings for conversion of the columns in the ResultSet object to the fields in the DB2JSONResultSet object are as follows:
## DB2 data type of a ResultSet column | JSON data type of a DB2JSONResultSet field
---|---
BIGINT | Number
DECIMAL | Number
DOUBLE | Number
FLOAT | Number
NUMERIC | Number
SMALLINT | Number
CHAR | String
VARCHAR | String

**Restrictions:**
- After you obtain the DB2JSONResultSet object, you can no longer retrieve data directly from the ResultSet object.
- You cannot create a DB2JSONResultSet object if the ResultSet object contains columns of the BLOB, CLOB, XML, VARBINARY, or BINARY data types.

3. Use the appropriate DB2JSONResultSet method to retrieve the data from the DB2JSONResultSet object.

<table>
<thead>
<tr>
<th>Method name</th>
<th>Function of the method</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAsciiStream</td>
<td>Returns the contents of the DB2JSONResultSet object as a java.io.InputStream object</td>
</tr>
<tr>
<td>getCharacterStream</td>
<td>Returns a java.io.Reader object on which applications can call methods to read the JSON document incrementally</td>
</tr>
<tr>
<td>getCurrentRow</td>
<td>Returns the current row of the DB2JSONResultSet as a JSON snippet</td>
</tr>
<tr>
<td>next</td>
<td>Moves the cursor to the next row in the DB2JSONResultSet object</td>
</tr>
<tr>
<td>toJSONString</td>
<td>Returns the rest of the DB2JSONResultSet object, starting with the current row, as a String</td>
</tr>
</tbody>
</table>

4. Close the DB2JSONResultSet object.

### Examples

The numbers to the right of statements in the following examples correspond to the previously described steps.

**Example:** Use DB2JSONResultSet.getCurrentRow to retrieve one row of a JSON document at a time.

```java
sql = "SELECT EMPNO, SALARY FROM EMPLOYEE";
ResultSet rs = stmt.executeQuery(sql); // ResultSet contains a CHAR and a DECIMAL column
DB2JSONResultSet jsonrs = ((DB2ResultSet)rs).toJSONResultSet(); // DB2JSONResultSet contains String and Number fields
while(jsonrs.next()) {
    String row = json.getCurrentRow(); // 3
    ...
    ... // code to process row
}
```
Example: Use DB2JSON.toJSONString to retrieve an entire JSON document into a String.

```java
sql = "SELECT EMPNO, PHONENO FROM EMPLOYEE";
ResultSet rs = stmt.executeQuery(sql);
DB2JSONResultSet jsonrs = ((DB2ResultSet) rs).toJSONResultSet();
String jsondoc = jsonrs.toJSONString(); ...
jsonrs.close(); // Close the DB2JSONResultSet
```

Example: Use DB2JSONResultSet.getAsciiStream to read the JSON document into an AsciiStream.

```java
sql = "SELECT EMPNO, PHONENO FROM EMPLOYEE";
ResultSet rs = stmt.executeQuery(sql);
DB2JSONResultSet jsonrs = ((DB2ResultSet) rs).toJSONResultSet();
InputStream is = jsonrs.getAsciiStream();
char [] buff = new char [1024];
int b = 0;
do {
b = is.read ();
if (b > 0) {
    System.out.print ((char)b);
}
} while (b > 0);
is.close();
jsonrs.close(); // Close the DB2JSONResultSet
```

Example: Use DB2JSONResultSet.getCharacterStream to retrieve an entire JSON document into a character stream. Then read the document contents into a buffer and print them.

```java
sql = "SELECT EMPNO, PHONENO FROM EMPLOYEE";
ResultSet rs = stmt.executeQuery(sql);
DB2JSONResultSet jsonrs = ((DB2ResultSet) rs).toJSONResultSet();
Stream reader = jsonrs.getCharacterStream();
char [] buff = new char[1024];
int len = 0;
do {
    len = reader.read (buff);
    if (len > 0) {
        System.out.println(String.copyValueOf(buff,0,len));
    }
}
reader.close();
jsonrs.close(); // Close the DB2JSONResultSet
```
Providing extended client information to the data source with IBM Data Server Driver for JDBC and SQLJ-only methods

A set of IBM Data Server Driver for JDBC and SQLJ-only methods provide extra information about the client to the server. This information can be used for accounting, workload management, or debugging.

About this task

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

In the IBM Data Server Driver for JDBC and SQLJ version 4.0 or later, the IBM Data Server Driver for JDBC and SQLJ-only methods are deprecated. You should use java.sql.Connection.setClientInfo instead.

The IBM Data Server Driver for JDBC and SQLJ-only methods are listed in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Information provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>setDB2ClientAccountingInformation</td>
<td>Accounting information</td>
</tr>
<tr>
<td>setDB2ClientApplicationInformation</td>
<td>Name of the application that is working with a connection</td>
</tr>
<tr>
<td>setDB2ClientDebugInfo</td>
<td>The CLIENT DEBUGINFO connection attribute for the Unified debugger</td>
</tr>
<tr>
<td>setDB2ClientProgramId</td>
<td>A caller-specified string that helps the caller identify which program is associated with a particular SQL statement. setDB2ClientProgramId does not apply to Db2 on Linux, UNIX, and Windows systems data servers.</td>
</tr>
<tr>
<td>setDB2ClientUser</td>
<td>User name for a connection</td>
</tr>
<tr>
<td>setDB2ClientWorkstation</td>
<td>Client workstation name for a connection</td>
</tr>
</tbody>
</table>

Procedure

To set the extended client information, follow these steps:
1. Create a Connection.
2. Cast the java.sql.Connection object to a com.ibm.db2.jcc.DB2Connection.
3. Call any of the methods shown in Table 12.
4. Execute an SQL statement to cause the information to be sent to the data server.

Example

The following code performs the previous steps to pass a user name and a workstation name to the data server. The numbers to the right of selected statements correspond to the previously-described steps.
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.

About this task

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 ConnectiongetClientInfo 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.

Procedure

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.
Example

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

```java
public class ClientInfoTest {
    public static void main(String[] args) {
        String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose";
        try {
            Class.forName("com.ibm.db2.jcc.DB2Driver");
            String user = "db2adm";
            String password = "db2adm";
            URLConnection conn = DriverManager.getConnection(url, user, password);
            conn.setClientInfo("ClientUser", "Michael L Thompson");
            conn.setClientInfo("ClientHostname", "sjwkstn1");
            conn.prepareStatement("SELECT * FROM SYSIBM.SYSDUMMY1" + "WHERE 0 = 1").executeQuery();
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
}
```

Figure 19. Example of passing extended client information to a data 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 on Linux, UNIX, and Windows systems and for Db2 for IBM i.

<table>
<thead>
<tr>
<th>NAME</th>
<th>MAX_LEN (bytes)</th>
<th>DEFAULT_VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationName</td>
<td>255</td>
<td>Empty string</td>
<td>The name of the application that is currently using the connection. This value is stored in DB2 special register <strong>CURRENT CLIENT_APPLNAME</strong>.</td>
</tr>
</tbody>
</table>

Chapter 3. JDBC application programming  89
### Table 13. Client info property values for Db2 on Linux, UNIX, and Windows systems and for Db2 for IBM i (continued)

<table>
<thead>
<tr>
<th>NAME</th>
<th>MAX_LEN (bytes)</th>
<th>DEFAULT_VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClientAccountingInformation</td>
<td>255</td>
<td>Empty string</td>
<td>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.</td>
</tr>
<tr>
<td>ClientHostname</td>
<td>255</td>
<td>The host name of the local host. For version 3.68 or 4.18, or later of the IBM Data Server Driver for JDBC and SQLJ, if ClientHostname is set to 'NODEFAULT', no value is sent to the data server.</td>
<td>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.</td>
</tr>
<tr>
<td>ClientUser</td>
<td>255</td>
<td>Empty string</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 14. Client info property values for type 4 connectivity to DB2 for z/OS

<table>
<thead>
<tr>
<th>NAME</th>
<th>MAX_LEN (bytes)</th>
<th>DEFAULT_VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationName</td>
<td>32</td>
<td>The string &quot;db2jcc_application&quot;.</td>
<td>The name of the application that is currently using the connection. This value is stored in DB2 special register CURRENT CLIENT_APPLNAME.</td>
</tr>
<tr>
<td>ClientAccountingInformation</td>
<td>200</td>
<td>A string of the form JCCversionclient-ip, where version is the driver version, and client-ip is the IP address of the client.</td>
<td>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.</td>
</tr>
<tr>
<td>ClientHostname</td>
<td>18</td>
<td>The string &quot;db2jcc_local&quot;. For version 3.68 or 4.18, or later of the IBM Data Server Driver for JDBC and SQLJ, if ClientHostname is set to 'NODEFAULT', no value is sent to the data server.</td>
<td>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.</td>
</tr>
<tr>
<td>ClientUser</td>
<td>16</td>
<td>The user ID that was specified when the connection was established.</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 15. Client info property values for type 2 connectivity on DB2 for z/OS

<table>
<thead>
<tr>
<th>NAME</th>
<th>MAX_LEN (bytes)</th>
<th>DEFAULT_VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationName</td>
<td>32</td>
<td>The string &quot;db2jcc_application&quot;.</td>
<td>The name of the application that is currently using the connection. This value is stored in DB2 special register CURRENT CLIENT_APPLNAME.</td>
</tr>
<tr>
<td>ClientAccountingInformation</td>
<td>22</td>
<td>Empty string.</td>
<td>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.</td>
</tr>
<tr>
<td>ClientHostname</td>
<td>18</td>
<td>The string &quot;RRSAF&quot;.</td>
<td>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.</td>
</tr>
<tr>
<td>ClientUser</td>
<td>16</td>
<td>The user ID that was specified for the connection. If no user ID was specified, the RACF user ID is used.</td>
<td>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.</td>
</tr>
</tbody>
</table>

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

### Table 16. Client info property values for IBM Informix

<table>
<thead>
<tr>
<th>NAME</th>
<th>MAX_LEN (bytes)</th>
<th>DEFAULT_VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationName</td>
<td>20</td>
<td>Empty string</td>
<td>The name of the application that is currently using the connection.</td>
</tr>
<tr>
<td>ClientAccountingInformation</td>
<td>199</td>
<td>Empty string</td>
<td>The value of the accounting string from the client information that is specified for the connection.</td>
</tr>
<tr>
<td>ClientHostname</td>
<td>20</td>
<td>The host name of the local host.</td>
<td>The host name of the computer on which the application that is using the connection is running.</td>
</tr>
<tr>
<td>ClientUser</td>
<td>1024</td>
<td>Empty string</td>
<td>The name of the user on whose behalf the application that is using the connection is running.</td>
</tr>
</tbody>
</table>

**Extended parameter information with the IBM Data Server Driver for JDBC and SQLJ**

IBM Data Server Driver for JDBC and SQLJ-only methods and constants let you assign the default value or no value to table columns or ResultSet columns.

The data server must support extended indicators before you can use the methods that provide extended indicator information in your Java applications. If you call
one of those methods against a data server that does not support extended indicators, an exception is thrown. Extended parameter information is supported by DB2 for z/OS Version 10 or later, or Db2 on Linux, UNIX, and Windows systems Version 9.7 or later.

The methods that provide extended parameter information are listed in the following table.

<table>
<thead>
<tr>
<th>Extended parameter information methods</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2PreparedStatement.setDBDefault,</td>
<td>Sets an input parameter to its default value.</td>
</tr>
<tr>
<td>DB2PreparedStatement.setJccDBDefaultAtName</td>
<td></td>
</tr>
<tr>
<td>DB2PreparedStatement.setDBUnassigned,</td>
<td>Indicates that an input parameter is unassigned. This action yields the same behavior that would occur if the input parameter did not appear in the SQL statement text.</td>
</tr>
<tr>
<td>DB2PreparedStatement.setJccDBUnassignedAtName</td>
<td></td>
</tr>
<tr>
<td>DB2ResultSet.updateDBDefault</td>
<td>Sets a column value in the current ResultSet row to its default value.</td>
</tr>
</tbody>
</table>

These methods are applicable only for parameter markers that appear in one of the following places:
- The SET list of an UPDATE statement
- The SET list of a MERGE statement
- The VALUES list of an INSERT statement
- The VALUES list of a MERGE statement
- The source table in a MERGE statement
- The SELECT list of an INSERT from SELECT statement

An SQLException is raised if you use these methods in any other context.

Alternatively, you can use the standard PreparedStatement.setObject or ResultSet.updateObject methods with IBM Data Server Driver for JDBC and SQLJ-only constants DB2PreparedStatement.DB_PARAMETER_DEFAULT or DB2PreparedStatement.DB_PARAMETER_UNASSIGNED to assign the default value or no value to parameters.

Extended parameter information can simplify application programs that have several input variables, each of which can send a value or the default value to the data server, or does not need to appear in the SQL statement. Instead of preparing separate statement strings for all combinations of variable values, you can prepare a single statement string. The resulting PreparedStatement object can be used in a homogeneous batch, whereas multiple different PreparedStatement objects cannot be used in a homogeneous batch.

**Related reference:**

“DB2PreparedStatement interface” on page 435

**Using DB2PreparedStatement methods or constants to provide extended parameter information**

Use DB2PreparedStatement methods or PreparedStatement methods with DB2PreparedStatement constants to assign default values to target columns or to assign no values to target columns.
About this task

Follow these steps to send extended client information for a PreparedStatement to the data server.

Procedure

1. Create a PreparedStatement object.
   The SQL statement is a INSERT, UPDATE, or MERGE statement.
2. If you are not using setObject to assign the values, cast the PreparedStatement object to a com.ibm.db2.jcc.DB2PreparedStatement object.
3. Call one of the following methods:
   - If you are not using setObject to assign the value:
     - To assign the default value of the target column to the input parameter, call DB2PreparedStatement.setDBDefault or DB2PreparedStatement.setJccDBDefaultAtName.
     - To mark the input parameter as unassigned, call DB2PreparedStatement.setDBUnassigned or DB2PreparedStatement.setJccDBUnassignedAtName.
   - If you are using setObject to assign the value:
     - To assign the default value of the target column to the input parameter, call PreparedStatement.setObject with DB2PreparedStatement.DB_PARAMETER_DEFAULT as the assigned value.
     - To mark the input parameter as unassigned, call PreparedStatement.setObject with DB2PreparedStatement.DB_PARAMETER_UNASSIGNED as the assigned value.
4. Execute the SQL statement.

Example

The following code assigns the default values of the target columns to the third and fifth parameters in an INSERT statement. The numbers to the right of selected statements correspond to the previously described steps.

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

Connection conn;
...
PreparedStatement p = conn.prepareStatement("INSERT INTO DEPARTMENT " +
"(DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION) " +
"VALUES (?, ?, ?, ?, ?)");
p.setString(1, "X00");
p.setString(2, "FACILITIES");
p.setString(4, "A00");
((com.ibm.db2.jcc.DB2PreparedStatement)p).setDBDefault(3);  // 2,3
((com.ibm.db2.jcc.DB2PreparedStatement)p).setDBDefault(5);
int uCount = p.executeUpdate();  // 4
...
p.close();  // Close PreparedStatement
```

The following code uses the PreparedStatement.setObject method and DB2PreparedStatement constants to perform the same function as in the previous example. The numbers to the right of selected statements correspond to the previously described steps.

Chapter 3. JDBC application programming 93
import java.sql.*;
import com.ibm.db2.jcc.*;

Connection conn;
...

PreparedStatement p = conn.prepareStatement("INSERT INTO DEPARTMENT " + "(DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION) " + "VALUES (?, ?, ?, ?, ?)");
p.setString(1, "X00");
p.setString(2, "FACILITIES");
p.setString(4, "A00");
p.setObject(3, DB2PreparedStatement.DB_PARAMETER_DEFAULT);
p.setObject(5, DB2PreparedStatement.DB_PARAMETER_DEFAULT);
int uCount = p.executeUpdate();
...
p.close(); // Close PreparedStatement

In these examples, use of the method DB2PreparedStatement.setDBDefault or the constant DB2PreparedStatement.DB_PARAMETER_DEFAULT simplifies programming of the INSERT operation. If DB2PreparedStatement.setDBDefault or DB2PreparedStatement.DB_PARAMETER_DEFAULT is not used, up to 32 different PreparedStatement objects are necessary to cover all combinations of default and non-default input values.

**Using DB2ResultSet methods or DB2PreparedStatement constants to provide extended parameter information**

Use DB2ResultSet methods or ResultSet methods with DB2PreparedStatement constants to assign default values to target columns in a DB2ResultSet.

**About this task**

Follow these steps to update a ResultSet with extended client information.

**Procedure**

1. Create a PreparedStatement object.
   The SQL statement is a SELECT statement.
2. Invoke PreparedStatement.setXXX methods to pass values to any input parameters.
3. Invoke the PreparedStatement.executeQuery method to obtain the result table from the SELECT statement in a ResultSet object.
4. Position the cursor to the row that you want to update or insert.
5. Update columns in the ResultSet row.
   - If you are not using updateObject to update a value:
     - To assign the default value to the target column of the ResultSet, cast the ResultSet to a DB2ResultSet, and call DB2ResultSet.updateDBDefault.
   - If you are using updateObject to assign the value:
     - To assign the default value to the target column of the ResultSet, call ResultSet.updateObject with DB2PreparedStatement.DB_PARAMETER_DEFAULT as the assigned value.
6. Execute ResultSet.updateRow if you are updating an existing row, or ResultSet.insertRow if you are inserting a new row.
Example

The following code inserts a row into a ResultSet with the default value in the second column, and does not modify the value in the first column. The numbers to the right of selected statements correspond to the previously described steps.

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

Connection conn;
...
PreparedStatement p = conn.prepareStatement (  
   "SELECT MGRNO, LOCATION " +  
   "FROM DEPARTMENT");
ResultSet rs = p.executeQuery ();
rs.next ();
rs.moveToInsertRow ();
((DB2ResultSet)rs).updateDBDefault (2);
rs.insertRow ();
...
rs.close ();   // Close ResultSet
p.close();   // Close PreparedStatement
```

The following code uses the ResultSet interface with DB2PreparedStatement constants to perform the same function as in the previous example. The numbers to the right of selected statements correspond to the previously described steps.

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

Connection conn;
...
PreparedStatement p = conn.prepareStatement (  
   "SELECT MGRNO, LOCATION " +  
   "FROM DEPARTMENT");
ResultSet rs = p.executeQuery ();
rs.next ();
rs.moveToInsertRow ();
r.updateObject (2,  
   DB2PreparedStatement.DB_PARAMETER_DEFAULT);
rs.insertRow ();
...
rs.close ();   // Close ResultSet
p.close();   // Close PreparedStatement
```

### XML data in JDBC applications

In JDBC applications, you can store data in XML columns and retrieve data from XML columns.

In database tables, the XML built-in data type is used to store XML data in a column as a structured set of nodes in a tree format.

JDBC applications can send XML data to the data server or retrieve XML data from the data server in one of the following forms:

- As textual XML data
- As binary XML data, if the data server supports it

In JDBC applications, you can:

- Store an entire XML document in an XML column using setXXX methods.
- Retrieve an entire XML document from an XML column using getXXX methods.
• Retrieve a sequence from a document in an XML column by using the SQL XMLQUERY function to retrieve the sequence into a serialized sequence in the database, and then using getXXX methods to retrieve the data into an application variable.

• Retrieve a sequence from a document in an XML column as a user-defined table by using the SQL XMLTABLE function to define the result table and retrieve it. Then use getXXX methods to retrieve the data from the result table into application variables.

JDBC 4.0 java.sql.SQLXML objects can be used to retrieve and update data in XML columns. Invocations of metadata methods, such as ResultSetMetaData.getColumnTypeName return the integer value java.sql.Types.SQLXML for an XML column type.

Related concepts:
“XML data retrieval in JDBC applications” on page 98
“XML column updates in JDBC applications”

Related reference:
“Data types that map to database data types in Java applications” on page 219

XML column updates in JDBC applications

In a JDBC application, you can update or insert data into XML columns of a table at a DB2 data server using XML textual data. You can update or insert data into XML columns of a table using binary XML data (data that is in the Extensible Dynamic Binary XML Db2 Client/Server Binary XML Format), if the data server supports binary XML data.

The following table lists the methods and corresponding input data types that you can use to put data in XML columns.

<table>
<thead>
<tr>
<th>Method</th>
<th>Input data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PreparedStatement.setAsciiStream</td>
<td>InputStream</td>
</tr>
<tr>
<td>PreparedStatement.setBinaryStream</td>
<td>InputStream</td>
</tr>
<tr>
<td>PreparedStatement.setBlob</td>
<td>Blob</td>
</tr>
<tr>
<td>PreparedStatement.setBytes</td>
<td>byte[]</td>
</tr>
<tr>
<td>PreparedStatement.setCharacterStream</td>
<td>Reader</td>
</tr>
<tr>
<td>PreparedStatement.setClob</td>
<td>Clob</td>
</tr>
<tr>
<td>PreparedStatement setObject</td>
<td>byte[], Blob, Clob, SQLXML, DB2Xml (deprecated), InputStream, Reader, String</td>
</tr>
<tr>
<td>PreparedStatement.setSQLXML</td>
<td>SQLXML</td>
</tr>
<tr>
<td>PreparedStatement.setString</td>
<td>String</td>
</tr>
</tbody>
</table>

Note:
1. This method requires JDBC 4.0 or later.

The encoding of XML data can be derived from the data itself, which is known as internally encoded data, or from external sources, which is known as externally encoded data. XML data that is sent to the database server as binary data is treated as internally encoded data. XML data that is sent to the data source as character data is treated as externally encoded data.
External encoding for Java applications is always Unicode encoding.

Externally encoded data can have internal encoding. That is, the data might be sent to the data source as character data, but the data contains encoding information. The data source handles incompatibilities between internal and external encoding as follows:

- If the data source is Db2 on Linux, UNIX, and Windows systems, the database source generates an error if the external and internal encoding are incompatible, unless the external and internal encoding are Unicode. If the external and internal encoding are Unicode, the database source ignores the internal encoding.
- If the database source is DB2 for z/OS, the database source ignores the internal encoding.

Character data in XML columns is stored in UTF-8 encoding. The database source handles conversion of the data from its internal or external encoding to UTF-8.

**Example:** The following example demonstrates inserting data from an SQLXML object into an XML column. The data is String data, so the database source treats the data as externally encoded.

```java
public void insertSQLXML()
{
    Connection con = DriverManager.getConnection(url);
    SQLXML info = con.createSQLXML();
    // Create an SQLXML object
    PreparedStatement insertStmt = null;
    String infoData =
    "<customerinfo xmlns="http://posample.org" Cid="1000">...</customerinfo>";
    info.setString(infoData);
    // Populate the SQLXML object
    int cid = 1000;
    try {
        sqls = "INSERT INTO CUSTOMER (CID, INFO) VALUES (?, ?)";
        insertStmt = con.prepareStatement(sqls);
        insertStmt.setInt(1, cid);
        insertStmt.setSQLXML(2, info);
        // Assign the SQLXML object value
        // to an input parameter
        if (insertStmt.executeUpdate() != 1) {
            System.out.println("insertSQLXML: No record inserted.");
        }
    }
    catch (IOException ioe) {
        ioe.printStackTrace();
    }
    catch (SQLException sqle) {
        System.out.println("insertSQLXML: SQL Exception: " + sqle.getMessage());
        System.out.println("insertSQLXML: SQL State: " + sqle.getSQLState());
        System.out.println("insertSQLXML: SQL Error Code: " + sqle.getErrorCode());
    }
}
```

**Example:** The following example demonstrates inserting data from a file into an XML column. The data is inserted as binary data, so the database server honors the internal encoding.
public void insertBinStream(Connection conn)
{
    PreparedStatement insertStmt = null;
    String sqls = null;
    int cid = 0;
    Statement stmt = null;
    try {
        sqls = "INSERT INTO CUSTOMER (CID, INFO) VALUES (?, ?)";
        insertStmt = conn.prepareStatement(sqls);
        insertStmt.setInt(1, cid);
        File file = new File(fn);
        insertStmt.setBinaryStream(2, new FileInputStream(file), (int)file.length());
        if (insertStmt.executeUpdate() != 1) {
            System.out.println("insertBinStream: No record inserted.");
        }
    }
    catch (IOException ioe) {
        ioe.printStackTrace();
    }
    catch (SQLException sqle) {
        System.out.println("insertBinStream: SQL Exception: " + sqle.getMessage());
        System.out.println("insertBinStream: SQL State: " + sqle.getSQLState());
    }
}

Example: The following example demonstrates inserting binary XML data from a file into an XML column.

... SQLXML info = conn.createSQLXML();
OutputStream os = info.setBinaryStream();
FileInputStream fis = new FileInputStream("c7.xml");
int read;
while ((read = fis.read()) != -1) {
    os.write(read);
}

PreparedStatement insertStmt = null;
String sqls = null;
int cid = 1015;
sqls = "INSERT INTO MyCustomer (Cid, Info) VALUES (?, ?)";
insertStmt = conn.prepareStatement(sqls);
insertStmt.setInt(1, cid);
insertStmt.setSQLXML(2, info);
insertStmt.executeUpdate();

Related reference:
"Data types that map to database data types in Java applications" on page 219

XML data retrieval in JDBC applications

In JDBC applications, you use ResultSet.getXXX or ResultSet.getObject methods to retrieve data from XML columns.

In a JDBC application, you can retrieve data from XML columns in a DB2 table as XML textual data. You can retrieve data from XML columns in a table as binary XML data (data that is in the Extensible Dynamic Binary XML Db2 Client/Server Binary XML Format), if the data server supports binary XML data.
You can use one of the following techniques to retrieve XML data:

- Use the `ResultSet.getSQLXML` method to retrieve the data. Then use a `SQLXML.getXXX` method to retrieve the data into a compatible output data type. This technique requires JDBC 4.0 or later.
  
  For example, you can retrieve data by using the `SQLXML.getBinaryStream` method or the `SQLXML.getSource` method.

- Use a `ResultSet.getXXX` method other than `ResultSet.getObject` to retrieve the data into a compatible data type.

- Use the `ResultSet.getObject` method to retrieve the data, and then cast it to the `DB2Xml` type and assign it to a `DB2Xml` object. Then use a `DB2Xml.getDB2XXX` or `DB2Xml.getDB2XmlXXX` method to retrieve the data into a compatible output data type.

  You need to use this technique if you are not using a version of the IBM Data Server Driver for JDBC and SQLJ that supports JDBC 4.0.

The following table lists the `ResultSet` methods and corresponding output data types for retrieving XML data.

<table>
<thead>
<tr>
<th>Method</th>
<th>Output data type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ResultSet.getAsciiStream</code></td>
<td><code>InputStream</code></td>
</tr>
<tr>
<td><code>ResultSet.getBinaryStream</code></td>
<td><code>InputStream</code></td>
</tr>
<tr>
<td><code>ResultSet.getBytes</code></td>
<td><code>byte[]</code></td>
</tr>
<tr>
<td><code>ResultSet.getCharacterStream</code></td>
<td><code>Reader</code></td>
</tr>
<tr>
<td><code>ResultSet.getObject</code></td>
<td><code>Object</code></td>
</tr>
<tr>
<td><code>ResultSet.getSQLXML</code></td>
<td><code>SQLXML</code></td>
</tr>
<tr>
<td><code>ResultSet.getString</code></td>
<td><code>String</code></td>
</tr>
</tbody>
</table>

The following table lists the methods that you can call to retrieve data from a `java.sql.SQLXML` or a `com.ibm.db2.jcc.DB2Xml` object, and the corresponding output data types and type of encoding in the XML declarations.

<table>
<thead>
<tr>
<th>Method</th>
<th>Output data type</th>
<th>Type of XML internal encoding declaration added</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SQLXML.getBinaryStream</code></td>
<td><code>InputStream</code></td>
<td>None</td>
</tr>
<tr>
<td><code>SQLXML.getCharacterStream</code></td>
<td><code>Reader</code></td>
<td>None</td>
</tr>
<tr>
<td><code>SQLXML.getSource</code></td>
<td><code>Source</code></td>
<td>None</td>
</tr>
<tr>
<td><code>SQLXML.getString</code></td>
<td><code>String</code></td>
<td>None</td>
</tr>
<tr>
<td><code>DB2Xml.getDB2AsciiStream</code></td>
<td><code>InputStream</code></td>
<td>None</td>
</tr>
<tr>
<td><code>DB2Xml.getDB2BinaryStream</code></td>
<td><code>InputStream</code></td>
<td>None</td>
</tr>
<tr>
<td><code>DB2Xml.getDB2Bytes</code></td>
<td><code>byte[]</code></td>
<td>None</td>
</tr>
<tr>
<td><code>DB2Xml.getDB2CharacterStream</code></td>
<td><code>Reader</code></td>
<td>None</td>
</tr>
<tr>
<td><code>DB2Xml.getDB2String</code></td>
<td><code>String</code></td>
<td>None</td>
</tr>
<tr>
<td><code>DB2Xml.getDB2XmlAsciiStream</code></td>
<td><code>InputStream</code></td>
<td><code>US-ASCII</code></td>
</tr>
<tr>
<td><code>DB2Xml.getDB2XmlBinaryStream</code></td>
<td><code>InputStream</code></td>
<td>Specified by <code>getDB2XmlBinaryStream</code> <code>targetEncoding</code> parameter</td>
</tr>
</tbody>
</table>
Table 19. SQLXML and DB2Xml methods, data types, and added encoding specifications (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Output data type</th>
<th>Type of XML internal encoding declaration added</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2Xml.getDB2XmlBytes</td>
<td>byte[]</td>
<td>Specified by DB2Xml.getDB2XmlBytes targetEncoding parameter</td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlCharacterStream Reader</td>
<td>ISO-10646-UCS-2</td>
<td></td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlString</td>
<td>String</td>
<td>ISO-10646-UCS-2</td>
</tr>
</tbody>
</table>

Note:
1. The class that is returned is specified by the invoker of getSource, but the class must extend javax.xml.transform.Source.

If the application executes the XMLSERIALIZE function on the data that is to be returned, after execution of the function, the data has the data type that is specified in the XMLSERIALIZE function, not the XML data type. Therefore, the driver handles the data as the specified type and ignores any internal encoding declarations.

Example: The following example demonstrates retrieving data from an XML column into an SQLXML object, and then using the SQLXML.getString method to retrieve the data into a string.

```java
public void fetchToSQLXML(long cid, java.sql.Connection conn) {
    System.out.println(">> fetchToSQLXML: Get XML data as an SQLXML object "+
                       "using getSQLXML");
    PreparedStatement selectStmt = null;
    String sqls = null, stringDoc = null;
    ResultSet rs = null;
    try{
        sqls = "SELECT info FROM customer WHERE cid = " + cid;
        selectStmt = conn.prepareStatement(sqls);
        rs = selectStmt.executeQuery();

        // Get metadata
        // Column type for XML column is the integer java.sql.Types.OTHER
        ResultSetMetaData meta = rs.getMetaData();
        int colType = meta.getColumnType(1);
        System.out.println("fetchToSQLXML: Column type = " + colType);
        while (rs.next()) {
            // Retrieve the XML data with getSQLXML.
            // Then write it to a string with
            // explicit internal ISO-10646-UCS-2 encoding.
            java.sql.SQLXML xml = rs.getSQLXML(1);
            System.out.println (xml.getString());
        }
        rs.close();
    } catch (SQLException sqle) {
        System.out.println("fetchToSQLXML: SQL Exception: " +
                            sqle.getMessage());
        System.out.println("fetchToSQLXML: SQL State: " +
                            sqle.getSQLState());
        System.out.println("fetchToSQLXML: SQL Error Code: " +
                            sqle.getErrorCode());
    }
}
```

Example: The following example demonstrates retrieving data from an XML column into an SQLXML object, and then using the SQLXML.getBinaryStream method to retrieve the data as binary data into an InputStream.
String sql = "SELECT INFO FROM Customer WHERE Cid='1000';";  
PreparedStatement pstmt = con.prepareStatement(sql);  
ResultSet resultSet = pstmt.executeQuery();  
// Get the result XML as a binary stream  
SQLXML sqlxml = resultSet.getSQLXML(1);  
InputStream binaryStream = sqlxml.getBinaryStream();

Example: The following example demonstrates retrieving data from an XML column into a String variable.

```java
public void fetchToString(long cid, java.sql.Connection conn) {
    System.out.println(">> fetchToString: Get XML data "+
                    "using getString");
    PreparedStatement selectStmt = null;
    String sqls = null, stringDoc = null;
    ResultSet rs = null;

    try{
        sqls = "SELECT info FROM customer WHERE.cid = " + cid;
        selectStmt = conn.prepareStatement(sqls);
        rs = selectStmt.executeQuery();

        // Get metadata  
        // Column type for XML column is the integer java.sql.Types.OTHER  
        ResultSetMetaData meta = rs.getMetaData();
        int colType = meta.getColumnType(1);
        System.out.println("fetchToString: Column type = " + colType);

        while (rs.next()) {
            stringDoc = rs.getString(1);
            System.out.println("Document contents:");
            System.out.println(stringDoc);
        }
    }
    catch (SQLException sqle) {
        System.out.println("fetchToString: SQL Exception: "+
                            sqle.getMessage());
        System.out.println("fetchToString: SQL State: "+
                            sqle.getSQLState());
        System.out.println("fetchToString: SQL Error Code: "+
                            sqle.getErrorCode());
    }
}
```

Example: The following example demonstrates retrieving data from an XML column into a DB2XML object, and then using the DB2XML.getDB2XMLString method to retrieve the data into a string with an added XML declaration with an ISO-10646-UCS-2 encoding specification.

```java
public void fetchToDB2Xml(long cid, java.sql.Connection conn) {
    System.out.println(">> fetchToDB2Xml: Get XML data as a DB2XML object "+
                        "using getObject");
    PreparedStatement selectStmt = null;
    String sqls = null, stringDoc = null;
    ResultSet rs = null;

    try{
        sqls = "SELECT info FROM customer WHERE.cid = " + cid;
        selectStmt = conn.prepareStatement(sqls);
        rs = selectStmt.executeQuery();

        // Get metadata  
        // Column type for XML column is the integer java.sql.Types.OTHER  
        ResultSetMetaData meta = rs.getMetaData();
        int colType = meta.getColumnType(1);
        System.out.println("fetchToDB2Xml: Column type = " + colType);
```
while (rs.next()) {
    // Retrieve the XML data with getObject, and cast the object
    // as a DB2Xml object. Then write it to a string with
    // explicit internal ISO-10646-UCS-2 encoding.
    com.ibm.db2.jcc.DB2Xml xml =
        (com.ibm.db2.jcc.DB2Xml) rs.getObject(1);
    System.out.println (xml.getDB2XmlString());
}
rs.close();
} catch (SQLException sqle) {
    System.out.println("fetchToDB2Xml: SQL Exception: "+
        sqle.getMessage());
    System.out.println("fetchToDB2Xml: SQL State:"+
        sqle.getSQLState());
    System.out.println("fetchToDB2Xml: SQL Error Code:"+
        sqle.getErrorCode());
}

Related reference:

"Data types that map to database data types in Java applications” on page 219

Invocation of routines with XML parameters in Java applications

Java applications can call stored procedures at Db2 on Linux, UNIX, and Windows systems or DB2 for z/OS data sources that have XML parameters.

For native SQL procedures, XML parameters in the stored procedure definition have the XML type. For external stored procedures and user-defined functions on Db2 on Linux, UNIX, and Windows systems data sources, XML parameters in the routine definition have the XML AS CLOB type. When you call a stored procedure or user-defined function that has XML parameters, you need to use a compatible data type in the invoking statement.

To call a routine with XML input parameters from a JDBC program, use parameters of the java.sql.SQLXML or com.ibm.db2.jcc.DB2Xml type. To register XML output parameters, register the parameters as the java.sql.Types.SQLXML or com.ibm.db2.jcc.DB2Types.XML type. (The com.ibm.db2.jcc.DB2Xml and com.ibm.db2.jcc.DB2Types.XML types are deprecated.)

Example: JDBC program that calls a stored procedure that takes three XML parameters: an IN parameter, an OUT parameter, and an INOUT parameter. This example requires JDBC 4.0 or later.

```java
java.sql.SQLXML in_xml = xmlvar;
java.sql.SQLXML out_xml = null;
java.sql.SQLXML inout_xml = xmlvar;  
// Declare an input, output, and
// INOUT XML parameter

Connection con;
CallableStatement cstmt;
ResultSet rs;
...
cstmt = con.prepareCall("CALL SP_xml(?,?,?)");
    // Create a CallableStatement object
    cstmt.setObject (1, in_xml);  // Set input parameter
    cstmt.setObject (3, inout_xml);  // Set inout parameter
    cstmt.registerOutParameter (2, java.sql.Types.SQLXML);  // Register out and input parameters
    cstmt.registerOutParameter (3, java.sql.Types.SQLXML);
    cstmt.executeUpdate();  // Call the stored procedure
```
out_xml = cstmt.getSQLXML(2); // Get the OUT parameter value
inout_xml = cstmt.getSQLXML(3); // Get the INOUT parameter value
System.out.println("Parameter values from SP_xml call: ");
System.out.println("Output parameter value ");
MyUtilities.printString(out_xml.getString());
    // Use the SQLXML.getString
    // method to convert the out_xml
    // value to a string for printing.
    // Call a user-defined method called
    // printString (not shown) to print
    // the value.
System.out.println("INOUT parameter value ");
MyUtilities.printString(inout_xml.getString());
    // Use the SQLXML.getString
    // method to convert the inout_xml
    // value to a string for printing.
    // Call a user-defined method called
    // printString (not shown) to print
    // the value.

To call a routine with XML parameters from an SQLJ program, use parameters of the java.sql.SQLXML or com.ibm.db2.jcc.DB2Xml type.

**Example:** SQLJ program that calls a stored procedure that takes three XML parameters: an IN parameter, an OUT parameter, and an INOUT parameter. This example requires JDBC 4.0 or later.

```sql
java.sql.SQLXML in_xml = xmlvar;
java.sql.SQLXML out_xml = null;
java.sql.SQLXML inout_xml = xmlvar;
    // Declare an input, output, and
    // INOUT XML parameter
...
#sql [myConnCtx] {CALL SP_xml(:IN in_xml,
    :OUT out_xml,
    :INOUT inout_xml);} ;
    // Call the stored procedure
System.out.println("Parameter values from SP_xml call: ");
System.out.println("Output parameter value ");
MyUtilities.printString(out_xml.getString());
    // Use the SQLXML.getString
    // method to convert the out_xml value
    // to a string for printing.
    // Call a user-defined method called
    // printString (not shown) to print
    // the value.
System.out.println("INOUT parameter value ");
MyUtilities.printString(inout_xml.getString());
    // Use the SQLXML.getString
    // method to convert the inout_xml
    // value to a string for printing.
    // Call a user-defined method called
    // printString (not shown) to print
    // the value.
```

**Binary XML format in Java applications**

The IBM Data Server Driver for JDBC and SQLJ can send XML data to the data server or retrieve XML data from the data server as binary XML data (data that is in the Extensible Dynamic Binary XML Db2 Client/Server Binary XML Format). The data server must provide support for binary XML data.

The IBM Data Server Driver for JDBC and SQLJ presents binary XML data to the application only through XML object interfaces. The user does not see the data in the binary XML format.
The format of XML data is transparent to the application. Storage and retrieval of binary XML data requires version 4.9 or later of the IBM Data Server Driver for JDBC and SQLJ. If you are using binary XML data in SQLJ applications, you also need version 4.9 or later of the sqlj4.zip package.

You use the property xmlFormat to control whether the data format for retrieval of XML data is textual XML format or binary XML format. You set xmlFormat to XML_FORMAT_BINARY (1) to enable binary XML format. The default is textual XML format.

For update of data in XML table columns, xmlFormat has no effect. If the input data is binary XML data, and the data server does not support binary XML data, the input data is converted to textual XML data. Otherwise, no conversion occurs.

When binary XML data 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.

There is no setXXX method defined on the Connection interface for the xmlFormat property. Therefore, to set the xmlFormat value when you use the Connection interface, you need to specify xmlFormat as a property when you execute the DriverManager.getConnection method. For example:

```java
properties.put("xmlFormat", "1");
DriverManager.getConnection(url, properties);
```

**Restriction:** When you send XML data in binary format to a DB2 for z/OS data server that supports binary XML data, you cannot use an InputStreamReader object with a CharSet object named UTF-16LE, UTF-8, or UTF-16BE for an XML document file that contains a byte order mark (BOM). To circumvent this restriction, take one of the following actions:

- Remove the BOM from the XML instance document in the input file.
- Use an InputStreamReader object with a CharSet object named UTF-16 for the input file.
- Use an InputStream object instead of an InputStreamReader object for the input file.

Binary XML format is most efficient for cases in which the input or output data is in a non-textual representation, such as SAX, StAX, or DOM. For example, these methods retrieve XML data in non-textual representations:

- `getSourcexxx(SAXSource.class)`
- `getSourcexxx(StAXSource.class)`
- `getSourcexxx(DOMSource.class)`

These methods update XML columns with data in non-textual representations:

- `setResult(SAXResult.class)`
- `setResult(StAXResult.class)`
- `setResult(DOMResult.class)`

The SAX representation is the most efficient way to retrieve data that is in the binary XML format because the data does not undergo extra conversions from binary format to textual format.
Suppose that you set xmlFormat to XML_FORMAT_BINARY (1). In the following JDBC example, the IBM Data Server Driver for JDBC and SQLJ retrieves data in the binary XML format, application uses the SAX parser to parse the retrieved data.

```java
... 
Statement stmt = conn.createStatement(); 
ResultSet rs = stmt.executeQuery("SELECT XMLCOL FROM XMLTABLE"); 
ContentHandler handler = new MyContentHandler(); 
while (rs.next()) {
    SQLXML sqlxml = rs.getSQLXML(1);
    SAXSource source = sqlxml.getSource(SAXSource.class);
    XMLReader reader = source.getXMLReader();
    reader.setContentHandler(handler);
    reader.parse(source.getInputStream());
} 
... 
```

The following SQLJ example performs the same actions.

```sql
#sql iterator SqlXmlIter(java.sql.SQLXML); { 
 ... 
 SqlXmlIter SQLXmlIter = null;
 java.sql.SQLXML outSqlXml = null;
 ContentHandler handler = new MyContentHandler();
 #sql [ctx] SqlXmlIter = {SELECT XMLCOL FROM XMLTABLE};
 #sql {FETCH :SqlXmlIter INTO :outSqlXml};
 while (!SQLXmlIter.endFetch()) {
    SAXSource source = outSqlXml.getSource(SAXSource.class);
    XMLReader reader = source.getXMLReader();
    reader.setContentHandler(handler);
    reader.parse(source.getInputStream());
    #sql {FETCH :SqlXmlIter INTO :outSqlXml};
 } 
... 
}
```

**Java support for XML schema registration and removal**

The IBM Data Server Driver for JDBC and SQLJ provides methods that let you write Java application programs to register and remove XML schemas and their components.

The methods are:

- **DB2Connection.registerDB2XMLSchema**
  Registers an XML schema in the database manager, using one or more XML schema documents. There are two forms of this method: one form for XML schema documents that are input from InputStream objects, and one form for XML schema documents that are in a String.

- **DB2Connection.deregisterDB2XMLObject**
  Removes an XML schema definition from the database manager.

- **DB2Connection.updateDB2XmlSchema**
  Replaces the XML schema documents in a registered XML schema with the XML schema documents from another registered XML schema. Optionally drops the XML schema whose contents are copied. This method is available only for connections to Db2 on Linux, UNIX, and Windows systems.

Before you can invoke these methods, the stored procedures that support these methods must be installed on the data server.
Example: Registration of an XML schema: The following example demonstrates the use of registerDB2XmlSchema to register an XML schema in DB2, using a single XML schema document (customer.xsd) that is read from an input stream. The SQL schema name for the registered schema is SYSXSR. The xmlSchemaLocations value is null, so DB2 will not find this XML schema on an invocation of DSN_XMLVALIDATE that supplies a non-null XML schema location value. No additional properties are registered.

```java
class Example {
    public static void registerSchema(
        Connection con,
        String schemaName)
    throws SQLException {
        // Define the registerDB2XmlSchema parameters
        String[] xmlSchemaNameQualifiers = new String[1];
        String[] xmlSchemaNames = new String[1];
        String[] xmlSchemaLocations = new String[1];
        InputStream[] xmlSchemaDocuments = new InputStream[1];
        int[] xmlSchemaDocumentsLengths = new int[1];
        java.io.InputStream[] xmlSchemaDocumentsProperties = new java.io.InputStream[1];
        int[] xmlSchemaDocumentsPropertiesLengths = new int[1];
        InputStream xmlSchemaProperties;
        int xmlSchemaPropertiesLength;
        // Set the parameter values
        xmlSchemaLocations[0] = "";
        FileInputStream fi = null;
        xmlSchemaNameQualifiers[0] = "SYSXSR";
        xmlSchemaNames[0] = schemaName;
        try {
            fi = new FileInputStream("customer.xsd");
            xmlSchemaDocuments[0] = new BufferedInputStream(fi);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        try {
            xmlSchemaDocumentsLengths[0] = (int) fi.getChannel().size();
            System.out.println(xmlSchemaDocumentsLengths[0]);
        } catch (IOException e1) {
            e1.printStackTrace();
        }
        xmlSchemaDocumentsProperties[0] = null;
        xmlSchemaDocumentsPropertiesLengths[0] = 0;
        xmlSchemaProperties = null;
        xmlSchemaPropertiesLength = 0;
        DB2Connection ds = (DB2Connection) con;
        // Invoke registerDB2XmlSchema
        ds.registerDB2XmlSchema(
            xmlSchemaNameQualifiers,
            xmlSchemaNames,
            xmlSchemaLocations,
            xmlSchemaDocuments,
            xmlSchemaDocumentsLengths,
            xmlSchemaDocumentsProperties,
            xmlSchemaDocumentsPropertiesLengths,
            xmlSchemaProperties,
            xmlSchemaPropertiesLength,
            false);
    }
}
```

Example: Removal of an XML schema: The following example demonstrates the use of deregisterDB2XmlObject to remove an XML schema from DB2. The SQL schema name for the registered schema is SYSXSR.

```java
class Example {
    public static void deregisterSchema(
        Connection con,
        String schemaName)
    throws SQLException {
    }
}
```
// Define and assign values to the deregisterDB2XmlObject parameters
String xmlSchemaNameQualifier = "SYSXSR";
String xmlSchemaName = schemaName;
DB2Connection ds = (DB2Connection) con;
// Invoke deregisterDB2XmlObject
ds.deregisterDB2XmlObject(
    xmlSchemaNameQualifier,
    xmlSchemaName);
}

Example: Update of an XML schema: The following example applies only to connections to Db2 on Linux, UNIX, and Windows systems. It demonstrates the use of updateDB2XmlSchema to update the contents of an XML schema with the contents of another XML schema. The schema that is copied is kept in the repository. The SQL schema name for both registered schemas is SYSXSR.

public static void updateSchema(
    Connection con,
    String schemaNameTarget,
    String schemaNameSource)
    throws SQLException {
    // Define and assign values to the updateDB2XmlSchema parameters
    String xmlSchemaNameQualifierTarget = "SYSXSR";
    String xmlSchemaNameQualifierSource = "SYSXSR";
    String xmlSchemaNameTarget = schemaNameTarget;
    String xmlSchemaNameSource = schemaNameSource;
    boolean dropSourceSchema = false;
    DB2Connection ds = (DB2Connection) con;
    // Invoke updateDB2XmlSchema
ds.updateDB2XmlSchema(
    xmlSchemaNameQualifierTarget,
    xmlSchemaNameTarget,
    xmlSchemaNameQualifierSource,
    xmlSchemaNameSource,
    dropSourceSchema);
}

Bidirectional (BiDi) text layout transformation

The IBM BiDi (bidirectional) engine can perform BiDi Layout transformation on data being stored and retrieved from the database.

The IBM Bidi engine bases text transformations on the following characteristics of text data:

- Numeral shape (Arabic or Hindi)
- Orientation (right-to-left or left-to-right)
- Shaping (shaped or unshaped)
- Symmetric swapping (yes or no)
- Text type (logical or visual)

Enable Bidi text layout transformation for your database by setting the enableBidiLayoutTransformation, clientBidiStringType, and serverBidiStringType properties on a Connection or DataSource. Set enableBidiLayoutTransformation to true, then specify the String type of the client application and data server with the clientBidiStringType and serverBidiStringType properties respectively. The following table summarizes which string type values to specify for particular combinations of text data attributes.
### Table 20. String type values and corresponding attributes for the clientBidiStringType and serverBidiStringType properties

<table>
<thead>
<tr>
<th>String type value integer constant</th>
<th>String type value</th>
<th>Text Attributes</th>
<th>Type of text</th>
<th>Orientation</th>
<th>Symmetric swapping</th>
<th>Numeral shape</th>
<th>Text shapes</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>DB2Base DataSource. BIDI_ST4</td>
<td>Visual</td>
<td>Left-to-right</td>
<td>No</td>
<td>Nominal</td>
<td>Shaped</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>DB2Base DataSource. BIDI_ST5</td>
<td>Implicit</td>
<td>Left-to-right</td>
<td>Yes</td>
<td>Nominal</td>
<td>Nominal</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>DB2Base DataSource. BIDI_ST6</td>
<td>Implicit</td>
<td>Right-to-left</td>
<td>Yes</td>
<td>Nominal</td>
<td>Nominal</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>DB2Base DataSource. BIDI_ST7</td>
<td>Visual</td>
<td>Right-to-left</td>
<td>No</td>
<td>Nominal</td>
<td>Shaped</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>DB2Base DataSource. BIDI_ST8</td>
<td>Visual</td>
<td>Right-to-left</td>
<td>No</td>
<td>Nominal</td>
<td>Shaped</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>DB2Base DataSource. BIDI_ST9</td>
<td>Implicit</td>
<td>Right-to-left</td>
<td>Yes</td>
<td>Nominal</td>
<td>Shaped</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>DB2Base DataSource. BIDI_ST10</td>
<td>Implicit</td>
<td>Contextual left-to-right</td>
<td>Yes</td>
<td>Nominal</td>
<td>Nominal</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>DB2Base DataSource. BIDI_ST11</td>
<td>Implicit</td>
<td>Contextual right-to-left</td>
<td>Yes</td>
<td>Nominal</td>
<td>Nominal</td>
<td></td>
</tr>
</tbody>
</table>

BiDi text layout transformations can be performed on:

- Strings enclosed in single apostrophes in SQL statements:
  ```java
  ResultSet rs = st.executeQuery("SELECT * FROM PARAM.JCCTEST WHERE product= '<text_to_transform>'");
  ```

- Parameter values for PreparedStatements and CallableStatements:
  ```java
  Connection conn = null;
  PreparedStatement pstmt =
  conn.prepareStatement("SELECT PRODCUT FROM PARAM.JCCTEST WHERE PRODCUT=?");
  pstmt.setString(1, "<text_to_transform>");
  ```

- Values retrieved through getXXX() methods on ResultSet and CallableStatement objects:
  ```java
  ResultSet rs = pstmt.executeQuery();
  while (rs.next()){
    System.out.println(rs.getString(1));
    // BiDi layout transformation will be performed the on return value from getString()
  }
  ```

---

108 Application Programming Guide and Reference for Java
Inserting data from file reference variables into tables in JDBC applications

You can use file reference variable objects with IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 9 or later to stream LOB or XML input data.

Before you begin

You need to store your LOB or XML input data in HFS files.

About this task

Use of file reference variables eliminates the need to materialize the LOB or XML data in memory before the data is stored in tables.

Procedure

To use file reference variables to store LOB or XML data in tables, follow these steps:

1. Invoke the Connection.prepareStatement method to create a PreparedStatement object from an INSERT statement.
   The parameter markers in the INSERT statement represent XML or LOB values.

2. Execute constructors for file reference variable objects of the appropriate types.
   The following table lists the types of data in the input files and the appropriate constructors.

<table>
<thead>
<tr>
<th>Input data type</th>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>com.ibm.db2.jcc.DB2BlobFileReference</td>
</tr>
<tr>
<td>CLOB</td>
<td>com.ibm.db2.jcc.DB2ClobFileReference</td>
</tr>
<tr>
<td>XML AS BLOB</td>
<td>com.ibm.db2.jcc.DB2XmlAsBlobFileReference</td>
</tr>
<tr>
<td>XML AS CLOB</td>
<td>com.ibm.db2.jcc.DB2XmlAsClobFileReference</td>
</tr>
</tbody>
</table>

   The first parameter in each constructor must specify the absolute path name for an existing HFS file.

3. If you are performing single-row INSERT operations, repeat these steps for each row that you want to insert:
   a. Invoke DB2PreparedStatement.setXXX to pass values to the input variables. Alternatively, you can use PreparedStatement setObject methods.
      The following table lists the types of data in the input files and the appropriate DB2PreparedStatement.setXXX methods to use for each data type.

<table>
<thead>
<tr>
<th>Input data type</th>
<th>DB2PreparedStatement.setXXX method</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>setDB2BlobFileReference</td>
</tr>
<tr>
<td>CLOB</td>
<td>setDB2ClobFileReference</td>
</tr>
<tr>
<td>XML AS BLOB</td>
<td>setDB2XmlAsBlobFileReference</td>
</tr>
<tr>
<td>XML AS CLOB</td>
<td>setDB2XmlAsClobFileReference</td>
</tr>
</tbody>
</table>

   If you use DB2PreparedStatement methods, you need to cast the PreparedStatement object that you created in step 1 to a

Chapter 3. JDBC application programming 109
DB2PreparedStatement object when you execute a
DB2PreparedStatement.setXXX method.

You can assign NULL values to input parameters in any of the following
ways:

- Using DB2PreparedStatement.setXXX methods, with null as the fileRef parameter value.
- Using PreparedStatement.setObject, with null as the x (second) parameter value and the appropriate value from com.ibm.db2.jcc.DB2Types for the targetJdbcType (third) parameter value.
- Using PreparedStatement.setNull, with the appropriate value from com.ibm.db2.jcc.DB2Types for the JdbcType (second) parameter value.

b. Invoke the PreparedStatement.executeUpdate method to update the table with the
column values.

4. If you are performing multi-row INSERT operations, execute these steps:
   a. Repeat these steps for every row that you want to insert:
      1) Invoke DB2PreparedStatement.setXXX to pass values to the input
         variables. Alternatively, you can use PreparedStatement.setObject methods.
      2) Invoke the PreparedStatement.addBatch method after you set the values
         for a row of the table.
   b. Invoke the PreparedStatement.executeUpdate method to update the table
      with the variable values.

5. Invoke the PreparedStatement.close method to close the PreparedStatement
   object when you have finished using that object.

Examples

The following code inserts a single row into a table. The code inserts values from
CLOB and BLOB file reference variables into CLOB and BLOB columns and a
NULL value into an XML column. The numbers to the right of selected statements
 correspond to the previously-described steps.

Connection conn;
...
PreparedStatement pstmt =
    conn.prepareStatement("INSERT INTO TEST02TB(RECID,CLOBCOL,BLOBCOL,XMLCOL) VALUES('003',?,?,?)"); // Create a PreparedStatement object
com.ibm.db2.jcc.DB2ClobFileReference clobFileRef =
    new com.ibm.db2.jcc.DB2ClobFileReference("
/u/usr001/jcc/test/TEXT.FILE","Cp037");
com.ibm.db2.jcc.DBBlobFileReference blobFileRef =
    new com.ibm.db2.jcc.DBBlobFileReference("/u/usr001/jcc/test/BINARY.FILE");
com.ibm.db2.jcc.DBXMLAsBlobFileReference xmlAsBlobFileRef =
    new com.ibm.db2.jcc.DBXMLAsBlobFileReference("/u/usr001/jcc/test/XML.FILE");
    // Execute constructors for the file reference
    // variable objects
((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).setDB2ClobFileReference(1,clobFileRef);
((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).setDB2BlobFileReference(2,blobFileRef);
pstmt.setNull(3,com.ibm.db2.jcc.DB2Types.XML_AS_BLOB_FILE);
    // Assign values to the CLOB and BLOB parameters.
    // Assign a null value to the XML parameter.
int numUpd = pstmt.executeUpdate();
    // Perform the update
pstmt.close(); // Close the PreparedStatement object

10  Application Programming Guide and Reference for Java
The following code uses multi-row INSERT to insert two rows in a table. The code inserts values from XML AS CLOB and XML AS BLOB file reference variables into XML columns. The numbers to the right of selected statements correspond to the previously-described steps.

```java
Connection conn;
...
PreparedStatement pstmt =
    conn.prepareStatement(
        "INSERT INTO TEST03TB(RECID,XMLCLOBCOL,XMLBLOBCOL) VALUES('003',?,?)");
// Create a PreparedStatement object 1
com.ibm.db2.jcc.DB2XmlAsClobFileReference xmlAsClobFileRef1 =
    new com.ibm.db2.jcc.DB2XmlAsClobFileReference("/u/usrt001/jcc/test/XMLCLOB1.FILE","Cp037");
com.ibm.db2.jcc.DB2XmlAsBlobFileReference xmlAsBlobFileRef1 =
    new com.ibm.db2.jcc.DB2XmlAsBlobFileReference("/u/usrt001/jcc/test/XMLBLOB1.FILE");
com.ibm.db2.jcc.DB2XmlAsClobFileReference xmlAsClobFileRef2 =
    new com.ibm.db2.jcc.DB2XmlAsClobFileReference("/u/usrt001/jcc/test/XMLCLOB2.FILE","Cp037");
com.ibm.db2.jcc.DB2XmlAsBlobFileReference xmlAsBlobFileRef2 =
    new com.ibm.db2.jcc.DB2XmlAsBlobFileReference("/u/usrt001/jcc/test/XMLBLOB2.FILE");
// Execute constructors for the file reference 2
// variable objects
((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).setDB2ClobFileReference(1,xmlAsClobFileRef1);
((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).setDB2BlobFileReference(2,xmlAsBlobFileRef1);
// Assign first set of values to the 4ai
// XML parameters
pstmt.addBatch();
// Add the first input parameters to the batch 4aii
((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).setDB2ClobFileReference(1,xmlAsClobFileRef2);
((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).setDB2BlobFileReference(2,xmlAsBlobFileRef2);
// Assign second set of values to the 4ai
// XML parameters
pstmt.addBatch();
// Add the second input parameters to the batch 4aii
int [] numUpd = pstmt.executeBatch();
// Perform the update 4b
pstmt.close(); // Close the PreparedStatement object 5
```

**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>
<thead>
<tr>
<th>JDBC value</th>
<th>DB2 isolation level</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.sql.Connection.TRANSACTION_SERIALIZABLE</code></td>
<td>Repeatable read</td>
</tr>
<tr>
<td><code>java.sql.Connection.TRANSACTION_REPEATABLE_READ</code></td>
<td>Read stability</td>
</tr>
<tr>
<td><code>java.sql.Connection.TRANSACTION_READ_COMMITTED</code></td>
<td>Cursor stability</td>
</tr>
<tr>
<td><code>java.sql.Connection.TRANSACTION_READ_UNCOMMITTED</code></td>
<td>Uncommitted read</td>
</tr>
</tbody>
</table>

Chapter 3. JDBC application programming 111
The following table shows the values of `level` that you can specify in the `Connection.setTransactionIsolation` method and their IBM Informix equivalents.

**Table 22. Equivalent JDBC and IBM Informix isolation levels**

<table>
<thead>
<tr>
<th>JDBC value</th>
<th>IBM Informix isolation level</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.sql.Connection.TRANSACTION_SERIALIZABLE</code></td>
<td>Repeatable read</td>
</tr>
<tr>
<td><code>java.sql.Connection.TRANSACTION_REPEATABLE_READ</code></td>
<td>Repeatable read</td>
</tr>
<tr>
<td><code>java.sql.Connection.TRANSACTION_READ_COMMITTED</code></td>
<td>Committed read</td>
</tr>
<tr>
<td><code>java.sql.Connection.TRANSACTION_READ_UNCOMMITTED</code></td>
<td>Dirty read</td>
</tr>
<tr>
<td><code>com.ibm.db2.jcc.DB2Connection.TRANSACTION_IDS_CURSOR_STABILITY</code></td>
<td>IBM Informix cursor stability</td>
</tr>
<tr>
<td><code>com.ibm.db2.jcc.DB2Connection.TRANSACTION_IDS_LAST_COMMITTED</code></td>
<td>Committed read, last committed</td>
</tr>
</tbody>
</table>

**Related concepts:**

"JDBC connection objects" on page 23

**Committing or rolling back JDBC transactions**

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

**About this task**

For example:

```java
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 or global transaction, the associated application cannot issue the `commit` or `rollback` methods.

While a connection is participating in a global transaction, the associated application cannot invoke the `setAutoCommit(true)` method.

**Related concepts:**

"Savepoints in JDBC applications" on page 75

**Related tasks:**

"Disconnecting from data sources in JDBC applications“ on page 123

"Making batch updates in JDBC applications” on page 33

**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>
<thead>
<tr>
<th>Type of data source</th>
<th>Default autocommit mode for local transactions</th>
<th>Default autocommit mode for global transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI-compliant database</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>Non-ANSI-compliant database without logging</td>
<td>false</td>
<td>not applicable</td>
</tr>
<tr>
<td>Non-ANSI-compliant database with logging</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

**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.
BatchUpdateException

A BatchUpdateException object contains the following items about an error that occurs during execution of a batch of SQL statements:
- A String object that contains a description of the error, or null.
- A String object that contains the SQLSTATE for the failing SQL statement, or null
- An integer value that contains the error code, or zero
- An integer array of update counts for SQL statements in the batch, or null
- A pointer to an SQLException object, or null

One BatchUpdateException is thrown for the entire batch. At least one SQLException object is chained to the BatchUpdateException object. The SQLException objects are chained in the same order as the corresponding statements were added to the batch. To help you match SQLException objects to statements in the batch, the error description field for each SQLException object begins with this string:
Error for batch element #n:

n is the number of the statement in the batch.

SQL warnings during batch execution do not throw BatchUpdateExceptions. To obtain information about warnings, use the Statement.getWarnings method on the object on which you ran the executeBatch method. You can then retrieve an error description, SQLSTATE, and error code for each SQLWarning object.

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 information specific to DB2. The information specific to DB2, for an SQLWarning object, is the same as the 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.

Procedure

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:
   
   ```java
   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.

      ```java
      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.getSql1Code` 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`.

Example

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.

```
import java.sql.*; // Import JDBC API package
import com.ibm.db2.jcc.DB2Diagnosable; // Import packages for DB2
import com.ibm.db2.jcc.DB2Sqlca; // SQLException support
java.io.PrintWriter printWriter; // For dumping all SQLException information

String url = "jdbc:db2://myhost:9999/myDB:" +
    "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 {
    // Code that could generate SQLExceptions

    } catch(SQLException sqle) {
        while(sqle != null) {
            // Check whether there are more
```

Figure 20. Processing an SQLException under the IBM Data Server Driver for JDBC and SQLJ

Chapter 3. JDBC application programming 117
Optional IBM Data Server Driver for JDBC and SQLJ-only error processing

```java
// 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;
diagnosable.printStackTrace (printWriter, "");
java.lang.Throwable throwable = diagnosable.getThrowable();
if (throwable != null) {
    // Extract java.lang.Throwable information such as message or stack trace...

    DB2Sqlca sqlca = diagnosable.getSqlca();
    // Get DB2Sqlca object
    if (sqlca != null) {
        // Check that DB2Sqlca is not null
        int sqlCode = sqlca.getSqlCode(); // Get the SQL error code
        String sqlErrmc = sqlca.getSqlErrmc();
        // Get the entire SQLERRMC
        String[] sqlErrmcTokens = sqlca.getSqlErrmcTokens();
        // You can also retrieve the individual SQLERRMC tokens
        String sqlErrp = sqlca.getSqlErrp();
        // Get the SQLERRP
        int[] sqlErrd = sqlca.getSqlErrd();
        // Get SQLERRD fields
        char[] sqlWarn = sqlca.getSqlWarn();
        // Get SQLWARN fields
        String sqlState = sqlca.getSqlState();
        // Get SQLSTATE
        String errMessage = sqlca.getMessage();
        // 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);
        System.err.println ("SQLERRMC Tokens: " + Arrays.toString(sqlErrmcTokens));
        System.err.println ("SQLERRP: " + sqlErrp);
        System.err.println ("SQLERRD: " + Arrays.toString(sqlErrd));
        System.err.println ("SQLWARN: " + Arrays.toString(sqlWarn));
        System.err.println ("SQLSTATE: " + sqlState);
    }
}
```

// portion of SQLException
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.

Procedure

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 specific warning information in DB2, perform the same steps that you perform to get specific information for an SQLException in DB2.
   f. Invoke the SQLWarning.getNextWarning method to retrieve the next SQLWarning.
Example

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

```java
String url = "jdbc:db2://myhost:9999/myDB:" + 
        "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
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
while (sqlwarn != null) {        // While there are warnings, get and
        // print warning information
            System.out.println("Warning description: " + sqlwarn.getMessage());  // Warning description:
            System.out.println("SQLSTATE: " + sqlwarn.getSQLState());  // SQLSTATE:
            System.out.println("Error code: " + sqlwarn.getErrorCode());  // Error code:
            sqlwarn = sqlwarn.getNextWarning();  // Get next SQLWarning
}
```

Figure 21. 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.

Procedure

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.

   ```java
   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.

Example

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.

```java
try {
    // Batch updates
    catch(BatchUpdateException buex) {
        System.err.println("Contents of BatchUpdateException:");
        System.err.println(" Update counts: ");
        int[] updateCounts = buex.getUpdateCounts();
        for (int i = 0; i < updateCounts.length; i++) {
            System.err.println(" Statement "+i+:" + updateCounts[i]);
        }
        ResultSet[] resultList = ((DBBatchUpdateException)buex).getDBGeneratedKeys();
        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());
        System.err.println(" SQLSTATE: "+ buex.getSQLState());
        System.err.println(" Error code: "+ buex.getErrorCode());
    }  
    SQLException ex = buex.getNextException();
    while (ex != null) {
```
In general, applications that use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity require more memory than applications that use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

With IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, an application receives data from the DB2 database server in network packets, and receives only the data that is contained in a particular row and column of a table.

Applications that run under IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS generally require more memory. IBM Data Server Driver for JDBC and SQLJ type 2 connectivity has a direct, native interface to DB2 for z/OS. For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity, the driver must provide memory in which DB2 for z/OS writes data. Because the amount of data that is needed can vary from row to row, and the driver has no information about how much memory is needed for each row, the driver must allocate the maximum amount of memory that any row might need. This value is determined from DESCRIBE information on the SELECT statement that generates the result table. For example, when an application that uses IBM Data Server Driver for JDBC and SQLJ type 2 connectivity selects a column that is defined as VARCHAR(32000), the driver must allocate 32000 bytes for each row of the result table.

The extra memory requirements can be particularly great for retrieval of LOB columns, which can be defined with lengths of up to 2 GB, or for CAST expressions that cast values to LOB types with large length attributes.

In general, even when you use a 64-bit JVM, all native connectivity to DB2 for z/OS is below the bar, with 32-bit addressing limits. Although the maximum size of any row is defined as approximately 2 GB, the practical maximum amount of available memory for use by IBM Data Server Driver for JDBC and SQLJ type 2 connectivity is generally significantly less. However, if the IBM Data Server Driver for JDBC and SQLJ can use limited block fetch to retrieve the data for a query, the data can be passed to the driver using full 64-bit addressing.

Two ways to alleviate excess memory use for LOB retrieval and manipulation are to use progressive streaming or LOB locators. You enable progressive streaming or LOB locator use by setting the progressiveStreaming property or the fullyMaterializeLobData property.

Related concepts:

"LOBs in JDBC applications with the IBM Data Server Driver for JDBC and SQLJ" on page 59

Related reference:
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.

Procedure

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.
For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, when you close the connection to a data source, the driver issues an implicit rollback to ensure consistency of the underlying RRSAF thread before thread termination.

Related concepts:

“How JDBC applications connect to a data source” on page 11
Chapter 4. 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 data server 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 22. Simple SQLJ application

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

#sql context EzSqljCtx;
#sql iteratr EzSqljNameIter (String LASTNAME);
public class EzSqlj {
    public static void main(String args[]) throws SQLException {
        EzSqljCtx ctx = null;
        String URLprefix = "jdbc:db2:"
        String url;
        url = new String(URLprefix + args[0]);
        // Location name is an input parameter
        String hvmgr="000010";
        String hvdeptno="A00";
        try {
            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);
                // Create a JDBC Connection
                con0.setAutoCommit(false);
                // set autocommit OFF
                ctx = new EzSqljCtx(con0);
                try {
                    // do something
                }
            }
        }
    } }
EzSqljNameIter iter;
int count=0;

#sql [ctx] iter =
( SELECT LASTNAME FROM EMPLOYEE);
   // Create result table of the SELECT
while (iter.next()) {
   System.out.println(iter.LASTNAME());
   count++;
}
System.out.println("Retrieved "+ count + " rows of data");
iter.close();  // Close the iterator
}
catch( SQLException e )
{
   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] {UPDATE DEPARTMENT SET MGRNO=:hvmgr
      WHERE DEPTNO=:hvdeptno};  // Update data for one department
   #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();
}
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 22 on page 125
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 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 or later.
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 a database application in any other language, you must be connected to a data source before you can execute SQL statements.

About this task

You can use one of six techniques to connect to a data source in an SQLJ program. Two use the JDBC DriverManager interface, two use the JDBC DataSource interface, one uses a previously created connection context, and one uses the default connection.

Related concepts:

"How JDBC applications connect to a data source" on page 11

SQLJ connection technique 1: JDBC DriverManager interface

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

Procedure

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.
• Invoke Class.forName this way:

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

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

3. Invoke the constructor for the connection context class that you created in step 1 on page 127.

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:

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

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

```java
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.

  If the data source is a DB2 for z/OS system, and you do not specify these parameters, DB2 uses the external security environment, such as the RACF security environment, that was previously established for the user. For a CICS connection, you cannot specify a user ID or password.

- **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.

**Example**

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.
SQLJ connection technique 2: JDBC DriverManager interface

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

Procedure

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.

   • Invoke `Class.forName` this way:
     
     ```java
     Class.forName("com.ibm.db2.jcc.DB2Driver");
     ```

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

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.

---

```sql
#sql context Ctx;
// Create connection context class Ctx
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");
}
catch (ClassNotFoundException e) {
    e.printStackTrace();
}
Ctx myConnCtx=
    new Ctx("jdbc:db2://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 23. Using connection technique 1 to connect to a data source
If the data source is a DB2 for z/OS system, and you do not specify these parameters, DB2 uses the external security environment, such as the RACF security environment, that was previously established for the user. For a CICS connection, you cannot specify a user ID or password.

4. **Invoke the constructor for the connection context class that you created in step 1 on page 129.**

   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:

   ```java
   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 129.

**Example**

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.

```java
#sql context Ctx;    // Create connection context class Ctx
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=
    DriverManager.getConnection("jdbc:db2://sysmvs1.stl.ibm.com:5021/NEWYORK",
                          userid,password); // Create JDBC connection object jdbccon
jdbccon.setAutoCommit(false); // Do not autocommit
Ctx myConnCtx=new Ctx(jdbccon); // 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 24. 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.

**Procedure**

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
   #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`:
   ```java
   getConnection();
   getConnection(user, password);
   ```
   The meanings of the `user` and `password` parameters are:
   ```markdown
   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.

   If the data source is a DB2 for z/OS system, and you do not specify these parameters, DB2 uses the external security environment, such as the RACF security environment, that was previously established for the user. For a CICS connection, you cannot specify a user ID or password.

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:
   ```java
   setAutoCommit(boolean autocommit);
   ```
   For environments other than the environments for CICS, stored procedures, and user-defined functions, the default autocommit mode for a JDBC connection is `true`. To disable autocommit, invoke `setAutoCommit(false)`.

5. Invoke the constructor for the connection context class that you created in step [1 on page 130](#).
   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:
   ```java
   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](#).

**Example**

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.*;
...

```java
#sql context CtxSqlj; // Create connection context class CtxSqlj
Context ctx=new InitialContext();
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");
Connection con=ds.getConnection();
String empname; // Declare a host variable
... con.setAutoCommit(false); // Do not autocommit
CtxSqlj myConnCtx=new CtxSqlj(con);
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
WHERE EMPNO='000010'};
// Use myConnCtx for executing an SQL statement
```

Figure 25. 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.

#### Procedure

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:
   ```java
   #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:
   ```java
   connection-context-class connection-context-object=
   new connection-context-class();
   ```
   ```java
   connection-context-class connection-context-object=
   new connection-context-class (String user,
   String password);
   ```
   
   The meanings of the *user* and *password* parameters are:
   ```java
   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.
   
   If the data source is a DB2 for z/OS system, and you do not specify these parameters, DB2 uses the external security environment, such as the RACF...
security environment, that was previously established for the user. For a CICS connection, you cannot specify a user ID or password.

Example

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.

```java
#sql public static context Ctx
    with (dataSource="jdbc/sampledb");
// 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);
// 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 26. 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.

About this task

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.

Example

Program CtxGen.sqlj declares connection context Ctx and creates instance oldCtx:

```java
#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:

```java
void useContext(sqlj.runtime.ConnectionContext oldCtx)
{  
    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
    ...
}
SQLJ connection technique 6: Use the default connection

SQLJ connection technique 6 uses the default connection to connect to the data source. It should be used only in situations where the database thread is controlled by another resource manager, such as the Java stored procedure environment.

About this task

You use the default connection by specifying your SQL statements without a connection context object. When you use this technique, you do not need to load a JDBC driver unless you explicitly use JDBC interfaces in your program.

The default connection context can be:

- The connection context that is associated with the data source that is bound to the logical name jdbc/defaultDataSource
- An explicitly created connection context that has been set as the default connection context with the ConnectionContext.setDefaultContext method. This method of creating a default connection context is not recommended.

In a stored procedure that runs on DB2 for z/OS, or for a CICS or IMS application, when you use the default connection, DB2 for z/OS uses the implicit connection.

Example

The following SQLJ execution clause does not have a connection context, so it uses the default connection context.

```sql
#sql (SELECT LASTNAME INTO :empname FROM EMPLOYEE
  WHERE EMPNO='000010'); // Use default connection 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 application programs in other languages, you use host variables to pass data between the application program and the data server. 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.

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.

Examples

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.

```java
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 the data server.
• 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.

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.

<table>
<thead>
<tr>
<th>Indicator value</th>
<th>Equivalent constant</th>
<th>Meaning of value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>sqlj.runtime.ExecutionContext.DBNull</td>
<td>Null</td>
</tr>
<tr>
<td>-2, -3, -4, -6</td>
<td></td>
<td>Null</td>
</tr>
<tr>
<td>-5</td>
<td>sqlj.runtime.ExecutionContext.DBDefault</td>
<td>Default</td>
</tr>
<tr>
<td>-7</td>
<td>sqlj.runtime.ExecutionContext.DBUassigned</td>
<td>Unassigned</td>
</tr>
<tr>
<td>short-value &gt;=0</td>
<td>sqlj.runtime.ExecutionContext.DBNNull</td>
<td>Non-null</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Indicator value</th>
<th>Equivalent constant</th>
<th>Meaning of value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>sqlj.runtime.ExecutionContext.DBNull</td>
<td>Null</td>
</tr>
<tr>
<td>-7 &lt;= short-value &lt; -1</td>
<td></td>
<td>Null</td>
</tr>
<tr>
<td>0</td>
<td>sqlj.runtime.ExecutionContext.DBNNull</td>
<td>Non-null</td>
</tr>
<tr>
<td>short-value &gt;0</td>
<td></td>
<td>Non-null</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Indicator value</th>
<th>Equivalent constant</th>
<th>Meaning of value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>sqlj.runtime.ExecutionContext.DBNull</td>
<td>Retrieved value is SQL NULL</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Retrieved value is not SQL NULL</td>
</tr>
</tbody>
</table>

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.

```java
import sqlj.runtime.*;
...
String dept = "F01";
String dName = "SHIPPING";
String rptDept = "A00";
String mgr, locn = null;
short deptInd, dNameInd, rptDeptInd, locnInd;
// Set indicator variables for dept, dName, rptDept to non-null
deptInd = dNameInd = rptDeptInd = ExecutionContext.DBNonNull;
mgrInd = ExecutionContext.DBDefault;
locnInd = ExecutionContext.DBDefault;
#sql [ctxt]
{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.

```java
import sqlj.runtime.*;
...
String dept = null;
String dName = "RECEIVING";
String rptDept = null;
String mgr, locn = null;
short deptInd, dNameInd, mgrInd, rptDeptInd, locnInd;
dNameInd = ExecutionContext.DBNonNull;
mgrInd = ExecutionContext.DBDefault;
```
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.

```java
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
  #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);
  // Retrieve the first row
  while (!positer.endFetch()) // Check whether the FETCH returned a row
  {
    if(indhrdate == ExecutionContext.DBNonNull {
      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
}
```

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
#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. You can use Java comments outside of SQLJ statements and SQL or Java comments in SQLJ statements.

You can include Java comments outside SQLJ clauses, wherever the Java language permits them. Within an SQLJ clause, you can use comments in the following places:

- Within a host expression (enclosed in /* and */ or preceded by //).
- Within an SQL statement in an executable clause, if the data server supports a comment within the SQL statement.
  - For connections to DB2 data servers or Informix data servers, comments can be:
    - Anywhere in the SQL statement text, and enclosed in /* and */ pairs. /* and */ pairs can be nested.
    - At the end of the SQL statement text, and preceded by two hyphens (--).
  - For connections to Informix data servers only, comments can be enclosed in left curly bracket ({{) and right curly bracket (}}) pairs.

**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
#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.

**Example**

The following executable statements demonstrate an INSERT, a searched UPDATE, and a searched DELETE:

```sql
#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 database applications in other languages, performing positioned UPDATEs and DELETEs with SQLJ is an extension of retrieving rows from a result table.

**Procedure**

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:

   ```java
   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.

   ```java
   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.

The authorization ID under which a positioned UPDATE or DELETE statement executes depends on whether the statement executes statically or dynamically.

If the statement executes statically, the authorization ID is the owner of the plan or package that includes the statement. If the statement executes dynamically...
the authorization ID is determined by the DYNAMICRULES behavior that is in effect. For the IBM Data Server Driver for JDBC and SQLJ, the behavior is always DYNAMICRULES BIND.

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.

Example

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:
Then, in another file, use `UpdByPos` for a positioned UPDATE, as shown in the following code fragment:

```java
import java.math.*; // Import this class for BigDecimal data type
import sqlj.runtime.*; // Import files for SQLJ and JDBC APIs
import java.sql.*; // 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

public context HSCtx; // Create a connection 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:db2:SANJOSE"); // Create a JDBC connection object
    HSjdbccon.setAutoCommit(false); // Set autocommit off so automatic commits 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
    String empnum; // Declares host variable to receive EMPNO
    BigDecimal sal; // and SALARY column values

    upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE
                WHERE WORKDEPT='D11'};
    // Assign result table to iterator object
    MY {FETCH :upditer INTO :empnum,:sal}; // Move cursor to next row
    while (!upditer.endFetch()) { // Check if on a row
        MY {UPDATE EMPLOYEE SET SALARY=SALARY*1.05
             WHERE CURRENT OF :upditer}; // Perform positioned update
        System.out.println("Updating row for " + empnum);
        MY {FETCH :upditer INTO :empnum,:sal}; // Move cursor to next row
    }
    upditer.close(); // Close the iterator
    MY {COMMIT}; // Commit the changes
    myConnCtx.close(); // Close the connection context
}
```

Figure 28. 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:

```java
import java.math.*; // Import this class for BigDecimal data type
$sql public iterator UpdByName implements sqlj.runtime.ForUpdate with(updateColumns="SALARY") (String EmpNo, BigDecimal Salary);
```

*Figure 29. 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:

```java
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 connection 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:db2:SANJOSE");
  // Create a JDBC connection object
  HSjdbccon.setAutoCommit(false);
  // Set autocommit off so automatic commits do not destroy the cursor between updates
  HSCtx myConnCtx=new HSCtx(HSjdbccon);
  // Create a connection context object
  UpdByName upditer; // Declare iterator object of UpdByName class

  String empnum; // Declare host variable to receive EmpNo column values

  $sql [myConnCtx]
  upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE
  WHERE WORKDEPT='D11'};
  // Assign result table to iterator object

  while (upditer.next()) // Move cursor to next row and check if on a row
  {
    empnum = upditer.EmpNo(); // Get employee number from current row

    $sql [myConnCtx]
    (UPDATE EMPLOYEE SET SALARY=SALARY*1.05
    WHERE CURRENT OF :upditer);
    // Perform positioned update
    System.out.println("Updating row for: "+ empnum);
  }

  upditer.close(); // Close the iterator

  $sql [myConnCtx] {COMMIT}; // Commit the changes
  myConnCtx.close(); // Close the connection context
}
```

*Figure 30. Example of performing a positioned UPDATE with a named iterator*

**Related concepts:**

"Iterators as passed variables for positioned UPDATE or DELETE operations in an SQLJ application" on page 145
Iterators as passed variables for positioned UPDATE or DELETE operations in an SQLJ application

SQLJ allows iterators to be passed between methods as variables.

An iterator that is used for a positioned UPDATE or DELETE statement can be identified only at runtime. The same SQLJ positioned UPDATE or DELETE statement can be used with different iterators at runtime. If you specify a value of YES for -staticpositioned when you customize your SQLJ application as part of the program preparation process, the SQLJ customizer prepares positioned UPDATE or DELETE statements to execute statically. In this case, the customizer must determine which iterators belong with which positioned UPDATE or DELETE statements. The SQLJ customizer does this by matching iterator data types to data types in the UPDATE or DELETE statements. However, if there is not a unique mapping of tables in UPDATE or DELETE statements to iterator classes, the SQLJ customizer cannot determine exactly which iterators and UPDATE or DELETE statements go together. The SQLJ customizer must arbitrarily pair iterators with UPDATE or DELETE statements, which can sometimes result in SQL errors. The following code fragments illustrate this point.

```java
#sql iterator GeneralIter implements sqlj.runtime.ForUpdate
   (String);

public static void main (String args[])
{
   ...
   GeneralIter iter1 = null;
   #sql [ctxt] iter1 = { SELECT CHAR_COL1 FROM TABLE1 };
   
   GeneralIter iter2 = null;
   #sql [ctxt] iter2 = { SELECT CHAR_COL2 FROM TABLE2 };
   
   doUpdate ( iter1 );
}

public static void doUpdate (GeneralIter iter)
{
   #sql [ctxt] { UPDATE TABLE1 ... WHERE CURRENT OF :iter };
}
```

Figure 31. Static positioned UPDATE that fails

In this example, only one iterator is declared. Two instances of that iterator are declared, and each is associated with a different SELECT statement that retrieves data from a different table. During customization and binding with -staticpositioned YES, SQLJ creates two DECLARE CURSOR statements, one for each SELECT statement, and attempts to bind an UPDATE statement for each cursor. However, the bind process fails with SQLCODE -509 when UPDATE TABLE1 ... WHERE CURRENT OF :iter is bound for the cursor for SELECT CHAR_COL2 FROM TABLE2 because the table for the UPDATE does not match the table for the cursor.

You can avoid a bind time error for a program like the one in Figure 31 by specifying the bind option SQLERROR(CONTINUE). However, this technique has
the drawback that it causes the database manager to build a package, regardless of
the SQL errors that are in the program. A better technique is to write the program
so that there is a one-to-one mapping between tables in positioned UPDATE or
DELETE statements and iterator classes. Figure 32 shows an example of how to do
this.

```java
#sql iterator Table2Iter(String);
#sql iterator Table1Iter(String);
    public static void main ( String args[] )
    {
        ...  
        Table2Iter iter2 = null;  
        #sql [ctxt] iter2 = { SELECT CHAR_COL2 FROM TABLE2 };

        Table1Iter iter1 = null;  
        #sql [ctxt] iter1 = { SELECT CHAR_COL1 FROM TABLE1 };

        ...
        doUpdate(iter1);
    } 

    public static void doUpdate ( Table1Iter iter )
    {  
        ...  
        #sql [ctxt] { UPDATE TABLE1 ... WHERE CURRENT OF :iter };
        ...  
    }

    public static void doUpdate ( Table2Iter iter )
    {  
        ...  
        #sql [ctxt] { UPDATE TABLE2 ... WHERE CURRENT OF :iter };
        ...  
    }
```

Figure 32. Static positioned UPDATE that succeeds

With this method of coding, each iterator class is associated with only one table.
Therefore, the data server bind process can always associate the positioned
UPDATE statement with a valid 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.

**About this task**

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.

**Procedure**

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.

```java
#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);
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);

#sql [c1] deptiter =
{SELECT MGRNO FROM DEPARTMENT};  // Assign the result table of the SELECT
  // to iterator object deptiter
#sql [c1] (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]
  {UPDATE EMPLOYEE SET SALARY=:(currentSalary+raise)
   WHERE EMPNO=:mgrnum};
  #sql [c1, ec] (FETCH :deptiter INTO :mgrnum );  // Fetch the next row
}
ec.executeBatch();
ect setBatching(false);
#sql [c1] {COMMIT};
depiter.close();  // Close the iterator
ec.close();  // Close the connection
}
```
The following example demonstrates batching of INSERT statements. Suppose that ATOMICCTBL is defined like this:

```sql
CREATE TABLE ATOMICCTBL(
    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.

```java
... 
TestContext ctx = new TestContext(url, username, password, false);
ctx.getExecutionContext().setBatching(true);
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 ATOMICCTBL values(:intVar, :strVar)};
        }
        int[] counts = ctx.getExecutionContext().executeBatch();
        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();
        }
    }
}
```

Related tasks:

"Controlling the execution of SQL statements in SQLJ" on page 170
"Connecting to a data source using SQLJ" on page 127

Related reference:

"sqlj.runtime.SQLNullException class" on page 381
"db2sqljcustomize - SQLJ profile customizer" on page 502
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 database 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
#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 database 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. Positioned 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.

Procedure

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 column data types. See "Java, JDBC, and SQL data types" for a list of the best mappings between Java data types and table column 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.

Example

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
#sql iterator ByName(String LastName, Date HireDate);
   // Declare named iterator ByName
{
   ... 
   ByName nameiter;  // Declare object of ByName class
   #sql [ctxt]
   nameiter={SELECT LASTNAME, HIREDATE FROM EMPLOYEE}; 
   // Assign the result table of the SELECT 
   // to iterator object nameiter
   while (nameiter.next()) // Move the iterator through the result 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
}
```

Figure 33. Example of using a named iterator

Related tasks:

“Performing positioned UPDATE and DELETE operations in an SQLJ application”
on page 141

“Using a positioned iterator in an SQLJ application”

**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.

**Procedure**

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 column data types. See "Java, JDBC, and SQL data types" for a list of the best mappings between Java data types and column 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
#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.

Example

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
#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}; // Assign the result table of the SELECT // to iterator object positer

 #sql {FETCH :positer INTO :name, :hrdate };
 // 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 34. Example of using a positioned iterator

Related concepts:
"Data retrieval in SQLJ applications" on page 151

Related tasks:
"Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 141
"Using a named iterator in an SQLJ application" on page 151

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.
For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS, support for multiple open iterators on a single SQL statement must be enabled. To do that, set the db2.jcc.allowSqljDuplicateStaticQueries configuration property to YES or true.

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:

```java
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 35.

```java
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 35. Example of concurrent table operations using iterators with different SQL statements

Figure 36 on page 157 demonstrates how you can perform the same operations when you have the capability for multiple, concurrently open iterators for a single 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.

```java
public MultiIter openIter(String EmpNo) {
    MultiIter iter;
    #sql [ctxt]
        iter = {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo};
        return iter;
    #endsql
    // Method returns an iterator instance
}
```

Figure 36. 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.

```java
public MultiIter openIter(String EmpNo) {
    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();
    ...
}
```

Figure 37. 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.
About this task

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 can be insensitive or sensitive. A sensitive scrollable iterator can be static or dynamic. Insensitive means that changes to the underlying table after the iterator is opened are not visible to the iterator. Insensitive iterators are read-only. Sensitive means that changes that the iterator or other processes make to the underlying table are visible to the iterator. Asensitive means that if the cursor is a read-only cursor, it behaves as an insensitive cursor. If it is not a read-only cursor, it behaves as a sensitive cursor.

If a scrollable iterator is static, the size of the result table and the order of the rows in the result table do not change after the iterator is opened. This means that you cannot insert into result tables, and if you delete a row of a result table, a delete hole occurs. If you update a row of the result table so that the row no longer qualifies for the result table, an update hole occurs. Fetching from a hole results in an SQLException.

Important: Like static scrollable cursors in any other language, SQLJ static scrollable iterators use declared temporary tables for their internal processing. This means that before you can execute any applications that contain static scrollable iterators, your database administrator needs to create a temporary database and temporary table spaces for those declared temporary tables.

If a scrollable iterator is dynamic, the size of the result table and the order of the rows in the result table can change after the iterator is opened. Rows that are inserted or deleted with INSERT and DELETE statements that are executed by the same application process are immediately visible. Rows that are inserted or deleted with INSERT and DELETE statements that are executed by other application processes are visible after the changes are committed.

Important: Db2 on Linux, UNIX, and Windows systems servers do not support dynamic scrollable cursors. You can use dynamic scrollable iterators in your SQLJ applications only if those applications access data on DB2 for z/OS servers, at Version 9 or later.

Procedure

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=sensitivity-attribute) or with (sensitivity=sensitivity-attribute, dynamic=true|false)`
     `sensitivity-attribute` indicates whether update or delete operations on the underlying table can be visible to the iterator. Possible values are `sqlj.runtime.ResultSetIteratorSENSITIVE`, `sqlj.runtime.ResultSetIteratorINSENSITIVE`, or `sqlj.runtime.ResultSetIteratorASENSITIVE`. `sqlj.runtime.ResultSetIteratorASENSITIVE` is the default.
dynamic=true|false indicates whether the size of the result table or the order of the rows in the result table can change after the iterator is opened. The default value of dynamic is false.

The iterator can be a named or positioned iterator.

**Example:** The following iterator declaration clause declares a positioned, sensitive, dynamic, scrollable iterator:

```sql
public iterator ByPos
    implements sqlj.runtime.Scrollable
    with (sensitivity=sqlj.runtime.ResultSetIterator.SENSITIVE, dynamic=true) (String);
```

**Example:** The following iterator declaration clause declares a named, insensitive, scrollable iterator:

```sql
public iterator ByName
    implements sqlj.runtime.Scrollable
    with (sensitivity=sqlj.runtime.ResultSetIterator.INSENSITIVE) (String EmpNo);
```

**Restriction:** You cannot use a scrollable iterator to select columns with the following data types from a table on a Db2 on Linux, UNIX, and Windows systems 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. If you want to give the SQLJ runtime environment a hint about the initial fetch direction, use the `setFetchDirection(int direction)` method. `direction` can be `FETCH_FORWARD` or `FETCH_REVERSE`. If you do not invoke `setFetchDirection`, the fetch direction is `FETCH_FORWARD`.

4. 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 24. `sqlj.runtime.Scrollable` methods for positioning a scrollable cursor

<table>
<thead>
<tr>
<th>Method</th>
<th>Positions the cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>first</code></td>
<td>On the first row of the result table</td>
</tr>
<tr>
<td><code>last</code></td>
<td>On the last row of the result table</td>
</tr>
<tr>
<td><code>previous</code></td>
<td>On the previous row of the result table</td>
</tr>
<tr>
<td><code>next</code></td>
<td>On the next row of the result table</td>
</tr>
<tr>
<td><code>absolute(int n)</code></td>
<td>If <code>n</code> is positive, on row <code>n</code> of the result table. If <code>n</code> is negative, on row <code>m+n+1</code> of the result table.</td>
</tr>
<tr>
<td><code>relative(int n)</code></td>
<td>If <code>n</code> is positive, on the row that is <code>n</code> rows after the current row. If <code>n</code> is negative, on the row that is <code>n</code> rows before the current row.</td>
</tr>
<tr>
<td><code>afterLast</code></td>
<td>After the last row in the result table</td>
</tr>
<tr>
<td><code>beforeFirst</code></td>
<td>Before the first row in the result table</td>
</tr>
</tbody>
</table>
Table 24. sqlj.runtime.Scrollable methods for positioning a scrollable cursor (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Positions the cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notes:</td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>This method does not apply to connections to IBM Informix.</td>
</tr>
<tr>
<td>2.</td>
<td>If the cursor is after the last row of the result table, this method positions the cursor on the last row.</td>
</tr>
<tr>
<td>3.</td>
<td>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.</td>
</tr>
<tr>
<td>4.</td>
<td>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&gt;0 ) and ( x+n&gt;m ), the iterator is positioned after the last row. If ( n&lt;0 ) and ( x+n&lt;1 ), the iterator is positioned before the first row.</td>
</tr>
</tbody>
</table>

b. If you need to know the current cursor position, use the \( \text{getRow} \), \( \text{isFirst} \), \( \text{isLast} \), \( \text{isBeforeFirst} \), or \( \text{isAfterLast} \) method to obtain this information.

If you need to know the current fetch direction, invoke the \( \text{getFetchDirection} \) 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 \( \text{getWarnings} \) 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 25 lists the clauses that you can use to position the cursor.

Table 25. FETCH clauses for positioning a scrollable cursor

<table>
<thead>
<tr>
<th>Method</th>
<th>Positions the cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST</td>
<td>On the first row of the result table</td>
</tr>
<tr>
<td>LAST</td>
<td>On the last row of the result table</td>
</tr>
<tr>
<td>PRIOR</td>
<td>On the previous row of the result table</td>
</tr>
<tr>
<td>NEXT</td>
<td>On the next row of the result table</td>
</tr>
<tr>
<td>ABSOLUTE((n))</td>
<td>If ( n&gt;0 ), on row ( n ) of the result table. If ( n&lt;0 ), and ( m ) is the number of rows in the result table, on row ( m+n+1 ) of the result table.</td>
</tr>
<tr>
<td>RELATIVE((n))</td>
<td>If ( n&gt;0 ), on the row that is ( n ) rows after the current row. If ( n&lt;0 ), on the row that is ( n ) rows before the current row. If ( n=0 ), on the current row.</td>
</tr>
<tr>
<td>AFTER</td>
<td>After the last row in the result table</td>
</tr>
<tr>
<td>BEFORE</td>
<td>Before the first row in the result table</td>
</tr>
</tbody>
</table>
Table 25. FETCH clauses for positioning a scrollable cursor (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Positions the cursor</th>
</tr>
</thead>
</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.

5. Invoke the close method to close the iterator.

Example

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 in reverse order. 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
    (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;
#sql [ctxt]
    scrliter=(SELECT EMPNO, LASTNAME FROM EMPLOYEE);
    scrliter.afterLast();
    while (scrliter.previous())
    {
        System.out.println(scrliter.EmpNo() + " "
                        + scrliter.LastName());
    }
    scrliter.close();
}
```

Related concepts:
"Data retrieval in SQLJ applications" on page 151

Related tasks:
"Using a positioned iterator in an SQLJ application" on page 153
"Using a named iterator in an SQLJ application" on page 151

Related information:
Calling stored procedures in SQLJ applications

To call a stored procedure, you use an executable clause that contains an SQL CALL statement.

About this task

You can execute the CALL statement with host identifier parameters. You can execute the CALL statement with literal parameters only if the data server on which the CALL statement runs supports execution of the CALL statement dynamically.

Procedure

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.

Example

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
String LastName="NARISINST";
String Address="IBM";
int CustNo;  // Output parameters
String Mark;
String MarkErrorText;
...
#sql [myConnCtx] {CALL ADD_CUSTOMER(:IN FirstName,
                     :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 " + FirstName + " + LastName + ": " + CustNo);  // Output
System.out.println(Mark);
If (MarkErrorText != null)
    System.out.println(" Error messages:" + MarkErrorText);
```

Figure 38. Example of calling a stored procedure in an SQLJ application

Related concepts:

“Retrieving multiple result sets from a stored procedure in an SQLJ application”

Retrieving multiple result sets from a stored procedure in an SQLJ application

Some stored procedures return one or more result sets to the calling program by including the DYNAMIC RESULT SETS n clause in the definition, with n>0, and opening cursors that are defined with the WITH RETURN clause. The calling program needs to retrieve the contents of those result sets.

To retrieve the rows from those result sets, you execute these steps:
1. Acquire an execution context for retrieving the result set from the stored procedure.

2. Associate the execution context with the CALL statement for the stored procedure.
   Do not use this execution context for any other purpose until you have retrieved and processed the last result set.

3. For each result set:
   a. Use the ExecutionContext method getNextResultSet to retrieve the result set.
   b. If you do not know the contents of the result set, use ResultSetMetaData methods to retrieve this information.
   c. Use an SQLJ result set iterator or JDBC ResultSet to retrieve the rows from the result set.

Result sets are returned to the calling program in the same order that their cursors are opened in the stored procedure. When there are no more result sets to retrieve, getNextResultSet returns a null value.

getNextResultSet has two forms:
getNextResultSet();
getNextResultSet(int current);

When you invoke the first form of getNextResultSet, SQLJ closes the currently-open result set and advances to the next result set. When you invoke the second form of getNextResultSet, the value of 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.

The following code calls a stored procedure that returns multiple result sets. For this example, it is assumed that the caller does not know the number of result sets to be returned or the contents of those result sets. It is also assumed that autoCommit is false. The numbers to the right of selected statements correspond to the previously-described steps.
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 uncustomized SQLJ statement that includes host expressions, the driver might need to do extra processing to determine data types. This extra processing can impact performance.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS, when the JDBC driver processes a CALL statement, the driver cannot determine the parameter data types.

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.

```
ExecutionContext execCtx=myConnCtx.getExecutionContext();
#sql [myConnCtx, execCtx] {CALL MULTRSSP();}
// MULTRSSP returns multiple result sets
ResultSet rs;
while ((rs = execCtx.getNextResultSet()) != null) {
    ResultSetMetaData rsmeta=rs.getMetadata();
    int numcols=rsmeta.getColumnCount();
    while (rs.next()) {
        for (int i=1; i<=numcols; i++) {
            String colval=rs.getString(i);
            System.out.println("Column "+i+" value is "+colval);
        }
    }
}
```

Figure 39. Retrieving result sets from a stored procedure
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
define 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.

For example, if you are using IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, you can use the IBM Data Server Driver for JDBC and SQLJ-only method `com.ibm.db2.jcc.t2zos.DB2LobFactory.createBlob` to create a java.sql.Blob object and populate the object with byte[] data:

```java
byte[] byteArray = {0, 1, 2, 3};
java.sql.Blob blobData =
    com.ibm.db2.jcc.t2zos.DB2LobFactory.createBlob(byteArray);
```

- 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
  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
  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.

For example, if you are using IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, you can use the IBM Data Server Driver for JDBC and SQLJ-only method `com.ibm.db2.jcc.t2zos.DB2LobFactory.createClob` to create a java.sql.Clob object and populate the object with String data:

```java
String stringVal = "Some Data";
java.sql.Clob clobData =
    com.ibm.db2.jcc.t2zos.DB2LobFactory.createClob(stringVal);
```
• Use one of the following types of stream IN parameters:
  - A `sqlj.runtime.CharacterStream` input parameter:
    ```java
    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:
      ```java
      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:
      ```java
      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
  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
  java.sql.Clob clobData = null;
  #sql {CALL STORPROC(:OUT clobData)};
  ```

• Use a `java.lang.String` output variable:
  ```java
  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 40 on page 168](#) 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

```java
public iterator ByName(String LastName, Date HireDate);  
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();
    String query = "SELECT LASTNAME, HIREDATE FROM EMPLOYEE";
    query+=whereClause;  // Build the query
    ResultSet rs = stmt.executeQuery(query);
    #sql [connCtx] nameiter = {CAST :rs};  // Convert the ResultSet
    while (nameiter.next()) {
        System.out.println(nameiter.LastName() + " was hired on " + nameiter.HireDate());
    }
    nameiter.close();
    stmt.close();
}
```

**Figure 40. Converting a JDBC result set to an SQLJ iterator**

Notes to [Figure 40]

<table>
<thead>
<tr>
<th>Note</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>This SQLJ clause creates the named iterator class ByName, which has accessor methods <code>LastName()</code> and <code>HireDate()</code> that return the data from result table columns <code>LASTNAME</code> and <code>HIREDATE</code>.</td>
</tr>
<tr>
<td>2</td>
<td>This statement and the following two statements build and prepare a query for dynamic execution using JDBC.</td>
</tr>
<tr>
<td>3</td>
<td>This JDBC statement executes the SELECT statement and assigns the result table to result set <code>rs</code>.</td>
</tr>
<tr>
<td>4</td>
<td>This iterator conversion clause converts the JDBC ResultSet <code>rs</code> to SQLJ iterator <code>nameiter</code>, and the following statements use <code>nameiter</code> to retrieve values from the result table.</td>
</tr>
<tr>
<td>5</td>
<td>The <code>nameiter.close()</code> method closes the SQLJ iterator and JDBC ResultSet <code>rs</code>.</td>
</tr>
</tbody>
</table>

**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 41 on page 169](#) 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};
ResultSet rs=iter.getResultSet();
while (rs.next())
  { System.out.println(rs.getString(1) + " was hired in " +
       rs.getDate(2));
  }
rs.close();
}

Figure 41. Converting an SQLJ iterator to a JDBC ResultSet

Notes to Figure 41

<table>
<thead>
<tr>
<th>Note</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>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.</td>
</tr>
<tr>
<td>2</td>
<td>The getResultSet() method accesses the ResultSet that underlies iterator iter.</td>
</tr>
<tr>
<td>3</td>
<td>The JDBC getString() and getDate() methods retrieve values from the ResultSet. The next() method moves the cursor to the next row in the ResultSet.</td>
</tr>
<tr>
<td>4</td>
<td>The rs.close() method closes the SQLJ iterator as well as the ResultSet.</td>
</tr>
</tbody>
</table>

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:

- Before you can access the columns of a remote table by name, through either a named iterator or an iterator that is converted to a JDBC ResultSet object, the DB2 for z/OS DESCSTAT subsystem parameter must be set to YES.
- 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.

Related reference:
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.

Procedure

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:
     ```java
     ExecutionContext execCtx = connCtx.getExecutionContext();
     ```
   
   • Create a new execution context by invoking the constructor for ExecutionContext. For example:
     ```java
     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.

Example

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.

```java
ExecutionContext execCtx = new ExecutionContext();
#sql [connCtx, execCtx] {DELETE FROM EMPLOYEE WHERE SALARY > 10000};
System.out.println("Deleted " + execCtx.getUpdateCount() + " rows");
```

Related tasks:

- “Handling an SQLWarning under the IBM Data Server Driver for JDBC and SQLJ” on page 119
- “Handling SQL warnings in an SQLJ application” on page 185
- “Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ” on page 116

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

DB2 for z/OS and Db2 for IBM 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
#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
  #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 42. 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.
TIMESTAMP WITH TIME ZONE values in SQLJ applications

DB2 for z/OS supports table columns with the TIMESTAMP WITH TIME ZONE data type. IBM Data Server Driver for JDBC and SQLJ supports update into and retrieval from a column with the TIMESTAMP WITH TIME ZONE data type in SQLJ programs.

When you update or retrieve a TIMESTAMP WITH TIME ZONE value, or call a stored procedure with a TIMESTAMP WITH TIME ZONE parameter, you need to use host variables that are com.ibm.db2.jcc.DBTimestamp objects to retain the time zone information. If you use java.sql.Timestamp objects to pass TIMESTAMP WITH TIME ZONE values to and from the data server, you lose the time zone information.

Because the com.ibm.db2.jcc.DBTimestamp class is a IBM Data Server Driver for JDBC and SQLJ-only class, if you run an uncustomized SQLJ application that uses com.ibm.db2.jcc.DBTimestamp objects, the application receives an SQLException.

Examples

Suppose that table TSTABLE has a single column, TSCOL, which has data type TIMESTAMP WITH TIME ZONE. The following code assigns a timestamp value with a time zone to the column, and retrieves the value from the column.

```java
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 43. Example of calling a stored procedure with a ROWID parameter
new com.ibm.db2.jcc.DBTimestamp(ts,estcal);
    // Create a datetime object that
    // includes the time zone

#sql[ctx] {INSERT INTO TSTABLE (TSCOL) VALUES (:dbts)};
    // Insert the datetime object in
    // the table

#sql[ctx] {COMMIT};

TSIter iter = null;
#sql [ctx] iter = {SELECT TSCOL FROM TSTABLE};
    // Assign the result table of the SELECT
while (iter.next()) {
    System.out.println("Timestamp = " +
    ((com.ibm.db2.jcc.DBTimestamp)iter.TSVar()).toDBString(true));
    // Use accessor method TSVar to retrieve
    // the TIMESTAMP WITH TIME ZONE value,
    // cast it to a DBTimestamp value,
    // and retrieve its string representation.
    // Value retrieved:
}

Suppose that stored procedure TSSP has a single INOUT parameter, TSPARM, which has data type TIMESTAMP WITH TIME ZONE. The following code calls the stored procedure with a timestamp value that includes a time zone, and retrieves a parameter value with a timestamp value that includes a time zone.

{ ...
    java.util.TimeZone esttz = java.util.TimeZone.getTimeZone("EST");
    // Set the time zone to UTC-5
    java.util.Calendar estcal = java.util.Calendar.getInstance(esttz);
    // Create a calendar instance
    // with the EST time zone
    java.sql.Timestamp ts =
    // Initialize a timestamp object
    // with the timestamp value that you
    // want to pass to the stored procedure
    com.ibm.db2.jcc.DBTimestamp dbts =
    new com.ibm.db2.jcc.DBTimestamp(ts,estcal);
    // Create a timestamp object that
    // includes the time zone to
    // pass to the stored procedure
    #sql[ctx] { CALL TSSP (:INOUT dbts) };
    System.out.println("Output parameter: " + dbts.toDBString (true));
    // Call the stored procedure with
    // the timestamp value as input,
    // and retrieve a timestamp value
    // with a time zone in the same
    // parameter
}

**Distinct types in SQLJ applications**

In an SQLJ program, you can create a distinct type using the CREATE DISTINCT TYPE statement in an executable clause.

You can also use CREATE TABLE in an executable clause to create a table that includes a column of that type. When you retrieve data from a column of that type, or update a column of that type, you use Java host variables or expressions with data types that correspond to the built-in types on which the distinct types are based.
The following example creates a distinct type that is based on an INTEGER type, creates a table with a column of that type, inserts a row into the table, and retrieves the row from the table:

```java
String empNumVar;
int shoeSizeVar;
...
#sql [myConnCtx] {CREATE DISTINCT TYPE SHOESIZE AS INTEGER WITH COMPARISONS};  // Create distinct type
#sql [myConnCtx] {COMMIT}; // Commit the create
#sql [myConnCtx] {CREATE TABLE EMP_SHOE (EMPNO CHAR(6), EMP_SHOE_SIZE SHOESIZE)};  // Create table using distinct type
#sql [myConnCtx] {COMMIT}; // Commit the create
#sql [myConnCtx] {INSERT INTO EMP_SHOE VALUES('000010',6)};  // Insert a row in the table
#sql [myConnCtx] {COMMIT}; // Commit the INSERT
#sql [myConnCtx] {SELECT EMPNO, EMP_SHOE_SIZE INTO :empNumVar, :shoeSizeVar FROM EMP_SHOE};  // Retrieve the row
System.out.println("Employee number: " + empNumVar + " Shoe size: " + shoeSizeVar);
```

Figure 44. Defining and using a distinct type

Related reference:

[CREATE TYPE (distinct) (DB2 SQL)]

**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.

```java
#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:db2://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
```

Figure 45. Setting, rolling back to, and releasing a savepoint in an SQLJ application
XML data in SQLJ applications

In SQLJ applications, you can store data in XML columns and retrieve data from XML columns.

In DB2 tables, the XML built-in data type is used to store XML data in a column as a structured set of nodes in a tree format.

SQLJ applications can send XML data to the data server or retrieve XML data from the data server in one of the following forms:

- As textual XML data
- As binary XML data (data that is in the Extensible Dynamic Binary XML Db2 Client/Server Binary XML Format), if the data server supports it

In SQLJ applications, you can:

- Store an entire XML document in an XML column using INSERT, UPDATE, or MERGE statements.
- Retrieve an entire XML document from an XML column using single-row SELECT statements or iterators.
- Retrieve a sequence from a document in an XML column by using the SQL XMLQUERY function to retrieve the sequence in the database, and then using single-row SELECT statements or iterators to retrieve the serialized XML string data into an application variable.
- You can update or retrieve XML data as textual XML data. Alternatively, for connections to a data server that supports binary XML data, you can update or retrieve XML data as binary XML data.

For data retrieval, you use the Datasource or Connection property xmlFormat to control whether the format of the retrieved data is textual XML or binary XML. For update of data in XML columns, xmlFormat has no effect. If the input data is binary XML data, and the data server does not support binary XML data, the input data is converted to textual XML data. Otherwise, no conversion occurs.

The format of XML data is transparent to the application. Storage and retrieval of binary XML data on a DB2 for z/OS data server requires version 4.9 or later of the IBM Data Server Driver for JDBC and SQLJ. Storage and retrieval of binary XML data on a Db2 on Linux, UNIX, and Windows systems data server requires version 4.11 or later of the IBM Data Server Driver for JDBC and SQLJ.
JDBC 4.0 java.sql.SQLXML objects can be used to retrieve and update data in XML columns. Invocations of metadata methods, such as ResultSetMetaData.getColumnType return the integer value java.sql.Types.SQLXML for an XML column type.

Related concepts:
“XML data retrieval in SQLJ applications” on page 178
“XML column updates in SQLJ applications”

**XML column updates in SQLJ applications**

In an SQLJ application, you can update or insert data into XML columns of a table at a DB2 data server using XML textual data. You can update or insert data into XML columns of a table using binary XML data (data that is in the Extensible Dynamic Binary XML Db2 Client/Server Binary XML Format), if the data server supports binary XML data.

The host expression data types that you can use to update XML columns are:
- java.sql.SQLXML (requires an SDK for Java Version 6 or later, and the IBM Data Server Driver for JDBC and SQLJ version 4.0 or later)
- com.ibm.db2.jcc.DB2Xml (deprecated)
- String
- byte
- Blob
- Clob
- sqlj.runtime.AsciiStream
- sqlj.runtime.BinaryStream
- sqlj.runtime.CharacterStream

The encoding of XML data can be derived from the data itself, which is known as *internally encoded* data, or from external sources, which is known as *externally encoded* data. XML data that is sent to the database server as binary data is treated as internally encoded data. XML data that is sent to the data source as character data is treated as externally encoded data. The external encoding is the default encoding for the JVM.

External encoding for Java applications is always Unicode encoding.

Externally encoded data can have internal encoding. That is, the data might be sent to the data source as character data, but the data contains encoding information. The data source handles incompatibilities between internal and external encoding as follows:
- If the data source is Db2 on Linux, UNIX, and Windows systems, the data source generates an error if the external and internal encoding are incompatible, unless the external and internal encoding are Unicode. If the external and internal encoding are Unicode, the data source ignores the internal encoding.
- If the data source is DB2 for z/OS, the data source ignores internal encoding.

Character data in XML columns is stored in UTF-8 encoding.

**Example:** Suppose that you use the following statement to insert data from String host expression xmlString into an XML column in a table. xmlString is a character type, so its external encoding is used, whether or not it has an internal encoding specification.

```sql
#sql [ctx] {INSERT INTO CUSTACC VALUES (1, :xmlString)};
```
**Example:** Suppose that you copy the data from xmlString into a byte array with CP500 encoding. The data contains an XML declaration with an encoding declaration for CP500. Then you insert the data from the byte[] host expression into an XML column in a table.

```java
byte[] xmlBytes = xmlString.getBytes("CP500");
#sql[ctx] {INSERT INTO CUSTACC VALUES (4, :xmlBytes)};
```

A byte string is considered to be internally encoded data. The data is converted from its internal encoding scheme to UTF-8, if necessary, and stored in its hierarchical format on the data source.

**Example:** Suppose that you copy the data from xmlString into a byte array with US-ASCII encoding. Then you construct an sqlj.runtimeAsciiStream host expression, and insert data from the sqlj.runtimeAsciiStream host expression into an XML column in a table on a data source.

```java
byte[] b = xmlString.getBytes("US-ASCII");
java.io.ByteArrayInputStream xmlAsciiInputStream =
   new java.io.ByteArrayInputStream(b);
sqlj.runtimeAsciiStream sqljXmlAsciiStream =
   new sqlj.runtimeAsciiStream(xmlAsciiInputStream, b.length);
#sql[ctx] {INSERT INTO CUSTACC VALUES (4, :sqljXmlAsciiStream)};
```

sqljXmlAsciiStream is a stream type, so its internal encoding is used. The data is converted from its internal encoding to UTF-8 encoding and stored in its hierarchical form on the data source.

**Example:** sqlj.runtime.CharacterStream host expression: Suppose that you construct an sqlj.runtime.CharacterStream host expression, and insert data from the sqlj.runtime.CharacterStream host expression into an XML column in a table.

```java
java.io.StringReader xmlReader =
   new java.io.StringReader(xmlString);
sqlj.runtimeCharacterStream sqljXmlCharacterStream =
   new sqlj.runtimeCharacterStream(xmlReader, xmlString.length());
#sql [ctx] {INSERT INTO CUSTACC VALUES (4, :sqljXmlCharacterStream)};
```

sqljXmlCharacterStream is a character type, so its external encoding is used, whether or not it has an internal encoding specification.

**Example:** Suppose that you retrieve a document from an XML column into a java.sql.SQLXML host expression, and insert the data into an XML column in a table.

```java
java.sql.ResultSet rs = s.executeQuery("SELECT * FROM CUSTACC");
rs.next();
java.sql.SQLXML xmlObject = (java.sql.SQLXML)rs.getObject(2);
#sql [ctx] {INSERT INTO CUSTACC VALUES (6, :xmlObject)};
```

After you retrieve the data it is still in UTF-8 encoding, so when you insert the data into another XML column, no conversion occurs.

**Example:** Suppose that you retrieve a document from an XML column into a com.ibm.db2.jdbc.DB2Xml host expression, and insert the data into an XML column in a table.

```java
java.sql.ResultSet rs = s.executeQuery("SELECT * FROM CUSTACC");
rs.next();
com.ibm.db2.jdbc.DB2Xml xmlObject = (com.ibm.db2.jdbc.DB2Xml)rs.getObject(2);
#sql [ctx] {INSERT INTO CUSTACC VALUES (6, :xmlObject)};
```
After you retrieve the data it is still in UTF-8 encoding, so when you insert the data into another XML column, no conversion occurs.

**XML data retrieval in SQLJ applications**

When you retrieve data from XML columns of a database table in an SQLJ application, the output data must be explicitly or implicitly serialized.

The host expression or iterator data types that you can use to retrieve data from XML columns are:
- `java.sql.SQLXML` (requires an SDK for Java Version 6 or later, and the IBM Data Server Driver for JDBC and SQLJ version 4.0 or later)
- `com.ibm.db2.jcc.DB2Xml` (deprecated)
- `String`
- `byte[]`
- `sqlj.runtime.AsciiStream`
- `sqlj.runtime.BinaryStream`
- `sqlj.runtime.CharacterStream`

If the application does not call the XMLSERIALIZE function before data retrieval, the data is converted from UTF-8 to the external application encoding for the character data types, or the internal encoding for the binary data types. No XML declaration is added. If the host expression is an object of the `java.sql.SQLXML` or `com.ibm.db2.jcc.DB2Xml` type, you need to call an additional method to retrieve the data from this object. The method that you call determines the encoding of the output data and whether an XML declaration with an encoding specification is added.

The following table lists the methods that you can call to retrieve data from a `java.sql.SQLXML` or a `com.ibm.db2.jcc.DB2Xml` object, and the corresponding output data types and type of encoding in the XML declarations.

<table>
<thead>
<tr>
<th>Method</th>
<th>Output data type</th>
<th>Type of XML internal encoding declaration added</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLXML.getBinaryStream</td>
<td>InputStream</td>
<td>None</td>
</tr>
<tr>
<td>SQLXML.getCharacterStream</td>
<td>Reader</td>
<td>None</td>
</tr>
<tr>
<td>SQLXML.getSource</td>
<td>Source</td>
<td>None</td>
</tr>
<tr>
<td>SQLXML.getString</td>
<td>String</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2AsciiStream</td>
<td>InputStream</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2BinaryStream</td>
<td>InputStream</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2Bytes</td>
<td>byte[]</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2CharacterStream</td>
<td>Reader</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2String</td>
<td>String</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlAsciiStream</td>
<td>InputStream</td>
<td>US-ASCII</td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlBinaryStream</td>
<td>InputStream</td>
<td>Specified by <code>getDB2XmlBinaryStream</code> <code>targetEncoding</code> parameter</td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlBytes</td>
<td>byte[]</td>
<td>Specified by <code>DB2Xml.getDB2XmlBytes</code> <code>targetEncoding</code> parameter</td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlCharacterStream</td>
<td>Reader</td>
<td>ISO-10646-UCS-2</td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlString</td>
<td>String</td>
<td>ISO-10646-UCS-2</td>
</tr>
</tbody>
</table>

178 Application Programming Guide and Reference for Java
If the application executes the XMLSERIALIZE function on the data that is to be returned, after execution of the function, the data has the data type that is specified in the XMLSERIALIZE function, not the XML data type. Therefore, the driver handles the data as the specified type and ignores any internal encoding declarations.

**Example:** Suppose that you retrieve data from an XML column into a String host expression.

```sql
#sql iterator XmlStringIter (int, String);
#sql [ctx] siter = {SELECT C1, CADOC from CUSTACC};
#sql {FETCH :siter INTO :row, :outString};
```

The String type is a character type, so the data is converted from UTF-8 to the external encoding, which is the default JVM encoding, and returned without any XML declaration.

**Example:** Suppose that you retrieve data from an XML column into a byte[] host expression.

```sql
#sql iterator XmlByteArrayIter (int, byte[]);
XmlByteArrayIter biter = null;
#sql [ctx] biter = {SELECT C1, CADOC from CUSTACC};
#sql {FETCH :biter INTO :row, :outBytes};
```

The byte[] type is a binary type, so no data conversion from UTF-8 encoding occurs, and the data is returned without any XML declaration.

**Example:** Suppose that you retrieve a document from an XML column into a java.sql.SQLXML host expression, but you need the data in a binary stream.

```sql
#sql iterator SqlXmlIter (int, java.sql.SQLXML);
SqlXmlIter SQLXMLiter = null;
java.sql.SQLXML outSqlXml = null;
#sql [ctx] SqlXmlIter = {SELECT C1, CADOC from CUSTACC};
#sql {FETCH :SqlXmlIter INTO :row, :outSqlXml};
java.io.InputStream XmlStream = outSqlXml.getBinaryStream();
```

The FETCH statement retrieves the data into the SQLXML object in UTF-8 encoding. The SQLXML.getBinaryStream stores the data in a binary stream.

**Example:** Suppose that you retrieve a document from an XML column into a com.ibm.db2.jcc.DB2Xml host expression, but you need the data in a byte string with an XML declaration that includes an internal encoding specification for UTF-8.

```sql
#sql iterator DB2XmlIter (int, com.ibm.db2.jcc.DB2Xml);
DB2XmlIter db2xmliter = null;
com.ibm.db2.jcc.DB2Xml outDB2Xml = null;
#sql [ctx] db2xmliter = {SELECT C1, CADOC from CUSTACC};
#sql {FETCH :db2xmliter INTO :row, :outDB2Xml};
byte[] byteArray = outDB2Xml.getDB2XmlBytes("UTF-8");
```

The FETCH statement retrieves the data into the DB2Xml object in UTF-8 encoding. The getDB2XmlBytes method with the UTF-8 argument adds an XML declaration with a UTF-8 encoding specification and stores the data in a byte array.

**XMLCAST in SQLJ applications**

Before you can use XMLCAST to cast a host variable to the XML data type in an SQLJ application, you need to cast the host variable to the corresponding SQL data type.
Example: The following code demonstrates a situation in which it is necessary to cast a String host variable to an SQL character type, such as VARCHAR, before you use XMLCAST to cast the value to the XML data type.

```java
String xmlresult = null;
String varchar_hv = "San Jose";
...
$sql [con] {SELECT XMLCAST(CAST(:varchar_hv AS VARCHAR(32)) AS XML) INTO :xmlresult FROM SYSIBM.SYSDUMMY1};
```

Inserting data from file reference variables into tables in SQLJ applications

You can use file reference variable objects with IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 9 or later to stream LOB or XML input data.

Before you begin

You need to store your LOB or XML input data in HFS files.

About this task

Use of file reference variables eliminates the need to materialize the LOB or XML data in memory before the data is stored in tables.

Procedure

To use file reference variables to store LOB or XML data in tables, follow these steps:

1. Execute constructors for file reference variable objects of the appropriate types.
   The following table lists the types of data in the input files and the appropriate constructors.

<table>
<thead>
<tr>
<th>Input data type</th>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>com.ibm.db2.jcc.DB2BlobFileReference</td>
</tr>
<tr>
<td>CLOB</td>
<td>com.ibm.db2.jcc.DB2ClobFileReference</td>
</tr>
<tr>
<td>XML AS BLOB</td>
<td>com.ibm.db2.jcc.DB2XmlAsBlobFileReference</td>
</tr>
<tr>
<td>XML AS CLOB</td>
<td>com.ibm.db2.jcc.DB2XmlAsClobFileReference</td>
</tr>
</tbody>
</table>

   The first parameter in each constructor must specify the absolute path name for an existing HFS file.

2. Execute an INSERT statement with the file reference variable object as the input host variable.

Example

Suppose that a table is defined like this:

```sql
CREATE TABLE TEST02TB (  
    RECID INTEGER,  
    CLOBCOL CLOB(100M),  
    BLOBCOL(200M),  
    XMLCOL XML)  
```
The following code uses file reference variables to insert a CLOB value, a BLOB value, and an XML AS BLOB value into the table. The numbers to the right of selected statements correspond to the previously described steps.

```java
com.ibm.db2.jcc.DB2ClobFileReference clobFileRef =
    new com.ibm.db2.jcc.DB2ClobFileReference("/u/usrt001/jcc/test/TEXT.FILE","Cp037");
com.ibm.db2.jcc.DB2BlobFileReference blobFileRef =
    new com.ibm.db2.jcc.DB2BlobFileReference("/u/usrt001/jcc/test/BINARY.FILE");
com.ibm.db2.jcc.DB2XmlAsBlobFileReference xmlAsBlobFileRef =
    new com.ibm.db2.jcc.DB2XmlAsBlobFileReference("/u/usrt001/jcc/test/XML.FILE");
    // Execute constructors for the file reference variable objects
    #sql [ctx] {"INSERT INTO TEST03TB(RECID,CLOBCOL,BLOBCOL,XMLCOL)
VALUES('003',:clobFileRef,:blobFileRef,:xmlAsBlobFileRef)};
```

### 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:

```java
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:

```java
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`:

```java
public @interface MyAnnot {}```
You also declare the following marker annotation in a program called MyAnnot2.java:
public @interface MyAnnot2 { }

You can then use those annotations in an SQLJ program:

// Class annotations
@MyAnnot2 public @MyAnot class TestAnnotation {

    // Field annotation
    @MyAnot
    private static final int field1 = 0;

    // Constructor annotation
    @MyAnnot2 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 {

    }

} // Class TestAnnotation

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:
  ```java
  List<List<String>> strList2 = new ArrayList<List<String>>();
  ```
- A HashMap in which the key/value pair has the String type:
  ```java
  Map<String, String> map = new HashMap<String, String>();
  ```
- A method that takes a List with elements of any type:
  ```java
  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.
```java
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.
```java
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.

About this task

The following table shows the values that you can specify in the SET TRANSACTION ISOLATION LEVEL clause and their data server equivalents.

<table>
<thead>
<tr>
<th>SET TRANSACTION value</th>
<th>Data server isolation level</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERIALIZABLE</td>
<td>Repeatable read</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>Read stability</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>Cursor stability</td>
</tr>
<tr>
<td>READ UNCOMMITTED</td>
<td>Uncommitted read</td>
</tr>
</tbody>
</table>

The isolation level affects the underlying JDBC connection as well as the SQLJ connection.

Related concepts:

“JDBC connection objects” on page 23

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.

Example

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};
```

Related tasks:

“Connecting to a data source using SQLJ” on page 127
“Committing or rolling back SQLJ transactions”

Handling SQL errors and warnings in SQLJ applications

SQLJ clauses throw SQLExceptions when SQL errors occur, but not when most SQL warnings occur.

About this task

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.
Procedure

- 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.

Procedure

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.

Example

The following code prints out the SQL error that occurred if a SELECT statement fails.

```java
try {
    #sql [ctxt]
    {SELECT LASTNAME INTO :empname
     FROM EMPLOYEE WHERE EMPNO='00000010'};
}
catch(SQLException e) {
    System.out.println("Error code returned: " + e.getErrorCode());
}
```

Related tasks:

- "Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ” on page 116

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.

About this task

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.jdbc.DB2Diagnosable` interface and the `com.ibm.db2.jcc.DB2Sqlca` class.

Procedure

To retrieve data-server-specific information about a warning:

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".

Example

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);
```

Related tasks:
"Handling SQL errors in an SQLJ application" on page 185

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 data server resources and SQLJ resources for the associated ConnectionContext object immediately.

About this task

If you do not close a ConnectionContext object after you use it, unexpected behavior might occur if a Java finalizer closes the ConnectionContext object. Examples of the unexpected behavior are:

- An ObjectClosedException on the underlying ResultSet or Statement objects
- Agent hangs in DB2 stored procedures

Procedure

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.

Example

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

Related tasks:

“Connecting to a data source using SQLJ” on page 127
Chapter 5. Java stored procedures and user-defined functions

Like stored procedures and user-defined functions in any other language, Java stored procedures and user-defined functions are programs that can contain SQL statements. You invoke Java stored procedures from a client program that is written in any supported language.

The following topics contain information that is specific to defining and writing Java user-defined functions and stored procedures.

In these topics, the word *routine* refers to either a stored procedure or a user-defined function.

**Related reference:**
- [Java sample JDBC stored procedure (DB2 for z/OS Stored Procedures: Through the CALL and Beyond)](#)
- [Java stored procedure returning a BLOB column (DB2 for z/OS Stored Procedures: Through the CALL and Beyond)](#)
- [Java stored procedure returning a CLOB column (DB2 for z/OS Stored Procedures: Through the CALL and Beyond)](#)
- [Debugging Java procedures on Linux, UNIX, and Windows (DB2 for z/OS Stored Procedures: Through the CALL and Beyond)](#)
- [Java sample SQLJ stored procedure (DB2 for z/OS Stored Procedures: Through the CALL and Beyond)](#)

Setting up the environment for Java routines

Before you can run Java routines, you need to set up a WLM environment and set Java environment variables.

**Before you begin**

Before you can prepare and run Java routines, you need to satisfy the following prerequisites:

- **Java 2 Technology Edition, V5 or later.**
  The IBM Data Server Driver for JDBC and SQLJ supports only 31-bit Java routines.
- **TCP/IP**
  TCP/IP is required on the client and all database servers to which you connect.
- **The version of the IBM Data Server Driver for JDBC and SQLJ that matches the DB2 for z/OS version.**
  If you are migrating from a previous release of DB2 for z/OS, you need to install the corresponding version of the IBM Data Server Driver for JDBC and SQLJ.

**About this task**

The steps in this ask are necessary for preparing and running Java routines.
If you plan to use IBM Optim Development Studio to prepare and run your Java routines, see [Developing DB2 for z/OS stored procedures (IBM Data Studio)](#).

**Procedure**

To set up the environment for running Java routines, you need to perform these tasks:

1. Ensure that your operating system, SDK for Java, and the IBM Data Server Driver for JDBC and SQLJ are at the correct levels, and that you have installed all prerequisite products.
   
   **Important:** If you have migrated the DB2 subsystem from a previous release of DB2 for z/OS, your existing Java stored procedures and user-defined function no longer work with the previous release of the IBM Data Server Driver for JDBC and SQLJ and the current release of DB2 for z/OS. You need to install the version of the IBM Data Server Driver for JDBC and SQLJ that matches the DB2 for z/OS release level, and update the WLM-managed stored procedure address space configuration and JAVAENV data set to use the current driver.

2. Create the Workload Manager for z/OS (WLM) application environment for running the routines.

3. Set up the run-time environment for z/OS (WLM) application environment for Java routines, which includes setting environment variables.

**Setting up the WLM application environment for Java routines**

You need different WLM application environments for Java routines from the WLM application environments that you use for other routines.

**About this task**

Setting up a WLM environment for Java routines involves the same basic steps as setting up a WLM environment for other routines.

**Procedure**

1. Create a WLM environment startup procedure for Java routines.

2. Define the WLM environment to WLM.

**WLM address space startup procedure for Java routines**

The WLM address space startup procedure for Java routines requires extra DD statements that other routines do not need.

The following figure shows an example of a startup procedure for an address space in which Java routines can run. The JAVAENV DD statement indicates to DB2 that the WLM environment is for Java routines.
Notes to Figure 46:

1. In this statement:
   - Change the DB2SSN value to your DB2 for z/OS subsystem name.
   - Change the APPLENV value to the name of the application environment that you set up for Java stored procedures.
   - Choose a maximum value of NUMTCB of between 5 and 8. For testing a Java stored procedure, NUMTCB=1 is recommended. With NUMTCB=1, only one JVM is started, so refreshing the WLM environment after you change the stored procedure takes less time.
   - Change the MNSPAS value to the minimum number of stored procedure address spaces that WLM starts and maintains. Valid values are 0 to 50. If you specify 0, WLM starts and shuts down stored procedure address spaces as applications require them. Specify a value of greater than 0 if the overhead of starting and shutting down stored procedure address spaces and JVMs makes your response time unacceptable.

2. JAVAENV specifies a data set that contains Language Environment® run-time options for Java stored procedures. The presence of this DD statement indicates to DB2 that the WLM environment is for Java routines. This data set must contain the environment variable JAVA_HOME. This environment variable indicates to DB2 that the WLM environment is for Java routines. JAVA_HOME also specifies the highest-level directory in the set of directories that contain the SDK for Java.

3. Specifies a data set into which DB2 puts information that you can use to debug your stored procedure. The information that DB2 collects is for assistance in debugging setup problems, and should be used only under the direction of IBM Software Support. You should comment out this DD statement during production.

Related concepts:

“WLM application environment values for Java routines”
“Runtime environment for Java routines” on page 193

WLM application environment values for Java routines
To define the application environment for Java routines to WLM, specify the appropriate values on WLM setup panels.

Use values like those that are shown in the following screen examples.
**Definition name**
Specify the name of the WLM application environment that you are setting up for stored procedures.

**Description**
Specify any value.

**Options**
Specify 9 (Application Environments).

**Subsystem Type**
Specify DB2.

**Procedure Name**
Specify a name that matches the name of the JCL startup procedure for the stored procedure address spaces that are associated with this application environment.

**Start Parameters**
If the DB2 subsystem in which the stored procedure runs is not in a sysplex, specify a DB2SSN value that matches the name of that DB2 subsystem. If the same JCL is used for multiple DB2 subsystems, specify DB2SSN=&IWMSSNM. The NUMTCB value depends on the type of stored procedure you are running. For running Java routines, specify a value between 5 and 8. Specify an APPLENV value that matches the value that you specify on the CREATE PROCEDURE or CREATE FUNCTION statement for the routines that run in this application environment.
Limit on starting server address spaces for a subsystem instance

Specify 1 (no limit).

Related concepts:

“WLM address space startup procedure for Java routines” on page 190
“Runtime environment for Java routines”

Runtime environment for Java routines

For Java routines, the startup procedure for the stored procedure address space contains a JAVAENV DD statement. This statement specifies a data set that contains Language Environment runtime options for the routines that run in the stored procedure address space.

Create the data set for the runtime options with the characteristics that are listed in the following table.

<table>
<thead>
<tr>
<th>Table 28. Data set characteristics for the JAVAENV data set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary space allocation</td>
</tr>
<tr>
<td>Secondary space allocation</td>
</tr>
<tr>
<td>Record format</td>
</tr>
<tr>
<td>Record length</td>
</tr>
<tr>
<td>Block size</td>
</tr>
</tbody>
</table>

After you create the data set, edit it to insert a Language Environment options string, which has this form:

```
XPLINK(ON),
ENVAR("environment-variable=setting"),
MSGFILE(ddname, recm, lrec, blksize, NOENQ, ENQ)
```

The maximum length of the Language Environment runtime options string in a JAVAENV data set for Java stored procedures is 245 bytes. If you exceed the maximum length, DB2 truncates the contents but does not issue a message. If you enter the contents of the JAVAENV data set on more than one line, DB2 concatenates the lines to form the runtime options string. The runtime options string can contain no leading or trailing blanks. Within the string, only blanks that are valid within an option are permitted.

If your environment variable list is long enough that the JAVAENV content is greater than 245 bytes, you can put the environment variable list in a separate data set in a separate file, and use the environment variable _CEE_ENVFILE to point to that file.

The descriptions of the parameters are:

-_CEE_ENVFILE_

Specifies a z/OS UNIX System Services data set that contains some or all of the settings for environment variables.
Use the _CEE_ENVFILE parameter if the length of environment variable string causes the total length of the JAVAENV content to exceed 245 bytes, which is the DB2 limit for the JAVAENV content.

The data set must be variable-length. The format for environment variable settings in this data set is:

```
environment-variable-1=setting-1
environment-variable-2=setting-2
...
environment-variable-n=setting-n
```

You can specify some of your environment variable settings as arguments of ENVAR and put some of the settings in this data set, or you can put all of your environment variable settings in this data set.

For example, to use file /u/db2a10/javasp/jspnolimit.txt for environment variable settings, specify:

```
_COUNTRY_ENVFILE=/u/db2a10/javasp/jspnolimit.txt
```

**ENVAR**

Sets the initial values for specified environment variables. The environment variables that you might need to specify are:

**CLASSPATH**

When you prepare your Java routines, if you do not put your routine classes into JAR files, include the directories that contain those classes. For example:

```
CLASSPATH=.:/U/DB2RES3/ACMEJOS
```

Do not include directories for JAR files for JDBC or the JDK in the CLASSPATH. If you use a DB2JccConfiguration.properties file, you need to include the directory that contains that file in the CLASSPATH.

**DB2_BASE**

The value of DB2_BASE is the highest-level directory in the set of HFS directories that contain DB2 for z/OS code.

For example:

```
DB2_BASE=/usr/lpp/db2a10/base
```

The default is /usr/lpp/db2a10/base.

**JAVA_HOME**

This environment variable indicates to DB2 that the WLM environment is for Java routines. The value of JAVA_HOME is the highest-level directory in the set of directories that contain the SDK for Java. For example:

```
JAVA_HOME=/usr/lpp/java/IBM/J6.0
```

**JCC_HOME**

The value of JCC_HOME is the highest-level directory in the set of directories that contain the JDBC driver. For example:

```
JCC_HOME=/usr/lpp/db2a10/jdbc
```

JCC_HOME must be set.

**JDBCSTD**

Specifies which version of the IBM Data Server Driver for JDBC and SQLJ that Java routines use. Possible values are:

```
3       Java routines use the version of the driver that supports JDBC 3.0.
```
Java routines use the version of the driver that supports JDBC 4.0.

**JVM_DEBUG_PORTRANGE**
This environment variable specifies a range of ports that the JVM listens on for debug connections, in the form low-port-number:high-port-number. The default is ports 8000 to 8050. For example:

```
JVM_DEBUG_PORTRANGE=8051::8055
```

Specify JVM_DEBUG_PORTRANGE only for WLM environments that are used for debugging Java routines.

**JVMPROPS**
This environment variable specifies the name of a z/OS UNIX System Services file that contains startup options for the JVM in which the stored procedure runs. For example:

```
JVMPROPS=/usr/lpp/java/properties/jvmsp
```

The following example shows the contents of a startup options file that you might use for a JVM in which Java stored procedures run:

```
# Properties file for JVM for Java stored procedures
# Sets the initial size of middleware heap within non-system heap
-Xms64M

# Sets the maximum size of nonsystem heap
-Xmx128M

# initial size of system heap
-Xinitsh512K
```

For information about JVM startup options, see IBM 31-bit and 64-bit SDKs for z/OS, Java 2 Technology Edition, Version 5 SDK and Runtime Environment User Guide, available at:

http://www.ibm.com/servers/eserver/zseries/software/java

Click the Reference Information link.

**LC_ALL**
Modify LC_ALL to change the locale to use for the locale categories when the individual locale environment variables specify locale information. This value needs to match the CCSID for the DB2 subsystem on which the stored procedures run. For example:

```
LC_ALL=En_US.IBM-037
```

**TZ**
Modify TZ to change the local timezone. For example:

```
TZ=PST08
```

The default is GMT (UTC).

**USE_LIBJVM_G**
Specifies whether the debug version of the JVM is used instead of the default, non-debug version of the JVM. The debug version of the JVM is in dynamic link library libjvm_g. If USE_LIBJVM_G is not specified, or its value is anything other than the capitalized string YES, the non-debug version of the JVM is used. For example, USE_LIBJVM_G=NO causes the non-debug version of the JVM to be used.

If USE_LIBJVM_G=YES, the JVMPROPS environment variable must specify a file that contains JVM startup options. That file must contain the startup option -Djava.execsuffix=_g.
Specify USE_LIBJVM_G=YES only under the direction of IBM Software Support.

**WORK_DIR**
Modify WORK_DIR to change the default destination for STDOUT and STDERR output.

**MSGFILE**
Specifies the DD name of a data set in which Language Environment puts runtime diagnostics. All subparameters in the MSGFILE parameter are optional. The default is MSGFILE(SYSOUT,FBA,121,0,NOENQ)

If you specify a data set name in the JSPDEBUG statement of your stored procedure address space startup procedure, you need to specify JSPDEBUG as the first parameter. If the NUMTCB value in the stored procedure address space startup procedure is greater than 1, you need to specify ENQ as the fifth subparameter. z/OS Language Environment Programming Reference contains complete information about MSGFILE.

**XPLINK(ON)**
Causes the initialization of the XPLINK environment. This option is required.

The following example shows the contents of a JAVAENV data set.

```
ENVAR("JCC_HOME=/usr/lpp/db2a10/jdbc",
"JAVA_HOME=/usr/lpp/java160/J6.0",
"WORK_DIR=/u/db2a10/tmp"),
MSGFILE(JSPDEBUG,,,,ENQ)
```

For information on environment variables that are related to locales, see z/OS C/C++ Programming Guide.

**Related concepts:**
- “WLM address space startup procedure for Java routines” on page 190
- “WLM application environment values for Java routines” on page 191

### Defining Java routines and JAR files to DB2

Before you can use a Java routine, you need to define it to DB2.

**About this task**

Use the following procedure to manually define a Java routine to DB2. If you use IBM Optim Development Studio, IBM Optim Development Studio creates the definitions.

**Procedure**

1. Execute the CREATE PROCEDURE or CREATE FUNCTION statement to define the routine to DB2. To alter the routine definition, use the ALTER PROCEDURE or ALTER FUNCTION statement.

2. Optional: If the routines are in JAR files, define the JAR files to DB2.
   - If the routines are in JAR files, it is recommended that you also define the JAR files to DB2. Alternatively, you can include the JAR file name in the CLASSPATH.
   - To define the JAR files to DB2:
     - Use the SQLJ.INSTALL_JAR or SQLJ.DB2_INSTALL_JAR built-in stored procedure to define the JAR files to DB2.
After you have installed a JAR, if that JAR references classes in other installed JARs, use the SQLJ.ALTER_JAVA_PATH stored procedure to specify the class resolution path that the JVM searches to resolve those class references.

To replace the JAR file, use the SQLJ.REPLACE_JAR or SQLJ.DB2_REPLACE_JAR stored procedure.

To remove the JAR file, use the SQLJ.REMOVE_JAR or SQLJ.DB2_REMOVE_JAR stored procedure.

SQLJ.INSTALL_JAR, SQLJ, SQLJ.REPLACE_JAR, and SQLJ.REMOVE_JAR can be used only with the local DB2 catalog. The other stored procedures can be used with remote or local DB2 catalogs.

Definition of a Java routine to DB2

Before you can use a Java routine, you need to define it to DB2 using the CREATE PROCEDURE or CREATE FUNCTION statement.

The definition for a Java routine is much like the definition for a routine in any other language. However, the following parameters have different meanings for Java routines.

**LANGUAGE**

Specifies the application programming language in which the routine is written.

Specify LANGUAGE JAVA.

You cannot specify LANGUAGE JAVA for a user-defined table function.

**EXTERNAL NAME**

Specifies the program that runs when the procedure name is specified in a CALL statement or the user-defined function name is specified in an SQL statement. For Java routines, the argument of EXTERNAL NAME is a string that is enclosed in single quotation marks. The EXTERNAL NAME clause for a Java routine has the following syntax:

```
EXTERNAL NAME
'B"class-name.method-name"(1)
(2)
JAR-name:package-name.(method-signature)'
```

Notes:
1. For compatibility with DB2, you can use an exclamation point (!) after JAR-name instead of a colon.
2. For compatibility with previous versions of DB2, you can use a slash (/) after package-name instead of a period.

Whether you include JAR-name depends on where the Java code for the routine resides. If you create a JAR file from the class file for the routine (the output from the javac command), you need to include JAR-name. You must create the JAR file and define the JAR file to DB2 before you execute the CREATE PROCEDURE or CREATE FUNCTION statement. If some other user executes the CREATE PROCEDURE or CREATE FUNCTION statement, you need to grant the USAGE privilege on the JAR to them.

If you use a JAR file, that JAR file must refer to classes that are contained in that JAR file, are found in the CLASSPATH, or are system-supplied. Classes
that are in directories that are referenced in DB2_HOME or JCC_HOME, and Java_HOME do not need to be included in the JAR file.

Whether you include *(method-signature)* depends on the following factors:

- The way that you define the parameters in your routine method
  - Each SQL data type has a corresponding default Java data type. If your routine method uses data types other than the default types, you need to include a method signature in the EXTERNAL NAME clause. A method signature is a comma-separated list of data types.
  - Whether you overload a Java routine
    - If you have several Java methods in the same class, with the same name and different parameter types, you need to specify the method signature to indicate which version of the program is associated with the Java routine.

If your stored procedure returns result sets, you also need to include a parameter in the method signature for each result set. The parameter can be in one of the following forms:

- java.sql.ResultSet[]
- An array of an SQLJ iterator class

You do not include these parameters in the parameter list of the SQL CALL statement when you invoke the stored procedure.

*Example: EXTERNAL NAME clause for a Java user-defined function:* Suppose that you write a Java user-defined function as method getSals in class S1Sal and package s1. You put S1Sal in a JAR file named sal_JAR and install that JAR in DB2. The EXTERNAL NAME parameter is:

```
EXTERNAL NAME 'sal_JAR:s1.S1Sal.getSals'
```

*Example: EXTERNAL NAME clause for a Java stored procedure:* Suppose that you write a Java stored procedure as method getSals in class S1Sal. You put S1Sal in a JAR file named sal_JAR and install that JAR in DB2. The stored procedure has one input parameter of type INTEGER and returns one result set. The Java method for the stored procedure receives one parameter of type java.lang.Integer, but the default Java data type for an SQL type of INTEGER is int, so the EXTERNAL NAME clause requires a signature clause. The EXTERNAL NAME parameter is:

```
EXTERNAL NAME 'sal_JAR:S1Sal.getSals(java.lang.Integer,java.sql.ResultSet[])'
```

**NO SQL**
- Indicates that the routine does not contain any SQL statements.
- For a Java routine that is stored in a JAR file, you cannot specify NO SQL.

**PARAMETER STYLE**
- Identifies the linkage convention that is used to pass parameters to the routine.
- For a Java routine, the only value that is valid is PARAMETER STYLE JAVA.
- You cannot specify PARAMETER STYLE JAVA for a user-defined table function.

**WLM ENVIRONMENT**
- Identifies the MVS workload manager (WLM) environment in which the routine is to run.
- If you do not specify this parameter, the routine runs in the default WLM environment that was specified when DB2 was installed.
PROGRAM TYPE
   Specifies whether Language Environment runs the routine as a main routine or a subroutine.

   This parameter value must be PROGRAM TYPE SUB.

RUN OPTIONS
   Specifies the Language Environment run-time options to be used for the routine.

   This parameter has no meaning for a Java routine. If you specify this parameter with LANGUAGE JAVA, DB2 issues an error.

SCRATCHPAD
   Specifies that when the user-defined function is invoked for the first time, DB2 allocates memory for a scratchpad.

   You cannot use a scratchpad in a Java user-defined function. Do not specify SCRATCHPAD when you create or alter a Java user-defined function.

FINAL CALL
   Specifies that a final call is made to the user-defined function, which the function can use to free any system resources that it has acquired.

   You cannot perform a final call when you call a Java user-defined function. Do not specify FINAL CALL when you create or alter a Java user-defined function.

DBINFO
   Specifies that when the routine is invoked, an additional argument is passed that contains environment information.

   You cannot pass the additional argument when you call a Java routine. Do not specify DBINFO when you call a Java routine.

SECURITY
   Specifies how the routine interacts with an external security product, such as RACF, to control access to non-SQL resources. The values of the SECURITY parameter are the same for a Java routine as for any other routine. However, the value of the SECURITY parameter determines the authorization ID that must have authority to access z/OS UNIX System Services. The values of SECURITY and the IDs that must have access to z/OS UNIX System Services are:

   DB2   The user ID that is defined for the stored procedure address space in the RACF started-procedure table.

   EXTERNAL
       The invoker of the routine.

   DEFINER
       The definer of the routine.

ALLOW DEBUG MODE, DISALLOW DEBUG MODE, or DISABLE DEBUG MODE
   Specifies whether a Java stored procedure can be run in debugging mode. When DYNAMICRULES run behavior is in effect, the default is determined by using the value of the CURRENT DEBUG MODE special register. Otherwise the default is DISALLOW DEBUG MODE.

   ALLOW DEBUG MODE
       Specifies that the procedure can be run in debugging mode.

   DISALLOW DEBUG MODE
       Specifies that the procedure cannot be run in debugging mode.
You can use an ALTER PROCEDURE statement to change this option to ALLOW DEBUG MODE.

**DISABLE DEBUG MODE**

Specifies that the procedure can never be run in debugging mode.

The procedure cannot be changed to specify ALLOW DEBUG MODE or DISALLOW DEBUG MODE once the procedure has been created or altered using DISABLE DEBUG MODE. To change this option, you must drop and recreate the procedure using the desired option.

*Example: Defining a Java stored procedure:* Suppose that you have written and prepared a stored procedure that has these characteristics:

- **Fully-qualified procedure name**: SYSPROC.S1SAL
- **Parameters**: DECIMAL(10,2) INOUT
- **Language**: Java
- **Collection ID for the stored procedure package**: DSNJDBC
- **Package, class, and method name**: s1.S1Sal.getSal
- **Type of SQL statements in the program**: Statements that modify DB2 tables
- **WLM environment name**: WLMIJAVA
- **Maximum number of result sets returned**: 1

This CREATE PROCEDURE statement defines the stored procedure to DB2:

```sql
CREATE PROCEDURE SYSPROC.S1SAL
  (DECIMAL(10,2) INOUT)
  FENCED
  MODIFIES SQL DATA
  COLLID DSNJDBC
  LANGUAGE JAVA
  EXTERNAL NAME 's1.S1Sal.getSal'
  WLM ENVIRONMENT WLMIJAVA
  DYNAMIC RESULT SETS 1
  PROGRAM TYPE SUB
  PARAMETER STYLE JAVA;
```

*Example: Defining a Java user-defined function:* Suppose that you have written and prepared a user-defined function that has these characteristics:

- **Fully-qualified function name**: MYSCHEMA.S2SAL
- **Input parameter**: INTEGER
- **Data type of returned value**: VARCHAR(20)
- **Language**: Java
- **Collection ID for the function package**: DSNJDBC
- **Package, class, and method name**: s2.S2Sal.getSals
- **Java data type of the method input parameter**: java.lang.Integer
- **JAR file that contains the function class**: sal_JAR
- **Type of SQL statements in the program**: Statements that modify DB2 tables
- **Function is called when input parameter is null?**: Yes
- **WLM environment name**: WLMIJAVA

This CREATE FUNCTION statement defines the user-defined function to DB2:

```sql
CREATE FUNCTION MYSCHEMA.S2SAL(INTEGER)
  RETURNS VARCHAR(20)
  FENCED
```

200 Application Programming Guide and Reference for Java
MODIFIES SQL DATA
COLLID DSNJDBC
LANGUAGE JAVA
EXTERNAL NAME 'sal_JAR:s2.S2Sal.getSals(java.lang.Integer)'
WLM ENVIRONMENT WLMIJAV
CALLED ON NULL INPUT
PROGRAM TYPE SUB
PARAMETER STYLE JAVA;

In this function definition, you need to specify a method signature in the EXTERNAL NAME clause because the data type of the method input parameter is different from the default Java data type for an SQL type of INTEGER.

Related concepts:
- “Definition of a JAR file for a Java routine to DB2”

Related reference:
- ALTER FUNCTION (external) (DB2 SQL)
- ALTER PROCEDURE (external) (DB2 SQL)
- CREATE FUNCTION (DB2 SQL)
- CREATE PROCEDURE (external) (DB2 SQL)

**Definition of a JAR file for a Java routine to DB2**

One way to organize the classes for a Java routine is to collect those classes into a JAR file. If you do this, you need to install the JAR file into the DB2 catalog.

DB2 provides built-in stored procedures that perform the following functions for the JAR file:

SQLJ.INSTALL_JAR
   Installs a JAR file into the local DB2 catalog.

SQLJ.DB2_INSTALL_JAR
   Installs a JAR file into the local DB2 catalog or a remote DB2 catalog.

SQLJ.REPLACE_JAR
   Replaces an existing JAR file in the local DB2 catalog.

SQLJ.DB2_REPLACE_JAR
   Replaces an existing JAR file in the local DB2 catalog or a remote DB2 catalog.

SQLJ.REMOVE_JAR
   Deletes a JAR file from the local DB2 catalog or a remote DB2 catalog.

SQLJ.ALTER_JAVA_PATH
   Modifies the class resolution path of an previously installed JAR file to a specified value.

You can use IBM Optim Development Studio to install JAR files into the DB2 catalog, or you can write a client program that executes SQL CALL statements to invoke these stored procedures.

Related concepts:
- “Definition of a Java routine to DB2” on page 197

**Java routine programming**

A Java routine is a Java application program that runs in a stored procedure address space. It can include JDBC methods or SQLJ clauses.
A Java routine is much like any other Java program and follows the same rules as routines in other languages. It receives input parameters, executes Java statements, optionally executes SQLJ clauses, JDBC methods, or a combination of both, and returns output parameters.

**Differences between Java routines and stand-alone Java programs**

Java routines differ in a few basic ways from stand-alone Java programs.

Those differences are:

- In a Java routine, a JDBC connection or an SQLJ connection context can use the connection to the data source that processes the CALL statement or the user-defined function invocation. The URL that identifies this default connection is jdbc:default:connection.
- The top-level method for a Java routine must be declared as static and public. Although you can use static and final variables in a Java routine without problems, you might encounter problems when you use static and non-final variables. You cannot guarantee that a static and non-final variable retains its value in the following circumstances:
  - Across multiple invocations of the same routine
  - Across invocations of different routines that reference that variable
- As in routines in other languages, the SQL statements that you can execute in the routine depend on whether you specify an SQL access level of CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA.

**Related concepts:**

- "Differences between Java routines and other routines"

**Related reference:**

[SQL statement data access classification for routines (DB2 SQL)]

**Differences between Java routines and other routines**

Java routines differ in a few basic ways from routines that are written in other programming languages.

A Java routine differs from stored procedures that are written in other languages in the following ways:

- A Java routine must be defined with PARAMETER STYLE JAVA. PARAMETER STYLE JAVA specifies that the routine uses a parameter-passing convention that conforms to the Java language and SQLJ specifications. DB2 passes INOUT and OUT parameters as single-entry arrays. This means that in your Java routine, you must declare OUT or INOUT parameters as arrays. For example, suppose that stored procedure sp_one_out has one output parameter of type int. You declare the parameter like this:

  public static void routine_one_out (int[] out_parm)

- Java routines that are Java main methods have these restrictions:
  - The method must have a signature of String[]. It must be possible to map all the parameters to Java variables of type java.lang.String.
  - The routine can have only IN parameters.
- You cannot make instrumentation facility interface (IFI) calls in Java routines.
- You cannot specify an SQL access level of NO SQL for Java routines.
• As in other Java programs, you cannot include the following statements in a Java routine:
  – CONNECT
  – RELEASE
  – SET CONNECTION

• Routine parameters have different mappings to host language data types than the mappings of routine parameters to host language parameters for other languages.

• The technique for returning result sets from Java stored procedures is different from the technique for returning result sets in other stored procedures.

• When a Java routine executes, Java dynamically loads classes when new class references occur in the class that is being executed. During the class loading process, a java.lang.ClassNotFoundException or java.lang.NoClassDefFoundError can be thrown. These failures can occur whether Java looks for the class in an installed JAR or in the CLASSPATH. If the Java routine does not catch these errors and exceptions, the routine terminates and an SQL error condition is reported.

Related concepts:
  “Differences between Java routines and stand-alone Java programs” on page 202

Related tasks:
  “Writing a Java stored procedure to return result sets” on page 204
  Creating an external stored procedure (DB2 Application programming and SQL)
  Writing an external user-defined function (DB2 Application programming and SQL)

Related reference:
  “Data types that map to database data types in Java applications” on page 219

Static and non-final variables in a Java routine

Using Java variables that are defined as static but not final can cause problems for Java routines.

The reasons for those problems are:

• Use of variables that are static and non-final reduces portability.
  Because the ANSI/ISO standard does not include support for static and non-final variables, different database products might process those variables differently.

• A sequence of routine invocations is not necessarily processed by the same JVM, and static variable values are not shared among different JVMs.
  For example, suppose that two stored procedures, INITIALIZE and PROCESS, use the same static variable, sv1. INITIALIZE sets the value of sv1, and PROCESS depends on the value of sv1. If INITIALIZE runs in one JVM, and then PROCESS runs in another JVM, sv1 in PROCESS does not contain the value that INITIALIZE set.
  Specifying NUMTCB=1 in the WLM-established stored process space startup procedure is not sufficient to guarantee that a sequence of routine invocations go to the same JVM. Under load, multiple stored procedure address spaces are initiated, and each address space has its own JVM. Multiple invocations might be directed to multiple address spaces.
In Java, the static variables for a class are initialized or reset whenever the class is loaded. However, for Java routines, it is difficult to know when initialization or reset of static variables occurs.

In certain cases, you need to declare variables as static and non-final. In those cases, you can use the following technique to make your routines work correctly with static variables.

To determine whether the values of static data in a routine have persisted across routine invocations, define a static boolean variable in the class that contains the routine. Initially set the variable to false, and then set it to true when you set the value of static data. Check the value of the boolean variable at the beginning of the routine. If the value is true, the static data has persisted. Otherwise, the data values need to be set again. With this technique, static data values are not set for most routine invocations, but are set more than once during the lifetime of the JVM. Also, with this technique, it is not a problem for a routine to execute on different JVMs for different invocations.

**Writing a Java stored procedure to return result sets**

You can write your Java stored procedures to return multiple query result sets to a client program.

**Before you begin**

A stored procedure can return multiple query result sets to a client program if the following conditions are satisfied:

- The client supports the DRDA code points that are used to return query result sets.
- The value of DYNAMIC RESULT SETS in the stored procedure definition is greater than 0.

**Procedure**

For each result set that you want to be returned, your Java stored procedure must perform the following actions:

1. For each result set, include an object of type java.sql.ResultSet[] or an array of an SQLJ iterator class in the parameter list for the stored procedure method. If the stored procedure definition includes a method signature, for each result set, include java.sql.ResultSet[] or the fully-qualified name of an array of a class that is declared as an SQLJ iterator in the method signature. These result set parameters must be the last parameters in the parameter list or method signature. Do not include a java.sql.ResultSet array or an iterator array in the SQL parameter list of the stored procedure definition.
2. Execute a SELECT statement to obtain the contents of the result set.
3. Retrieve any rows that you do not want to return to the client.
4. Assign the contents of the result set to element 0 of the java.sql.ResultSet[] object or array of an SQLJ iterator class that you declared in step 1.
5. Do not close the ResultSet, the statement that generated the ResultSet, or the connection that is associated with the statement that generated the ResultSet. DB2 does not return result sets for ResultSets that are closed before the stored procedure terminates.
Example

The following code shows an example of a Java stored procedure that uses an SQLJ iterator to retrieve a result set.

```java
package s1;

import sqlj.runtime.*;
import java.sql.*;
import java.math.*;

#sql iterator NameSal(String LastName, BigDecimal Salary);
public class S1Sal
{
    public static void getSals(BigDecimal[] AvgSalParm,
        java.sql.ResultSet[] rs) throws SQLException
    {
        NameSal iter1;
        try
        {
            #sql iter1 = {SELECT LASTNAME, SALARY FROM EMP
                WHERE SALARY>0 ORDER BY SALARY DESC};
            #sql {SELECT AVG(SALARY) INTO :AvgSalParm[0] FROM EMP};
            catch (SQLException e)
            {
                System.out.println("SQLCODE returned: "+ e.getErrorCode());
                throw(e);
            }
            rs[0] = iter1.getResultSet();
        }
    }
}
```

Figure 47. Java stored procedure that returns a result set

Notes to Figure 47:

1. This SQLJ clause declares the iterator named NameSal, which is used to retrieve the rows that will be returned to the stored procedure caller in a result set.
2. The declaration for the stored procedure method contains declarations for a single passed parameter, followed by the declaration for the result set object.
3. This SQLJ clause executes the SELECT to obtain the rows for the result set, constructs an iterator object that contains those rows, and assigns the iterator object to variable iter1.
4. This SQLJ clause retrieves a value into the parameter that is returned to the stored procedure caller.
5. This statement uses the getResultSet method to assign the contents of the iterator to the result set that is returned to the caller.

Related concepts:

"Retrieving multiple result sets from a stored procedure in an SQLJ application" on page 162

Related tasks:

"Retrieving multiple result sets from a stored procedure in a JDBC application" on page 56

Techniques for testing a Java routine

You can test your Java routines as stand-alone programs, use the IBM Data Studio routine debugger, or write your own debug information from the routines.
Test your routine as a stand-alone program

Before you invoke your Java routines from SQL applications, it is a good idea to run the routines as stand-alone programs, which are easier to debug. A Java program that runs as a routine requires only a DB2 package. However, before you can run the program as a stand-alone program, you need to bind a DB2 plan for it.

Use the IBM Data Studio routine debugger (stored procedures only)

The IBM Data Studio routine debugger is available with DB2 Database for Linux, UNIX, and Windows. The IBM Data Studio routine debugger provides a GUI interface for debugging Java stored procedures. For information on the IBM Data Studio routine debugger, see Routine debugger (IBM Data Studio).

To set up a DB2 for z/OS subsystem to work with the IBM Data Studio routine debugger, when you set up your stored procedure environment, follow these additional steps:

1. Customize and run the DSNTIJRT program to define stored procedures that provide server support for the IBM Data Studio routine debugger.
   DSNTIJSD is in the prefix.SDSNSAMP data set. The job prolog contains customization instructions.
2. Define the stored procedure that you intend to test with the ALLOW DEBUG MODE option in the CREATE PROCEDURE or ALTER PROCEDURE statement.
3. When you prepare the stored procedure for execution, specify the -g option in the javac command.
   -g causes the compiler to generate all debugging information for the program.
4. Grant the DEBUGSESSION privilege to the user who runs the debug client.
5. Make the following modifications to the WLM environment for the stored procedure:
   - In the WLM environment startup procedure, set NUMTCB=1.
   - In the WLM environment startup procedure, include a PSMDEBUG DD statement to direct the debug diagnostic log to a data set. You can allocate to a SYSPUT data set or to a preallocated data set. The data set needs to be created with the RECFM=VBA and LRECL=4096 characteristics.
   - In the ENVAR settings in the JAVAENV data set, set USE_LIBJVM_G=YES.
   - If the debug port range of 8000::8050 is not acceptable, in the ENVAR settings in the JAVAENV data set, set JVM_DEBUG_PORTRANGE to the range of ports that the JVM listens on for debug connections.

Enable collection of DB2 debug information

Include a JSPDEBUG DD statement in your startup procedure for the stored procedure address space. This DD statement specifies a data set to which DB2 writes debug information for use by IBM Software Support.

Write your own debug information from your routine

A useful technique for debugging is to include System.out.println and System.err.println calls in your program to write messages to the STDERR and STDOUT files.
STDERR and STDOUT output is written to the directory that is specified by the WORK_DIR parameter in the JAVAENV data set, if that directory exists. If no WORK_DIR parameter is specified, output goes to the default directory, /tmp/java, if that directory exists.

Related concepts:

“Runtime environment for Java routines” on page 193
Chapter 6. Preparing and running JDBC and SQLJ programs

You prepare and run DB2 for z/OS Java programs in the z/OS UNIX System Services environment.

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 or bind steps are required.

For example, to prepare the Sample01.java program for execution, execute this command from the /usr/lpp/db2a10/jdbc/ directory:

javac Sample01.java

Program preparation for SQLJ programs

Program preparation for SQLJ programs involves translating, compiling, customizing, and binding programs.

About this task

The following figure shows the steps of the program preparation process for a program that uses the IBM Data Server Driver for JDBC and SQLJ.

Figure 48. The SQLJ program preparation process
Procedure

The basic steps in SQLJ program preparation are:

1. Run the sqlj command from the z/OS UNIX System Services command line to translate and compile the source code.
   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.

2. Run the db2sqljcustomize command from the z/OS UNIX System Services command line to customize the serialize profiles and bind DB2 packages.
   The db2sqljcustomize command performs these tasks:
   • Customizes the serialized profiles.
   • Optionally does online checking to ensure that application variable types are compatible with the corresponding column data types.
     The default behavior is to do online checking. For better performance, you should do online checking.
   • Optionally binds DB2 packages on a specified database server.
     The default behavior is to bind the DB2 packages. However, you can disable automatic creation of packages and use the db2sqljbind command to bind the packages later.
     You might also need to run the db2sqljbind command under these circumstances:
     – If a bind fails when db2sqljcustomize runs
     – if you want to create identical packages at multiple locations for the same serialized profile

3. Optional: Bind the DB2 packages into a plan.
   Use the DB2 BIND command to do that.

Related reference:

“sqlj - SQLJ translator” on page 498
“db2sqljbind - SQLJ profile binder” on page 513
“db2sqljcustomize - SQLJ profile customizer” on page 502

Binding SQLJ applications to access multiple database servers

After you prepare an SQLJ program to run on one DB2 database server, you might want to port that application to other environments that access different database servers. For example, you might want to move your application from a test environment to a production environment.

Procedure

The general steps for enabling access of an existing SQLJ application to additional database servers are:

1. Bind packages on each database server that you want to access.
   Do not re-customize the serialized profiles. Customization stores a new package timestamp in the serialized profile, which makes the new serialized profile incompatible with the original package.
   You can use one of the following methods to bind the additional DB2 packages:
• Run the db2sqbjbind command against each of the database servers.

• Run the DB2 BIND PACKAGE command with the COPY option to copy the original packages to each of the additional database servers.

You might need a different qualifier for unqualified DB2 objects on each of the database servers. In that case, you need to specify a value for the QUALIFIER bind option when you bind the new packages. If you use the db2sqbjbind command, you specify the QUALIFIER option in the -bindoptions parameter, not in the -qualifier parameter. The -qualifier parameter applies to online checking only.

2. Specify the package collection for the DB2 packages.

By default, when an SQLJ application runs, the DB2 database server looks for packages using the collection ID that is stored in the serialized profile. If the collection ID for the additional DB2 packages that you create is different from the collection ID in the serialized profile, you need to override the collection ID that is in the serialized profile. You can do that in one of the following ways:

• Specify the collection ID with the pkList DataSource property or the db2.jcc.pkList global property.

• Follow these steps:
  a. Bind a plan for the application that includes the following packages:
     – The package collection that you bound in the previous step
     – The IBM Data Server Driver for JDBC and SQLJ packages
  b. Specify the plan name in the planName DataSource property or the db2.jcc.planName global property.

Binding a plan might simplify authorization for the application. You can authorize users to execute the plan, rather than authorizing them to execute each of the packages in the plan.

Example

An existing SQLJ application was customized and bound using the following db2sqljcustomize invocation:

```
db2sqljcustomize -url jdbc:db2://system1.svl.ibm.com:8000/ZOS1
     -user user01 -password mypass
     -rootPkgName WRKSQLJ
     -qualifier WRK1
     -collection MYCOL1
     -bindoptions "CURRENTDATA NO QUALIFIER WRK1"
     -staticpositioned YES WrkTraceTest_SJProfile0.ser
```

In addition to accessing data at the location that is indicated by URL jdbc:db2://system1.svl.ibm.com:8000/ZOS1, you want to use the application to access data at the location that is indicated by jdbc:db2://system2.svl.ibm.com:8000/ZOS2. On the ZOS2 system, DB2 objects have a qualifier of WRK2, and the packages need to be in collection MYCOL2. You therefore need to bind packages at location ZOS2, change the default qualifier to WRK2, and specify the MYCOL2 collection for the packages. Use one of the following methods to bind the packages:

• Run DB2 BIND with COPY to copy each of the packages (one for each isolation level) from the ZOS1 system to the ZOS2 system:

```
BIND PACKAGE (ZOS2.MYCOL2) OWNER(USER01) QUALIFIER(WRK2) - COPY(MYCOL.WRKSQLJ1) CURRENTDATA(NO)
BIND PACKAGE (ZOS2.MYCOL2) OWNER(USER01) QUALIFIER(WRK2) - COPY(MYCOL.WRKSQLJ2) CURRENTDATA(NO)
```
- Run the `db2sqljbind` command to create DB2 packages on ZOS2 from the serialized profile on ZOS1:
  
  ```
  db2sqljbind -url jdbc:db2://system2.svl.ibm.com:8000/ZOS2
  -user user01 -password mypass
  -bindoptions "COLLECTION MYCOL2 QUALIFIER WRK2"
  -staticpositioned YES WrkTraceTest_SJProfile0.ser
  ```

  After you bind the packages, you need to ensure that when the application runs, the DB2 database server at ZOS2 can find the packages. The collection ID in the serialized profile is MYCOL1, so the DB2 database server looks in MYCOL1 for the packages. When you run the application against the ZOS2 system, you need to access packages in MYCOL2.

  For applications that use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity, use one of the following methods to tell the database server to look in MYCOL2 as well as MYCOL1:
  
  - Specify "MYCOL1.*,MYCOL2.*" in the pkList `DataSource` property:
    ```
    pklist = MYCOL1.*,MYCOL2.*
    ```
  - Bind a plan for the application that includes the packages in MYCOL2 and the IBM Data Server Driver for JDBC and SQLJ packages:
    ```
    BIND PLAN(WRKSQLJ) PKLIST(MYCOL1.*,MYCOL2.*,JDBCCOL.*)
    ```
    Then specify WRKSQLJ in the `planName` `DataSource` property:
    ```
    planName = WRKSQLJ
    ```

  For applications that use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, specify "MYCOL1,MYCOL2" in the `currentPackagePath` `DataSource` property.

  Related tasks:
  
  "Program preparation for SQLJ programs" on page 209

  Related reference:
  
  "db2sqljbind - SQLJ profile binder" on page 513

---

**Program preparation for Java routines**

The program preparation process for Java routines varies, depending on whether the routines contain SQLJ clauses.

The following topics contain detailed information on program preparation for Java routines.

**Preparation of Java routines with no SQLJ clauses**

Java routines that contain no SQLJ clauses are written entirely in JDBC. You can use one of three methods to prepare Java routines with no SQLJ statements.

Those methods are:

- Prepare the Java routine to run from a JAR file. Running Java routines from JAR files is recommended.
- Prepare the Java routine with no JAR file.
• Use IBM Optim Development Studio to prepare the routine.
  You can use this method regardless of whether the routine is in a JAR file.

**Preparing Java routines with no SQLJ clauses to run from a JAR file**

The recommended method of running Java routines is to run them from a JAR file.

**About this task**

The steps in this task prepare a JDBC routine for execution, create a JAR file for the routine, define the JAR file and routine to DB2, and grant access on the routine to users.

**Procedure**

1. Run the `javac` command to compile the Java program to produce Java bytecodes.
2. Run the `jar` command to collect the class files that contain the methods for your routine into a JAR file. See “Creating JAR files for Java routines” for information on creating the JAR file.
3. Call the `INSTALL_JAR` stored procedure to define the JAR file to DB2.
4. If the installed JAR references classes in other installed JARs, call the `SQLJ.ALTER_JAVA_PATH` stored procedure to specify the class resolution path that the JVM searches to resolve those class references.
5. If another user defines the routine to DB2, execute the SQL GRANT USAGE ON JAR statement to grant the privilege to use the JAR file to that user.
6. Execute the SQL CREATE PROCEDURE or CREATE FUNCTION statement to define the routine to DB2. Specify the `EXTERNAL_NAME` parameter with the name of the JAR that you defined to DB2 in step 3.
7. Execute the SQL GRANT statement to grant the EXECUTE privilege on the routine to the appropriate users.

**Related concepts:**

“Program preparation for JDBC programs” on page 209
“Definition of a JAR file for a Java routine to DB2” on page 201

**Related tasks:**

“Creating JAR files for Java routines” on page 217

**Preparing Java routines with no SQLJ clauses and no JAR file**

If you do not use a JAR file for a Java routine that has no SQLJ clauses, you need to include the directories for the routine classes in the CLASSPATH.

**About this task**

The steps in this task compile source code, add the locations of the resulting class files to the CLASSPATH, define the routine to DB2, and grant access on the routine to users.

**Procedure**

1. Run the `javac` command to compile the Java program to produce Java bytecodes.
2. Ensure that the zFS or HFS directory that contains the class files for your routine is in the CLASSPATH for the WLM-established stored procedure address space.
You specify this CLASSPATH in the JAVAENV data set. You specify the JAVAENV data set using a JAVAENV DD statement in the startup procedure for the WLM-established stored procedure address space.

If you need to modify the CLASSPATH environment variable in the JAVAENV data set to include the directory for the Java routine's classes, you must restart the WLM address space to make it use the modified CLASSPATH.

3. Execute the SQL CREATE PROCEDURE or CREATE FUNCTION statement to define the routine to DB2. Specify the EXTERNAL NAME parameter without a JAR name.

4. Execute the SQL GRANT statement to grant the EXECUTE privilege on the routine to the appropriate users.

Related concepts:
- “Program preparation for JDBC programs” on page 209
- “Runtime environment for Java routines” on page 193

Preparation of Java routines with SQLJ clauses

You can use one of three methods to prepare Java routines with SQLJ clauses.

Those methods are:
- Prepare the routine Java routine to run from a JAR file. Running Java routines from JAR files is recommended.
- Prepare the routine Java routine with no JAR file.
- Use IBM Optim Development Studio to prepare the routine.
  You can use this method regardless of whether the routine is in a JAR file.

Preparing Java routines with SQLJ clauses to run from a JAR file

The recommended method of running Java routines with SQLJ clauses is to run them from a JAR file.

About this task

The steps in this task prepare an SQLJ routine for execution, create JAR files for the methods in the routine, define the JAR files to DB2, define the routine to DB2, and grant access on the routine to users.

Procedure

1. Run the sqlj command to translate the source code to produce generated Java source code and serialized profiles, and to compile the Java program to produce Java bytecodes.
2. Run the db2sqljcustomize command to produce serialized profiles that are customized for DB2 for z/OS and DB2 packages.
3. Run the jar command to package the class files that contain the methods for your routine, and the profiles that you generated in step 2 into a JAR file. See "Creating JAR files for Java routines" for information on creating the JAR file.
4. Call the INSTALL_JAR stored procedure to define the JAR file to DB2.
5. If the installed JAR references classes in other installed JARs, call the SQLJ.ALTER_JAVA_PATH stored procedure to specify the class resolution path that the JVM searches to resolve those class references.
6. If another user defines the routine to DB2, execute the SQL GRANT USAGE ON JAR statement to grant the privilege to use the JAR file to that user.
7. Execute the SQL CREATE PROCEDURE or CREATE FUNCTION statement to define the routine to DB2. Specify the EXTERNAL NAME parameter with the name of the JAR that you defined to DB2 in step 4 on page 214.

8. Execute the SQL GRANT statement to grant the EXECUTE privilege on the routine to the appropriate users.

Example

The following example demonstrates how to prepare a Java stored procedure that contains SQLJ clauses for execution from a JAR file.

1. On z/OS UNIX System Services, run the sqlj command to translate and compile the SQLJ source code.

   Assume that the path for the stored procedure source program is /u/db2res3/s1/s1sal.sqlj. Change to directory /u/db2res3/s1, and issue this command:

   sqlj s1sal.sqlj

   After this process completes, the /u/db2res3/s1 directory contains these files:

   s1sal.java
   s1sal.class
   s1sal_SJProfile0.ser

2. On z/OS UNIX System Services, run the db2sqljcustomize command to produce serialized profiles that are customized for DB2 for z/OS and to bind the DB2 packages for the stored procedure.

   Change to the /u/db2res3 directory, and issue this command:

   db2sqljcustomize -url jdbc:db2://mvs1:446/SJCEC1 \
   -user db2adm -password db2adm \
   -bindoptions "EXPLAIN YES" \
   -collection ADMCOLL \
   -rootpkgname S1SAL \
   s1sal_SJProfile0.ser

   After this process completes, s1sal_SJProfile0.ser contains a customized serialized profile. The DB2 subsystem contains these packages:

   S1SAL1
   S1SAL2
   S1SAL3
   S1SAL4

3. On z/OS UNIX System Services, run the jar command to package the class files that you created in step 1 and the customized serialized profile that you created in step 2 into a JAR file.

   Change to the /u/db2res3 directory, and issue this command:

   jar -cvf s1sal.jar s1/*.class s1/*.ser

   After this process completes, the /u/db2res3 directory contains this file:

   s1sal.jar

4. Call the INSTALL_JAR stored procedure, which is on DB2 for z/OS, to define the JAR file to DB2.

   You need to execute the CALL statement from a static SQL program or from an ODBC or JDBC program. The CALL statement looks similar to this:

   CALL SQLJ.INSTALL_JAR('file:/u/db2res3/s1sal.jar','MYSCHEMA.S1SAL',0);

   The exact form of the CALL statement depends on the language of the program that issues the CALL statement.
After this process completes, the DB2 catalog contains JAR file MYSCHEMA.S1SAL.

5. If the installed JAR references classes in other installed JARs, call the SQLJ.ALTER_JAVA_PATH stored procedure, which is on DB2 for z/OS, to specify the class resolution path that the JVM searches to resolve those class references. You need to execute the CALL statement from a static SQL program or from an ODBC or JDBC program.

6. If another user defines the routine to DB2, on DB2 for z/OS, execute the SQL GRANT USAGE ON JAR statement to grant the privilege to use the JAR file to that user.

   Suppose that you want any user to be able to define the stored procedure to DB2. This means that all users need the USAGE privilege on JAR MYSCHEMA.S1SAL. To grant this privilege, execute this SQL statement:
   
   ```sql
   GRANT USAGE ON JAR MYSCHEMA.S1SAL TO PUBLIC;
   ```

7. On DB2 for z/OS, execute the SQL CREATE PROCEDURE statement to define the stored procedure to DB2:

   ```sql
   CREATE PROCEDURE SYSPROC.S1SAL
       (DECIMAL(10,2) INOUT)
       FENCED
       MODIFIES SQL DATA
       COLLID ADMCOLL
       LANGUAGE JAVA
       EXTERNAL NAME 'MYSCHEMA.S1SAL:s1.S1Sal.getSals'
       WLM ENVIRONMENT WLMIJAVA
       DYNAMIC RESULT SETS 1
       PROGRAM TYPE SUB
       PARAMETER STYLE JAVA;
   ```

8. On DB2 for z/OS, execute the SQL GRANT EXECUTE statement to grant the privilege to run the routine to that user.

   Suppose that you want any user to be able to run the routine. This means that all users need the EXECUTE privilege on SYSPROC.S1SAL. To grant this privilege, execute this SQL statement:

   ```sql
   GRANT EXECUTE ON PROCEDURE SYSPROC.S1SAL TO PUBLIC;
   ```

Related concepts:

- "Definition of a JAR file for a Java routine to DB2” on page 201

Related tasks:

- "Program preparation for SQLJ programs” on page 209
- "Creating JAR files for Java routines” on page 217

Preparing Java routines with SQLJ clauses and no JAR file

If you do not use a JAR file for a Java routine that contains SQLJ clauses, you need to include the directories for the routine classes in the CLASSPATH.

About this task

The steps in this task prepare an SQLJ routine for execution, specify the class files for the routine in the CLASSPATH, define the routine to DB2, and grant access on the routine to users.

Procedure

1. Run the sqlj command to translate the source code to produce generated Java source code and serialized profiles, and to compile the Java program to produce Java bytecodes.
2. Run the db2sqljcustomize command to produce serialized profiles that are customized for DB2 for z/OS and DB2 packages.

3. Ensure that the zFS or HFS directory that contains the class files for your routine is in the CLASSPATH for the WLM-established stored procedure address space.
   
   You specify this CLASSPATH in the JAVAENV data set. You specify the JAVAENV data set using a JAVAENV DD statement in the startup procedure for the WLM-established stored procedure address space.
   
   If you need to modify the CLASSPATH environment variable in the JAVAENV data set to include the directory for the Java routine's classes, you must restart the WLM address space to make it use the modified CLASSPATH.

4. Use the SQL CREATE PROCEDURE or CREATE FUNCTION statement to define the routine to DB2. Specify the EXTERNAL NAME parameter without a JAR name.

5. Execute the SQL GRANT statement to grant the EXECUTE privilege on the routine to the appropriate users.

Related concepts:
“Runtime environment for Java routines” on page 193

Related tasks:
“Program preparation for SQLJ programs” on page 209

Creating JAR files for Java routines

A convenient way to ensure that all modules of a Java routine are accessible is to store those modules in a JAR file. You create the JAR file by running the jar command in z/OS UNIX System Services.

Before you begin

For a JDBC routine, before you can create a JAR file, you need to compile the source code. For an SQLJ routine, before you can create a JAR file, you need to translate, compile, and customize the source code.

Procedure

1. If the Java source file does not contain a package statement, change to the directory that contains the class file for the Java routine, which you created by running the javac command.
   
   For example, if JDBC routine Add_customer.java is in /u/db2res3/acmejos, change to directory /u/db2res3/acmejos.
   
   If the Java source file contains a package statement, change to the directory that is one level above the directory that is named in the package statement.
   
   For example, suppose the package statement is:
   
   ```
   package lvlOne.lvlTwo.lvlThree;
   ```
   
   Change to the directory that contains lvlOne as an immediate subdirectory.

2. Run the jar command.
   
   You might need to specify at least these options:

   - `c` Creates a new or empty archive.
   - `v` Generates verbose output on stderr.
   - `f` Specifies that the argument immediately after the options list is the name of the JAR file to be created.
For example, to create a JAR file named acmejos.jar from Add_customer.class, which is in package acmejos, execute this jar command:

```
jar -cvf acmejos.jar acmejos/Add_customer.class
```

To create a JAR file for an SQLJ routine, you also need to include all generated class files, such as classes that are generated for iterators, and all serialized profile files. For example, suppose that all classes are declared to be in package acmejos, and all class files, including generated class files, and all serialized profile files for SQLJ routine Add_customer.sqlj are in directory /u/db2res3/acmejos/. To create a JAR file named acmejos.jar, change the the /u/db2res3 directory, and then issue this jar command:

```
jar -cvf acmejos.jar acmejos/*.class acmejos/*.ser
```

---

## 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.

### About this task

These steps allow you to run a JDBC or SQLJ program.

### Procedure

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 on the z/OS UNIX System Services command line, with the top-level file name in the program as the argument.

### Example

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
```

### Related concepts:

[“Environment variables for the IBM Data Server Driver for JDBC and SQLJ” on page 520](#)
Chapter 7. 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 on Linux, UNIX, and Windows systems, 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, and for input host expressions in SQLJ programs. When more than one Java data type is listed, the first data type is the recommended data type.

<table>
<thead>
<tr>
<th>Java data type</th>
<th>Database data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>short, java.lang.Short</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>boolean, byte, java.lang.Boolean, java.lang.Byte</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>int, java.lang.Integer</td>
<td>INTEGER</td>
</tr>
<tr>
<td>long, java.lang.Long</td>
<td>BIGINT</td>
</tr>
<tr>
<td>java.math.BigInteger</td>
<td>BIGINT</td>
</tr>
<tr>
<td>java.math.BigInteger</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>float, java.lang.Float</td>
<td>REAL</td>
</tr>
<tr>
<td>double, java.lang.Double</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>DECFLOAT(n)</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>GRAPHIC(m)</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>VARGRAPHIC(m)</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>CLOB</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>XML</td>
</tr>
<tr>
<td>byte[]</td>
<td>CHAR(n) FOR BIT DATA</td>
</tr>
<tr>
<td>byte[]</td>
<td>VARCHAR(n) FOR BIT DATA</td>
</tr>
<tr>
<td>byte[]</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>byte[]</td>
<td>VARBINARY(n)</td>
</tr>
<tr>
<td>byte[]</td>
<td>BLOB</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1998, 2017
<table>
<thead>
<tr>
<th>Java data type</th>
<th>Database data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte[]</td>
<td>ROWID</td>
</tr>
<tr>
<td>byte[]</td>
<td>XML</td>
</tr>
<tr>
<td>java.sql.Blob</td>
<td>BLOB</td>
</tr>
<tr>
<td>java.sql.Blob</td>
<td>XML</td>
</tr>
<tr>
<td>java.sql.Clob</td>
<td>CLOB</td>
</tr>
<tr>
<td>java.sql.Clob</td>
<td>DBCLOB</td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>DATE</td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>TIME</td>
</tr>
<tr>
<td>java.sql.Timestamp</td>
<td>TIMESTAMP, TIMESTAMP(p), TIMESTAMP WITH TIME ZONE,</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP(p) WITH TIME ZONE,</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>java.io.ByteArrayInputStream</td>
<td>BLOB</td>
</tr>
<tr>
<td>java.io.StringReader</td>
<td>CLOB</td>
</tr>
<tr>
<td>java.io.ByteArrayInputStream</td>
<td>CLOB</td>
</tr>
<tr>
<td>java.io.InputStream</td>
<td>XML</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2RowID (deprecated)</td>
<td>ROWID</td>
</tr>
<tr>
<td>java.sql.RowId</td>
<td>ROWID</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2Xml (deprecated)</td>
<td>XML</td>
</tr>
<tr>
<td>java.sql.SQLXML</td>
<td>XML</td>
</tr>
<tr>
<td>java.util.Date</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>java.util.Date</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>java.util.Date</td>
<td>DATE</td>
</tr>
<tr>
<td>java.util.Date</td>
<td>TIME</td>
</tr>
<tr>
<td>java.util.Date</td>
<td>TIMESTAMP, TIMESTAMP(p), TIMESTAMP WITH TIME ZONE,</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP(p) WITH TIME ZONE,</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Calendar</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>java.util.Calendar</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>java.util.Calendar</td>
<td>DATE</td>
</tr>
<tr>
<td>java.util.Calendar</td>
<td>TIME</td>
</tr>
<tr>
<td>java.util.Calendar</td>
<td>TIMESTAMP, TIMESTAMP(p), TIMESTAMP WITH TIME ZONE,</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP(p) WITH TIME ZONE,</td>
</tr>
</tbody>
</table>
Table 29. Mappings of Java data types to database server data types for updating database tables (continued)

<table>
<thead>
<tr>
<th>Java data type</th>
<th>Database data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notes:</td>
<td></td>
</tr>
<tr>
<td>1. For column updates, the data server has no exact equivalent for the Java boolean or byte data types, but the best fit is SMALLINT.</td>
<td></td>
</tr>
<tr>
<td>2. ( p ) is the decimal precision and ( s ) is the scale of the table column. You should design financial applications so that java.math.BigDecimal columns map to DECIMAL columns. If you know the precision and scale of a DECIMAL column, updating data in the DECIMAL column with data in a java.math.BigDecimal variable results in better performance than using other combinations of data types.</td>
<td></td>
</tr>
<tr>
<td>3. ( n=16 ) or ( n=34 ).</td>
<td></td>
</tr>
<tr>
<td>4. DECFLOAT is valid for connections to DB2 Version 9.1 for z/OS, DB2 V9.5 for Linux, UNIX, and Windows, or Db2 for IBM iV6R1, or later database servers. Use of DECFLOAT requires the SDK for Java Version 5 (1.5) or later.</td>
<td></td>
</tr>
<tr>
<td>5. ( n\leq255 ).</td>
<td></td>
</tr>
<tr>
<td>6. ( m\leq127 ).</td>
<td></td>
</tr>
<tr>
<td>7. ( n\leq32704 ).</td>
<td></td>
</tr>
<tr>
<td>8. ( m\leq16352 ).</td>
<td></td>
</tr>
<tr>
<td>9. This mapping is valid only if the database server can determine the data type of the column.</td>
<td></td>
</tr>
<tr>
<td>10. XML is valid for connections to DB2 Version 9.1 for z/OS or later database servers or DB2 V9.1 for Linux, UNIX, and Windows or later database servers.</td>
<td></td>
</tr>
<tr>
<td>11. This mapping is valid only for IBM Data Server Driver for JDBC and SQLJ version 4.13 or later.</td>
<td></td>
</tr>
<tr>
<td>12. BIGINT is valid for connections to DB2 Version 9.1 for z/OS or later database servers, DB2 V9.1 for Linux, UNIX, and Windows or later database servers, and all supported Db2 for IBM i database servers.</td>
<td></td>
</tr>
<tr>
<td>13.</td>
<td></td>
</tr>
<tr>
<td>14. ( p ) indicates the timestamp precision, which is the number of digits in the fractional part of the timestamp. ( 0\leq p \leq 12 ). The default is 6. TIMESTAMP(p) is supported for connections to Db2 on Linux, UNIX, and Windows systems V9.7 and later and DB2 for z/OS V10 and later only.</td>
<td></td>
</tr>
<tr>
<td>15. The WITH TIME ZONE clause is supported for connections to DB2 for z/OS V10 and later only.</td>
<td></td>
</tr>
</tbody>
</table>

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 30. Mappings of database server data types to Java data types for retrieving data from database server tables

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>Recommended Java data type or Java object type</th>
<th>Other supported Java data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>short</td>
<td>byte, int, long, float, double, java.math.BigDecimal, boolean, java.lang.String</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
<td>short, byte, long, float, double, java.math.BigDecimal, boolean, java.lang.String</td>
</tr>
<tr>
<td>BIGINT</td>
<td>long</td>
<td>int, short, byte, float, double, java.math.BigDecimal, boolean, java.lang.String</td>
</tr>
<tr>
<td>DECIMAL((p,s)) or NUMERIC((p,s))</td>
<td>java.math.BigDecimal</td>
<td>long, int, short, byte, float, double, boolean, java.lang.String</td>
</tr>
</tbody>
</table>
Table 30. Mappings of database server data types to Java data types for retrieving data from database server tables (continued)

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>Recommended Java data type or Java object type</th>
<th>Other supported Java data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL</td>
<td>float</td>
<td>long, int, short, byte, double, java.math.BigDecimal, boolean, java.lang.String</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double</td>
<td>long, int, short, byte, float, java.math.BigDecimal, boolean, java.lang.String</td>
</tr>
<tr>
<td>CHAR(n) FOR BIT DATA</td>
<td>byte[]</td>
<td>java.lang.String, java.io.InputStream, java.io.Reader</td>
</tr>
<tr>
<td>VARCHAR(n) FOR BIT DATA</td>
<td>byte[]</td>
<td>java.lang.String, java.io.InputStream, java.io.Reader</td>
</tr>
<tr>
<td>BINARY(n) [3]</td>
<td>byte[]</td>
<td>None</td>
</tr>
<tr>
<td>VARBINARY(n) [4]</td>
<td>byte[]</td>
<td>None</td>
</tr>
<tr>
<td>GRAPHIC(m)</td>
<td>java.lang.String</td>
<td>long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Timestamp, java.io.InputStream, java.io.Reader</td>
</tr>
<tr>
<td>VARGRAPHIC(m)</td>
<td>java.lang.String</td>
<td>long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Timestamp, java.io.InputStream, java.io.Reader</td>
</tr>
<tr>
<td>CLOB(n)</td>
<td>java.sql.Clob</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>BLOB(n)</td>
<td>java.sql.Blob</td>
<td>byte[]</td>
</tr>
<tr>
<td>DBCLOB(m)</td>
<td>No exact equivalent. Use java.sql.Clob.</td>
<td></td>
</tr>
<tr>
<td>ROWID</td>
<td>java.sql.RowId</td>
<td>byte[], com.ibm.db2.jcc.DB2RowID (deprecated)</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
<td>java.sql.String, java.sql.Timestamp</td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Time</td>
<td>java.sql.String, java.sql.Timestamp</td>
</tr>
</tbody>
</table>
Table 30. Mappings of database server data types to Java data types for retrieving data from database server tables (continued)

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>Recommended Java data type or Java object type</th>
<th>Other supported Java data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP, TIMESTAMP(p), TIMESTAMP WITH TIME ZONE, TIMESTAMP(p) WITH TIME ZONE</td>
<td>java.sql.Timestamp</td>
<td>java.sql.String, java.sql.Date, java.sql.Time, java.sql.Timestamp</td>
</tr>
</tbody>
</table>

Notes:
1. \( n=16 \) or \( n=34 \).
2. DECFLOAT is valid for connections to DB2 Version 9.1 for z/OS, DB2 V9.5 for Linux, UNIX, and Windows, or Db2 for IBM i V6R1, or later database servers. Use of DECFLOAT requires the SDK for Java Version 5 (1.5) or later.
3. This mapping is valid only if the database server can determine the data type of the column.
4. XML is valid for connections to DB2 Version 9.1 for z/OS or later database servers or DB2 V9.1 for Linux, UNIX, and Windows or later database servers.
5. BIGINT is valid for connections to v Version 9.1 for z/OS or later database servers, DB2 V9.1 for Linux, UNIX, and Windows or later database servers, and all supported Db2 for IBM i database servers.
6. 
7. \( p \) indicates the timestamp precision, which is the number of digits in the fractional part of the timestamp. \( 0 \leq p \leq 12 \). The default is 6. TIMESTAMP(p) is supported for connections to Db2 on Linux, UNIX, and Windows systems V9.7 and later and DB2 for z/OS V10 and later only.
8. The WITH TIME ZONE clause is supported for connections to DB2 for z/OS V10 and later only.

---

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 31. Mappings of Java, JDBC, and SQL data types for calling stored procedures and user-defined functions

<table>
<thead>
<tr>
<th>Java data type</th>
<th>JDBC data type</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean java.lang.Boolean</td>
<td>BIT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>byte java.lang.Byte</td>
<td>TINYINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>short java.lang.Short</td>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>int java.lang.Integer</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>long java.lang.Long</td>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>float java.lang.Float</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>double java.lang.Double</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>java.types.OTHER</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>com.ibm.db2.jcc.DB2Types.DECFLOAT</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>Java data type</td>
<td>JDBC data type</td>
<td>SQL data type</td>
</tr>
<tr>
<td>------------------------------</td>
<td>----------------</td>
<td>--------------</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>CHAR</td>
<td>CHAR</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>CHAR</td>
<td>GRAPHIC</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>VARCHAR</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>VARCHAR</td>
<td>VARGRAPHIC</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>LONGVARCHAR</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>VARCHAR</td>
<td>CLOB</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>LONGVARCHAR</td>
<td>CLOB</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>CLOB</td>
<td>CLOB</td>
</tr>
<tr>
<td>byte[]</td>
<td>BINARY</td>
<td>CHAR FOR BIT DATA</td>
</tr>
<tr>
<td>byte[]</td>
<td>VARBINARY</td>
<td>VARCHAR FOR BIT DATA</td>
</tr>
<tr>
<td>byte[]</td>
<td>BINARY</td>
<td>BINARY</td>
</tr>
<tr>
<td>byte[]</td>
<td>VARBINARY</td>
<td>BINARY</td>
</tr>
<tr>
<td>byte[]</td>
<td>LONGVARBINARY</td>
<td>VARCHAR FOR BIT DATA</td>
</tr>
<tr>
<td>byte[]</td>
<td>VARBINARY</td>
<td>BLOB</td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>java.sql.Timestamp</td>
<td>TIME</td>
<td>TIME</td>
</tr>
<tr>
<td>java.sql.Timestamp</td>
<td>TIMESTAMP</td>
<td>TIMESTAMP, TIMESTAMP(p), TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH TIME ZONE(p)</td>
</tr>
<tr>
<td>java.sql.Blob</td>
<td>BLOB</td>
<td>BLOB</td>
</tr>
<tr>
<td>java.sql.Clob</td>
<td>CLOB</td>
<td>CLOB</td>
</tr>
<tr>
<td>java.sql.Clob</td>
<td>CLOB</td>
<td>DBCLOB</td>
</tr>
<tr>
<td>java.io.ByteArrayInputStream</td>
<td>None</td>
<td>BLOB</td>
</tr>
<tr>
<td>java.io.StringReader</td>
<td>None</td>
<td>CLOB</td>
</tr>
<tr>
<td>java.io.ByteArrayInputStream</td>
<td>None</td>
<td>CLOB</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2RowID</td>
<td>com.ibm.db2.jdbc.DB2Types.ROWID</td>
<td>ROWID</td>
</tr>
<tr>
<td>java.sql.RowId</td>
<td>java.sql.Types.ROWID</td>
<td>ROWID</td>
</tr>
<tr>
<td>java.sql.SQLXML</td>
<td>java.sql.Types.SQLXML</td>
<td>XML</td>
</tr>
<tr>
<td>java.sql.ResultSet</td>
<td>com.ibm.db2.jdbc.DB2Types.CURSOR</td>
<td>CURSOR type</td>
</tr>
</tbody>
</table>
Table 31. Mappings of Java, JDBC, and SQL data types for calling stored procedures and user-defined functions (continued)

<table>
<thead>
<tr>
<th>Java data type</th>
<th>JDBC data type</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notes:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. A DB2 for z/OS stored procedure or user-defined function parameter cannot have the XML data type. |
2. 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. |
3. DECFLOAT parameters in Java routines are valid only for connections to DB2 Version 9.1 for z/OS or later database servers. DECFLOAT parameters in Java routines are not supported for connections to for Linux, UNIX, and Windows or Db2 for IBM i. Use of DECFLOAT requires the SDK for Java Version 5 (1.5) or later. |
4. This mapping is valid only if the database server can determine the data type of the column. |
5. |
6. BIGINT is valid for connections to DB2 Version 9.1 for z/OS or later database servers, DB2 V9.1 for Linux, UNIX, and Windows or later database servers, and all supported v database servers. |
7. p indicates the timestamp precision, which is the number of digits in the fractional part of the timestamp. 0<=p<=12. The default is 6. TIMESTAMP(p) is supported for connections to Db2 on Linux, UNIX, and Windows systems V9.7 and later and DB2 for z/OS V10 and later only. |
8. The WITH TIME ZONE clause is supported for connections to DB2 for z/OS V10 and later only. |

Data types in Java stored procedures and user-defined functions

The following table summarizes mappings of the SQL parameter data types in a CREATE PROCEDURE or CREATE FUNCTION statement to the data types in the corresponding Java stored procedure or user-defined function method.

For Db2 on Linux, UNIX, and Windows systems, if more than one Java data type is listed for an SQL data type, only the first Java data type is valid.

For DB2 for z/OS, if more than one Java data type is listed, and you use a data type other than the first data type as a method parameter, you need to include a method signature in the EXTERNAL clause of your CREATE PROCEDURE or CREATE FUNCTION statement that specifies the Java data types of the method parameters.

Table 32. Mappings of SQL data types in a CREATE PROCEDURE or CREATE FUNCTION statement to data types in the corresponding Java stored procedure or user-defined function program

<table>
<thead>
<tr>
<th>SQL data type in CREATE PROCEDURE or CREATE FUNCTION</th>
<th>Data type in Java stored procedure or user-defined function method</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>short, java.lang.Integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int, java.lang.Integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>long, java.lang.Long</td>
</tr>
<tr>
<td>REAL</td>
<td>float, java.lang.Float</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double, java.lang.Double</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>CHAR</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>CHAR FOR BIT DATA</td>
<td>byte[]</td>
</tr>
<tr>
<td>VARCHAR FOR BIT DATA</td>
<td>byte[]</td>
</tr>
</tbody>
</table>
Table 32. Mappings of SQL data types in a CREATE PROCEDURE or CREATE FUNCTION statement to data types in the corresponding Java stored procedure or user-defined function program (continued)

<table>
<thead>
<tr>
<th>SQL data type in CREATE PROCEDURE or CREATE FUNCTION</th>
<th>Data type in Java stored procedure or user-defined function method</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY</td>
<td>byte[]</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>byte[]</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Time</td>
</tr>
<tr>
<td>TIMESTAMP, TIMESTAMP(p), TIMESTAMP WITH TIME ZONE,</td>
<td>java.sql.Timestamp</td>
</tr>
<tr>
<td>TIMESTAMP(p) WITH TIME ZONE</td>
<td></td>
</tr>
<tr>
<td>BLOB</td>
<td>java.sql.Blob</td>
</tr>
<tr>
<td>CLOB</td>
<td>java.sql.Clob</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>java.sql.Clob</td>
</tr>
<tr>
<td>ROWID</td>
<td>java.sql.Types.ROWID</td>
</tr>
</tbody>
</table>

Notes:
1. A DB2 for z/OS stored procedure or user-defined function parameter cannot have the XML data type.
2. For a stored procedure or user-defined function on a Db2 on Linux, UNIX, and Windows systems server, only the first data type is valid.
3. BIGINT is valid for connections to DB2 Version 9.1 for z/OS or later database servers or DB2 V9.1 for Linux, UNIX, and Windows or later database servers.
4. DECIMAL parameters in Java routines are valid only for connections to DB2 Version 9.1 for z/OS or later database servers. DECIMAL parameters in Java routines are not supported for connections to Linux, UNIX, and Windows or Db2 for IBM i. Use of DECIMAL requires the SDK for Java Version 5 (1.5) or later.
5. The WITH TIME ZONE clause is supported for connections to DB2 for z/OS V10 and later only.
6. p indicates the timestamp precision, which is the number of digits in the fractional part of the timestamp. $0 \leq p \leq 12$. The default is 6. TIMESTAMP(p) is supported for connections to Db2 on Linux, UNIX, and Windows systems V9.7 and later and DB2 for z/OS V10 and later only.
7. The WITH TIME ZONE clause is supported for connections to DB2 for z/OS V10 and later only.

**Date, time, and timestamp values that can cause problems in JDBC and SQLJ applications**

You might receive unexpected results in JDBC and SQLJ applications if you use date, time, and timestamp values that do not correspond to real dates and times.

The following items might cause problems:
- Use of the hour '24' to represent midnight
- Use of a date between October 5, 1582, and October 14, 1582, inclusive

**Problems with using the hour '24' as midnight**

The IBM Data Server Driver for JDBC and SQLJ uses Java data types for its internal processing of input and output parameters and ResultSet content in JDBC and SQLJ applications. The Java data type that is used by the driver is based on the best match for the corresponding SQL type when the target SQL type is known to the driver.
For values that are assigned to or retrieved from DATE, TIME, or TIMESTAMP SQL types, the IBM Data Server Driver for JDBC and SQLJ uses java.sql.Date for DATE SQL types, java.sql.Time for TIME SQL types, and java.sql.Timestamp for TIMESTAMP SQL types.

When you assign a string value to a DATE, TIME, or TIMESTAMP target, the IBM Data Server Driver for JDBC and SQLJ uses Java facilities to convert the string value to a java.sql.Date, java.sql.Time, or java.sql.Timestamp value. If a string representation of a date, time, or timestamp value does not correspond to a real date or time, Java adjusts the value to a real date or time value. In particular, Java adjusts an hour value of '24' to '00' of the next day. This adjustment can result in an exception for a timestamp value of '9999-12-31 24:00:00.0', because the adjusted year value becomes '10000'.

**Important:** To avoid unexpected results when you assign or retrieve date, time, or timestamp values in JDBC or SQLJ applications, ensure that the values are real date, time, or timestamp values. In addition, do not use '24' as the hour component of a time or timestamp value.

If a value that does not correspond to a real date or time, such as a value with an hour component of '24', is stored in a TIME or TIMESTAMP column, you can avoid adjustment during retrieval by executing the SQL CHAR function against that column in the SELECT statement that defines a ResultSet. Executing the CHAR function converts the date or time value to a character string value on the database side. However, if you use the getTime or getTimestamp method to retrieve that value from the ResultSet, the IBM Data Server Driver for JDBC and SQLJ converts the value to a java.sql.Time or java.sql.Timestamp type, and Java adjusts the value. To avoid date adjustment, execute the CHAR function against the column value, *and* retrieve the value from the ResultSet with the getString method.

The following examples show the results of updating DATE, TIME, or TIMESTAMP columns in JDBC or SQLJ applications, when the application data does not represent real dates or times.

**Table 33. Examples of updating DATE, TIME, or TIMESTAMP SQL values with Java date, time, or timestamp values that do not represent real dates or times**

<table>
<thead>
<tr>
<th>String input value</th>
<th>Target type in database</th>
<th>Value sent to table column, or exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-13-35</td>
<td>DATE</td>
<td>2009-02-04</td>
</tr>
<tr>
<td>25:00:00</td>
<td>TIME</td>
<td>01:00:00</td>
</tr>
<tr>
<td>24:00:00</td>
<td>TIME</td>
<td>00:00:00</td>
</tr>
<tr>
<td>2008-15-36</td>
<td>TIMESTAMP</td>
<td>2009-04-06 05:04:14.0</td>
</tr>
<tr>
<td>28:63:74.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9999-12-31</td>
<td>TIMESTAMP</td>
<td>Exception, because the adjusted value (10000-01-01 00:00:00.0) exceeds the maximum year of 9999.</td>
</tr>
<tr>
<td>24:00:00.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following examples demonstrate the results of retrieving data from TIMESTAMP columns in JDBC or SQLJ applications, when the values in those columns do not represent real dates or times.
### Table 34. Results of retrieving DATE, TIME, or TIMESTAMP SQL values that do not represent real dates or times into Java application variables

<table>
<thead>
<tr>
<th>SELECT statement</th>
<th>Value in TIMESTAMP column TS_COL</th>
<th>Target type in application (getXXX method for retrieval)</th>
<th>Value retrieved from table column</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT TS_COL FROM TABLE1</td>
<td>2000-01-01 24:00:00.000000</td>
<td>java.sql.Timestamp (getTimestamp)</td>
<td>2000-01-02 00:00:00.000000</td>
</tr>
<tr>
<td>SELECT TS_COL FROM TABLE1</td>
<td>2000-01-01 24:00:00.000000</td>
<td>String (getString)</td>
<td>2000-01-02 00:00:00.000000</td>
</tr>
<tr>
<td>SELECT CHAR(TS_COL) FROM TABLE1</td>
<td>2000-01-01 24:00:00.000000</td>
<td>java.sql.Timestamp (getTimestamp)</td>
<td>2000-01-02 00:00:00.000000</td>
</tr>
<tr>
<td>SELECT CHAR(TS_COL) FROM TABLE1</td>
<td>2000-01-01 24:00:00.000000</td>
<td>String (getString)</td>
<td>2000-01-01 24:00:00.000000 (no adjustment by Java)</td>
</tr>
</tbody>
</table>

### Problems with using dates in the range October 5, 1582, through October 14, 1582

The Java `java.util.Date` and `java.util.Timestamp` classes use the Julian calendar for dates before October 4, 1582, and the Gregorian calendar for dates starting with October 4, 1582. In the Gregorian calendar, October 4, 1582, is followed by October 15, 1582. If a Java program encounters a `java.util.Date` or `java.util.Timestamp` value that is between October 5, 1582, and October 14, 1582, inclusive, Java adds 10 days to that date. Therefore, a DATE or TIMESTAMP value in a DB2 table that has a value between October 5, 1582, and October 14, 1582, inclusive, is retrieved in a Java program as a `java.util.Date` or `java.util.Timestamp` value between October 15, 1582, and October 24, 1582, inclusive. A `java.util.Date` or `java.util.Timestamp` value in a Java program that is between October 5, 1582, and October 14, 1582, inclusive, is stored in a DB2 table as a DATE or TIMESTAMP value between October 15, 1582, and October 24, 1582, inclusive.

**Example:** Retrieve October 10, 1582, from a DATE column.
// DATETABLE has one date column with one row.
// Its value is 1582-10-10.
java.sql.ResultSet rs =
    statement.executeQuery(select * from DATETABLE);
rs.next();
System.out.println(rs.getDate(1)); // Value is retrieved as 1582-10-20

**Example:** Store October 10, 1582, in a DATE column.
java.sql.Date d = java.sql.Date.valueOf("1582-10-10");
java.sql.PreparedStatement ps =
    c.prepareStatement("Insert into DATETABLE values(?)");
ps.setDate(1, d);
ps.executeUpdate(); // Value is inserted as 1582-10-20

To retrieve a value in the range October 5, 1582, to October 14, 1582, from a DB2 table without date adjustment, execute the SQL CHAR function against the DATE or TIMESTAMP column in the SELECT statement that defines a ResultSet. Executing the CHAR function converts the date or time value to a character string value on the database side.

To store a value in the range October 5, 1582, to October 14, 1582 in a DB2 table without date adjustment, you can use one of the following techniques:
• For a JDBC or an SQLJ application, use the setString method to assign the value to a String input parameter. Cast the input parameter as VARCHAR, and execute the DATE or TIMESTAMP function against the result of the cast. Then store the result of the DATE or TIMESTAMP function in the DATE or TIMESTAMP column.

• For a JDBC application, set the Connection or DataSource property sendDataAsIs to true, and use the setString method to assign the date or timestamp value to the input parameter. Then execute an SQL statement to assign the String value to the DATE or TIMESTAMP column.

Example: Retrieve October 10, 1582, from a DATE column without date adjustment.

```java
// DATETABLE has one date column called DATECOL with one row.
// Its value is 1582-10-10.
java.sql.ResultSet rs = 
    statement.executeQuery("SELECT CHAR(DATECOL) FROM DATETABLE");
rs.next();
System.out.println(rs.getString(1)); // Value is retrieved as 1582-10-10
```

Example: Store October 10, 1582, in a DATE column without date adjustment.

```java
String s = "1582-10-10";
java.sql.Statement stmt = c.createStatement;
java.sql.PreparedStatement ps =
    c.prepareStatement("Insert INTO DATETABLE VALUES " + 
    "(DATE(CAST (? AS VARCHAR)))");
ps.setString(1, s);
ps.executeUpdate(); // Value is inserted as 1582-10-10
```

To avoid date adjustment, set the SQLJ option sqljAvoidTimeStampConversion to true. The SQLJ option sqljAvoidTimeStampConversion=true returns the timestamp data in CHAR format.

Example: Retrieve October 12, 1582, from a DATE column without date adjustment.

```java
import java.sql.*;
import javax.sql.DataSource;
import sqlj.runtime.ref.DefaultContext;
import com.ibm.db2.jcc.DBTimestamp;
import com.ibm.db2.jcc.DB2BaseDataSource;

#sql context DBContext;
public class UpdateTest
{
    // Set the timestamp conversion off.
    static boolean SqljAvoidTimeStampConversion = true;
    static DefaultContext ctx = null;
    java.sql.Connection con;
    public static void main(String[] args) throws Exception 
    {
        java.sql.DataSource ds = new com.ibm.db2.jcc.DB2SimpleDataSource();
          ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setServerName
            ("myserver.svl.ibm.com");
          ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setPortNumber(446);
          ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setDatabaseName("MYDB");
          ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setDriverType(4);
          ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setTimestampOutputType
            (DB2BaseDataSource.JCC_DBTIMESTAMP);
          ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setSqljAvoidTimeStampConversion
            (true);
        java.sql.Connection con = ds.getConnection("myid", "mypwd");
        // Create a connection context instance to pass to the
        // insertAndSelectTimestamp method.
```
Data loss for timestamp data in JDBC and SQLJ applications

For DB2 for z/OS Version 10 or later, or Db2 on Linux, UNIX, and Windows systems Version 9.7 or later, you can specify the precision of the fractional part of a TIMESTAMP column, with a maximum precision of 12 digits. The fractional part of a Java timestamp value can have up to 9 digits of precision. Depending on the column definition, data loss can occur when you update a TIMESTAMP(p) column or retrieve data from a TIMESTAMP(p) column.

Data loss for input data

If you use a setTimestamp call to pass a timestamp value to a TIMESTAMP(p) column, the maximum precision of the Java value that is sent to the data source is 9. If you use a setTimestamp call to pass a timestamp value to a TIMESTAMP column at a data source that does not support TIMESTAMP(p), the maximum precision of the Java value that is sent to the data source is 6. For input to a TIMESTAMP(p) column, if the precision of the target column is less than the precision of the input value, the data source truncates the excess digits in the fractional part of the timestamp.

If you use a setString call to pass the input value, it is possible to send a value with a precision of greater than 9 to the data source.

For IBM Data Server Driver for JDBC and SQLJ version 3.59 or later, no data loss occurs if the TIMESTAMP(p) column is big enough to accommodate the input value. For IBM Data Server Driver for JDBC and SQLJ version 3.58 or earlier, data loss depends on the setting of the deferPrepares property and the sendDataAsIs property:

- If sendDataAsIs is set to true, the IBM Data Server Driver for JDBC and SQLJ sends the string to the data source as-is, so the fractional part of the timestamp value can be more than 9 digits. If the value of p in the TIMESTAMP(p) column is greater than or equal to the number of digits in the fractional part of the input data, no data loss occurs.
- If sendDataAsIs is set to false, data loss depends on the deferPrepares setting.
If deferPrepares is set to true, the first time that an UPDATE statement is executed, the IBM Data Server Driver for JDBC and SQLJ sends the string to the data source as-is, so the fractional part of the timestamp value can be more than 9 digits. If the value of $p$ in the `TIMESTAMP(p)` column is greater than or equal to the number of digits in the fractional part of the input data, no data loss occurs.

For subsequent executions of the UPDATE statement, the IBM Data Server Driver for JDBC and SQLJ can determine that the target data type is a TIMESTAMP data type. If the data source supports `TIMESTAMP(p)` columns, the driver converts the input value to a `java.sql.Timestamp` value with a maximum precision of 9. If the data source does not support `TIMESTAMP(p)` columns, the driver converts the input value to a `java.sql.Timestamp` value with a maximum precision of 6. Data loss occurs if the original value has more precision than the converted `java.sql.Timestamp` value, or if the `java.sql.Timestamp` value has more precision than the `TIMESTAMP(p)` column.

If deferPrepares is set to false, the IBM Data Server Driver for JDBC and SQLJ can determine that the target data type is a TIMESTAMP data type. If the data source supports `TIMESTAMP(p)` columns, the driver converts the input value to a `java.sql.Timestamp` value with a maximum precision of 9. If the data source does not support `TIMESTAMP(p)` columns, the driver converts the input value to a `java.sql.Timestamp` value with a maximum precision of 6. Data loss occurs if the original value has more precision than the converted `java.sql.Timestamp` value, or if the `java.sql.Timestamp` value has more precision than the `TIMESTAMP(p)` column.

You can lessen data loss for input timestamp values by using a `setString` call and setting `sendDataAsIs` to true. However, if you set `sendDataAsIs` to true, you need to ensure that application data types are compatible with data source data types.

**Data loss for output data**

When you use a `getTimestamp` or `getString` call to retrieve data from a `TIMESTAMP(p)` column, the IBM Data Server Driver for JDBC and SQLJ converts the value to a `java.sql.Timestamp` value with a maximum precision of 9. If the source value has a precision of greater than 9, the driver truncates the fractional part of the retrieved value to nine digits. If you do not want truncation to occur, in the `SELECT` statement that retrieves the `TIMESTAMP(p)` value, you can cast the `TIMESTAMP(p)` value to a character data type, such as VARCHAR, and use `getString` to retrieve the value from the `ResultSet`.

**Retrieval of special values from DECFLOAT columns in Java applications**

Special handling is necessary if you retrieve values from DECFLOAT columns, 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, 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 or java.sql.ResultSet.getString method.

Suppose that the following SQL statements were used to create and populate a table.

```sql
CREATE TABLE TEST.DECFLOAT_TEST
(
    INT_VAL INT,
    DECFLOAT_VAL DECFLOAT
);
```

```sql
INSERT INTO TEST.DECFLOAT_TEST (INT_VAL, DECFLOAT_VAL) VALUES (1, 123.456),
```

```sql
INSERT INTO TEST.DECFLOAT_TEST (INT_VAL, DECFLOAT_VAL) VALUES (2, INFINITY),
```

```sql
INSERT INTO TEST.DECFLOAT_TEST (INT_VAL, DECFLOAT_VAL) VALUES (3, -123.456),
```

```sql
INSERT INTO TEST.DECFLOAT_TEST (INT_VAL, DECFLOAT_VAL) VALUES (4, -INFINITY),
```

```sql
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.

```java
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("Row \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("Caught 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.getBigDecimal(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();
        }
    }
}
```

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.getString method.

```java
final static int DECFLOAT_SPECIALVALUE_ENCOUNTERED = -4231;
java.sql.Connection con =
    java.sql.DriverManager.getConnection("jdbc:db2://localhost:50000/sample",
    "userid", "password");
```
```java
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("Caught SQLException" + e.getMessage());
        if (e.getErrorCode() == DECFLOAT_SPECIALVALUE_ENCOUNTERED) {
            // getBigDecimal failed because the retrieved value is NaN,  
            // INFINITY, or -INFINITY, so retry with getString.  
            System.out.println("DECFLOAT_VAL = " + rs.getString(2));
        } else {
            e.printStackTrace();
        }
    }
}
```

**Use of PreparedStatement.setTimestamp to set values in TIMESTAMP WITH TIME ZONE columns**

When you use PreparedStatement.setTimestamp to set a value in a TIMESTAMP WITH TIME ZONE column, you should specify a com.ibm.db2.jcc.DBTimestamp object for the input value.

Use of a com.ibm.db2.jcc.DBTimestamp object ensures that the correct time zone is assigned to the target column.

In certain cases, the target data type for a table update is not known. Possible reasons are:

- The deferPrepares and sendDataAsIs properties are set so that the target data type is not known.
- The input parameter is for a CALL statement, and the stored procedure is on a DB2 for z/OS data server.

If the target data type is not known, the IBM Data Server Driver for JDBC and SQLJ must choose a target data type. When an input parameter has type com.ibm.db2.jcc.DBTimestamp, and the target data server supports TIMESTAMP WITH TIME ZONE, the driver always chooses TIMESTAMP with TIMEZONE as the target data type.

Before version 3.63 or 4.13 of the IBM Data Server Driver for JDBC and SQLJ, if the target data type is not known, the target data server supports TIMESTAMP WITH TIME ZONE, and the input data type is java.sql.Timestamp, the driver chooses the TIMESTAMP WITH TIME ZONE as the target type.

Starting with version 3.63 or 4.13, if the target data type is not known, the target data server supports TIMESTAMP WITH TIME ZONE, and the input data type is java.sql.Timestamp, the driver chooses TIMESTAMP WITH TIME ZONE as the target type, except when the input object has a value of 0001-01-01-00:00:00.000000 or 9999-12-31-23:59:59.999999. In those cases, the driver chooses the TIMESTAMP type, without a time zone. Use of the TIMESTAMP data type in those two cases prevents an overflow condition from occurring because of adjustment of the value for the implied time zone. The implied time zone is the time zone of the Java virtual machine (JVM). Starting with version 3.65 or 4.15, the timestamps for which
the driver chooses the TIMESTAMP type, without the time zone, are 0001-01-01, with any time, or 9999-12-31, with any time.

**Migration consideration**

TIMESTAMP WITH TIME ZONE is first supported in DB2 for z/OS Version 10 new-function mode. Before Version 10 new-function mode, if the value that is assigned to a column (the second parameter of PreparedStatement.setTimestamp) has the java.sql.Timestamp data type, and the column data type is not known, the IBM Data Server Driver for JDBC and SQLJ chooses TIMESTAMP as the column data type. However, starting with DB2 for z/OS Version 10 new-function mode, unless the value that is assigned to the column is 0001-01-01-00:00:00.000000 or 9999-12-31-23:59:59.999999, the driver chooses TIMESTAMP WITH TIME ZONE as the column data type. If the driver chooses the TIMESTAMP data type, and the column type is actually TIMESTAMP WITH TIME ZONE, the database manager sets the time zone in the target column using the value of the IMPLICIT_TIMEZONE DECP value. This value might differ from the value that is inserted prior to Version 10 new-function mode.

To produce the same results before and after new-function mode when PreparedStatement.setTimestamp is executed, specify a com.ibm.db2.jcc.DBTimestamp value as the second parameter.

**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.

IBM Data Server Driver for JDBC and SQLJ property names are case-sensitive.

**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.DB2XADDataSource

- 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.

If you specify a property name that does not exist, the IBM Data Server Driver for JDBC and SQLJ does nothing with the property setting, and does not issue an error.

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:

```java
String url = 
  "jdbc:db2://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);
```

Related concepts:

“LOBs in JDBC applications with the IBM Data Server Driver for JDBC and SQLJ” on page 59

Chapter 10, “Security under the IBM Data Server Driver for JDBC and SQLJ,” on page 547

“IBM Data Server Driver for JDBC and SQLJ support for SSL” on page 560

Methods for keeping prepared statements after commit points (DB2 Performance)

Related tasks:

“Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ” on page 13

“Connecting to a data source using the DataSource interface” on page 20

Related reference:

“IBM Data Server Driver for JDBC and SQLJ extensions to JDBC” on page 383

SIGNON function for RRSAF (DB2 Application programming and SQL)

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:

**accessToken**

Used to get connection used for authentication by dash DB when security mechanism is PLUGIN_SECURITY. The data type of this property is String.

The IBM Data Server Driver for JDBC and SQLJ uses this option only if securityMechanism is set to PLUGIN_SECURITY(15). If no value is set for the parameter pluginName, connection is attempted by using default value of gss_dashdb_iam to pluginName.

If the property is set along with user or apiKey, a connection error is thrown indicating only one of them must be used.

Maximum length of the accessToken is set to 8192.

This value applies only to SSL connections to dash DB. Non-SSL connection is not supported. This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

**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`. `ResultSet.next()` returns `false`, 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 value 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.

`apiKey`  
Used to get connection used for authentication by dash DB when security mechanism is `PLUGIN_SECURITY`. The data type of this property is String.
The IBM Data Server Driver for JDBC and SQLJ uses this option only if securityMechanism is set to PLUGIN_SECURITY(15). If no value is set for the parameter pluginName, connection is attempted by using default value of gss_dashdb_iam to pluginName.

If the property is set along with user or accessToken, a connection error is thrown indicating only one of them must be used.

Maximum length of the apiKey is set to 8192.

This value applies only to SSL connections to dash DB. Non-SSL connection is not supported. This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

**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 on Linux, UNIX, and Windows systems 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.executeUpdate.

**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.executeUpdate. This value is the default.
**autocommit**
Sets the default autocommit value for a Connection or DataSource instance. All connections that are derived from a DataSource inherit the autocommit property set on the DataSource.

**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.

**clientBidiStringType**
Used if enableBidiLayoutTransformation is enabled. Specifies the string type that is used by the client application. Possible values for the property and their corresponding text attributes are specified in the following table:

<table>
<thead>
<tr>
<th>String type value integer constant</th>
<th>String type value</th>
<th>Type of text</th>
<th>Orientation</th>
<th>Symmetric swapping</th>
<th>Numeral shape</th>
<th>Text shapes</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>DB2Base DataSource. BIDI_ST4</td>
<td>Visual</td>
<td>Left-to-right</td>
<td>No</td>
<td>Nominal</td>
<td>Shaped</td>
</tr>
<tr>
<td>5</td>
<td>DB2Base DataSource. BIDI_ST5</td>
<td>Implicit</td>
<td>Left-to-right</td>
<td>Yes</td>
<td>Nominal</td>
<td>Nominal</td>
</tr>
<tr>
<td>6</td>
<td>DB2Base DataSource. BIDI_ST6</td>
<td>Implicit</td>
<td>Right-to-left</td>
<td>Yes</td>
<td>Nominal</td>
<td>Nominal</td>
</tr>
<tr>
<td>7</td>
<td>DB2Base DataSource. BIDI_ST7</td>
<td>Visual</td>
<td>Right-to-left</td>
<td>No</td>
<td>Nominal</td>
<td>Shaped</td>
</tr>
<tr>
<td>8</td>
<td>DB2Base DataSource. BIDI_ST8</td>
<td>Visual</td>
<td>Right-to-left</td>
<td>No</td>
<td>Nominal</td>
<td>Shaped</td>
</tr>
<tr>
<td>9</td>
<td>DB2Base DataSource. BIDI_ST9</td>
<td>Implicit</td>
<td>Right-to-left</td>
<td>Yes</td>
<td>Nominal</td>
<td>Shaped</td>
</tr>
<tr>
<td>10</td>
<td>DB2Base DataSource. BIDI_ST10</td>
<td>Implicit</td>
<td>Contextual left-to-right</td>
<td>Yes</td>
<td>Nominal</td>
<td>Nominal</td>
</tr>
<tr>
<td>11</td>
<td>DB2Base DataSource. BIDI_ST11</td>
<td>Implicit</td>
<td>Contextual right-to-left</td>
<td>Yes</td>
<td>Nominal</td>
<td>Nominal</td>
</tr>
</tbody>
</table>

**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 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems 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

**clientRerouteServerListJNDIClassContext**

Specifies the JNDI context that is used for binding and lookup of the `DB2ClientRerouteServerList` instance. `clientRerouteServerListJNDIClassContext` 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 by using system properties or the jndi.properties file.

clientRerouteServerListJNDIContext can be set only by using the following method:

```java
public void setClientRerouteServerListJNDIContext(javax.naming.Context registry)
```

**commandTimeout**

Specifies the maximum time in seconds that an application that runs under the IBM Data Server Driver for JDBC and SQLJ waits for SQL operations to complete before the driver throws an SQLException. The wait time includes time to obtain a transport, perform failover if needed, send the request, and wait for a response. The data type of this parameter is int. The default is 0, which means that there is no timeout.

If the `java.sql.Statement.setQueryTimeout` method is invoked, the query timeout value that is set through `Statement.setQueryTimeout` overrides the `commandTimeout` value.

`commandTimeout` applies to the execution of `Statement`, `PreparedStatement`, and `CallableStatement` methods `execute`, `executeQuery`, and `executeUpdate`. `commandTimeout` also applies to the `executeBatch` method if property `queryTimeoutInterruptProcessingMode` has the value `INTERRUPT_PROCESSING_MODE_CLOSE_SOCKET` (2).

The SQL error code that is returned with the `SQLException` depends on the data server and the value of property `queryTimeoutInterruptProcessingMode`:

- For connections to DB2 for z/OS data servers, -30108 is returned.
  - Automatic client reroute processing is not initiated if the `commandTimeout` value is exceeded.

- For connections to other data servers:
  - If the `queryTimeoutInterruptProcessingMode` value is `INTERRUPT_PROCESSING_MODE_STATEMENT_CANCEL` (1), -952 is returned.
  - If the `queryTimeoutInterruptProcessingMode` value is `INTERRUPT_PROCESSING_MODE_CLOSE_SOCKET` (2), -30108 is returned.

If configuration property `db2.jcc.enableInetAddrGetHostName` is set to true, the following situations might occur:

- Actual wait times might exceed the `commandTimeout` value. This situation can occur when the driver needs to do several DNS lookup operations to resolve IP addresses to host names. The amount by which the wait time exceeds the `commandTimeout` value depends on the number of DNS lookup operations, and the amount of time that each DNS lookup operation takes.

- The extra time that is required for DNS lookup operations might cause more timeout conditions than if `db2.jcc.enableInetAddrGetHostName` is set to false.

**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.

**DB2BaseDataSource.CONNECTION_CLOSE_WITH_COMMIT (3)**

When a connection is closed in the middle of a transaction, the transaction is committed, and no SQLException is thrown. Use this option only when migrating applications to DB2.

**connectionTimeout**

Specifies the maximum time in seconds that the IBM Data Server Driver for JDBC and SQLJ waits for a reply from a group of data servers when the driver attempts to establish a connection. If the driver does not receive a reply after the amount of time that is specified by connectionTimeout, the driver throws an SQLException with SQL error code -4499. The data type of this parameter is int. The default value is 0.

If connectionTimeout is set to a positive value, that value overrides any other timeout values that are set on a connection, such as loginTimeout. A connection is attempted to the member of the group of data servers with the greatest load capacity. If none of the members are up, a connection is attempted to the group IP address that is specified on the DataSource. If the connection cannot be established with any of the data servers within the amount of time that is specified by connectionTimeout, an SQLException is thrown.

If connectionTimeout is set to 0, and automatic client reroute is not enabled, there is no time limit.

If connectionTimeout is set to 0, and automatic client reroute is enabled against a DB2 for z/OS data sharing group, Db2 pureScale® instance, or IBM Informix high availability cluster, automatic client reroute properties such as maxRetriesForClientReroute and retryIntervalForClientReroute control the amount of time that is needed to establish the connection.

If configuration property db2.jcc.enableInetAddressGetHostName is set to true, the following situations might occur:

- Actual wait times might exceed the connectionTimeout value. This situation can occur when the driver needs to do several DNS lookup operations to resolve IP addresses to host names. The amount by which the wait time exceeds the connectionTimeout value depends on the number of DNS lookup operations, and the amount of time that each DNS lookup operation takes.

- The extra time that is required for DNS lookup operations might cause more timeout conditions than if db2.jcc.enableInetAddressGetHostName is set to false.

**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 on Linux, UNIX, and Windows systems 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:

• The databaseName value is the location name for the data source. The
  location name is defined in the SYSIBM.LOCATIONS catalog table.

If the databaseName property is not set, the connection location depends on
the type of environment in which the connection is made. If the connection
is made in an environment such as a stored procedure, CICS, or IMS
environment, where a DB2 connection to a location is previously established,
that connection is used. The connection URL for this case is
jdbc:default:connection. If a connection to DB2 is not previously established,
the connection is to the local site. The connection URL for this case is
jdbc:db2os390: or jdbc:db2os390sqlj.

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 value 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 value 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>
<thead>
<tr>
<th>Type of data source</th>
<th>Default isolation level</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI-compliant database with logging</td>
<td>java.sql.Connection.TRANSACTION_SERIALIZABLE</td>
</tr>
<tr>
<td>Database without logging</td>
<td>java.sql.Connection.TRANSACTION_READ_UNCOMMITTED</td>
</tr>
<tr>
<td>Non-ANSI-compliant database with logging</td>
<td>java.sql.Connection.TRANSACTION_READ_COMMITTED</td>
</tr>
</tbody>
</table>

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.executeUpdate 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 on Linux, UNIX, and Windows systems, and for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Possible values are:

ture Statement preparation on the data source does not occur until the PreparedStatement.executeUpdate method is executed. This value 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 on Linux,
  UNIX, and Windows systems 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. Therefore, 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 value 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 or false.
• Whether the sendDataAsIs property value is true or 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 value is the default.

**enableBidiLayoutTransformation**
Specifies whether the driver needs to perform BiDi (bidirectional) Layout
transformation on SQL statements, input parameters, or data received from the
database. The data type of this property is boolean and the default value is
false.

**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. Therefore, the driver does not throw an SQLException with SQL error code -4498, after a failed connection is reestablished, 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 is established, the driver reissues 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 SQL error code -4498, which indicates that a connection failed but was successfully reestablished. An SQLException with SQL error code -4498 informs the application that it must retry the transaction during which the connection failure occurred. If the driver cannot reestablish a connection, it throws an SQLException with SQL error code -4499.

**DB2BaseDataSource.NOT_SET** (0)

The IBM Data Server Driver for JDBC and SQLJ does not use seamless failover. This value 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.

Unless you have a Db2 on Linux, UNIX, and Windows systems pureScale database, set enableSysplexWLB to false.

enableSysplexWLB 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 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
fetchAllSize 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 value 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.

**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 as needed. 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 by using scrollable cursors. LOB stored procedure parameters are always fully materialized. LOBs that are fetched by using scrollable cursors use LOB locators if progressive streaming is not in effect.

**implicitRollbackOption**
Specifies the actions that the IBM Data Server Driver for JDBC and SQLJ takes when a transaction encounters a deadlock or a timeout. Possible values are:

**DB2BaseDataSource.IMPLICIT_ROLLBACK_OPTION_NOT_CLOSE_CONNECTION (1)**
The IBM Data Server Driver for JDBC and SQLJ throws an SQLException with an SQL error code that indicates that a deadlock or timeout occurred. The SQL error code is the SQL error code that is generated by the data server after a deadlock or timeout. The driver does not close the connection.

**DB2BaseDataSource.IMPLICIT_ROLLBACK_OPTION_CLOSE_CONNECTION (2)**
The IBM Data Server Driver for JDBC and SQLJ throws a DisconnectException with SQL error code -4499 when a deadlock or timeout occurs. The driver closes the connection. If automatic client reroute or Sysplex workload balancing is enabled, the driver disables automatic failover behavior.

**DB2BaseDataSource.IMPLICIT_ROLLBACK_OPTION_NOT_SET (0)**
This value is the default. The IBM Data Server Driver for JDBC and SQLJ throws an SQLException with an SQL error code that indicates that a deadlock or timeout occurred. The SQL error code is the SQL error code that is generated by the data server after a deadlock or timeout. The driver does not close the connection.
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.

DB2BaseDataSource.INTERRUPT_PROCESSING_MODE_CLOSE_SOCKET (2)
When an application executes `Statement.cancel`, the IBM Data Server Driver for JDBC and SQLJ drops the underlying socket. The connection is not closed and can be reused to resubmit the statement. When the connection is reused, the driver obtains a new socket.

For connections to DB2 for z/OS data servers, the IBM Data Server Driver for JDBC and SQLJ always uses this value, regardless of the value that is specified.

If `interruptProcessingMode` is set to `DB2BaseDataSource.INTERRUPT_PROCESSING_MODE_STATEMENTCancelar` or `DB2BaseDataSource.INTERRUPT_PROCESSING_MODE_CLOSE_SOCKET`, and `Statement.cancel` is executed while an application is executing a method on a `ResultSet` object, the operation on the `ResultSet` object might not be canceled. When the `Statement.cancel` statement is executed, if the underlying processing for the `ResultSet` method is in the data server, execution of the operation is canceled. If the underlying processing for the `ResultSet` method is in the driver, execution of the operation is not canceled.

keepAliveTimeOut
The maximum time in seconds before each TCP KeepAlive signal is sent to the data server. The data type of this property is int. The default is 15 seconds.

IBM Data Server Driver for JDBC and SQLJ type 4 connectivity uses the TCP/IP protocol to communicate with data servers. To prevent potential failover issues that are caused by timeouts within the TCP/IP layer, it is necessary to adjust the TCP/IP KeepAlive parameters on the client. Decreasing the KeepAlive values on the client improves timely detection of server failures.

A value of 0 means that the timeout value is the default system timeout value.

keepAliveTimeOut is supported only for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

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.

248 Application Programming Guide and Reference for Java
If the data server environment is a DB2 for z/OS Sysplex workload balancing environment or a Db2 pureScale environment, the wait time for a connection is determined by a combination of loginTimeout, maxRetriesForClientReroute, and retryIntervalForClientReroute. loginTimeout determines only the time for a single attempt to establish a connection to a data server. There might be multiple attempts to establish a connection, based on the maxRetriesForClientReroute value. There might also be gaps between attempts to establish a connection, based on the retryIntervalForClientReroute value.

During automatic client reroute processing, the memberConnectTimeout property takes precedence over the loginTimeout property.

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.PrintWriter. The default value is null, which means that no logging or tracing for the DataSource is output.

maxRetriesForClientReroute
During automatic client reroute, limits the number of retries if the primary connection to the data server fails.

The data type of this property is int.

The meaning of a retry and the default depends on the data server:

- **For connections to Db2 on Linux, UNIX, and Windows systems or IBM Informix data servers:**
  - **Meaning of a retry:** If 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), an attempt to connect to each server that is specified by the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber values counts as one retry. Each server connection is retried the number of times that is specified by maxRetriesForClientReroute.
  - For a connection to a Db2 pureScale instance:
    - For version 3.67 or 4.17, or later, a retry is an attempt to connect to all members of the Db2 pureScale instance.
    - For versions of the IBM Data Server Driver for JDBC and SQLJ before 3.67 or 4.17, a retry is an attempt to connect to a single member.
  - **Default:** If enableClientAffinitiesList is set to DB2BaseDataSource.NO (2), and maxRetriesForClientReroute and retryIntervalForClientReroute are not set, the connection is retried for 10 minutes. A wait time exists between retries that increases as the length of time from the first retry increases.
  - If enableClientAffinitiesList is DB2BaseDataSource.YES (1), the default is 3.

- **For connections to DB2 for z/OS data servers:**
  - **Meaning of a retry:**
  - For version 3.66 or 4.16, or later, one retry means one attempt to connect to all members of the data sharing group other than the failed member, and to the group IP address.
  - For versions of the IBM Data Server Driver for JDBC and SQLJ before 3.66 or 4.16, one retry means an attempt to connect to one member of the data sharing group.
- **Default:**
  - For version 3.66 or 4.16, or later, of the IBM Data Server Driver for JDBC and SQLJ, the default is 1.
  - For versions 3.64, 4.14, 3.65, or 4.15, the default is 5.
  - For versions of the IBM Data Server Driver for JDBC and SQLJ before 3.64 and 4.14, the connection is retried for 10 minutes. A wait time exists between retries that increases as the length of time from the first retry increases.

If the value of maxRetriesForClientReroute is 0, client reroute processing does not occur.

**maxStatements**

Controls an internal statement cache that is associated with a Connection. The data type of this property is int. Possible values are:

**positive integer**

- Enables the internal statement cache for a Connection, 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 Connection. 0 is the default.

`com.ibm.db2.jcc.DB2SimpleDataSource.maxStatements` controls the internal statement cache that is associated with a Connection only when the Connection object is created. `com.ibm.db2.jcc.DB2SimpleDataSource.maxStatements` has no effect on caching in an existing Connection object.

`com.ibm.db2.jcc.DB2SimpleDataSource.maxStatements` applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

**memberConnectTimeout**

Specifies the amount of time in seconds before an attempt to open a socket to a member of a DB2 for z/OS data sharing group, Db2 pureScale instance, or IBM Informix high availability cluster fails. The data type of this property is int.

memberConnectTimeout applies only to socket connection attempts to different members during automatic client reroute processing. The memberConnectTimeout property takes precedence over the loginTimeout property.

For connections to DB2 for z/OS data servers, the default is 1 second. For connections to other data servers, the default is 0.

If the memberConnectTimeout value is less than or equal to 0, the driver uses the loginTimeout value to determine how long to wait before it fails a connection request.

The memberConnectTimeout value is used for every socket open operation to each member in a member list.

For a connection to a DB2 for z/OS data sharing group, after all attempts to open a socket to all members fail, the driver retries the socket open by using a group IP address. For that retry, the driver uses the loginTimeout value to determine how long to wait before it fails the connection request.

**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.
This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

**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 on Linux, UNIX, and Windows systems 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. This step is 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 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 connectivity Db2 on Linux, UNIX, and Windows systems 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 value is the default. The following table describes the behavior for a connection to each type of data source.

<table>
<thead>
<tr>
<th>Data source</th>
<th>Version</th>
<th>Data sharing environment</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2 for z/OS</td>
<td>Version 10</td>
<td>Data sharing or non-data sharing</td>
<td>QUERY_CLOSE_IMPLICIT_COMMIT</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Version 9 with APAR PK68746</td>
<td>Non-data sharing, or in a data sharing group but not in coexistence mode with Version 8 members</td>
<td>QUERY_CLOSE_IMPLICIT_COMMIT</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Version 9 without APAR PK68746</td>
<td>Non-data sharing, or in a data sharing group but not in coexistence mode with Version 8 members</td>
<td>QUERY_CLOSE_IMPLICIT_YES</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Version 9 with APAR PK68746</td>
<td>In a data sharing group in coexistence mode with Version 8 members</td>
<td>QUERY_CLOSE_IMPLICIT_COMMIT</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Version 9 without APAR PK68746</td>
<td>In a data sharing group in coexistence mode with Version 8 members</td>
<td>QUERY_CLOSE_IMPLICIT_YES</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Version 8 with or without APAR PK68746</td>
<td>In a data sharing group in coexistence mode with Version 8 members</td>
<td>QUERY_CLOSE_IMPLICIT_YES</td>
</tr>
<tr>
<td>Db2 on Linux, UNIX, and Windows systems</td>
<td>Version 9.7</td>
<td>QUERY_CLOSE_IMPLICIT_NOT_SET</td>
<td></td>
</tr>
</tbody>
</table>

**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 (32 KB), 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>
<thead>
<tr>
<th>Data source</th>
<th>Product Version</th>
<th>Default</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Valid values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Db2 on Linux, UNIX, and Windows systems</td>
<td>All</td>
<td>32767</td>
<td>4096</td>
<td>262143</td>
<td>4096-32767, 98303, 131071, 163839, 196607, 229375, 262143</td>
</tr>
<tr>
<td>IBM Informix</td>
<td>All</td>
<td>32767</td>
<td>4096</td>
<td>10485760</td>
<td>4096-10485760</td>
</tr>
<tr>
<td>Db2 for IBM i</td>
<td>V5R4</td>
<td>32767</td>
<td>4096</td>
<td>65535</td>
<td>4096-65535</td>
</tr>
<tr>
<td>Db2 for IBM i</td>
<td>V6R1</td>
<td>32767</td>
<td>4096</td>
<td>262143</td>
<td>4096-65535, 98303, 131071, 163839, 196607, 229375, 262143</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Version 8 (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>32767</td>
<td>32767</td>
<td>32767</td>
<td>32767</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Version 9 (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>32767</td>
<td>32767</td>
<td>65535</td>
<td>32767, 65535</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Version 10 (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>32767</td>
<td>32767</td>
<td>262143</td>
<td>32767, 65535, 98303, 131071, 163839, 196607, 229375, 262143</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Version 10 (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity)</td>
<td>32767</td>
<td>32767</td>
<td>1048575</td>
<td>32767, 65535, 98303, 131071, 163839, 196607, 229375, 262143</td>
</tr>
</tbody>
</table>

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.

queryTimeoutInterruptProcessingMode

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. The IBM Data Server Driver for JDBC and SQLJ also throws an exception with SQL error -952, if the data server supports the interruption of SQL statements. If the data server does not support the interruption of SQL statements, the driver throws an exception that indicates that the feature is not supported.

For connections to data servers other than DB2 for z/OS, INTERRUPT_PROCESSING_MODE_STATEMENT_CANCEL is the default.

For connections to DB2 for z/OS data servers, INTERRUPT_PROCESSING_MODE_STATEMENT_CANCEL is not a possible value. If it is specified, the driver uses INTERRUPT_PROCESSING_MODE_CLOSE_SOCKET instead.

**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.

For connections to data servers other than DB2 for z/OS, when the Statement object times out:

- If automatic client reroute is not enabled and enableSysplexWLB is set to false, an exception with SQL 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 is enabled, and enableSysplexWLB is set to false, the IBM Data Server Driver for JDBC and SQLJ tries to reestablish a connection. If a new connection is successfully reestablished, the driver returns an SQL error code of -4498. However, the driver does not execute the timed-out SQL statements again, even if enableSeamlessFailover is set to DB2BaseDataSource.YES (1).
- If enableSysplexWLB is set to true, the IBM Data Server Driver for JDBC and SQLJ tries to reestablish a connection. If a new connection is successfully reestablished, the driver returns an SQL error code of -30108. However, the driver does not execute the timed-out SQL statements again, even if enableSeamlessFailover is set to DB2BaseDataSource.YES (1).

For connections to DB2 for z/OS, when the Statement object times out:

- If enableSysplexWLB is set to false, an exception with SQL 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 enableSysplexWLB is set to true, the IBM Data Server Driver for JDBC and SQLJ tries to reestablish a connection. If a new connection is successfully reestablished, the driver returns an SQL error code of...
-30108. 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 value is the default. 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 on Linux, UNIX, and Windows systems, or Db2 for IBM 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 only the DB2Sqlca.getMessage method for IBM Data Server Driver for JDBC and SQLJ 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 meaning of a retry and the default depends on the data server:
• For connections to Db2 on Linux, UNIX, and Windows systems or IBM Informix data servers:
  – **Meaning of a retry:** If `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), an attempt to connect to each server that is specified by the `clientRerouteAlternateServerName` and `clientRerouteAlternatePortNumber` values counts as one retry. Each server connection is retried the number of times that is specified by `maxRetriesForClientReroute`.
  – **Default:** If `enableClientAffinitiesList` is set to `DB2BaseDataSource.NO` (2), and `maxRetriesForClientReroute` and `retryIntervalForClientReroute` are not set, the connection is retried for 10 minutes. A wait time exists between retries that increases as the length of time from the first retry increases. If `enableClientAffinitiesList` is `DB2BaseDataSource.YES` (1), the default is 0.

• For connections to DB2 for z/OS data servers:
  – **Meaning of a retry:**
    - For version 3.66 or 4.16, or later, one retry means one attempt to connect to all members of the data sharing group other than the failed member, and to the group IP address.
    - For versions of the IBM Data Server Driver for JDBC and SQLJ before 3.66 or 4.16, one retry means an attempt to connect to one member of the data sharing group.
  – **Default:**
    - For version 3.64 or 4.14, or later, of the IBM Data Server Driver for JDBC and SQLJ, the default is 0.
    - For versions of the IBM Data Server Driver for JDBC and SQLJ before 3.64 and 4.14, the connection is retried for 10 minutes. A wait time exists between retries that increases as the length of time from the first retry increases.

**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 on Linux,
UNIX, and Windows systems only.

ENCRYPTED_USER_ONLY_SECURITY (16)
Encrypted user ID. This value does not apply to connections to IBM
Informix.

TLS_CLIENT_CERTIFICATE_SECURITY (18)
Client certificate security, by using SSL. This value applies to
connections to DB2 for z/OS Version 10 and later only.

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 provided by the
db2.jcc.securityMechanism configuration property. If the
db2.jcc.securityMechanism configuration property is also not specified, the
default value for securityMechanism is CLEAR_TEXT_PASSWORD_SECURITY.

If the data 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 upgrades 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.

This property does not apply to IBM Data Server Driver for JDBC and SQLJ
type 2 connectivity on DB2 for z/OS.

Security mechanisms ENCRYPTED_PASSWORD_SECURITY,
ENCRYPTED_USER_AND_PASSWORD_SECURITY,
ENCRYPTED_USER_AND_DATA_SECURITY,
ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY, and
ENCRYPTED_USER_ONLY_SECURITY use DRDA encryption. DRDA
encryption is not intended to provide confidentiality and integrity of
passwords or data over a network that is not secure, such as the Internet.
DRDA encryption uses an anonymous key exchange, Diffie-Hellman, which
does not provide authentication of the server or the client. DRDA encryption is
vulnerable to man-in-the-middle attacks.

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 must 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.

serverBidiStringType
Used if enableBidiLayoutTransformation is enabled. Specifies the string type
that is used by the server. The data type of this property is int.
serverBidiStringType takes on the same possible values as clientBidiStringType.
**serverName**

The host name or the TCP/IP address of the data source. 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.

**sslCertLocation**

Specifies that an application can configure the location of a trusted certificate file. For applications that have the database server certificate, sslCertLocation is the only property that is needed to be configured to instruct IBM Data Server Driver for JDBC and SQLJ to trust the certificate for SSL connections. This property removes the need to import the certificate into a Java truststore database and related driver configurations.

The default value for sslCertLocation is provided by the db2.jcc.sslCertLocation configuration property. If the db2.jcc.sslCertLocation configuration property is not specified, sslTruststoreLocation, and sslCertLocation properties are not configured, and sslConnection is set to true, IBM Data Server Driver for JDBC and SQLJ uses the default truststore for the Java runtime environment.

The sslCertLocation property accepts certificate location values in the following forms:

- File extensions: .arm, .pem, .cert, .crt, and .der.
- Full path to the certificate file: sslCertLocation=/path/to/cert.arm
- Path relative to the current class path: sslCertLocation=classpath:relative/cert.arm
- DER-encoded certificates in binary or Base64 ASCII encoding. If the certificate is provided in Base64 encoding, the file content must be bound at the beginning by "-----BEGIN CERTIFICATE-----" and at the end by "-----END CERTIFICATE-----".

**sslCipherSuites**

Specifies the set of cipher suites to use when you establish SSL connections to the server. Configure this property only when you would not like to use the default cipher suites from the JRE (Java Runtime Environment). Specify the names of cipher suites as string values separated by commas.

**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.

The default value for sslConnection is provided by the db2.jcc.sslConnection configuration property. If the db2.jcc.sslConnection configuration property is also not specified, the default value for sslConnection is false.

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 sslTrustStoreLocation 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.

The default value for sslTrustStoreLocation is provided by the db2.jcc.sslTrustStoreLocation configuration property. If the db2.jcc.sslTrustStoreLocation configuration property is also not specified, the default value for sslTrustStoreLocation is null.

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.

The default value for sslTrustStorePassword is provided by the db2.jcc.sslTrustStorePassword configuration property. If the db2.jcc.sslTrustStorePassword configuration property is also not specified, the default value for sslTrustStorePassword is null.

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 value 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.

**timerLevelForQueryTimeOut**

Specifies the level at which the IBM Data Server Driver for JDBC and SQLJ creates a java.util.Timer object for waiting for query execution to time out. Possible values are:

- **DB2BaseDataSource.QUERYTIMEOUT_STATEMENT_LEVEL (1)**
  The IBM Data Server Driver for JDBC and SQLJ creates a Timer object
for each Statement object. When the Statement object is closed, the driver deletes the Timer object. This value is the default.

**DB2BaseDataSource.QUERYTIMEOUT_CONNECTION_LEVEL (2)**
The IBM Data Server Driver for JDBC and SQLJ creates a Timer object for each Connection object. When the Connection object is closed, the driver deletes the Timer object.

**DB2BaseDataSource.QUERYTIMEOUT_DISABLED (-1)**
The IBM Data Server Driver for JDBC and SQLJ does not create a Timer object to control query execution timeout.

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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer value</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.ISO</td>
<td>1</td>
<td>yyyy-mm-dd- hh:mm:ss.nnnnnnnn</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.JDBC</td>
<td>5</td>
<td>yyyy-mm-dd hh:mm:ss.nnnnnnnn</td>
</tr>
</tbody>
</table>

**Note:**
1. The number of digits in the fractional part of the timestamp depends on the precision of the TIMESTAMP(p) column in the source table. If p<9, p digits are returned. If p>=9, 9 digits are returned, and the remaining digits are truncated.

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 value is the default.

For example:

- 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.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity, the db2.jcc.override.traceFile configuration property value overrides the traceFile property value.

**Recommendation:** Set the db2.jcc.override.traceFile configuration property, rather than setting the traceFile property for individual connections.

**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_T22OS (X'10000')
- 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_SYSPLEX (X'80000') (for version 3.68 or 4.18, or later, of the IBM Data Server Driver for JDBC and SQLJ, and connections to DB2 for z/OS data sharing groups)
- 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:
  \[\text{TRACE\_DRDA\_FLOWS}\mid\text{TRACE\_CONNECTION\_CALLS}\]

- 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:
  \[\sim\text{TRACE\_DRDA\_FLOWS}\]

**traceFileCount**
Specifies the maximum number of trace files for circular tracing. The IBM Data Server Driver for JDBC and SQLJ uses this property only when traceOption is set to `DB2BaseDataSource.TRACE_OPTION_CIRCULAR (1)`. The data type of this property is int. The default value is 2.

**traceFileSize**
Specifies the maximum size of each trace file, for circular tracing. The IBM Data Server Driver for JDBC and SQLJ uses this property only when traceOption is set to `DB2BaseDataSource.TRACE_OPTION_CIRCULAR (1)`. The data type of this property is int. The default value is 10485760 (10 MB).

**useJDBC41DefinitionForGetColumns**
Specifies whether the `DatabaseMetaData.getColumns` method returns a result set with a column with the name `SCOPE_CATALOG` or `SCOPE_CATLOG`. Possible values are:

- `DB2BaseDataSource.NOT_SET (0)`: Specifies that for version 4.13 or later of the IBM Data Server Driver for JDBC and SQLJ, the result set from `DatabaseMetaData.getColumns` contains a column named `SCOPE_CATALOG`. For version 4.12 or earlier of the IBM Data Server Driver for JDBC and SQLJ, that column is named `SCOPE_CATLOG`.

- `DB2BaseDataSource.YES (1)`: Specifies that for version 4.13 or later of the IBM Data Server Driver for JDBC and SQLJ, the result set from `DatabaseMetaData.getColumns` contains a column named `SCOPE_CATALOG`. For version 4.12 or earlier of the IBM Data Server Driver for JDBC and SQLJ, that column is named `SCOPE_CATLOG`.

- `DB2BaseDataSource.NO (2)`: Specifies that for all versions of the IBM Data Server Driver for JDBC and SQLJ, the result set from `DatabaseMetaData.getColumns` contains a column named `SCOPE_CATALOG`.

**traceOption**
Specifies how trace data is collected. The data type of this property is int. Possible values are:

- `DB2BaseDataSource.NOT_SET (0)`: Specifies that a single trace file is generated, and that there is no limit to the size of the file. This value is the default.
  
  If the value of traceOption is NOT_SET, the traceFileSize and traceFileCount properties are ignored.

- `DB2BaseDataSource.TRACE_OPTION_CIRCULAR (1)`: Specifies that the IBM Data Server Driver for JDBC and SQLJ does circular tracing. Circular tracing is done as follows:
1. When an application writes its first trace record, the driver creates a file.
2. The driver writes trace data to the file.
3. When the size of the file is equal to the value of property traceFileSize, the driver creates another file.
4. The driver repeats steps 2 and 3 until the number of files to where data is written is equal to the value of property traceFileCount.
5. The driver writes data to the first trace file, overwriting the existing data.
6. The driver repeats steps 3 through 5 until the application completes.

The file names for the trace files are the file names that are determined by the traceFile or traceDirectory property. The file name is appended with .1 for the first file, .2 for the second file, and so on.

**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:

```java
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 the 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 the 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.

### 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 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems 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 db2jcthread-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 on Linux, UNIX, and Windows systems 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. The default is NULLID for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity. For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity, if a value for currentPackageSet is not specified, the property value is not set. 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 IBM 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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer value</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.ISO</td>
<td>1</td>
<td>yyyy-mm-dd</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.USA</td>
<td>2</td>
<td>mm/dd/yyyy</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.EUR</td>
<td>3</td>
<td>dd.mm.yyyy</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.JIS</td>
<td>4</td>
<td>yyyy-mm-dd</td>
</tr>
</tbody>
</table>

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 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems 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.

enableExtendedDescribe
Specifies whether the IBM Data Server Driver for JDBC and SQLJ requests extended describe information from the data server when it prepares a statement.

Extended describe information provides:
• Additional descriptive information for a cursor or a result set
• Information about whether a column:
  – Can be updated
  – Is a primary key or a preferred candidate key member
  – Is an expression or a table column
  – is a generated column or a table column
• The fully qualified view or table name
• The fully qualified column name

Possible values are:

**DB2BaseDataSource.NOT_SET (0)**
The IBM Data Server Driver for JDBC and SQLJ requests extended describe information. This is the default.

**DB2BaseDataSource.YES (1)**
The IBM Data Server Driver for JDBC and SQLJ requests extended describe information.

**DB2BaseDataSource.NO (2)**
The IBM Data Server Driver for JDBC and SQLJ does not request extended describe information.

Setting enableExtendedDescribe to **DB2BaseDataSource.NO** can result in a performance benefit because it avoids the extra processing that the driver must do to provide the additional information. However, if you specify this is option, some methods throw an exception or return unexpected results. The following table lists the behavior of methods when enableExtendedDescribe is set to **DB2BaseDataSource.NO**.

<table>
<thead>
<tr>
<th>Method</th>
<th>Result when extended describe is off</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection.findAutoGeneratedKeysColumn</td>
<td>Returns an array of empty strings (&quot;&quot;&quot;)</td>
</tr>
<tr>
<td>DB2ResultSetMetaData.getDBTemporalColumnType</td>
<td>Returns -1</td>
</tr>
<tr>
<td>ResultSet.getMetaData on the ResultSet object that is returned by PreparedStatement.getGeneratedKeys</td>
<td>Returns null</td>
</tr>
<tr>
<td>ResultSet.insertRow, ResultSet.deleteRow, ResultSet.updateRow</td>
<td>SQLException with error code -4474, SQLSTATE 42808 (column not updatable)</td>
</tr>
<tr>
<td>ResultSet.updateXXX methods</td>
<td>SQLException with error code -4474, SQLSTATE 42808 (column not updatable)</td>
</tr>
<tr>
<td>ResultSetMetaData.getTableName,</td>
<td>Returns an empty string (&quot;&quot;)</td>
</tr>
<tr>
<td>ResultSetMetaData.getSchemaName,</td>
<td></td>
</tr>
<tr>
<td>ResultSetMetaData.getColumnName</td>
<td></td>
</tr>
<tr>
<td>ResultSetMetaData.isAutoIncrement</td>
<td>Returns false</td>
</tr>
</tbody>
</table>

**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 server supports multiple-row FETCH. The data type of this property is int.

For connections to DB2 for z/OS, 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 server 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 on Linux, UNIX, and Windows systems, multiple-row fetch is used for scrollable cursors, if the data server 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 to DB2 for z/OS, multiple-row fetch is used if useRowsetCursor is set to true.
- For connections to Db2 on Linux, UNIX, and Windows systems, multiple row fetch is used for scrollable cursors, if the data server supports multiple-row FETCH.

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.
This value is the default, unless configuration property db2.jcc.encryptionAlgorithm provides a different default.

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.

For AES encryption, you need an unrestricted policy file for JCE. That file is available at the following location:
encryptionAlgorithm can be specified only if the securityMechanism or 
db2.jcc.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
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.STMT_CONCENTRATOR_OFF (1)**
The IBM Data Server Driver for JDBC and SQLJ does not use the data source's statement concentrator functionality.

- **DB2BaseDataSource.STMT_CONCENTRATOR_WITH_LITERALS (2)**
The IBM Data Server Driver for JDBC and SQLJ uses the data source's statement concentrator functionality.

- **DB2BaseDataSource.STMT_CONCENTRATOR_NOT_SET (0)**
Enables the data server's default behavior for statement concentrator functionality. This is the default value.

For Db2 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems 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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer value</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.ISO</td>
<td>1</td>
<td>hh:mm:ss</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.USA</td>
<td>2</td>
<td>hh:mm am or hh:mm pm</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.EUR</td>
<td>3</td>
<td>hh.mm.ss</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.JIS</td>
<td>4</td>
<td>hh:mm:ss</td>
</tr>
</tbody>
</table>

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 a PreparedStatement object. The data type of useCachedCursor is boolean.

If useCachedCursor is set to true, the cursor for PreparedStatement objects is cached, which can improve performance.

Set useCachedCursor to false if PreparedStatement objects access tables whose column types or lengths change between executions of those PreparedStatement objects.

The default for useCachedCursor is:

- false, if the data server is Db2 on Linux, UNIX, and Windows systems, and the driver is at one of the following levels:
  - Version 3.67 or 4.17, or later
  - Version 3.64 or 4.14
- true, if the data server and driver version are one of the following combinations:
  - For a DB2 for z/OS data server, any version of the driver other than the versions for which useCachedCursor is false

If the driver version is 3.67 or 4.17, or later, or 3.64 or 4.14, and the deferPrepares property is set to true, the driver behaves as if useCachedCursor is set to false, regardless of the useCachedCursor setting.

The useCachedCursor property is deprecated in version 3.69 or 4.19 of the driver. You can achieve better performance when the property is not set by the application. If you need the older driver behavior, explicitly set the property to true or false.

**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 Specified 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.

useJDBC4ColumnNameAndLabelSemantics
Specifications 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 on Linux, UNIX, and Windows systems Version 8.1 or later
• Db2 for IBM i 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 on Linux, UNIX, and Windows systems Version 8.1 or later
  − Db2 for IBM i V5R3 or later

• For connections to earlier versions of these database systems, the default behavior for useJDBC4ColumnNameAndLabelSemantics is DB2BaseDataSource.NO.

**xmlFormat**

Specifies the format that is used to 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 the default XML format is used. The default is textual XML format.

- **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.
**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 to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

**Related concepts:**

- “Examples of ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnLabel values” on page 487

**Related reference:**

- Setting properties locally for individual connections that use the IBM Data Server Driver for JDBC and SQLJ

---

**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.

  enableConnectionConcentrator applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

  The following table shows the interaction between the enableConnectionConcentrator and enableSysplexWLB property settings:

<table>
<thead>
<tr>
<th>Table 38. Result of enableConnectionConcentrator and enableSysplexWLB settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableConnectionConcentrator setting</td>
</tr>
<tr>
<td>-------------------------------------</td>
</tr>
<tr>
<td>false</td>
</tr>
</tbody>
</table>
Table 38. Result of enableConnectionConcentrator and enableSysplexWLB settings (continued)

<table>
<thead>
<tr>
<th>enableConnectionConcentrator setting</th>
<th>enableSysplexWLB setting</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>true</td>
<td>The connection of the application to the data server chooses a transport based on the weights that are returned by the data server. A transport is chosen at every transaction boundary. This action balances the load on different DB2 data sharing members.</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>Several application connections can share the same transport for executing their transactions. The switching of the transport between connections occurs at each transaction boundary. Because many connections share one transport to the data server, fewer resources can be used on the data server.</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>The connection of the application to the data server chooses a transport based on the weights that are returned by the data server. A transport is chosen at every transaction boundary. This action balances the load on different DB2 data sharing members. This is the same behavior as the behavior when enableConnectionConcentrator is false and enableSysplexWLB is true.</td>
</tr>
</tbody>
</table>

**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.

For version 3.63 or 4.13 or later of the IBM Data Server Driver for JDBC and SQLJ, the default value for `maxTransportObjects` is 1000. For earlier versions of the IBM Data Server Driver for JDBC and SQLJ, 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.

Related concepts:

"Example of enabling DB2 for z/OS Sysplex workload balancing and automatic client reroute in Java applications" on page 611

Common IBM Data Server Driver for JDBC and SQLJ properties for IBM Informix and Db2 on Linux, UNIX, and Windows systems

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply to IBM Informix and Db2 on Linux, UNIX, and Windows systems database servers.

Properties that apply to IBM Informix and Db2 on Linux, UNIX, and Windows systems are:

**currentLockTimeout**

Specifies whether Db2 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply only to Db2 on Linux, UNIX, and Windows systems servers.
Those properties are:

**alternateGroupDatabaseName**
Specifies the database names for alternate groups to which an application can connect. The data type of this property is String. For a connection to a DB2 for z/OS data server, this value is the location name for a data sharing group. For a connection to a Db2 on Linux, UNIX, and Windows systems data server, each of these values is the database name for a single-member or multiple-member database instance. If more than one database name is specified, the database names must be separated by commas.

For connections to DB2 for z/OS, only one value can be specified.

**alternateGroupPortNumber**
Specifies the port numbers for alternate groups to which an application can connect. The data type of this property is String. For a connection to a DB2 for z/OS data server, this value is the TCP/IP server port number that is assigned to the data sharing group. For a connection to a Db2 on Linux, UNIX, and Windows systems data server, each of these values is the TCP/IP server port number that is assigned to a single-member or multiple-member database instance. If more than one port number is specified, the port numbers must be separated by commas.

For connections to DB2 for z/OS, only one value can be specified.

**alternateGroupServerName**
Specifies the host names for alternate groups to which an application can connect. The data type of this property is String. The data type of this property is String. For a connection to a DB2 for z/OS data server, this value is the domain name or IP address that is assigned to the data sharing group. For a connection to a Db2 on Linux, UNIX, and Windows systems data server, each of these values is the domain name or IP address that is assigned to a a single-member or multiple-member database instance. If more than one host name is specified, the host names must be separated by commas.

For connections to DB2 for z/OS, only one value can be specified.

**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 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems.

**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 on Linux, UNIX, and Windows systems Version 1 is performed to generate an access plan.

2 Specifies a level of optimization higher than that of Db2 on Linux, UNIX, and Windows systems 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.

**enableAlternateGroupSeamlessACR**
Specifies whether failover to an alternate group is seamless or non-seamless. The data type of this property is boolean. The possible values are:

- **false** Failover is non-seamless. *false* is the default.
  - With non-seamless behavior, if an application that is currently connected to a primary group is executing a transaction, and the entire primary group goes down, the IBM Data Server Driver for JDBC and SQLJ fails over to alternate group. If failover is successful, the driver throws an *SQLException* with SQL error code -30108.

- **true** Failover is seamless.
  - With seamless behavior, if an application that is currently connected to a primary group is executing a transaction, and the entire primary group goes down, the IBM Data Server Driver for JDBC and SQLJ fails over to alternate group. If the transaction is eligible for seamless failover, the connection is retried. If the connection is successful, no *SQLException* is thrown.

For connections to DB2 for z/OS, only one value can be specified.

**enableTimeoutForCursors**
For *DatabaseMetaData* or *ResultSet* methods that use *Statement* objects in their implementations, specifies whether the commandTimeout and queryTimeoutInterruptProcessingMode property values control the timeout behavior for those *Statement* objects.

Examples of methods that use *Statement* objects in their implementations are:
- *ResultSet.updateRow*
- *ResultSet.insertRow*
• ResultSet.deleteRow
• DatabaseMetaData.getProcedures
• DatabaseMetaData.getTables
• DatabaseMetaData.getColumns

The data type of this property is int. The possible values are:

com.ibm.db2.jcc.DB2BaseDataSource.YES (1) or
com.ibm.db2.jcc.DB2BaseDataSource.NOT_SET (0)

A Statement object that is used in the implementation of a
DatabaseMetaData or ResultSet method is controlled by the
commandTimeout and queryTimeoutInterruptProcessingMode
properties. This behavior is the default behavior.

com.ibm.db2.jcc.DB2BaseDataSource.NO (2)

A Statement object that is used in the implementation of a
DatabaseMetaData or ResultSet method is not controlled by the
commandTimeout and queryTimeoutInterruptProcessingMode
properties.

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 on Linux, UNIX, and Windows systems
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. The 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 on Linux, UNIX, and
Windows systems 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 on Linux, UNIX, and Windows systems server that uses a partitioned database environment.
- 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 direcetable statement in the transaction. Db2 on Linux, UNIX, and Windows systems 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.

The following list includes the properties for the IBM Data Server Driver for JDBC and SQLJ:

**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. The accounting interval also 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 sign-on call. If the value of subsystem parameter ACCUMACC is not NO, the ACCUMACC value overrides the accountingInterval setting.

The accountingInterval property applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. The accountingInterval property is not applicable to connections under CICS or IMS, or for Java stored procedures.

The accountingInterval property overrides the db2jcc.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 char0utputSize to the largest expected length for any Types.CHAR INOUT or OUT parameter.

The char0utputSize property 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.

Before you choose the value for char0utputSize, you need to consider 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 char0utputSize 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 3 bytes. (The euro symbol is an example of a 3-byte UTF-8 character). To ensure that the value of char0utputSize is large enough, if you have no information about the output data, set char0utputSize 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.

currentLocaleLCType
Specifies the LC_CTYPE locale that is used to execute SQL statements that use a built-in function that references a locale. The data type of this property is String. If currentLocaleLCType is set, the IBM Data Server Driver for JDBC and SQLJ sets the CURRENT LOCALE LC_CTYPE special register on the data server to the property value. The currentLocaleLCType property has no default.

The currentLocaleLCType property can be set only at the start of a connection, and cannot be changed while the connection is active.

currentSQLID
The currentSQLID property specifies the following information:
- 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.

The `currentSQLID` property 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 during a connection. The `enableMultiRowInsertSupport` property must be set to false if INSERT FROM SELECT statements are executed in a batch. Otherwise, the driver produces a BatchUpdateException error.

`enableT2zosLBF`  
Specifies whether limited block fetch is used for connections that use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to retrieve data from a table on a local DB2 for z/OS data server. The data type of this property is an int. You must use one of the following possible values of the property:

- `com.ibm.db2.jcc.DB2BaseDataSource.NOT_SET(0)` or `not specified`: This value is the default setting.
  - For a connection to a DB2 for z/OS data server in Version 10 conversion mode, specifies that limited block fetch is not used for retrieving data from a local table that uses the IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.
  - For a connection to a DB2 for z/OS data server in Version 10 new-function mode or later, specifies that limited block fetch is used for retrieving data from a local table that uses the IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

- `com.ibm.db2.jcc.DB2BaseDataSource.YES(1)`: Specifies that limited block fetch is used for retrieving data from a local table that uses the IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

- `com.ibm.db2.jcc.DB2BaseDataSource.NO(2)`: Specifies that limited block fetch is not used for retrieving data from a local table that uses the IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

The `enableT2zosLBF` property applies to DB2 for z/OS data servers in Version 10 new-function mode or later. The `enableT2zosLBF` property does not apply to retrieval of stored procedure result sets.

`enableT2zosLBFSResultSets`  
Specifies whether limited block fetch is used for connections that use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to retrieve data from a stored procedure result set on a local DB2 for z/OS data server. The data type of this property is int. The following are possible values of the property:

- `com.ibm.db2.jcc.DB2BaseDataSource.YES(1)` or `com.ibm.db2.jcc.DB2BaseDataSource.NOT_SET(0)`: Specifies that limited block fetch is used for retrieving data from a
stored procedure result set that uses the IBM Data Server Driver for JDBC and SQLJ type 2 connectivity. This value is the default.

**com.ibm.db2.jcc.DB2BaseDataSource.NO**(2)

Specifies that limited block fetch is used for retrieving data from a stored procedure result set that uses the IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

The `enableT2zosLBFSResultSets` property applies to DB2 for z/OS data servers in Version 10 new-function mode or later.

**enableXACleanTransaction**

Specifies whether to reset session data at XA End. The data type of `enableXACleanTransaction` is Boolean. Possible values of the property include the following Boolean values:

- **true** Enables clean transaction mode. All database resources such as cursors and DGTIs, are implicitly closed and dropped on XA_END.
- **false** Enable dirty transaction mode. This value is the default. Session data is not reset at XA End, and persists across transactions.

**extendedTableInfo**

The `extendedTableInfo` property specifies whether information about extended table types is returned from a `DatabaseMetaData.getTables` method call. Currently, there exists one extended table type: ACCEL-ONLY TABLE.

**com.ibm.db2.jcc.DB2BaseDataSource.NO**(2)

The result set that is returned by the `DatabaseMetaData.getTables` method does not contain columns for extended table types.

Rows for extended table types are returned only if "TABLE" is explicitly specified in the `types` parameter value. In this case, extended table types are listed as TABLE in the `TABLE_TYPE` column of the result set.

**com.ibm.db2.jcc.DB2BaseDataSource.YES**(1)

The result set that is returned by the `DatabaseMetaData.getTables` method contains rows and columns for extended table types. The following list depicts the result set in more detail:

- The result set contains these extra columns after the columns that are always returned in the result set from `DatabaseMetaData.getTables`:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEMPORAL_TABLE_TYPE</td>
<td>String</td>
<td>Contains the type of temporal table. The following shows possible values of the temporal table:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>SYSTEM</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>System-period temporal table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>APPLICATION</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Application-period temporal table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>BITEMPORAL</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitemporal table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Empty string</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not a temporal table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This row is returned for connections to DB2 for z/OS Version 10 or later.</td>
</tr>
</tbody>
</table>
Table 39. Extra columns returned by DatabaseMetaData.getTables (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS_ACCELERATED</td>
<td>String</td>
<td>Indicates whether the table is an accelerated table. Possible values are YES or NO.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This row is returned for connections to DB2 for z/OS Version 10 or later.</td>
</tr>
<tr>
<td>ACCEL_ARCHIVE_STATUS</td>
<td>String</td>
<td>Contains the archive status of the table in the accelerator database. See the description of the ARCHIVE column in SYSACCEL.SYSACCELERATEDTABLES table (DB2 SQL) for the possible values and their meanings.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This row is returned for connections to DB2 for z/OS Version 10 or later.</td>
</tr>
<tr>
<td>IS_ARCHIVE_ENABLED</td>
<td>String</td>
<td>Indicates whether the table is an archive-enabled table. Possible values are YES or NO.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This row is returned for connections to DB2 for z/OS Version 11 or later.</td>
</tr>
</tbody>
</table>

- Rows for extended table types are returned under the following circumstances:
  - All table types are implicitly requested by specifying null in the types parameter value.
  - An extended table type name is explicitly specified in the types parameter value.
In this case, the extended table type is listed by its extended table type name in the TABLE_TYPE column of the result set.

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 contain bounded IBM Data Server Driver for JDBC and SQLJ packages at the server by 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.

The jdbcCollection setting does not apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

maxConnCachedParamBufferSize
Specifies the maximum size of an internal buffer that is used for caching input parameter values for PreparedStatement objects. The buffer caches values on the native code side that are passed from the driver's Java code side for the IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. The buffer is used by all PreparedStatement objects for a Connection. The default is 1048576 (1 MB). The default value must be adequate for most users. Set maxConnCachedParamBufferSize to a larger value if many applications that run under the driver instance have PreparedStatement objects with large numbers of input parameters or large input parameters. The maxConnCachedParamBufferSize value must be larger than the maximum size of all input parameter data for a Connection. However, you also need to consider
the total number of connections and the maximum amount of memory that is available when you set the `maxConnCachedParamBufferSize` value.

The buffer exists for the life of a `Connection`, unless it reaches the maximum size. If that happens, the buffer is freed on each call to the native code. The corresponding buffer on the Java code side is freed on `PreparedStatement.clearParameters` and `PreparedStatement.close` calls. The buffers are not cleared if an application calls `PreparedStatement.clearParameters`, and the buffers do not reach the maximum size.

**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.

The `maxRowsetSize` property applies only to the IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

**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. The `pkList` property applies only to the IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

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 `planName`**.

**Recommendation:** Use `pkList` instead of `planName`.

The following is the format of the package list:

```
<collection-ID>.<collection-ID>...
```

The `pkList` property overrides the value of the `db2jcc.pkList` configuration property. If `pkList`, `planName`, and `db2jcc.pkList` are not specified, then the value of `pkList` is `NULLID.*`.

**planName**

Specifies a DB2 plan name that is used for the underlying RRSAF CREATE THREAD call when a JDBC or SQLJ connection to a data source is established. `planName` applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

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 `pkList`**.

The `planName` property overrides the value of the `db2.jcc.planName` configuration property. If `pkList`, `planName`, and `db2.jcc.planName` are not specified, `NULLID.*` is used as the package list for the underlying CREATE THREAD call.

**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:
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 value is the default for DB2 for z/OS Version 9 or later.

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 of the result set. The method also returns a LONG VARCHAR or LONG VARGRAPHIC in the TYPE_NAME column of the result set.

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.

The following shows possible values of the property:

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.

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. Therefore, 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 length of the data. UTF-8 data
can require up to 3 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 `s.t:h:m`.
The `s` in the format is the sign, `h` is the time zone hour, and `m` is time zone
minutes. The range of valid values is -12:59 to +14:00.

**sqljAvoidTimeStampConversion**
Specifies whether a date that falls in the range of October 5, 1582 to October
14, 1582, inclusive, is retrieved from the data server. The retrieval will occur
without date adjustment to a value between October 15, 1582, and October 24,
1582, inclusive. The value for `sqljAvoidTimeStampConversion` must be `true` or
`false`. The default is `false`.

This property applies only to SQLJ.

**sqljEnableClassLoaderSpecificProfiles**
Specifies whether the IBM Data Server Driver for JDBC and SQLJ allows the
use 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`. The `sqljEnableClassLoaderSpecificProfiles` property 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 by using the 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.

The `ssid` property 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 ends.
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.

The `ssid` property applies only to IBM Data Server Driver for JDBC and SQLJ
type 2 connectivity to DB2 for z/OS.

**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 must set useRowSetCursor to false. Those applications do not operate properly if the IBM Data Server Driver for JDBC and SQLJ uses multiple-row FETCH.

Related reference:

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:

```java
boolean dbDate = DB2BaseDataSource.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-".

Related reference:

[DDF/RRSAF ACCUM field (ACCUMACC subsystem parameter) (DB2 Installation and Migration)](url)
**DBPATH**

Specifies a colon-separated list of values that identify the database servers that contain databases. The data 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.

OPTOFCD
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 10. 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>
<thead>
<tr>
<th>Configuration property name</th>
<th>Connection or DataSource property name</th>
<th>Introduced in driver version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.accessToken (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity and dash DB only)</td>
<td>accessToken</td>
<td>4.24</td>
<td><a href="#">1</a> <a href="#">2</a> <a href="#">3</a></td>
</tr>
<tr>
<td>db2.jcc.accountingInterval</td>
<td>accountingInterval</td>
<td>3.6</td>
<td><a href="#">4</a></td>
</tr>
<tr>
<td>db2.jcc.allowSqljDuplicateStaticQueries</td>
<td></td>
<td>2.11</td>
<td><a href="#">4</a></td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.apiKey (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity and dash DB only)</td>
<td>apiKey</td>
<td>4.24</td>
<td><a href="#">3</a> <a href="#">4</a></td>
</tr>
<tr>
<td>db2.jcc.charOutputSize</td>
<td>charOutputSize</td>
<td>2.9</td>
<td><a href="#">5</a> <a href="#">6</a></td>
</tr>
</tbody>
</table>

Table 40. Summary of Configuration properties and corresponding Connection and DataSource properties

Chapter 7. JDBC and SQLJ reference information 295
Table 40. Summary of Configuration properties and corresponding Connection and DataSource properties (continued)

<table>
<thead>
<tr>
<th>Configuration property name</th>
<th>Connection or DataSource property name</th>
<th>Introduced in driver version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>db2.jcc.currentSchema</td>
<td>currentSchema</td>
<td>1.2</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.override.currentSchema</td>
<td>currentSchema</td>
<td>1.2</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.currentSQLID</td>
<td>currentSQLID</td>
<td>1.3</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.override.currentSQLID</td>
<td>currentSQLID</td>
<td>1.3</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.decimalRoundingMode</td>
<td>decimalRoundingMode</td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.override.decimalRoundingMode</td>
<td>decimalRoundingMode</td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.defaultSQLState</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>db2.jcc.diagLevelExceptionCode</td>
<td></td>
<td>3.69, 4.19</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.disableSQLExceptionOnSchedule</td>
<td></td>
<td>1.8</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.disableTimezone</td>
<td></td>
<td>3.68, 4.18</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.dumpDiagLevel</td>
<td></td>
<td>3.69, 4.19</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.dumpPool</td>
<td>dumpPool</td>
<td>3.52, 4.2</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.dumpPoolStatisticsOnSchedule</td>
<td>dumpPoolStatisticsOnSchedule</td>
<td>3.52, 4.2</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.dumpPoolStatisticsOnScheduleFile</td>
<td>dumpPoolStatisticsOnScheduleFile</td>
<td>3.52, 4.2</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.enableNetAddressGetHostName</td>
<td></td>
<td>3.63, 4.13</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.override.enableMultirowInsertSupport</td>
<td>enableMultirowInsertSupport</td>
<td>3.62, 4.12</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.enableTzosLBF</td>
<td>enableTzosLBF</td>
<td>3.61, 4.11</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.override.enableTzosLBF</td>
<td>enableTzosLBF</td>
<td>3.61, 4.11</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.override.enableTzosLBFSPResultSets</td>
<td>enableTzosLBFSPResultSets</td>
<td>3.63, 4.13</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.encryptionAlgorithm</td>
<td>encryptionAlgorithm</td>
<td>3.65, 4.15</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.override.encryptionAlgorithm</td>
<td>encryptionAlgorithm</td>
<td>3.65, 4.15</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.extendedTableInfo</td>
<td>extendedTableInfo</td>
<td>3.69, 4.19</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.jacksonEnabled</td>
<td></td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.lobOutputSize</td>
<td></td>
<td>1.8</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.maxConnCachedParamBufferSize</td>
<td>maxConnCachedParamBufferSize</td>
<td>3.63, 4.13</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.maxRefreshInterval</td>
<td></td>
<td>3.68, 4.13</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.maxTransportObjectIdleTime</td>
<td></td>
<td>3.52, 4.2</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.maxTransportObjectWaitTime</td>
<td></td>
<td>3.52, 4.2</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.maxTransportObjects</td>
<td>maxTransportObjects</td>
<td>2.6</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.minTransportObjects</td>
<td></td>
<td>3.52, 4.2</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.outputDirectory</td>
<td></td>
<td>3.61, 4.11</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.pktList</td>
<td></td>
<td>1.4</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.planName</td>
<td>planName</td>
<td>1.4</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.progressiveStreaming</td>
<td>progressiveStreaming</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.override.progressiveStreaming</td>
<td>progressiveStreaming</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.rollbackOnShutdown</td>
<td></td>
<td>3.50, 4.0</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.securityMechanism</td>
<td>securityMechanism</td>
<td>3.65, 4.15</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.override.securityMechanism</td>
<td>securityMechanism</td>
<td>3.65, 4.15</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.sendCharInputsUTF8</td>
<td>sendCharInputsUTF8</td>
<td>3.50, 4.0</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.sqljStmtCacheSize</td>
<td></td>
<td>3.66, 4.16</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.sqljToolsExitJVMOnCompletion</td>
<td></td>
<td>3.62, 4.12</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.sqljUncustomizedWarningOrException</td>
<td></td>
<td>2.2</td>
<td></td>
</tr>
<tr>
<td>db2.jcc.ssld</td>
<td>ssid</td>
<td>3.6</td>
<td></td>
</tr>
</tbody>
</table>
The following definitions describe the meanings of the configuration properties:

**accessToken**

Used to get connection used for authentication by dash DB when security mechanism is PLUGIN_SECURITY. The data type of this property is String.

The IBM Data Server Driver for JDBC and SQLJ uses this option only if `securityMechanism` is set to PLUGIN_SECURITY(15). If no value is set for the parameter `pluginName`, connection is attempted by using default value of `gss_dashdb_iam` to `pluginName`.

If the property is set along with user or `apiKey`, a connection error is thrown indicating only one of them must be used.

Maximum length of the `accessToken` is set to 8192.
This value applies only to SSL connections to dash DB. Non-SSL connection is not supported. This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

\textbf{db2.jcc.accountingInterval}

Specifies whether DB2 accounting records are produced at commit points or on termination of the physical connection to the data source. If the value of \texttt{db2.jcc.accountingInterval} is COM\texttt{MIT}, DB2 accounting records are produced at commit points. The property is illustrated in the following example:

\begin{quote}
\texttt{db2.jcc.accountingInterval=COMMIT}
\end{quote}

Otherwise, accounting records are produced on termination of the physical connection to the data source.

\texttt{db2.jcc.accountingInterval} applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. \texttt{db2.jcc.accountingInterval} is not applicable to connections under CICS or IMS, or for Java stored procedures.

You can override \texttt{db2.jcc.accountingInterval} by setting the accounting\texttt{Interval} property for a Connection or DataSource object.

This configuration property applies only to DB2 for z/OS.

\textbf{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 \texttt{db2.jcc.allowSqljDuplicateStaticQueries} to \texttt{YES} or \texttt{true}.

\textbf{db2.jcc.alternateUTF8Encoding}

Specifies whether the JDBC driver that runs under IBM JRE Version 8 and above uses the UTF8J implementation in place of UTF8, when working with UTF8 data. Java 8 enforces stricter UTF8 implementation rules that might cause malformed input exceptions to occur with certain data, when upgrading to Java 8. The UTF8J implementation available in IBM JRE is expected to be compatible with prior versions of Java 8 UTF8 implementation that allows such data to be processed successfully with IBM JRE 8 and above.

\begin{itemize}
\item \texttt{0} Default value that means use the standard UTF8 implementation that is provided by the JRE for UTF8 data.
\item \texttt{1} Value that means use UTF8J implementation for UTF8 data, if running under IBM JRE.
\end{itemize}

\textbf{apiKey}

Used to get connection used for authentication by dash DB when security mechanism is PLUGIN\_SECURITY. The data type of this property is String.

The IBM Data Server Driver for JDBC and SQLJ uses this option only if security\texttt{Mechanism} is set to PLUGIN\_SECURITY(15). If no value is set for the parameter plugin\texttt{Name}, connection is attempted by using default value of gss\_dashdb\_iam to plugin\texttt{Name}.

If the property is set along with user or accessToken, a connection error is thrown indicating only one of them must be used.

Maximum length of the apiKey is set to 8192.
This value applies only to SSL connections to dash DB. Non-SSL connection is not supported. This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

\textbf{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 \texttt{db2.jcc.charOutputSize} to the largest expected length for any Types.CHAR INOUT or OUT parameter.

\texttt{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.

When choosing the value for \texttt{db2.jcc.charOutputSize} 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 \texttt{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 3 bytes. (The euro symbol is an example of a 3-byte UTF-8 character). To ensure that the value of \texttt{db2.jcc.charOutputSize} is large enough, if you have no information about the output data, set \texttt{db2.jcc.charOutputSize} to three times the defined length of the largest CHAR parameter.

This configuration property applies only to DB2 for z/OS.

\textbf{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 on Linux, UNIX, and Windows systems.

\textbf{db2.jcc.currentSQLID or db2.jcc.override.currentSQLID}

The properties specify the following statements:

- 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.

\texttt{currentSQLID} sets the value in the CURRENT SQLID special register on a DB2 for z/OS server. If the \texttt{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 on Linux, UNIX, and Windows systems data servers.

The following includes possible values of the properties:

- **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 represent 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 on Linux, UNIX, and Windows systems 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=xxxxx**
The value xxxxx in the format 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 5 bytes, the driver truncates the value to 5 bytes. If xxxxx is shorter than 5 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.diagLevelExceptionCode**
Specifies whether to collect detailed diagnostic information for SQL errors or not.

The following values depict possible values of the property:
- -1 Collect diagnostic data for all SQL codes.
- 0 Collection of diagnostic data is disabled. 0 is the default value.
- 1 Collect diagnostic data for SQL code -30108.
- 2 Collect diagnostic data for SQL code -1224.
- 4 Collect diagnostic data for SQL code -204.
- 8 Collect diagnostic data for SQL code -4499.
- 16 Collect diagnostic data for SQL code -20542.

To specify multiple SQL codes, combine the diagnostic constants that correspond to the SQL codes. For example, specify 13 to collect diagnostic data for SQL codes -30108, -204, and -4499.

Negative signs in SQL warnings are optional. SQL codes can be suffixed by 'n' for negative SQL codes and 'p' for positive SQL Codes.

To collect diagnostic information for -204, -30108, the db2.jcc.diagLevelExceptionCode entry in the properties file would look like one of the three examples:
- db2.jcc.diagLevelExceptionCode=-204,30108n
- db2.jcc.diagLevelExceptionCode=204,30108
- db2.jcc.diagLevelExceptionCode=-204,30108,4499n

SQL Warnings must be configured with a positive sign or suffix with 'p'. For example, to collect diagnostic information for SQL warning +100, SQL error code -30108 and SQL warning +222, the properties file must have an entry that looks like one of the following statements:
- db2.jcc.diagLevelExceptionCode=+100,-30108,222p
In versions 3.69 and 4.19 of the IBM Data Server Driver for JDBC and SQLJ, to log diagnostic information for error codes -204 and -30108, you must add the integer constants that are assigned to them. These integer constants that are related to the IBM Data Server Driver for JDBC and SQLJ are now discontinued.

**db2.jcc.dumpDiagLevel**
Specifies whether to collect diagnostic information for all reason codes or only critical reason codes.

The following values show possible values of the property:

1. Collect diagnostic data for all reason codes of SQL codes that are specified in `db2.jcc.diagLevelExceptionCode`.
2. Collect diagnostic data for only critical reason codes of SQL codes that are specified in `db2.jcc.diagLevelExceptionCode`. 2 is the default value.

**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.

The following list illustrates types of statistics with the `db2.jcc.dumpPool` property, that you can specify one or more of:
- **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.enableInetAddressGetHostName**

Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses the `InetAddress.getHostName` and `InetAddress.getCanonicalHostName` methods to determine the host name for an IP address.

`db2.jcc.enableInetAddressGetHostName` applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity. Possible values of the property include the following values:

- **true** The IBM Data Server Driver for JDBC and SQLJ uses the `InetAddress.getHostName` and `InetAddress.getCanonicalHostName` methods to determine the host name for an IP address.
  
  When you specify `true`, applications might take longer to run because of the additional time that is required for DNS lookup operations.

- **false** The IBM Data Server Driver for JDBC and SQLJ uses the `InetAddress.getHostAddress` method to return the IP address. The returned IP address is used in place of the host name.

  For versions 3.65 and 4.15 or later of the IBM Data Server Driver for JDBC and SQLJ, the default is `false`. For versions 3.64 and 4.14 or earlier, the default is `true`. 
**db2.jcc.override.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 default is true.

`db2.jcc.override.enableMultiRowInsertSupport` must be set to `false` if INSERT FROM SELECT statements are executed in a batch. Otherwise, the driver throws a `BatchUpdateException`.

Possible values are:

- **true**: Specifies the default value that 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.

- **false**: Specifies that the IBM Data Server Driver for JDBC and SQLJ does not use 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.

**db2.jcc.enableT2zosLBF or db2.jcc.override.enableT2zosLBF**

Specifies whether limited block fetch is used for connections that use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to retrieve data from a table on a local DB2 for z/OS data server. Possible values are:

- **0 or not specified**: This value is the default.
  - For a connection to a DB2 for z/OS data server in Version 10 conversion mode, specifies that limited block fetch is not used for retrieving data from a local table that uses IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.
  - For a connection to a DB2 for z/OS data server in Version 10 new-function mode or later, specifies that limited block fetch is used for retrieving data from a local table that uses IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

- **1**: Specifies that limited block fetch is used for retrieving data from a local table that uses IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

- **2**: Specifies that limited block fetch is not used for retrieving data from a local table that uses IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

`db2.jcc.enableT2zosLBF` and `db2.jcc.override.enableT2zosLBF` apply to DB2 for z/OS data servers in Version 10 new-function mode or later. These properties do not apply to retrieval of stored procedure result sets.

**db2.jcc.enableT2zosLBFSPResultSets or db2.jcc.override.enableT2zosLBFSPResultSets**

Specifies whether limited block fetch is used for connections that use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to retrieve data from a stored procedure result set on a local DB2 for z/OS data server. Possible values are:

- **0 or 1**: Specifies the default value that limited block fetch is used for retrieving data from a stored procedure result set by using IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.
2 Specifies that limited block fetch is not used for retrieving data from a stored procedure result set by using IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

db2.jcc.enableT2zosLBFSResultSets and db2.jcc.override.enableT2zosLBFSResultSets apply to DB2 for z/OS data servers in Version 10 new-function mode or later.

db2.jcc.encryptionAlgorithm or db2.jcc.override.encryptionAlgorithm
Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses 56-bit DES (weak) encryption or 256-bit AES (strong) encryption.

db2.jcc.encryptionAlgorithm or db2.jcc.override.encryptionAlgorithm can be specified only if db2.jcc.securityMechanism or db2.jcc.securityMechanism is set to 7 or 9.

Possible values of the properties include the following values:

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.

For AES encryption, you need an unrestricted policy file for JCE. That file is available at the following location:


db2.jcc.encryptionAlgorithm can be specified only if the db2.jcc.securityMechanism, db2.jcc.override.securityMechanism, or securityMechanism value is set for encrypted password security or encrypted user ID and password security.

db2.jcc.extendedTableInfo
Specifies whether information about extended table types is returned from a DatabaseMetaData.getTables method call. Currently, one extended table type exists: ACCEL-ONLY TABLE.

0 The result set that is returned by the DatabaseMetaData.getTables method does not contain columns for extended table types.

Rows for extended table types are returned only if "TABLE" is explicitly specified in the types parameter value. In this case, extended table types are listed as TABLE in the TABLE_TYPE column of the result set.

1 The result set that is returned by the DatabaseMetaData.getTables method contains rows and columns for extended table types. In particular:

• The result set contains these extra columns after the columns that are always returned in the result set from DatabaseMetaData.getTables:
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEMPORAL_TABLE_TYPE</td>
<td>String</td>
<td>Contains the type of temporal table. The following shows possible values of the temporal table:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>SYSTEM</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>System-period temporal table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>APPLICATION</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Application-period temporal table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>BITEMPORAL</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitemporal table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Empty string</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not a temporal table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This row is returned for connections to DB2 for z/OS Version 10 or later.</td>
</tr>
<tr>
<td>IS_ACCELERATED</td>
<td>String</td>
<td>Indicates whether the table is an accelerated table. Possible values are YES or NO.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This row is returned for connections to DB2 for z/OS Version 10 or later.</td>
</tr>
<tr>
<td>ACCEL_ARCHIVE_STATUS</td>
<td>String</td>
<td>Contains the archive status of the table in the accelerator database. See the description of the ARCHIVE column in <code>SYSACCEL_SYSACCELERATEDTABLES</code> table (DB2 SQL) for the possible values and their meanings.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This row is returned for connections to DB2 for z/OS Version 10 or later.</td>
</tr>
<tr>
<td>IS_ARCHIVE_ENABLED</td>
<td>String</td>
<td>Indicates whether the table is an archive-enabled table. Possible values are YES or NO.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This row is returned for connections to DB2 for z/OS Version 11 or later.</td>
</tr>
</tbody>
</table>

- Rows for extended table types are returned under the following circumstances:
  - All table types are implicitly requested by specifying null in the `types` parameter value.
  - An extended table type name is explicitly specified in the `types` parameter value.

In this case, the extended table type is listed by its extended table type name in the `TABLE_TYPE` column of the result set.

**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 of the property include the following values:

**true or yes**

Indicates that JMX is enabled.

**Any other value**

Indicates the default value that JMX is disabled.

**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.maxConnCachedParamBufferSize**

Specifies the maximum size of an internal buffer that is used for caching input parameter values for `PreparedStatement` objects. The buffer caches values on the native code side that are passed from the driver's Java code side for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. The buffer is used by all `PreparedStatement` objects for a `Connection`. The default is 1048576 (1 MB). The default must be adequate for most users. Set `db2.jcc.maxConnCachedParamBufferSize` to a larger value if many applications that run under the driver instance have `PreparedStatement` objects with large numbers of input parameters or large input parameters. The `db2.jcc.maxConnCachedParamBufferSize` must be larger than the maximum size of all input parameter data for a `Connection`. However, you also need to consider the total number of connections and the maximum amount of memory that is available when you set the `db2.jcc.maxConnCachedParamBufferSize` value.

The buffer exists for the life of a `Connection`, unless it reaches the maximum specified size. If that happens, the buffer is freed on each call to the native code. The corresponding buffer on the Java code side is freed on `PreparedStatement.clearParameters` and `PreparedStatement.close` calls. The buffers are not cleared if an application calls `PreparedStatement.clearParameters`, and the buffers did not reach the maximum size.

**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 minimum valid value is 1.

For version 3.63 or 4.13 or later of the IBM Data Server Driver for JDBC and SQLJ, the default is 10 seconds. For earlier versions of the driver, the default is 30 seconds.

**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 10. Setting `db2.jcc.maxTransportObjectIdleTime` to a value less than 0 causes unused transport objects to be deleted from the pool immediately. Setting the value below 0 is not recommended because it can cause severe performance degradation.

**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.

For version 3.63 or 4.13 or later of the IBM Data Server Driver for JDBC and SQLJ, the default is 1000. For earlier versions of the driver, the default is -1.

Any value that is less than or equal to 0 means that no limit to the number of transport objects in the global transport object pool exists.

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 is not reached yet. 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.

Any negative value means that applications wait forever.

For version 3.63 or 4.13 or later of the IBM Data Server Driver for JDBC and SQLJ, the default is 1 second. For earlier versions of the driver, the default is -1.

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, no transport objects exist 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.

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 on Linux, UNIX, and Windows systems.

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:

```
<collection-ID>.*
```

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 an 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, 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 also known as dynamic data format.

Valid values for the properties include the following values:

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.

Valid values for the properties include the following values:

**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 default value that IBM Data Server Driver for JDBC and SQLJ takes no action with respect to rollback processing during processing of JVM shutdown hooks.

This configuration property applies only to DB2 for z/OS.

**db2.jcc.securityMechanism or db2.jcc.override.securityMechanism**
Specifies the DRDA security mechanism. Possible values are:

- **3** User ID and password
- **4** User ID only
- **7** User ID, encrypted password
- **9** Encrypted user ID and password
- **11** Kerberos. This value does not apply to connections to IBM Informix.
- **12** Encrypted user ID and encrypted security-sensitive data. This value applies to connections to DB2 for z/OS only.
- **13** Encrypted user ID and password, and encrypted security-sensitive data. This value does not apply to connections to IBM Informix.
- **15** Plug-in security. This value applies to connections to Db2 on Linux, UNIX, and Windows systems only.
- **16** Encrypted user ID. This value does not apply to connections to IBM Informix.
- **18** Client certificate security, that uses SSL. This value applies to connections to DB2 for z/OS Version 10 and later only.

The security mechanism that is specified by this property 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 db2.jcc.securityMechanism is 3. If the server does not support user ID and password security, but supports encrypted user ID and password security (9), the IBM Data Server Driver for JDBC and SQLJ driver upgrades the security mechanism to encrypted user ID 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.

This property does not apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on 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.

Valid values for the properties include the following values:

**no, false, or 2**

Specifies the default value 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.

**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 length of the data. UTF-8 data can require up to 3 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:

```bash
db2.jcc.sendCharInputsUTF8=yes
db2.jcc.sendCharInputsUTF8=true
db2.jcc.sendCharInputsUTF8=1
```

This configuration property applies only to DB2 for z/OS.

---

**db2.jcc.sqljStmtCacheSize**

Specifies the maximum number of statements that are in the SQLJ statement
cache for each DefaultContext instance and each JVM thread. This value applies to SQLJ stored procedures that run in a 64-bit, multi-threaded environment. The default is 10 statements.

In a multi-threaded environment, the IBM Data Server Driver for JDBC and SQLJ caches statements that are associated with each instance of a DefaultContext object that is used by each JVM thread. When the driver attempts to cache a statement after the db2.jcc.sqljStmtCacheSize value is reached. The least recently used cached statement is purged and replaced by the new statement.

This property applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 11 or later.

db2.jcc.sqljToolsExitJvmOnCompletion
Specifies whether the Java programs that underlie SQLJ tools such as db2sqljcustomize and db2sqljbinder 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. The value 0 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 on Linux, UNIX, and Windows systems.

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.

The property is illustrated in the following 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 ends. 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.ssoid` 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.sslCertLocation` or `db2.jcc.override.sslCertLocation`

Specifies that an application can configure the location of a trusted certificate file. For applications in possession of the database server certificate, `sslCertLocation` is the only property that is needed to be configured to instruct IBM Data Server Driver for JDBC and SQLJ to trust the certificate for SSL connections. This property removes the need to import the certificate into a Java truststore database and related driver configurations.

The default value for `sslCertLocation` is provided by the `db2.jcc.sslCertLocation` configuration property. If the `db2.jcc.sslCertLocation` configuration property is not specified, `sslTruststoreLocation`, and `sslCertLocation` properties are not configured, and `sslConnection` is set to true, IBM Data Server Driver for JDBC and SQLJ uses the default truststore for the Java runtime environment.

The `db2.jcc.override.sslCertLocation` property overrides the `sslCertLocation` property for a `Connection` or `DataSource` object.

`db2.jcc.sslConnection` or `db2.jcc.override.sslConnection`

Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses an SSL socket to connect to the data source. If the value is true, the connection uses an SSL socket. If the value is `false`, the connection uses a plain socket.

The `db2.jcc.override.sslConnection` property overrides the `sslConnection` property for a `Connection` or `DataSource` object.

If no property is specified, the default value is `false`.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

`db2.jcc.sslTrustStoreLocation` or `db2.jcc.override.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 `db2.jcc.sslConnection`, `db2.jcc.override.sslConnection`, or `sslConnection` property is set to true.
If db2.jcc.sslTrustStoreLocation or db2.jcc.override.sslTrustStoreLocation, or
sslTrustStoreLocation is set, and db2.jcc.sslConnection, db2.jcc.override.sslConnection, or sslConnection is set to true, the IBM Data
Server Driver for JDBC and SQLJ uses the db2.jcc.sslTrustStoreLocation,
db2.jcc.override.sslTrustStoreLocation, or sslTrustStoreLocation value instead of
the value in the javax.net.ssl.trustStore Java property.

The db2.jcc.override.sslTrustStoreLocation property overrides the
sslTrustStoreLocation property for a Connection or DataSource object.

If no property is specified, the default value is null.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ
type 4 connectivity.

`db2.jcc.sslTrustStorePassword` or `db2.jcc.override.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
db2.jcc.sslConnection, db2.jcc.override.sslConnection, or sslConnection property
is set to true.

If db2.jcc.sslTrustStorePassword, db2.jcc.override.sslTrustStorePassword, or
sslTrustStorePassword is set, and db2.jcc.sslConnection, db2.jcc.override.sslConnection, or sslConnection is set to true, the IBM Data
Server Driver for JDBC and SQLJ uses the db2.jcc.sslTrustStorePassword,
db2.jcc.override.sslTrustStorePassword, or sslTrustStorePassword value instead of
the value in the javax.net.ssl.trustStorePassword Java property.

The db2.jcc.override.sslTrustStorePassword property overrides the
sslTrustStorePassword property for a Connection or DataSource object.

If no property is specified, the default value is null.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ
type 4 connectivity.

`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. These
properties do not apply to IBM Data Server Driver for JDBC and SQLJ type 2
connectivity on DB2 for z/OS. 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.
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 must 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.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. The db2.jcc.traceFile property does not apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

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 must 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. These properties do not apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. 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 must set the trace properties under the direction of IBM Software Support.

**db2.jcc.traceFileCount**

Specifies the maximum number of trace files for circular tracing. The IBM Data 
Server Driver for JDBC and SQLJ uses this property only when 
db2.jcc.traceOption is set to 1. The default value is 2.

This property does not apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

You must set the trace properties under the direction of IBM Software Support.

**db2.jcc.traceFileSize**

Specifies the maximum size of each trace file, for circular tracing. The IBM 
Data Server Driver for JDBC and SQLJ uses this property only when 
db2.jcc.traceOption is set to 1. The default value is 10485760 (10 MB).

This property does not apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

You must set the trace properties under the direction of IBM Software Support.

**db2.jcc.traceOption**

Specifies how trace data is collected. The data type of this property is int. 
Possible values of the property include the following values:

0  Specifies the default value that means a single trace file is generated, 
and that no limit to the size of the file exist.

1  Specifies that the IBM Data Server Driver for JDBC and SQLJ does 
circular tracing. Circular tracing is done as follows:

1. When an application writes its first trace record, the driver creates a 
file.

2. The driver writes trace data to the file.

3. When the size of the file is equal to the value of property 
db2.jcc.traceFileSize, the driver creates another file.

4. The driver repeats steps 2 and 3 until the number of files to which 
data written is equal to the value of property 
db2.jcc.traceFileCount.

5. The driver writes data to the first trace file, overwriting the existing 
data.

6. The driver repeats steps 3 through 5 until the application 
completes.

The file names for the trace files are the file names that are determined 
by the db2.jcc.traceFile, db2.jcc.override.traceFile, db2.jcc.traceDirectory, 
db2.jcc.override.traceDirectory property, which is appended with .1 for 
the first file, .2 for the second file, and so on.

This property does not apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

You must 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.
The default is false for version 3.69 or earlier of the IBM Data Server Driver for
JDBC and SQLJ, and the default is true for versions 3.69 and 4.19 or later of
the IBM Data Server Driver for JDBC and SQLJ.

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.t2zosTraceFile`
- `db2.jcc.t2zosTraceBufferSize`
- `db2.jcc.t2zosTraceWrap`

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.jcc.t2zosTraceFile property value.

For example, specifying the following setting for `db2.jcc.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 KB, 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:

<table>
<thead>
<tr>
<th>Specified value ( n )</th>
<th>Trace buffer size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;64</td>
<td>64</td>
</tr>
<tr>
<td>64&lt;=( n )&lt;128</td>
<td>64</td>
</tr>
<tr>
<td>128&lt;=( n )&lt;256</td>
<td>128</td>
</tr>
<tr>
<td>256&lt;=( n )&lt;512</td>
<td>256</td>
</tr>
<tr>
<td>512&lt;=( n )&lt;1024</td>
<td>512</td>
</tr>
<tr>
<td>1024&lt;=( n )&lt;2048</td>
<td>1024</td>
</tr>
<tr>
<td>2048&lt;=( n )&lt;4096</td>
<td>2048</td>
</tr>
<tr>
<td>( n )\geq 4096</td>
<td>4096</td>
</tr>
</tbody>
</table>

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 must 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.
2. Do not wrap the trace.

The default is 1. This parameter is optional. The following example shows how to set the property:

```
DB2SQLJ_TRACE_WRAP=0
```

You must 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. The value `false` is the default.

This configuration property applies only to DB2 for z/OS.

**Related concepts:**

- Customization of IBM Data Server Driver for JDBC and SQLJ configuration properties
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 support them. The drivers and their supported platforms are:

Table 42. JDBC drivers for DB2 and IBM Informix database systems

<table>
<thead>
<tr>
<th>JDBC driver name</th>
<th>Associated data source</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Data Server Driver for JDBC and SQLJ</td>
<td>Db2 on Linux, UNIX, and Windows systems, DB2 for z/OS, or IBM Informix</td>
</tr>
<tr>
<td>IBM Informix JDBC Driver (IBM Informix JDBC Driver)</td>
<td>IBM Informix</td>
</tr>
</tbody>
</table>

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 43. Support for java.sql.Array methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>free</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getArray</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBaseType</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBaseTypeName</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getResultSet</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Notes:
1. Under the IBM Data Server Driver for JDBC and SQLJ, Array methods are supported for connections to Db2 on Linux, UNIX, and Windows systems data sources only.
2. This is a JDBC 4.0 method.

Table 44. Support for java.sql.BatchUpdateException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getUpdateCounts</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 45. Support for java.sql.Blob methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>free</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBinaryStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBytes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>length</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>position</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setBinaryStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setByte</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>truncate</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 45. Support for `java.sql.Blob` methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
</table>

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 46. Support for `java.sql.CallableStatement` methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from <code>java.sql.Statement</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from <code>java.sql.PreparedStatement</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getArray</code></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getBigDecimal</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getBlob</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getBoolean</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getByte</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getBytes</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getClob</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getDate</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getDouble</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getFloat</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getInt</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getLong</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getObject</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getRef</code></td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>getRowId</code></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>getShort</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getString</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getTime</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getTimestamp</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getURL</code></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>registerOutParameter</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>setAsciiStream</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>setBigDecimal</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>setBinaryStream</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>setBoolean</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>setByte</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>setBytes</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Chapter 7. JDBC and SQLJ reference information 321
### Table 46. Support for java.sql.CallableStatement methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>setCharacterStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setDate</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setDouble</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setFloat</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setInt</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLong</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setNull</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setObject</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setShort</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setString</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setTime</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setTimestamp</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setURL</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>wasNull</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**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: `getXXX(String parameterName)`
4. The following JDBC 4.1 method is supported: `getObject(int parameterIndex, java.lang.Class<T> type)` `getObject(java.lang.String parameterName, java.lang.Class<T> type)`
5. 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.
6. The following form of the getobject method is not supported: `getObject(int parameterIndex, java.util.Map map)`
7. The following form of the registerOutParameter method is not supported: `registerOutParameter(int parameterIndex, int jdbcType, String typeName)`
8. The following forms of CallableStatement.setXXX methods are not supported if the data source is DB2 for z/OS: `setXXX(String parameterName,...)`
9. The following form of setNull is not supported: `setNull(int parameterIndex, int jdbcType, String typeName)`

### Table 47. Support for java.sql.Clob methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>free</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getAsciiStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getCharacterStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSubString</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

---

322  Application Programming Guide and Reference for Java
**Table 47. Support for java.sql.Clob methods (continued)**

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>position</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setAsciiStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setCharacterStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setString</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>truncate</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**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 48. Support for javax.sql.CommonDataSource methods**

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getLoginTimeout</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLogWriter</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getParentLogger</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>setLoginTimeout</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLogWriter</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Notes:**

1. This is a JDBC 4.1 method.

**Table 49. Support for java.sql.Connection methods**

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>clearWarnings</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>close</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>commit</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>createBlob</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>createClob</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>createStatement</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getAutoCommit</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getCatalog</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getClientInfo</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getHoldability</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMetaData</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getNetworkTimeout</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
## Table 49. Support for `java.sql.Connection` methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getSchema</code></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>getTransactionIsolation</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getTypeMap</code></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getWarnings</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>isClosed</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>isReadOnly</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>isValid</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>nativeSQL</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>prepareCall</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>prepareStatement</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>releaseSavepoint</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>rollback</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>setAutoCommit</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>setCatalog</code></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>setClientInfo</code></td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>setNetworkTimeout</code></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>setReadOnly</code></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>setSavepoint</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>setSchema</code></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><code>setTransactionIsolation</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>setTypeMap</code></td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Notes:

1. This is a JDBC 4.1 method.
2. This is a JDBC 4.0 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 `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 50. Support for `javax.sql.ConnectionEvent` methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from <code>java.util.EventObject</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>getSQLException</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

## Table 51. Support for `javax.sql.ConnectionEventListener` methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connectionClosed</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 51. Support for javax.sql.ConnectionEventListener methods  (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectionErrorOccurred</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 52. Support for javax.sql.ConnectionPoolDataSource methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getLoginTimeout</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLogWriter</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getPooledConnection</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLoginTimeout</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLogWriter</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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 53. Support for java.sql.DatabaseMetaData methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>allProceduresAreCallable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>allTablesAreSelectable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>dataDefinitionCausesTransactionCommit</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>dataDefinitionIgnoredInTransactions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>deletesAreDetected</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>doesMaxRowSizeIncludeBlobs</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>generatedKeyAlwaysReturned</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getAttributes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBestRowIdentifier</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getCatalogs</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getCatalogSeparator</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getCatalogTerm</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getClientInfoProperties</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>getColumnPrivileges</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getColumns</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getConnection</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getCrossReference</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDatabaseMajorVersion</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDatabaseMinorVersion</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDatabaseProductName</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDatabaseProductVersion</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDefaultTransactionIsolation</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JDBC method</td>
<td>IBM Data Server Driver for JDBC and SQLJ support</td>
<td>IBM Informix JDBC Driver support</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>getDriverMajorVersion</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDriverMinorVersion</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDriverName</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDriverVersion</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getExportedKeys</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getFunctionColumns</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getFunctions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getExtraNameCharacters</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getIdentifierQuoteString</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getImportedKeys</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getIndexInfo</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getJDBCMajorVersion</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getJDBCMinorVersion</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxBinaryLiteralLength</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxCatalogNameLength</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxCharLiteralLength</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxColumnNameLength</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxColumnsInGroupBy</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxColumnsInIndex</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxColumnsInOrderBy</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxColumnsInSelect</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxColumnsInTable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxConnections</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxCursorNameLength</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxIndexLength</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxProcedureNameLength</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxRowSize</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxSchemaNameLength</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxStatementLength</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxStatements</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxTableNameLength</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxTablesInSelect</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxUserNameLength</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getNumericFunctions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getPrimaryKeys</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getProcedureColumns</td>
<td>Yes [8 on page 330]</td>
<td>Yes</td>
</tr>
<tr>
<td>getProcedures</td>
<td>Yes [8 on page 330]</td>
<td>Yes</td>
</tr>
<tr>
<td>getProcedureTerm</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 53. Support for java.sql.DatabaseMetaData methods  (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getPseudoColumns</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>getResultSetHoldability</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getRowIdLifetime</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>getSchemas</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>getSchemaTerm</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>getSearchStringEscape</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSQLKeywords</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSQLStateType</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getStringFunctions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSuperTables</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSuperTypes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSystemFunctions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTablePrivileges</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTables</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>getTableTypes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTimeDateFunctions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTypeInfo</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getUDTs</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>getURL</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getUserName</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getVersionColumns</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>insertsAreDetected</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isCatalogAtStart</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isReadOnly</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>locatorsUpdateCopy</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>nullPlusNonNullIsNull</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>nullsAreSortedAtEnd</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>nullsAreSortedAtStart</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>nullsAreSortedHigh</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>nullsAreSortedLow</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>othersDeletesAreVisible</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>othersInsertsAreVisible</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>othersUpdatesAreVisible</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ownDeletesAreVisible</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ownInsertsAreVisible</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ownUpdatesAreVisible</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>storesLowerCaseIdentifiers</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>storesLowerCaseQuotedIdentifiers</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JDBC method</td>
<td>IBM Data Server Driver for JDBC and SQLJ support</td>
<td>IBM Informix JDBC Driver support</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>--------------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>storesMixedCaseIdentifiers</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>storesMixedCaseQuotedIdentifiers</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>storesUpperCaseIdentifiers</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>storesUpperCaseQuotedIdentifiers</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsAlterTableWithAddColumn</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsAlterTableWithDropColumn</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsANSI92EntryLevelSQL</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsANSI92FullSQL</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsANSI92IntermediateSQL</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsBatchUpdates</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsCatalogsInDataManipulation</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsCatalogsInIndexDefinitions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsCatalogsInPrivilegeDefinitions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsCatalogsInProcedureCalls</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsCatalogsInTableDefinitions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SupportsColumnAliasing</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsConvert</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsCoreSQLGrammar</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsCorrelatedSubqueries</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsDataDefinitionAndDataManipulationTransactions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsDataManipulationTransactionsOnly</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsDifferentTableCorrelationNames</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsExpressionsInOrderBy</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsExtendedSQLGrammar</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsFullOuterJoins</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsGetGeneratedKeys</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsGroupBy</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsGroupByBeyondSelect</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsGroupByUnrelated</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsIntegrityEnhancementFacility</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsLikeEscapeClause</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsLimitedOuterJoins</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsMinimumSQLGrammar</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsMixedCaseIdentifiers</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsMixedCaseQuotedIdentifiers</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsMultipleOpenResults</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsMultipleResultSets</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsMultipleTransactions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 53. Support for java.sql.DatabaseMetaData methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>supportsNamedParameters</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsNonNullableColumns</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsOpenCursorsAcrossCommit</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsOpenCursorsAcrossRollback</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsOpenStatementsAcrossCommit</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsOpenStatementsAcrossRollback</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsOrderByUnrelated</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsOuterJoins</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsPositionedDelete</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsPositionedUpdate</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsResultSetConcurrency</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsResultSetHoldability</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsResultSetType</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSavepoints</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSchemasInDataManipulation</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSchemasInIndexDefinitions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSchemasInPrivilegeDefinitions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSchemasInProcedureCalls</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSchemasInTableDefinitions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSelectForUpdate</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsStoredProcedures</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSubqueriesInComparisons</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSubqueriesInExists</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSubqueriesInIns</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSubqueriesInQuantifieds</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSuperTables</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>supportsSuperTypes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>supportsTableCorrelationNames</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsTransactionIsolationLevel</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsTransactions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsUnion</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsUnionAll</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updatesAreDetected</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>usesLocalFilePerTable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>usesLocalFiles</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 53. Support for java.sql.DatabaseMetaData methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. DB2 data sources return false for this method. IBM Informix data sources return true.
2. This is a JDBC 4.1 method.
3. This method is supported for connections to Db2 on Linux, UNIX, and Windows systems and IBM Informix only.
4. 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.
5. 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.
6. DB2 data sources return true for this method. IBM Informix data sources return false.
7. This is a JDBC 4.0 method.
8. This method returns the additional column that is described by the JDBC 4.0 specification.
9. 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."
10. The JDBC 4.0 form and previous forms of this method are supported.
11. The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows does not support the JDBC 3.0 form of this method.
12. The method can be executed, but it returns an empty ResultSet.

Table 54. Support for java.sql.DataSource methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getConnection</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLoginTimeout</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLogWriter</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLoginTimeout</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLogWriter</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Notes:
1. This method is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

Table 55. Support for java.sql.DataTruncation methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.sql.SQLException</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.sql.SQLWarning</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDataSize</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getIndex</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getParameter</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getRead</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

330 Application Programming Guide and Reference for Java
Table 55. Support for java.sql.DataTruncation methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getTransferSize</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 56. Support for java.sql.Driver methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>acceptsURL</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>connect</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMajorVersion</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMinorVersion</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getParentLogger</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>getPropertyInfo</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>jdbcCompliant</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 57. Support for java.sql.DriverManager methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>deregisterDriver</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getConnection</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDriver</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDrivers</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLoginTimeout</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLogStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLogWriter</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>println</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>registerDriver</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLoginTimeout</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLogStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLogWriter</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Notes:

1. This method is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

Table 58. Support for java.sql.ParameterMetaData methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getParameterClassName</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>getParameterCount</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getParameterMode</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getParameterType</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getParameterTypeName</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getPrecision</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 58. Support for java.sql.ParameterMetaData methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getScale</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isNullable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isSigned</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 59. Support for javax.sql.PooledConnection methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>addConnectionEventListener</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>addStatementEventListener</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>close</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getConnection</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>removeConnectionEventListener</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>removeStatementEventListener</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Notes:
1. This is a JDBC 4.0 method.

Table 60. Support for java.sql.PreparedStatement methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.sql.Statement</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>addBatch</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>clearParameters</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>execute</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>executeQuery</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>executeUpdate</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMetaData</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getParameterMetaData</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setArray</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setAsciiStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setBigDecimal</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setBinaryStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setBlob</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setBoolean</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setByte</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setBytes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setCharacterStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setClob</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setDate</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setDouble</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setFloat</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 60. Support for java.sql.PreparedStatement methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>setInt</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLong</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setNull</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setObject</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setRef</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>setRowId</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>setShort</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setString</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setTime</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setTimestamp</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setUnicodeStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setURL</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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 a binary data type.

Table 61. Support for java.sql.Ref methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getBaseTypeName</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 62. Support for java.sql.ResultSet methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>afterLast</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>beforeFirst</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>cancelRowUpdates</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>clearWarnings</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>close</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>deleteRow</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>findColumn</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>first</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getArray</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>getAsciiStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBigDecimal</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBinaryStream</td>
<td>Yes[1]</td>
<td>Yes</td>
</tr>
<tr>
<td>getBlob</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBoolean</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getByte</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBytes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getCharacterStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getClob</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getConcurrency</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getCursorName</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDate</td>
<td>Yes[1]</td>
<td>Yes[1]</td>
</tr>
<tr>
<td>getDouble</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getFetchDirection</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getFetchSize</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getFloat</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getFloat</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getInt</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLong</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMetaData</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getObject</td>
<td>Yes[1]</td>
<td>Yes[1]</td>
</tr>
<tr>
<td>getRef</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getRow</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getRowId[2]</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>getShort</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getStatement</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getString</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTime</td>
<td>Yes[1]</td>
<td>Yes[1]</td>
</tr>
<tr>
<td>getTimestamp</td>
<td>Yes[1]</td>
<td>Yes[1]</td>
</tr>
<tr>
<td>getType</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 62. Support for java.sql.ResultSet methods  (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getUnicodeStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getURL</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getWarnings</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>insertRow</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isAfterLast</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isBeforeFirst</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isFirst</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isLast</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>last</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>moveToCurrentRow</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>moveToInsertRow</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>next</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>previous</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>refreshRow</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>relative</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>rowDeleted</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>rowInserted</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>rowUpdated</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>setFetchDirection</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setFetchSize</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateArray</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>updateAsciiStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateBigDecimal</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateBinaryStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateBlob</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateBoolean</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateByte</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateBytes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateCharacterStream</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateClob</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateDate</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateDouble</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateFloat</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateInt</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateLong</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateNull</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateObject</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateRef</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateRow</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 62. Support for java.sql.ResultSet methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>updateRowId</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>updateShort</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateString</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateTime</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateTimeStamp</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>wasNull</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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 `getDateTime`, `getTime`, or `getTimestamp` method that includes a java.util.Calendar parameter.
4. The following form of the `getObject` method is not supported:
   ```java
generateObject(int parameterIndex, java.util.Map map)
   ``
5. Supported forms of this method include the following JDBC 4.0 forms:
   ```java
   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:
   ```java
   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:
   ```java
   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:
   ```java
   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:
   ```java
   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 63. Support for java.sql.ResultSetMetaData methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getCatalogName</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getColumnClassName</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>getColumnCount</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getColumnDisplaySize</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 63. Support for java.sql.ResultSetMetaData methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getColumnType</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getColumnTypeName</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getPrecision</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getScale</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSchemaName</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTableName</td>
<td>Yes[1]</td>
<td>Yes</td>
</tr>
<tr>
<td>isAutoIncrement</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isCaseSensitive</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isCurrency</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isDefinitelyWritable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isNullable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isReadOnly</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isSearchable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isSigned</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isWritable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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 64. Support for java.sql.RowId methods[1]

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>equals</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>getBytes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>hashCode</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>toString</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

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 65. Support for java.sql.SQLException methods[1]

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>getFailedProperties</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Note:
1. This is a JDBC 4.0 class.
Table 66. Support for java.sql.SQLData methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSQLType_Name</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readSQL</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>writeSQL</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 67. Support for java.sql.SQLDataException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Note:
1. This is a JDBC 4.0 class.

Table 68. Support for java.sql.SQLDataException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Note:
1. This is a JDBC 4.0 class.

Table 69. Support for java.sql.SQLException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSQLState</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getErrorCode</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getNextException</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setNextException</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 70. Support for java.sql.SQLFeatureNotSupported methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Note:
1. This is a JDBC 4.0 class.
### Table 71. Support for java.sql.SQLInput methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>readArray</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readAsciiStream</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readBigDecimal</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readBinaryStream</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readBlob</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readBoolean</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readByte</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readBytes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readCharacterStream</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readClob</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readDate</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readDouble</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readFloat</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readInt</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readLong</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readObject</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readRef</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readShort</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readString</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readTime</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>readTimestamp</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>wasNull</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Table 72. Support for java.sql.SQLIntegrityConstraintViolationException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Note:**
1. This is a JDBC 4.0 class.

### Table 73. Support for java.sql.SQLInvalidAuthorizationSpecException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Note:**
1. This is a JDBC 4.0 class.
### Table 74. Support for java.sql.SQLNonTransientConnectionException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Note:**
1. This is a JDBC 4.0 class.

### Table 75. Support for java.sql.SQLNonTransientException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Note:**
1. This is a JDBC 4.0 class.

### Table 76. Support for java.sql.SQLOutput methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>writeArray</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeAsciiStream</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeBigDecimal</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeBinaryStream</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeBlob</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeBoolean</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeByte</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>writeBytes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>writeCharacterStream</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeClob</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeDate</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>writeDouble</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>writeFloat</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>writeInt</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>writeLong</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeObject</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeRef</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeShort</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>writeString</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeStruct</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeTime</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>writeTimestamp</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 77. Support for java.sql.SQLRecoverableException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Note:
1. This is a JDBC 4.0 class.

Table 78. Support for java.sql.SQLSyntaxErrorException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Note:
1. This is a JDBC 4.0 class.

Table 79. Support for java.sql.SQLTimeoutException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Note:
1. This is a JDBC 4.0 class.

Table 80. Support for java.sql.SQLTransientConnectionException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Note:
1. This is a JDBC 4.0 class.

Table 81. Support for java.sql.SQLTransientException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Note:
1. This is a JDBC 4.0 class.
Table 82. Support for java.sqlTransientRollbackException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Note:
1. This is a JDBC 4.0 class.

Table 83. Support for java.sql.SQLXML methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>free</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>getBinaryStream</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>getCharacterStream</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>getSource</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>getString</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>setBinaryStream</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>setCharacterStream</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>setResult</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>setString</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Notes:
1. These are JDBC 4.0 methods. These methods are not supported for connections to IBM Informix servers.

Table 84. Support for java.sql.Statement methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>addBatch</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>cancel</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>clearBatch</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>clearWarnings</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>close</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>closeOnCompletion</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>execute</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>executeBatch</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>executeQuery</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>executeUpdate</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getConnection</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getFetchDirection</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getFetchSize</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getGeneratedKeys</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxFieldSize</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 84. Support for java.sql.Statement methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getMaxRows</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMoreResults</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getQueryTimeout</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getResultSet</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getResultSetConcurrency</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getResultSetHoldability</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getResultSetConcurrency</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getUpdateCount</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getWarnings</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isCloseOnCompletion</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>isClosed</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isPoolable</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>setCursorName</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setEscapeProcessing</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setFetchDirection</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setFetchSize</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setMaxFieldSize</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setMaxRows</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setPoolable</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>setQueryTimeout</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 84. Support for java.sql.Statement methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notes:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. This is a JDBC 4.1 method.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. For the IBM Data Server Driver for JDBC and SQLJ, Statement.cancel is supported only in the following environments:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Type 2 and type 4 connectivity from a Linux, UNIX, or Windows client to a Db2 on Linux, UNIX, and Windows systems server, Version 8 or later</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Type 2 and type 4 connectivity from a Linux, UNIX, or Windows client to a DB2 for z/OS server, Version 9 or later</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Type 4 connectivity from a z/OS client to a Db2 on Linux, UNIX, and Windows systems server, Version 8 or later</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Type 4 connectivity from a z/OS client to a DB2 for z/OS server, Version 8 or later</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The action that the IBM Data Server Driver for JDBC and SQLJ takes when the application executes Statement.cancel depends on the setting of the DB2BaseDataSource.interruptProcessingMode property.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>In addition, if DB2BaseDataSource.interruptProcessingMode is set to DB2BaseDataSource INTERRUPT_PROCESSING_MODE_STATEMENT_CANCEL or DB2BaseDataSource INTERRUPT_PROCESSING_MODE_CLOSE_SOCKET, and Statement.cancel is executed while an application is executing a method on a ResultSet object, the operation on the ResultSet object might not be canceled. When the Statement.cancel statement is executed, if the underlying processing for the ResultSet method is currently in the data server, execution of the operation is canceled. If the underlying processing for the ResultSet method is in the driver, execution of the operation is not canceled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Not supported for stored procedure ResultSets.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. For DB2 for i, this method is supported only for a seconds value of 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. 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.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on Db2 on Linux, UNIX, and Windows systems, Statement.setQueryTimeout is supported only if Connection or DataSource property queryTimeoutInterruptProcessingMode is set to INTERRUPT_PROCESSING_MODE_STATEMENT_CANCEL.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. For the IBM Data Server Driver for JDBC and SQLJ Version 4.0 and later, Statement.setQueryTimeout is supported for the following methods:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Statement.execute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Statement.executeUpdate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Statement.executeQuery</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The action that the IBM Data Server Driver for JDBC and SQLJ takes when the application executes Statement.setQueryTimeout is supported for the Statement.executeUpdate method only when property queryTimeoutInterruptProcessingMode is set to INTERRUPT_PROCESSING_MODE_CLOSE_SOCKET (2).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. This is a JDBC 4.0 method.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 85. Support for java.sql.Struct methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSQLTypeName</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>getAttributes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 86. Support for java.sql_WRAPPER methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>isWrapperFor</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>unWrap</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
**Table 86. Support for java.sql.Wrapper methods (continued)**

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
</table>

**Notes:**
1. These are JDBC 4.0 methods.

**Table 87. Support for javax.sql.XAConnection methods**

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from javax.sql.PooledConnection</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getXAResource</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Notes:**
1. These methods are supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to a Db2 on Linux, UNIX, and Windows systems server or IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

**Table 88. Support for javax.sql.XDataSource methods**

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getLoginTimeout</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLogWriter</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getXAConnection</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLoginTimeout</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLogWriter</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Table 89. Support for javax.transaction.xa.XAResource methods**

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>IBM Informix JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>commit</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>end</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>forget</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTransactionTimeout</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isSameRM</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>prepare</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>recover</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>rollback</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setTransactionTimeout</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>start</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Notes:**
1. This method is supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to a Db2 on Linux, UNIX, and Windows systems server or IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.
2. When the end method is called, the IBM Data Server Driver for JDBC and SQLJ closes the underlying cursor, even if the TMSUSPEND flag is specified.
3. This method is supported for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to Db2 on Linux, UNIX, and Windows systems 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

```
#sql
connection-declaration-clause
iterator-declaration-clause
executable-clause
```

Usage notes

Keywords in an SQLJ clause are case sensitive, unless those keywords are part of an SQL statement in an executable clause.

Related reference:

- “SQLJ iterator-declaration-clause” on page 351
- “SQLJ executable-clause” on page 353
- “SQLJ connection-declaration-clause” on page 350

SQLJ host-expression

A host expression is a Java variable or expression that is referenced by SQLJ clauses in an SQLJ application program.

Syntax

```
IN
OUT
INOUT
```

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.

**INDICATOR :simple-variable or INDICATOR :(complex-expression)**
Specifies the optional indicator variable for the corresponding Java host variable. The data type of the indicator variable must be the Java short type. The only valid values for :simple-variable or :(complex-expression) are:

For customized applications, and for input, only these values are valid:

<table>
<thead>
<tr>
<th>Indicator value</th>
<th>Equivalent constant</th>
<th>Meaning of value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>sqlj.runtime.ExecutionContextDBNull</td>
<td>Null</td>
</tr>
<tr>
<td>-2, -3, -4, -6</td>
<td></td>
<td>Null</td>
</tr>
<tr>
<td>-5</td>
<td>sqlj.runtime.ExecutionContext.DBDefault</td>
<td>Default</td>
</tr>
<tr>
<td>-7</td>
<td>sqlj.runtime.ExecutionContext.DBUassigned</td>
<td>Unassigned</td>
</tr>
<tr>
<td>short-value &gt;=0</td>
<td>sqlj.runtime.ExecutionContext.DBNonNull</td>
<td>Non-null</td>
</tr>
</tbody>
</table>

For uncustomized applications, and for input, only these values are valid:

<table>
<thead>
<tr>
<th>Indicator value</th>
<th>Equivalent constant</th>
<th>Meaning of value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>sqlj.runtime.ExecutionContext.DBNull</td>
<td>Null</td>
</tr>
<tr>
<td>-7 &lt;= short-value &lt; -1</td>
<td></td>
<td>Null</td>
</tr>
<tr>
<td>0</td>
<td>sqlj.runtime.ExecutionContext.DBNonNull</td>
<td>Non-null</td>
</tr>
<tr>
<td>short-value &gt;0</td>
<td></td>
<td>Non-null</td>
</tr>
</tbody>
</table>

For customized or uncustomized applications, and for output, SQLJ sets one of these values:

<table>
<thead>
<tr>
<th>Indicator value</th>
<th>Equivalent constant</th>
<th>Meaning of value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>sqlj.runtime.ExecutionContext.DBNull</td>
<td>Retrieved value is SQL NULL</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Retrieved value is not SQL NULL</td>
</tr>
</tbody>
</table>

**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.
- Indicator variables are required in the following cases:
  - For input, when a Java primitive type is used to assign the NULL value to a table column.
  - For output, when a Java primitive type is used for a host variable, and the source column can return NULL values.
    If an SQL NULL value is returned, and no indicator variable is defined, an SQLException is thrown.

Indicator variables are not required for input or output of a Java null value as an SQL NULL, if the data type of the host variable is:
- The data type of a Java class
- A custom database type that the driver supports

• , ... variable-n

- For output, indicator variables are valid in the following types of statements:
  - CALL statement with OUT or INOUT parameters
  - FETCH positioned-iterator INTO variable-1, ..., variable-n
  - SELECT column-1, ..., column-n INTO variable-1, ..., variable-n

Related concepts:
"Variables in SQLJ applications" on page 135

**SQLJ implements-clause**

The implements clause derives one or more classes from a Java interface.

**Syntax**

```plaintext
implement<interface-element>:

interface-element:

sqlj.runtime.ForUpdate

sqlj.runtime.Scrollable

user-specified-interface-class
```

**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 SQL.

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.

Related tasks:
"Using scrollable iterators in an SQLJ application" on page 157
"Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 141

**SQLJ with-clause**

The with clause specifies a set of one or more attributes for an iterator or a
connection context.

**Syntax**
with-element:

- holdability=true
- holdability=false
- sensitivity=sqlj.runtime.ResultSetIterator.ASENSITIVE
- sensitivity=sqlj.runtime.ResultSetIterator.INSENSITIVE
- sensitivity=sqlj.runtime.ResultSetIterator.SENSITIVE
- dynamic=false
- dynamic=true
- updateColumns="column-name"
- Java-ID=Java-constant-expression
- dataSource="logical-datasource-name"

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 sqlj.runtime.ResultSetIterator.INSENSITIVE, sqlj.runtime.ResultSetIterator.SENSITIVE, or sqlj.runtime.ResultSetIterator.ASENSITIVE. The default is sqlj.runtime.ResultSetIterator.ASENSITIVE.

For connections to IBM Informix, only sqlj.runtime.ResultSetIterator.INSENSITIVE is supported.

dynamic

For an iterator that is defined with sensitivity=sqlj.runtime.ResultSetIterator.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 on Linux, UNIX, and Windows systems 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.

**Related concepts:**
- “SQLJ and JDBC in the same application” on page 166

**Related tasks:**
- “Using scrollable iterators in an SQLJ application” on page 157
- “Performing positioned UPDATE and DELETE operations in an SQLJ application” on page 141
- “Connecting to a data source using SQLJ” on page 127

### SQLJ connection-declaration-clause

The connection declaration clause declares a connection to a data source in an SQLJ application program.

#### Syntax

```
Java-modifiers context Java-class-name implements-clause with-clause
```

#### 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.

Related tasks:
- "Connecting to a data source using SQLJ" on page 127

Related reference:
- "SQLJ with-clause" on page 348
- "SQLJ implements-clause" on page 348

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

```
Java-modifiers iterator Java-class-name implements-clause with-clause

(positioned-iterator-column-declarations)
(named-iterator-column-declarations)
```

positioned-iterator-column declarations:

```
Java-data-type
```

named-iterator-column-declarations:

```
Java-data-type Java-ID
```
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.

Related concepts:
"Data retrieval in SQLJ applications” on page 151

Related tasks:
"Using scrollable iterators in an SQLJ application” on page 157
"Using a positioned iterator in an SQLJ application” on page 153
"Using a named iterator in an SQLJ application” on page 151

Related reference:
"SQLJ with-clause” on page 348
"SQLJ implements-clause” on page 348
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

```
context-clause
►►
statement-clause
assignment-clause

```

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.

Related reference:
- “SQLJ statement-clause” on page 354
- “SQLJ context-clause”
- “SQLJ assignment-clause” on page 357

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

```
[connection-context
execution-context
connection-context,
execution-context
connection-context
execution-context

```

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.

Related tasks:
SQLJ statement-clause

A statement clause contains an SQL statement or a SET TRANSACTION clause.

Syntax

```
{(SQL-statement
  SET-TRANSACTION-clause)}
```

Description

**SQL-statement**

You can include SQL statements in Table 90 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 90. Valid SQL statements in an SQLJ statement clause

<table>
<thead>
<tr>
<th>Statement</th>
<th>Applicable data sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER DATABASE</td>
<td>1 2</td>
</tr>
<tr>
<td>ALTER FUNCTION</td>
<td>1 2 3</td>
</tr>
<tr>
<td>ALTER INDEX</td>
<td>1 2 3</td>
</tr>
<tr>
<td>ALTER PROCEDURE</td>
<td>1 2 3</td>
</tr>
<tr>
<td>ALTER STOGROUP</td>
<td>1 2</td>
</tr>
<tr>
<td>ALTER TABLE</td>
<td>1 2 3</td>
</tr>
<tr>
<td>ALTER TABLESPACE</td>
<td>1 2</td>
</tr>
<tr>
<td>CALL</td>
<td>1 2 3</td>
</tr>
<tr>
<td>COMMENT ON</td>
<td>1 2</td>
</tr>
<tr>
<td>COMMIT</td>
<td>1 2 3</td>
</tr>
<tr>
<td>Compound SQL (BEGIN ATOMIC...END)</td>
<td>2</td>
</tr>
<tr>
<td>CREATE ALIAS</td>
<td>1 2</td>
</tr>
<tr>
<td>CREATE DATABASE</td>
<td>1 2 3</td>
</tr>
<tr>
<td>CREATE DISTINCT TYPE</td>
<td>1 2 3</td>
</tr>
<tr>
<td>CREATE FUNCTION</td>
<td>1 2 3</td>
</tr>
<tr>
<td>CREATE GLOBAL TEMPORARY TABLE</td>
<td>1 2</td>
</tr>
<tr>
<td>CREATE TEMP TABLE</td>
<td>3</td>
</tr>
<tr>
<td>CREATE INDEX</td>
<td>1 2 3</td>
</tr>
<tr>
<td>CREATE PROCEDURE</td>
<td>1 2 3</td>
</tr>
<tr>
<td>CREATE STOGROUP</td>
<td>1 2</td>
</tr>
<tr>
<td>CREATE SYNONYM</td>
<td>1 2 3</td>
</tr>
<tr>
<td>CREATE TABLE</td>
<td>1 2 3</td>
</tr>
<tr>
<td>CREATE TABLESPACE</td>
<td>1 2</td>
</tr>
<tr>
<td>Statement</td>
<td>Applicable data sources</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>CREATE TYPE (cursor)</td>
<td>2</td>
</tr>
<tr>
<td>CREATE TRIGGER</td>
<td>1 2 3</td>
</tr>
<tr>
<td>CREATE VIEW</td>
<td>1 2 3</td>
</tr>
<tr>
<td>DECLARE GLOBAL TEMPORARY TABLE</td>
<td>1 2</td>
</tr>
<tr>
<td>DELETE</td>
<td>1 2 3</td>
</tr>
<tr>
<td>DROP ALIAS</td>
<td>1 2</td>
</tr>
<tr>
<td>DROP DATABASE</td>
<td>1 2 3 on page 356</td>
</tr>
<tr>
<td>DROP DISTINCT TYPE</td>
<td>1 2</td>
</tr>
<tr>
<td>DROP TYPE</td>
<td>3</td>
</tr>
<tr>
<td>DROP FUNCTION</td>
<td>1 2 3</td>
</tr>
<tr>
<td>DROP INDEX</td>
<td>1 2 3</td>
</tr>
<tr>
<td>DROP PACKAGE</td>
<td>1 2</td>
</tr>
<tr>
<td>DROP PROCEDURE</td>
<td>1 2 3</td>
</tr>
<tr>
<td>DROP STOGROUP</td>
<td>1 2</td>
</tr>
<tr>
<td>DROP SYNONYM</td>
<td>1 2 3</td>
</tr>
<tr>
<td>DROP TABLE</td>
<td>1 2</td>
</tr>
<tr>
<td>DROP TABLESPACE</td>
<td>1 2</td>
</tr>
<tr>
<td>DROP TRIGGER</td>
<td>1 2 3</td>
</tr>
<tr>
<td>DROP VIEW</td>
<td>1 2 3</td>
</tr>
<tr>
<td>FETCH</td>
<td>1 2 3</td>
</tr>
<tr>
<td>GRANT</td>
<td>1 2 3</td>
</tr>
<tr>
<td>INSERT</td>
<td>1 2 3</td>
</tr>
<tr>
<td>LOCK TABLE</td>
<td>1 2 3</td>
</tr>
<tr>
<td>MERGE</td>
<td>1 2</td>
</tr>
<tr>
<td>REVOKE</td>
<td>1 2 3</td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>1 2 3</td>
</tr>
<tr>
<td>SAVEPOINT</td>
<td>1 2 3</td>
</tr>
<tr>
<td>SELECT INTO</td>
<td>1 2 3</td>
</tr>
<tr>
<td>SET CURRENT APPLICATION ENCODING SCHEME</td>
<td>1</td>
</tr>
<tr>
<td>SET CURRENT DEBUG MODE</td>
<td>1</td>
</tr>
<tr>
<td>SET CURRENT DEFAULT TRANSFORM GROUP</td>
<td>2</td>
</tr>
<tr>
<td>SET CURRENT DEGREE</td>
<td>1 2</td>
</tr>
<tr>
<td>SET CURRENT EXPLAIN MODE</td>
<td>2</td>
</tr>
<tr>
<td>SET CURRENT EXPLAIN SNAPSHOT</td>
<td>2</td>
</tr>
<tr>
<td>SET CURRENT ISOLATION</td>
<td>1 2</td>
</tr>
<tr>
<td>SET CURRENT LOCALSE LC_CTYPE</td>
<td>1</td>
</tr>
<tr>
<td>SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION</td>
<td>1 2</td>
</tr>
<tr>
<td>SET CURRENT OPTIMIZATION HINT</td>
<td>1 2</td>
</tr>
<tr>
<td>SET CURRENT PACKAGE PATH</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 90. Valid SQL statements in an SQLJ statement clause (continued)

<table>
<thead>
<tr>
<th>Statement</th>
<th>Applicable data sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET CURRENT PACKAGESET (USER is not supported)</td>
<td>1, 2</td>
</tr>
<tr>
<td>SET CURRENT PRECISION</td>
<td>1, 2</td>
</tr>
<tr>
<td>SET CURRENT QUERY ACCELERATION</td>
<td>1</td>
</tr>
<tr>
<td>SET CURRENT QUERY OPTIMIZATION</td>
<td>2</td>
</tr>
<tr>
<td>SET CURRENT REFRESH AGE</td>
<td>1, 2</td>
</tr>
<tr>
<td>SET CURRENT ROUTINE VERSION</td>
<td>1</td>
</tr>
<tr>
<td>SET CURRENT RULES</td>
<td>1</td>
</tr>
<tr>
<td>SET CURRENT SCHEMA</td>
<td>2</td>
</tr>
<tr>
<td>SET CURRENT SQLID</td>
<td>1</td>
</tr>
<tr>
<td>SET PATH</td>
<td>1, 2</td>
</tr>
<tr>
<td>TRUNCATE</td>
<td>1</td>
</tr>
<tr>
<td>UPDATE</td>
<td>1, 2, 3</td>
</tr>
</tbody>
</table>

**Note:** The SQL statement applies to connections to the following data sources:
1. DB2 for z/OS
2. Db2 on Linux, UNIX, and Windows systems
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.

### Related tasks:
- “Setting the isolation level for an SQLJ transaction” on page 184

### Related reference:
- “SQLJ SET-TRANSACTION-clause”
- Statements (DB2 SQL)

### SQLJ SET-TRANSACTION-clause
The SET TRANSACTION clause sets the isolation level for the current unit of work.

### Syntax

```
SET TRANSACTION ISOLATION LEVEL ...
```

### Description

**ISOLATION LEVEL**

Specifies one of the following isolation levels:
**READ COMMITTED**
Specifies that the current isolation level is cursor stability.

**READ UNCOMMITTED**
Specifies that the current isolation level is uncommitted read.

**REPEATABLE READ**
Specifies that the current isolation level is read stability.

**SERIALIZABLE**
Specifies that the current isolation level is repeatable read.

**Usage notes**
You can execute SET TRANSACTION only at the beginning of a transaction.

**Related tasks:**
"Setting the isolation level for an SQLJ transaction" on page 184

---

**SQLJ assignment-clause**
The assignment clause assigns the result of an SQL operation to a Java variable.

**Syntax**
```
Java-ID = { fullselect [ order-by-clause ] |
optimize-for-clause |
isolation-clause |
queryno-clause |
fetch-first-clause |
iterator-conversion-clause }
```

**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

\[ \text{CAST} \text{host-expression} \]

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.

Related concepts:

“SQLJ and JDBC in the same application” on page 166

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>
<thead>
<tr>
<th>Interface name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionContext</td>
<td>Manages the SQL operations that are performed during a connection to a data source.</td>
</tr>
<tr>
<td>ForUpdate</td>
<td>Implemented by iterators that are used in a positioned UPDATE or DELETE statement.</td>
</tr>
<tr>
<td>NamedIterator</td>
<td>Implemented by iterators that are declared as named iterators.</td>
</tr>
</tbody>
</table>
Table 91. Summary of sqlj.runtime interfaces (continued)

<table>
<thead>
<tr>
<th>Interface name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>PositionedIterator</td>
<td>Implemented by iterators that are declared as positioned iterators.</td>
</tr>
<tr>
<td>ResultSetIterator</td>
<td>Implemented by all iterators to allow query results to be processed using a JDBC ResultSet.</td>
</tr>
<tr>
<td>Scrollable</td>
<td>Provides a set of methods for manipulating scrollable iterators.</td>
</tr>
</tbody>
</table>

sqlj.runtime classes

The following table summarizes the classes in sqlj.runtime.

Table 92. Summary of sqlj.runtime classes

<table>
<thead>
<tr>
<th>Class name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsciiStream</td>
<td>A class for handling an input stream whose bytes should be interpreted as ASCII.</td>
</tr>
<tr>
<td>BinaryStream</td>
<td>A class for handling an input stream whose bytes should be interpreted as binary.</td>
</tr>
<tr>
<td>CharacterStream</td>
<td>A class for handling an input stream whose bytes should be interpreted as Character.</td>
</tr>
<tr>
<td>DefaultRuntime</td>
<td>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.</td>
</tr>
<tr>
<td>ExecutionContext</td>
<td>Implemented when an SQLJ execution context is declared, to control the execution of SQL operations.</td>
</tr>
<tr>
<td>Indicator</td>
<td>Defines constants for indicator variable values.</td>
</tr>
<tr>
<td>RuntimeContext</td>
<td>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.</td>
</tr>
<tr>
<td>SQLNullException</td>
<td>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.</td>
</tr>
<tr>
<td>StreamWrapper</td>
<td>Wraps a java.io.InputStream instance.</td>
</tr>
<tr>
<td>UnicodeStream</td>
<td>A class for handling an input stream whose bytes should be interpreted as Unicode.</td>
</tr>
</tbody>
</table>

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(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 getExecutionContext()

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 CtxsetDefaultContext(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:

eндFetch
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:

```java
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:

```java
public abstract void afterLast() throws SQLException
```

Moves the iterator after the last row of the result table.

**beforeFirst()**

Format:

```java
public abstract void beforeFirst() throws SQLException
```

Moves the iterator before the first row of the result table.

**first()**

Format:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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 row, 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:
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:
```java
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:
```java
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:
```java
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:
```java
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:
```java
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:
```java
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:
```java
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.

eexecuteQuery  
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  
The 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:

```java
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:

```java
sqlj.runtime.ResultSetIterator.FETCH_FORWARD
```

Rows are fetched in a forward direction. This is the default.

```java
sqlj.runtime.ResultSetIterator.FETCH_REVERSE
```

Rows are fetched in a backward direction.

```java
sqlj.runtime.ResultSetIterator.FETCH_UNKNOWN
```

The order of fetching is unknown.

Any other input value results in an `SQLException`.

**setFetchSize**

Format:

```java
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.

```java
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.

```java
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.

When `setMaxRows` is invoked at run time on a statically executed SELECT statement, `setMaxRows` limits the maximum number of rows in the result table through IBM Data Server Driver for JDBC and SQLJ processing only. Data server optimization that limits the number of rows in the result table does not occur unless the FETCH FIRST n ROWS ONLY clause is also added to the SELECT statement. If FETCH FIRST n rows ONLY is added to the SELECT statement, and `setMaxRows(m)` is called, the maximum number of rows is the smaller of n and m. The driver discards the rest of the 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.

```java
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.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS data servers, `setQueryTimeout` is supported only if `Connection` or `DataSource` property `queryTimeoutInterruptProcessingMode` is set to `INTERRUPT_PROCESSING_MODE_CLOSE_SOCKET`.

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.

Related tasks:
"Controlling the execution of SQL statements in SQLJ" on page 170

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.


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.

Related reference:
"sqlj.runtime.UnicodeStream class" on page 371
"sqlj.runtime.CharacterStream class" on page 371
"sqlj.runtime.BinaryStream class" on page 371
"sqlj.runtime.AsciiStream class" on page 370

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:
**InputStream**

The InputStream object that SQLJ interprets as an UnicodeStream object.

**length**

The length of the InputStream object that SQLJ interprets as an UnicodeStream object.

**Related reference:**

- "sqlj.runtime.StreamWrapper class" on page 381
- "sqlj.runtime.CharacterStream class" on page 371
- "sqlj.runtime.BinaryStream class" on page 371
- "sqlj.runtime.AsciiStream class" on page 370

---

**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:

```java
javax.sql.DataSource ds =
new com.ibm.db2.jcc.DB2SimpleDataSource();
((com.ibm.db2.jcc.DB2BaseDataSource) ds).setServerName("sysmvs1.stl.ibm.com");
```

**Table 93** summarizes the IBM Data Server Driver for JDBC and SQLJ-only interfaces.

**Table 93. Summary of IBM Data Server Driver for JDBC and SQLJ-only interfaces provided by the IBM Data Server Driver for JDBC and SQLJ**

<table>
<thead>
<tr>
<th>Interface name</th>
<th>Applicable data sources</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2CallableStatement</td>
<td>1, 2</td>
<td>Extends the java.sql.CallableStatement and the com.ibm.db2.jcc.DB2PreparedStatement interfaces.</td>
</tr>
<tr>
<td>DB2Connection</td>
<td>1, 2, 3</td>
<td>Extends the java.sql.Connection interface.</td>
</tr>
<tr>
<td>DB2DatabaseMetaData</td>
<td>1, 2, 3</td>
<td>Extends the java.sql.DatabaseMetaData interface.</td>
</tr>
<tr>
<td>DB2Diagnosable</td>
<td>1, 2, 3</td>
<td>Provides a mechanism for getting DB2 diagnostics from a DB2 SQLException.</td>
</tr>
<tr>
<td>DB2JSONResultSet</td>
<td>1, 2, 3</td>
<td>Used for converting the contents of JDBC ResultSet objects to JSON documents and snippets.</td>
</tr>
<tr>
<td>DB2ParameterMetaData</td>
<td>2</td>
<td>Extends the java.sql.ParameterMetaData interface.</td>
</tr>
<tr>
<td>DB2PreparedStatement</td>
<td>1, 2, 3</td>
<td>Extends the com.ibm.db2.jcc.DB2PreparedStatement and java.sql.PreparedStatement interfaces.</td>
</tr>
<tr>
<td>DB2ResultSet</td>
<td>1, 2, 3</td>
<td>Extends the java.sql.ResultSet interface.</td>
</tr>
<tr>
<td>DB2RowID</td>
<td>1, 2</td>
<td>Used for declaring Java objects for use with the ROWID data type.</td>
</tr>
<tr>
<td>DB2Statement</td>
<td>1, 2, 3</td>
<td>Extends the java.sql.Statement interface.</td>
</tr>
<tr>
<td>DB2Struct</td>
<td>2</td>
<td>Provides methods for working with java.sql.Struct objects.</td>
</tr>
</tbody>
</table>
Table 93. Summary of IBM Data Server Driver for JDBC and SQLJ-only interfaces provided by the IBM Data Server Driver for JDBC and SQLJ (continued)

<table>
<thead>
<tr>
<th>Interface name</th>
<th>Applicable data sources</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2SystemMonitor</td>
<td>1, 2, 3</td>
<td>Used for collecting system monitoring data for a connection.</td>
</tr>
<tr>
<td>DB2TraceManagerMXBean</td>
<td>1, 2, 3</td>
<td>Provides the MBean interface for the remote trace controller.</td>
</tr>
<tr>
<td>DB2Xml</td>
<td>1, 2</td>
<td>Used for updating data in XML columns and retrieving data from XML columns.</td>
</tr>
<tr>
<td>DBBatchUpdateException</td>
<td>1, 2, 3</td>
<td>Used for retrieving error information about batch execution of statements that return automatically generated keys.</td>
</tr>
</tbody>
</table>

Note: The interface applies to connections to the following data sources:
1. DB2 for z/OS
2. Db2 on Linux, UNIX, and Windows systems
3. IBM Informix

Table 94 summarizes the IBM Data Server Driver for JDBC and SQLJ-only classes.

Table 94. Summary of IBM Data Server Driver for JDBC and SQLJ-only classes provided by the IBM Data Server Driver for JDBC and SQLJ

<table>
<thead>
<tr>
<th>Class name</th>
<th>Applicable data sources</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2Administrator</td>
<td>2</td>
<td>Instances of the DB2Administrator class are used to retrieve DB2CataloguedDatabase objects.</td>
</tr>
<tr>
<td>DB2BaseDataSource</td>
<td>1, 2, 3</td>
<td>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.</td>
</tr>
<tr>
<td>DB2Binder</td>
<td>1, 2</td>
<td>Provides the runJDBCBinder method as an alternative to the DB2Binder utility for binding IBM Data Server Driver for JDBC and SQLJ packages.</td>
</tr>
<tr>
<td>DB2BlobFileReference</td>
<td>1</td>
<td>A subclass of DB2FileReference for creating BLOB file reference variable objects.</td>
</tr>
<tr>
<td>DB2CataloguedDatabase</td>
<td>2</td>
<td>Contains methods that retrieve information about a local Db2 on Linux, UNIX, and Windows systems database.</td>
</tr>
<tr>
<td>DB2ClientRerouteServerList</td>
<td>1, 2</td>
<td>Implements the java.io.Serializable and java.naming.Referenceable interfaces.</td>
</tr>
<tr>
<td>DB2ClobFileReference</td>
<td>1</td>
<td>A subclass of DB2FileReference for creating CLOB file reference variable objects.</td>
</tr>
<tr>
<td>DB2ConnectionPoolDataSource</td>
<td>1, 2, 3</td>
<td>A factory for PooledConnection objects.</td>
</tr>
<tr>
<td>DB2DataSource</td>
<td>1, 2, 3</td>
<td>Extends the extends DB2BaseDataSource class, and implements the javax.sql.DataSource, java.io.Serializable, and java.naming.Referenceable interfaces.</td>
</tr>
<tr>
<td>DB2Driver</td>
<td>1, 2, 3</td>
<td>Extends the java.sql.Driver interface.</td>
</tr>
<tr>
<td>DB2ExceptionFormatter</td>
<td>1, 2, 3</td>
<td>Contains methods for printing diagnostic information to a stream.</td>
</tr>
</tbody>
</table>
Table 94. Summary of IBM Data Server Driver for JDBC and SQLJ-only classes provided by the IBM Data Server Driver for JDBC and SQLJ (continued)

<table>
<thead>
<tr>
<th>Class name</th>
<th>Applicable data sources</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2FileReference</td>
<td>1</td>
<td>Provides methods for inserting data into tables from file reference variables.</td>
</tr>
<tr>
<td>DB2JCCPlugin</td>
<td>2</td>
<td>The abstract class for implementation of JDBC security plug-ins.</td>
</tr>
<tr>
<td>DB2PooledConnection</td>
<td>1, 2, 3</td>
<td>Provides methods that an application server can use to switch users on a preexisting trusted connection.</td>
</tr>
<tr>
<td>DB2PoolMonitor</td>
<td>1, 2</td>
<td>Provides methods for monitoring the global transport objects pool for the connection concentrator and Sysplex workload balancing.</td>
</tr>
<tr>
<td>DB2SimpleDataSource</td>
<td>1, 2, 3</td>
<td>Extends the DatabaseDataSource class. Does not support connection pooling or distributed transactions.</td>
</tr>
<tr>
<td>DB2Sqlca</td>
<td>1, 2, 3</td>
<td>An encapsulation of the DB2 SQLCA.</td>
</tr>
<tr>
<td>DB2TraceManager</td>
<td>1, 2, 3</td>
<td>Controls the global log writer.</td>
</tr>
<tr>
<td>DB2Types</td>
<td>1</td>
<td>Defines data type constants.</td>
</tr>
<tr>
<td>DB2Version</td>
<td>1, 2, 3</td>
<td>Provides methods that return information about the IBM Data Server Driver for JDBC and SQLJ version.</td>
</tr>
<tr>
<td>DB2XADatasource</td>
<td>1, 2, 3</td>
<td>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).</td>
</tr>
<tr>
<td>DB2XmlAsBlobFileReference</td>
<td>1</td>
<td>A subclass of DB2FileReference for creating XML AS BLOB file reference variable objects.</td>
</tr>
<tr>
<td>DB2XmlAsClobFileReference</td>
<td>1</td>
<td>A subclass of DB2FileReference for creating XML AS CLOB file reference variable objects.</td>
</tr>
<tr>
<td>DBTimestamp</td>
<td>1, 2, 3</td>
<td>A subclass of Timestamp that handles timestamp values with extra precision or time zone information.</td>
</tr>
</tbody>
</table>

Note: The class applies to connections to the following data sources:
1. DB2 for z/OS
2. Db2 on Linux, UNIX, and Windows systems
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.

The DB2BaseDataSource properties implement 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
• maxTransportObjectIdleTime
• maxTransportObjectWaitTime
• minTransportObjects
• xmlFormat

A setXXX method has this form:
void setProperty-name(data-type property-value)

A getXXX method has this form:
data-type getProperty-name()

The Property-name property 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>
<thead>
<tr>
<th>Property name</th>
<th>Applicable data sources</th>
<th>Data type</th>
<th>Introduced in driver version</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.accessToken</td>
<td>2</td>
<td>String</td>
<td>4.24</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.accountingInterval</td>
<td>2</td>
<td>String</td>
<td>3.6</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.alternateGroupDatabaseName</td>
<td>3</td>
<td>String</td>
<td>3.66, 4.16</td>
</tr>
</tbody>
</table>

386  Application Programming Guide and Reference for Java
Table 95. DB2BaseDataSource properties and their data types (continued)

<table>
<thead>
<tr>
<th>Property name</th>
<th>Applicable data sources</th>
<th>Data type</th>
<th>Introduced in driver version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.alternateGroupPortNumber</td>
<td>0</td>
<td>String</td>
<td>3.66, 4.16</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.alternateGroupServerName</td>
<td>0</td>
<td>String</td>
<td>3.63, 4.13</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.affinityFailbackInterval</td>
<td>0</td>
<td>int</td>
<td>3.58, 4.8</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.allowNextOnExhaustedResultSet</td>
<td>0</td>
<td>int</td>
<td>3.51, 4.1</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.allowNullResultSetForExecuteQuery</td>
<td>0</td>
<td>int</td>
<td>3.59, 4.9</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.apiKey (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity and dash DB only)</td>
<td>0</td>
<td>String</td>
<td>4.24</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.atomicMultiRowInsert</td>
<td>0</td>
<td>int</td>
<td>3.57, 4.7</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.blockingReadConnectionTimeout</td>
<td>0</td>
<td>int</td>
<td>2.8</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.charOutputSize</td>
<td>2</td>
<td>short</td>
<td>2.10</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientAccountingInformation</td>
<td>0, 2</td>
<td>String</td>
<td>1.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientApplicationInformation</td>
<td>0, 2</td>
<td>String</td>
<td>1.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientDebugInfo (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>0, 2</td>
<td>String</td>
<td>3.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientProgramId</td>
<td>0, 2</td>
<td>String</td>
<td>2.3</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientProgramName (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>0, 2</td>
<td>String</td>
<td>2.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteAlternateServerName</td>
<td>0</td>
<td>String</td>
<td>3.4</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteAlternatePortNumber</td>
<td>0</td>
<td>String</td>
<td>3.4</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteServerListJNDIContext</td>
<td>0</td>
<td>javax.naming.Context</td>
<td>3.3</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteServerListJNDIName</td>
<td>0</td>
<td>String</td>
<td>2.1</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientUser (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DR2 for z/OS only)</td>
<td>0</td>
<td>String</td>
<td>1.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientWorkstation (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DR2 for z/OS only)</td>
<td>0</td>
<td>String</td>
<td>1.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.commandTimeout (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>0</td>
<td>int</td>
<td>3.64, 4.14</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.connectionCloseWithinFlightTransaction</td>
<td>0</td>
<td>String</td>
<td>3.59, 4.9</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.concurrentAccessResolution</td>
<td>0, 2</td>
<td>int</td>
<td>3.53, 4.3</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.connectNode</td>
<td>0</td>
<td>int</td>
<td>3.4</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.connectionTimeout (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>0</td>
<td>int</td>
<td>3.64, 4.14</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentDegree</td>
<td>0, 2</td>
<td>String</td>
<td>3.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentExplainMode</td>
<td>0, 2</td>
<td>String</td>
<td>2.6</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentExplainSnapshot</td>
<td>0</td>
<td>String</td>
<td>2.6</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentFunctionPath</td>
<td>0, 2</td>
<td>String</td>
<td>1.3</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentLocaleLCType</td>
<td>0</td>
<td>String</td>
<td>3.64, 4.14</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentLockTimeout</td>
<td>0</td>
<td>int</td>
<td>2.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentMaintainedTableTypesForOptimization</td>
<td>0, 2</td>
<td>String</td>
<td>2.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentPackagePath</td>
<td>0, 2</td>
<td>String</td>
<td>1.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentPackageSet</td>
<td>0, 2</td>
<td>String</td>
<td>1.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentQueryOptimization</td>
<td>0</td>
<td>int</td>
<td>2.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentRefreshAge</td>
<td>0, 2</td>
<td>long</td>
<td>2.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentSchema</td>
<td>0, 2</td>
<td>String</td>
<td>1.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.cursorSensitivity</td>
<td>0</td>
<td>int</td>
<td>1.5</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentSQLID</td>
<td>0</td>
<td>String</td>
<td>1.3</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.databaseName</td>
<td>0</td>
<td>String</td>
<td>1.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.dateFormat</td>
<td>0, 2</td>
<td>int</td>
<td>3.3</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.decimalRoundingMode</td>
<td>0</td>
<td>int</td>
<td>3.4</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.decimalSeparator</td>
<td>0</td>
<td>int</td>
<td>3.53, 4.3</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.decimalStringFormat</td>
<td>0</td>
<td>int</td>
<td>3.8</td>
</tr>
<tr>
<td>Property name</td>
<td>Applicable data sources</td>
<td>Data type</td>
<td>Introduced in driver version</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------</td>
<td>-----------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.defaultIsolationLevel</td>
<td>1</td>
<td>int</td>
<td>3.4</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.deferPrepares</td>
<td>1</td>
<td>Boolean</td>
<td>1.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.description</td>
<td>1</td>
<td>String</td>
<td>1.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.downgradeHoldCursorsUnderXa</td>
<td>1</td>
<td>Boolean</td>
<td>3.3</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.driverType</td>
<td>1</td>
<td>int</td>
<td>1.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.dumpPool</td>
<td>1</td>
<td>int</td>
<td>3.52, 4.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.dumpPoolStatisticsOnSchedule</td>
<td>1</td>
<td>int</td>
<td>3.52, 4.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.dumpPoolStatisticsOnScheduleFile</td>
<td>1</td>
<td>String</td>
<td>3.52, 4.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableAlternateGroupSeamlessACR</td>
<td>1</td>
<td>Boolean</td>
<td>3.66, 4.16</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableClientAffinitiesList</td>
<td>1</td>
<td>int</td>
<td>3.51, 4.1</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableExtendedIndicators</td>
<td>2</td>
<td>int</td>
<td>3.59, 4.9</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableNamedParameterMarkers</td>
<td>1</td>
<td>int</td>
<td>3.57, 4.7</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableConnectionConcentrator (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>2</td>
<td>Boolean</td>
<td>2.6</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableMultiRowInsertSupport</td>
<td>0</td>
<td>Boolean</td>
<td>3.58, 4.8</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableRowsetSupport</td>
<td>0</td>
<td>Boolean</td>
<td>3.7</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableSeamlessFailover</td>
<td>1</td>
<td>int</td>
<td>3.51, 4.1</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableSuspendWLB (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>1</td>
<td>Boolean</td>
<td>2.6</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.encryptionAlgorithm</td>
<td>2</td>
<td>int</td>
<td>2.11</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableExtendedDescribe</td>
<td>2</td>
<td>int</td>
<td>3.63, 4.13</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableT2zosLBF (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS only)</td>
<td>2</td>
<td>int</td>
<td>3.61, 4.11</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableT2zosLBFSResultSets (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS only)</td>
<td>2</td>
<td>int</td>
<td>3.63, 4.13</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableTimeoutForCursors</td>
<td>0</td>
<td>int</td>
<td>3.66, 4.16</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.extendedTableInfo</td>
<td>0</td>
<td>int</td>
<td>3.68, 4.18</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.fetchSize</td>
<td>1</td>
<td>int</td>
<td>3.53, 4.3</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.floatingPointStringFormat</td>
<td>0</td>
<td>int</td>
<td>3.58, 4.8</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.fullyMaterializeInputStreams</td>
<td>1</td>
<td>Boolean</td>
<td>2.7</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.fullyMaterializeLobData</td>
<td>1</td>
<td>Boolean</td>
<td>1.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.gssCredential</td>
<td>1</td>
<td>Object</td>
<td>1.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.implicitRollbackOption</td>
<td>0</td>
<td>int</td>
<td>3.64, 4.14</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.interruptProcessingMode (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>1</td>
<td>int</td>
<td>3.59, 4.9</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.jdbcCollection</td>
<td>2</td>
<td>String</td>
<td>1.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.keepAliveTimeOut (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>1</td>
<td>int</td>
<td>3.63, 4.13</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.keepDynamic</td>
<td>2</td>
<td>int</td>
<td>1.5</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.kerberosServerPrincipal</td>
<td>3</td>
<td>String</td>
<td>1.1</td>
</tr>
<tr>
<td>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)</td>
<td>1</td>
<td>int</td>
<td>1.4</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.logWriter</td>
<td>1</td>
<td>PrintWriter</td>
<td>1.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.maxConnCachedParamBufferSize (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS only)</td>
<td>2</td>
<td>int</td>
<td>3.63, 4.13</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.maxRetriesForClientReroute</td>
<td>0</td>
<td>int</td>
<td>2.7</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.maxStatements</td>
<td>0</td>
<td>int</td>
<td>3.63, 4.13</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.maxRowsetSize (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS only)</td>
<td>2</td>
<td>int</td>
<td>3.7</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.maxTransportObjectIdleTime</td>
<td>4</td>
<td>int</td>
<td>3.52, 4.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.maxTransportObjectWaitTime</td>
<td>4</td>
<td>int</td>
<td>3.52, 4.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.maxTransportObjects</td>
<td>2</td>
<td>int</td>
<td>2.6</td>
</tr>
</tbody>
</table>
### Table 95. `DB2BaseDataSource` properties and their data types (continued)

<table>
<thead>
<tr>
<th>Property name</th>
<th>Applicable data sources</th>
<th>Data type</th>
<th>Introduced in driver version</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.memberConnectTimeout (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>1</td>
<td>int</td>
<td>3.65, 4.15</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.minTransportObjects</td>
<td>5</td>
<td>int</td>
<td>3.52, 4.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.optimizationProfile</td>
<td>6</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.optimizationProfileToFlush</td>
<td>6</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.password</td>
<td>7</td>
<td>String</td>
<td>1.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.pgsqlProperties</td>
<td>2</td>
<td>String</td>
<td>3.52, 4.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.pkList (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity)</td>
<td>2</td>
<td>String</td>
<td>1.4</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.planName (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity only)</td>
<td>2</td>
<td>String</td>
<td>1.4</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.plugin</td>
<td>3</td>
<td>Object</td>
<td>2.8</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.pluginName</td>
<td>3</td>
<td>String</td>
<td>2.8</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.portNumber</td>
<td>1</td>
<td>int</td>
<td>1.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.progressiveStreaming</td>
<td>1</td>
<td>int</td>
<td>3.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.queryClosedImplicit</td>
<td>1</td>
<td>int</td>
<td>3.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.queryDataSize</td>
<td>1</td>
<td>int</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.queryTimeoutInterruptProcessingMode</td>
<td>1</td>
<td>int</td>
<td>3.62, 4.12</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.readOnly</td>
<td>3</td>
<td>Boolean</td>
<td>1.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.reportLongTypes</td>
<td>2</td>
<td>short</td>
<td>3.6</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.resultSetHoldability</td>
<td>1</td>
<td>int</td>
<td>1.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.resultSetHoldabilityForCatalogQueries</td>
<td>2</td>
<td>int</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.retrieveMessagesFromServerOnGetMessage</td>
<td>1</td>
<td>Boolean</td>
<td>1.1</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.retryIntervalForClientReroute</td>
<td>1</td>
<td>int</td>
<td>2.7</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.retryWithAlternativeSecurityMechanism (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>3</td>
<td>int</td>
<td>3.6</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.returnAlias</td>
<td>4</td>
<td>short</td>
<td>2.5</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.securityMechanism</td>
<td>1</td>
<td>int</td>
<td>1.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sendCharInputsUTF8</td>
<td>2</td>
<td>int</td>
<td>3.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sendDataAsIs</td>
<td>1</td>
<td>Boolean</td>
<td>3.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.serverName</td>
<td>1</td>
<td>String</td>
<td>1.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sessionTimeZone</td>
<td>2</td>
<td>String</td>
<td>3.59, 4.9</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sqljAvoidTimeStampConversion</td>
<td>3</td>
<td>Boolean</td>
<td>3.69, 4.19</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sqljEnableClassLoaderSpecificProfiles</td>
<td>2</td>
<td>Boolean</td>
<td>2.10</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.ssl (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS only)</td>
<td>2</td>
<td>String</td>
<td>3.6</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sslCertificate (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>1</td>
<td>String</td>
<td>3.69, 4.19</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sslCipherSuites (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>1</td>
<td>String</td>
<td>3.71, 4.21</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sslConnection (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>1</td>
<td>Boolean</td>
<td>3.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sslTrustStoreLocation</td>
<td>1</td>
<td>String</td>
<td>3.53, 4.3</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sslTrustStorePassword</td>
<td>1</td>
<td>String</td>
<td>3.53, 4.3</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.statemntConcentrator</td>
<td>2</td>
<td>int</td>
<td>3.57, 4.7</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.streamBufferSize</td>
<td>2</td>
<td>int</td>
<td>3.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.stripTrailingZerosForDecimalNumbers</td>
<td>1</td>
<td>int</td>
<td>3.59, 4.9</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.supportsAsynchronousXAExpiration</td>
<td>1</td>
<td>int</td>
<td>2.7</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sysSchema</td>
<td>2</td>
<td>String</td>
<td>2.5</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.timerLevelForQueryTimeout</td>
<td>1</td>
<td>int</td>
<td>3.59, 4.9</td>
</tr>
</tbody>
</table>
Table 95. DB2BaseDataSource properties and their data types (continued)

<table>
<thead>
<tr>
<th>Property name</th>
<th>Applicable data sources</th>
<th>Data type</th>
<th>Introduced in driver version</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.timeFormat</td>
<td>2</td>
<td>int</td>
<td>3.3</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.timestampFormat</td>
<td>3</td>
<td>int</td>
<td>3.6</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.timestampOutputType</td>
<td>3</td>
<td>int</td>
<td>3.59, 4.9</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.timestampPrecisionReporting</td>
<td>3</td>
<td>int</td>
<td>3.8</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.traceDirectory</td>
<td>2</td>
<td>String</td>
<td>1.5</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.traceFile</td>
<td>2</td>
<td>String</td>
<td>1.1</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.traceFileAppend</td>
<td>2</td>
<td>Boolean</td>
<td>1.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.traceFileCount</td>
<td>2</td>
<td>int</td>
<td>3.63, 4.13</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.traceFileSize</td>
<td>2</td>
<td>int</td>
<td>3.63, 4.13</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.traceLevel</td>
<td>2</td>
<td>int</td>
<td>1.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.traceOption</td>
<td>2</td>
<td>int</td>
<td>3.63, 4.13</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.useCachedCursor</td>
<td>2</td>
<td>Boolean</td>
<td>2.2</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.useIdentityValLocalForAutoGeneratedKeys</td>
<td>2</td>
<td>Boolean</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.useJDBC4ColumnNameAndLabelSemantics</td>
<td>2</td>
<td>int</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.xaNetworkOptimization</td>
<td>2</td>
<td>Boolean</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.xmlFormat</td>
<td>2</td>
<td>int</td>
<td>3.53, 4.3</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.DBANSIWARN</td>
<td>4</td>
<td>Boolean</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.DBDATE</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.DBPATH</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.DBSYSTEM</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.DTBTEMP</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.DBUTEMP</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.DELIMIDENT</td>
<td>4</td>
<td>Boolean</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.IFX_DIRECTIVES</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.IFX_EXTERNALDIRECTIVES</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.IFX_UPDESC</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.IFX_XASTDCOMPLIANCE_XAEND</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.INFORMIXOPCACHE</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.INFORMIXSTACKSIZE</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.NODEFDAC</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.OPTCOMPIND</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.OPTOFC</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.PDQPRIORITY</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.PSORT_DBTEMP</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.PSORT_NPROCS</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.STMT_CACHE</td>
<td>4</td>
<td>String</td>
<td>3.50, 4.0</td>
</tr>
</tbody>
</table>

Note: The property applies to connections to the following data sources:
1. All data servers
2. DB2 for z/OS
3. Db2 on Linux, UNIX, and Windows systems
4. IBM Informix
DB2BaseDataSource fields

The following constants are defined only for the IBM Data Server Driver for JDBC and SQLJ.

`public final static int IMPLICIT_ROLLBACK_OPTION_NOT_SET = 0`
A constant for the implicitRollbackOption property. This value indicates that a connection is not closed when a deadlock or timeout occurs. This value causes the same behavior as `IMPLICIT_ROLLBACK_OPTION_NOT_CLOSE_CONNECTION`.

`public final static int IMPLICIT_ROLLBACK_OPTION_NOT_CLOSE_CONNECTION = 1`
A constant for the implicitRollbackOption property. This value indicates that a connection is not closed when a deadlock or timeout occurs. The IBM Data Server Driver for JDBC and SQLJ returns the error code that the data server generates for a deadlock or timeout.

`public final static int IMPLICIT_ROLLBACK_OPTION_CLOSE_CONNECTION = 2`
A constant for the implicitRollbackOption property. This value indicates that a connection is closed when a deadlock or timeout occurs.

`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 runs `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 NOT_SET = 0`
The default value for properties.

`public final static int YES = 1`
The YES value for properties.

`public final static int NO = 2`
The NO value for properties.

`public final static int QUERYTIMEOUT_DISABLED = -1`
A constant for the timerLevelForQueryTimeOut property. This value indicates that Timer objects for waiting for queries to time out are not created.

`public final static int QUERYTIMEOUT_STATEMENT_LEVEL = 1`
A constant for the timerLevelForQueryTimeOut property. This value indicates that Timer objects for waiting for queries to time out are created at the Statement level.

`public final static int QUERYTIMEOUT_CONNECTION_LEVEL = 2`
A constant for the timerLevelForQueryTimeOut property. This value indicates that Timer objects for waiting for queries to time out are created at the Connection level.

`public final static int TRACE_OPTION_CIRCULAR = 1`
A constant for the traceOption property. This value indicates that the IBM Data Server Driver for JDBC and SQLJ uses circular tracing.
**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**

The format of the `getReference` method is illustrated in the following example:

```java
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.

**Related reference:**

[Properties for the IBM Data Server Driver for JDBC and SQLJ](#)

---

**DB2Binder class**

The `com.ibm.db2.jcc.DB2Binder` class provides the `runJDBCBinder` method as an alternative to the DB2Binder utility for binding IBM Data Server Driver for JDBC and SQLJ packages.

**DB2Binder methods**

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

**runJDBCBinder**

Format:

```java
public void runJDBCBinder(DB2Connection connection, java.util.Properties db2BinderOptions)
throws SQLException
```

Binds a IBM Data Server Driver for JDBC and SQLJ package.

Parameters:

- `connection`:
  A `DB2Connection` object for a connection that was established at the data server on which the packages are being bound.

- `db2BinderOptions`:
  A `java.util.Properties` object that contains key and value pairs, in which each key is the name of a DB2Binder utility option, and each value is the value to which you want to set that DB2Binder option. See "DB2Binder utility" on page 526 for a list of DB2Binder options.

  For example, suppose that `con` is a previously defined `Connection` object. Set the property values in the following way.

  ```java
  DB2Binder db2binder = new DB2Binder();
  Properties prop = new Properties();
  prop.put("action", "replace");
  prop.put("bindoptions", "DEFER(PREPARE) IMMEDIATE(NO) REOPT(NONE)");
  db2binder.runJDBCBinder((DB2Connection)con,prop);
  ```

  This method is not supported for connections to IBM Informix.
DB2BlobFileReference class

The com.ibm.db2.jcc.DB2BlobFileReference class is subclass of DB2FileReference that is used for creating BLOB file reference variable objects. This class applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS Version 9 or later.

DB2BlobFileReference constructor

The following constructor is defined only for the IBM Data Server Driver for JDBC and SQLJ.

```
public DB2BlobFileReference(String fileName)
    throws java.sql.SQLException

Constructs a DB2BlobFileReference object for a BLOB file reference variable.
```

Parameter descriptions:

```
fileName
The name of the file for the file reference variable. The name must specify the absolute path name for an existing HFS file.
```

DB2CallableStatement interface

The com.ibm.db2.jcc.DB2CallableStatement interface extends the java.sqlCallableStatement and the com.ibm.db2.jcc.DB2PreparedStatement interfaces.

DB2CallableStatement methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getDBTimestamp
Formats:

```
public DBTimestamp getDBTimestamp(int parameterIndex)
    throws SQLException
public DBTimestamp getDBTimestamp(String parameterName)
    throws SQLException
```

Returns the value of a TIMESTAMP OUT or INOUT parameter as a DBTimestamp object. If the value of the parameter is NULL, the returned value is null.

Parameters:

```
parameterIndex
The number of the parameter whose value is being retrieved.
```

```
parameterName
The name of the parameter whose value is being retrieved.
```

This method is not supported for connections to IBM Informix.

getJccArrayAtName
Format:

```
public java.sql.Array getJccArrayAtName(String parameterMarkerName)
    throws java.sql.SQLException
```
Retrieves an ARRAY value that is designated by a named parameter marker as a java.sql.Array value.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

**parameterMarkerName**

The name of the parameter marker for which a value is retrieved.

### getJccBigDecimalAtName

**Format:**

```java
public java.math.BigDecimal getJccBigDecimalAtName(String parameterMarkerName)
throws java.sql.SQLException
```

```
public java.math.BigDecimal getJccBigDecimalAtName(String parameterMarkerName,
int scale)
throws java.sql.SQLException
```

Retrieves a DECIMAL value that is designated by a named parameter marker as a java.math.BigDecimal value.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

**parameterMarkerName**

The name of the parameter marker for which a value is retrieved.

**scale**

The scale of the parameter value that is retrieved.

### getJccBlobAtName

**Formats:**

```java
public java.sql.Blob getJccBlobAtName(String parameterMarkerName)
throws java.sql.SQLException
```

Retrieves a BLOB value that is designated by a named parameter marker as a java.sql.Blob value.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

**parameterMarkerName**

The name of the parameter marker for which a value is retrieved.

### getJccBooleanAtName

**Format:**

```java
public boolean getJccBooleanAtName(String parameterMarkerName)
throws java.sql.SQLException
```

Retrieves a BIT or BOOLEAN value that is designated by a named parameter marker as a boolean value.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

**parameterMarkerName**

The name of the parameter marker for which a value is retrieved.
getJccByteAtName

Format:
public byte getJccByteAtName(String parameterMarkerName)
 throws java.sql.SQLException

Retrieves a TINYINT value that is designated by a named parameter marker as a byte value.
This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker for which a value is retrieved.

getJccBytesAtName

Format:
public byte[] getJccBytesAtName(String parameterMarkerName)
 throws java.sql.SQLException

Retrieves a BINARY or VARBINARY value that is designated by a named parameter marker as an array of byte values.
This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker for which a value is retrieved.

getJccClobAtName

Format:
public java.sql.Blob getJccClobAtName(String parameterMarkerName)
 throws java.sql.SQLException

Retrieves a CLOB value that is designated by a named parameter marker as a java.sql.Clob value.
This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker for which a value is retrieved.

getJccDateAtName

Formats:
public java.sql.Date getJccDateAtName(String parameterMarkerName)
 throws java.sql.SQLException
public java.sql.Date getJccDateAtName(String parameterMarkerName, java.util.Calendar cal)
 throws java.sql.SQLException

Retrieves a DATE value that is designated by a named parameter marker as a java.sql.Date value.
This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:
**parameterMarkerName**
The name of the parameter marker for which a value is retrieved.

**cal**
The java.util.Calendar object that the IBM Data Server Driver for JDBC and SQLJ uses to construct the date.

**getJccDoubleAtName**
Format:
```java
public double getJccDoubleAtName(String parameterMarkerName)
   throws java.sql.SQLException
```
Retrieves a DOUBLE value that is designated by a named parameter marker as a double value.
This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).
Parameters:

**parameterMarkerName**
The name of the parameter marker for which a value is retrieved.

**getJccFloatAtName**
Format:
```java
public double getJccFloatAtName(String parameterMarkerName)
   throws java.sql.SQLException
```
Retrieves a FLOAT value that is designated by a named parameter marker as a double value.
This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).
Parameters:

**parameterMarkerName**
The name of the parameter marker for which a value is retrieved.

**getJccIntAtName**
Format:
```java
public int getJccIntAtName(String parameterMarkerName)
   throws java.sql.SQLException
```
Retrieves a INTEGER value that is designated by a named parameter marker as a int value.
This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).
Parameters:

**parameterMarkerName**
The name of the parameter marker for which a value is retrieved.

**getJccLongAtName**
Format:
```java
public long getJccLongAtName(String parameterMarkerName)
   throws java.sql.SQLException
```
Retrieves a BIGINT value that is designated by a named parameter marker as a long value.
This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

*parameterMarkerName*

The name of the parameter marker for which a value is retrieved.

getJccObjectAtName

Formats:

```java
public java.sql.Object getJccObjectAtName(String parameterMarkerName)
    throws java.sql.SQLException
```

```java
public java.sql.Object getJccObjectAtName(String parameterMarkerName,
    Map map)
    throws java.sql.SQLException
```

Retrieves a value that is designated by a named parameter marker as a java.sql.Object value.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

*parameterMarkerName*

The name of the parameter marker for which a value is retrieved.

map

The mapping from SQL type names to Java classes.

getJccRowIdAtName

Format:

```java
public java.sql.RowId getJccRowIdAtName(String parameterMarkerName)
    throws java.sql.SQLException
```

Retrieves a ROWID value that is designated by a named parameter marker as a java.sql.RowId value.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

This method requires the IBM Data Server Driver for JDBC and SQLJ Version 4.8 or later.

Parameters:

*parameterMarkerName*

The name of the parameter marker for which a value is retrieved.

getJccShortAtName

Format:

```java
public short getJccShortAtName(String parameterMarkerName)
    throws java.sql.SQLException
```

Retrieves a SMALLINT value that is designated by a named parameter marker as a short value.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

*parameterMarkerName*

The name of the parameter marker for which a value is retrieved.
getJccSQLXMLAtName
Format:
public java.sql.SQLXML getJccSQLXMLAtName(String parameterMarkerName)
    throws java.sql.SQLException

Retrieves a SQLXML value that is designated by a named parameter marker as
a java.sql.SQLXML value.
This method can be called only if the enableNamedParameterMarkers property
is set to DB2BaseDataSource.YES (1).
This method requires the IBM Data Server Driver for JDBC and SQLJ Version
4.8 or later.
Parameters:
    parameterMarkerName
    The name of the parameter marker for which a value is retrieved.

getJccStringAtName
Format:
public java.lang.String getJccStringAtName(String parameterMarkerName)
    throws java.sql.SQLException

Retrieves a CHAR, VARCHAR, or LONGVARCHAR value that is designated
by a named parameter marker as a java.lang.String value.
This method can be called only if the enableNamedParameterMarkers property
is set to DB2BaseDataSource.YES (1).
Parameters:
    parameterMarkerName
    The name of the parameter marker for which a value is retrieved.

getJccTimeAtName
Formats:
public java.sql.Time getJccTimeAtName(String parameterMarkerName)
    throws java.sql.SQLException
public java.sql.Time getJccTimeAtName(String parameterMarkerName,
    java.util.Calendar cal)
    throws java.sql.SQLException

Retrieves a TIME value that is designated by a named parameter marker as a
java.sql.Time value.
This method can be called only if the enableNamedParameterMarkers property
is set to DB2BaseDataSource.YES (1).
Parameters:
    parameterMarkerName
    The name of the parameter marker for which a value is retrieved.
    cal
    The java.util.Calendar object that the IBM Data Server Driver for JDBC
    and SQLJ uses to construct the time.

getJccTimestampAtName
Formats:
public java.sql.Timestamp getJccTimestampAtName(String parameterMarkerName)
    throws java.sql.SQLException
public java.sql.Timestamp getJccTimestampAtName(String parameterMarkerName, java.util.Calendar cal)
    throws java.sql.SQLException

Retrieves a TIMESTAMP value that is designated by a named parameter marker as a java.sql.Timestamp value.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

* **parameterMarkerName**
  The name of the parameter marker for which a value is retrieved.

* **cal**
  The java.util.Calendar object that the IBM Data Server Driver for JDBC and SQLJ uses to construct the timestamp.

**registerJccOutParameterAtName**

Formats:

public void registerJccOutParameterAtName(String parameterMarkerName, int sqlType)
    throws java.sql.SQLException
public void registerJccOutParameterAtName(String parameterMarkerName, int sqlType, int scale)
    throws java.sql.SQLException
public void registerJccOutParameterAtName(String parameterMarkerName, int sqlType, String typeName)
    throws java.sql.SQLException

Registers an OUT parameter that is identified by parameterMarkerName as the JDBC type sqlType.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

* **parameterMarkerName**
  The name of the parameter marker for the parameter that is to be registered.

* **sqlType**
  The JDBC type code, as defined in java.sql.Types, of the parameter that is to be registered.

* **scale**
  The scale of the parameter that is to be registered. This parameter applies only to this case:
  - If sqlType is java.sql.Types.DECIMAL or java.sql.Types.NUMERIC, scale is the number of digits to the right of the decimal point.

* **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 parameter that is to be registered.

**setDBTimestamp**

Format:
public void setDBTimestamp(String parameterName,
   DBTimestamp timestamp)
   throws java.sql.SQLException

Assigns a DBTimestamp value to an IN or INOUT parameter.
Parameters:

   parameterName
       The name of the parameter to which a DBTimestamp variable value is
       assigned.

   timestamp
       The DBTimestamp value that is assigned to the parameter.

This method is not supported for connections to IBM Informix.

**setJccXXXAtName methods**
These methods are inherited from DB2PreparedStatement.

**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.

**Related concepts:**
Chapter 11, “Java client support for high availability on IBM data servers,” on page 569

**DB2ClobFileReference class**

The `com.ibm.db2.jcc.DB2ClobFileReference` class is subclass of `DB2FileReference` that is used for creating CLOB file reference variable objects. This class applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS Version 9 or later.

**DB2ClobFileReference constructor**

The following constructor is defined only for the IBM Data Server Driver for JDBC and SQLJ.

**DB2ClobFileReference**

Format:
public DB2ClobFileReference(String fileName,
int fileCcsid)
throws java.sql.SQLException
public DB2ClobFileReference(String fileName,
String fileEncoding)
throws java.sql.SQLException

Constructs a `DB2ClobFileReference` object for a CLOB file reference variable.

**Parameter descriptions:**

**fileName**

The name of the file for the file reference variable. The name must specify the absolute path name for an existing HFS file.

**fileCcsid**

The CCSID of the data in the file for the file reference variable.

**fileEncoding**

The encoding scheme of the data in the file for the file reference variable.

**DB2Connection interface**

The `com.ibm.db2.jcc.DB2Connection` interface extends the `java.sql.Connection` interface.
DB2Connection implements the java.sql.Wrapper interface.

**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.

**deregisterDB2XmlObject**

Formats:
```
public void deregisterDB2XmlObject(String sqlIdSchema,
String sqlIdName)
throws SQLException
```

Removes a previously registered XML schema from the data source.

Parameter descriptions:
SQLIdSchema

The SQL schema name for the XML schema. sqlIdSchema is a String value with a maximum length of 128 bytes. The value of sqlIdSchema must be the string 'SYSXSR' or null. If the value of sqlIdSchema is null, the database system uses the string 'SYSXSR'.

SQLIdName

The SQL name for the XML schema. sqlIdName is a String value with a maximum length of 128 bytes. The value of sqlIdName must conform to the rules for an SQL identifier and cannot be null.

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 on Linux, UNIX, and Windows systems 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:**

```java
String getDB2Correlator()
  throws java.sql.SQLException
```

Returns the value of the `crtkn` (correlation token) instance variable that DRDA sends with the ACCRDB command. The correlation token uniquely identifies a logical connection to a server.

### getDB2CurrentPackagePath

**Format:**

```java
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:**

```java
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.

### getDB2ProgressiveStreaming

**Format:**

```java
public int getDB2ProgressiveStreaming()
  throws java.sql.SQLException
```

Returns the current progressive streaming setting for the connection.

The returned value depends on whether the data source supports progressive streaming, how the progressiveStreaming property is set, and whether `DB2Connection.setProgressiveStreaming` was called:

- If the data source does not support progressive streaming, 2 (NO) is always returned, regardless of the progressiveStreaming property setting.
- If the data source supports progressive streaming, and `DB2Connection.setProgressiveStreaming` was called, the returned value is the value that `DB2Connection.setProgressiveStreaming` set.
- If the data source supports progressive streaming, and `DB2Connection.setProgressiveStreaming` was not called, the returned value is 2 (NO) if progressiveStreaming was set to `DB2BaseDataSource.NO`. If progressiveStreaming was set to `DB2BaseDataSource.YES` or was not set, the returned value is 1 (YES).
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 on Linux, UNIX, and Windows systems.

getDBProgressiveStreaming
Format:
public int getDB2ProgressiveStreaming()
    throws java.sql.SQLException

Returns the current progressive streaming setting for the connection.
The returned value depends on whether the data source supports progressive streaming, how the progressiveStreaming property is set, and whether DB2Connection.setProgressiveStreaming was called:
• If the data source does not support progressive streaming, 2 (NO) is always returned, regardless of the progressiveStreaming property setting.
• If the data source supports progressive streaming, and DB2Connection.setProgressiveStreaming was called, the returned value is the value that DB2Connection.setProgressiveStreaming set.
• If the data source supports progressive streaming, and DB2Connection.setProgressiveStreaming was not called, the returned value
is 2 (NO) if progressiveStreaming was set to DB2BaseDataSource.NO. If progressiveStreaming was set to DB2BaseDataSource.YES or was not set, the returned value is 1 (YES).

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.

installDB2JavaStoredProcedure
Format:
public void DB2Connection.installDB2JavaStoredProcedure(
java.io.InputStream jarFile,
int jarFileLength,
String jarId)
throws java.sql.SQLException

Invokes the SQLJ.DB2_INSTALL_JAR stored procedure on a DB2 for z/OS server to create a new definition of a JAR file in the catalog for that server.

Parameter descriptions:
jarFile
 The contents of the JAR file that is to be defined to the server.

jarFileLength
 The length of the JAR file that is to be defined to the server.

jarId
 The name of the JAR in the database, in the form schema.JAR-id or JAR-id.
This is the name that you use when you refer to the JAR in SQL statements. If you omit schema, the database system uses the SQL authorization ID that is in the CURRENT SCHEMA special register. The owner of the JAR is the authorization ID in the CURRENT SQLID special register.

This method does not apply to connections to IBM Informix data sources.

**isDB2Alive**

Format:

```java
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.

**isDBValid**

Format:

```java
public boolean DB2Connection.isDBValid(boolean throwException, int timeout)
throws java.sql.SQLException
```

Returns true if the connection has not been closed and is still valid. Returns false otherwise.

Parameter descriptions:

**throwException**

Specifies whether isDBValid throws an SQLException if the connection is not valid. Possible values are:

- `true` isDBValid throws an SQLException if the connection is not valid.
- `false` isDBValid throws an SQLException only if the value of `timeout` is not valid.

**timeout**

The time in seconds to wait for completion of a database operation that the driver submits. The driver submits that database operation to the data source to validate the connection. If the timeout period expires before the database operation completes, isDBValid returns false. A value of 0 indicates that there is no timeout period for the database operation.

For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, isDBValid throws an SQLException if the value of `timeout` is less than 0.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity, isDBValid throws an SQLException if the value of `timeout` is not equal to 0.

This method does not apply to connections to IBM Informix data sources.

**reconfigureDB2Connection**

Format:

```java
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 on Linux, UNIX, and Windows systems 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.

**registerDB2XmlSchema**

Formats:

```java
public void registerDB2XmlSchema(String[] sqlIdSchema,  
  String[] sqlIdName,  
  String[] xmlSchemaLocations,  
  InputStream[] xmlSchemaDocuments,  
  int[] xmlSchemaDocumentsLengths,  
  InputStream[] xmlSchemaDocumentsProperties,  
  int[] xmlSchemaDocumentsPropertiesLengths,  
  InputStream xmlSchemaProperties,  
  int xmlSchemaPropertiesLength,  
  boolean isUsedForShredding)
  throws SQLException
```

```java
public void registerDB2XmlSchema(String[] sqlIdSchema,  
  String[] sqlIdName,  
  String[] xmlSchemaLocations,  
  String[] xmlSchemaDocuments,  
  String xmlSchemaDocumentsProperties,  
  String xmlSchemaProperties,  
  boolean isUsedForShredding)
  throws SQLException
```

Registers an XML schema with one or more XML schema documents. If multiple XML schema documents are processed with one call to `registerDB2XmlSchema`, those documents are processed as part of a single transaction.

The first form of `registerDB2XmlSchema` is for XML schema documents that are read from an input stream. The second form of `registerDB2XmlSchema` is for XML schema documents that are read from strings.

Parameter descriptions:

**sqlIdSchema**

The SQL schema name for the XML schema. Only the first element of the `sqlIdSchema` array is used. `sqlIdSchema` is a String value with a maximum length of 128 bytes. The value of `sqlIdSchema` must be the string 'SYSXSR' or null. If the value of `sqlIdSchema` is null, the database system uses the string 'SYSXSR'.

**sqlIdName**

The SQL name for the XML schema. Only the first element of the `sqlIdName` array is used. `sqlIdName` is a String value with a maximum length of 128 bytes. The value of `sqlIdName` must conform to the rules for an SQL identifier and cannot be null.

**xmlSchemaLocations**

XML schema locations for the primary XML schema documents of the
schemas that are being registered. XML schema location values are normally in URI format. Each xmlSchemaLocations value is a String value with a maximum length of 1000 bytes. The value is used only to match the information that is specified in the XML schema document that references this document. The database system does no validation of the format, and no attempt is made to resolve the URI.

**xmlSchemaDocuments**

The content of the primary XML schema documents. Each xmlSchemaDocuments value is a String or InputStream value with a maximum length of 30 MB. The values must not be null.

**xmlSchemaDocumentsLengths**

The lengths of the XML schema documents in the xmlSchemaDocuments parameter, if the first form of registerDB2XmlSchema is used. Each xmlSchemaDocumentsLengths value is an int value.

**xmlSchemaDocumentsProperties**

Contains properties of the primary XML schema documents, such as properties that are used by an external XML schema versioning system. The database system does no validation of the contents of these values. They are stored in the XSR table for retrieval and used in other tools and XML schema repository implementations. Each xmlSchemaDocumentsProperties value is a String or InputStream value with a maximum length of 5 MB. A value is null if there are no properties to be passed.

**xmlSchemaDocumentsPropertiesLengths**

The lengths of the XML schema properties in the xmlSchemaDocumentsProperties parameter, if the first form of registerDB2XmlSchema is used. Each xmlSchemaDocumentsPropertiesLengths value is an int value.

**xmlSchemaProperties**

Contains properties of the entire XML schema, such as properties that are used by an external XML schema versioning system. The database system does no validation of the contents of this value. They are stored in the XSR table for retrieval and used in other tools and XML schema repository implementations. The xmlSchemaProperties value is a String or InputStream value with a maximum length of 5 MB. The value is null if there are no properties to be passed.

**xmlSchemaPropertiesLengths**

The length of the XML schema property in the xmlSchemaProperties parameter, if the first form of registerDB2XmlSchema is used. The xmlSchemaPropertiesLengths value is an int value.

**isUsedForShredding**

Indicates whether there are annotations in the schema that are to be used for XML decomposition. isUsedForShredding is a boolean value.

The isUsedForShredding parameter value must be false for connections to DB2 for z/OS data sources.

This method does not apply to connections to IBM Informix data sources.

**setDBConcurrentAccessResolution**

Format:

```java
public void setDBConcurrentAccessResolution(int concurrentAccessResolution)
throws java.sql.SQLException
```

Chapter 7. JDBC and SQLJ reference information
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 on Linux, UNIX, and Windows systems.

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.

**setDBProgressiveStreaming**
Format:
```
public void setDB2ProgressiveStreaming(int newSetting)
throws java.sql.SQLException
```

Sets the progressive streaming setting for all ResultSet objects that are created on the connection.

Parameter descriptions:

**newSetting**
The new progressive streaming setting. Possible values are:

- **DB2BaseDataSource.YES** (1)
  Enable progressive streaming. If the data source does not support progressive streaming, this setting has no effect.

- **DB2BaseDataSource.NO** (2)
  Disable progressive streaming.

**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 on Linux, UNIX, and Windows systems 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.

**removeDB2JavaStoredProcedure**

Format:

```java
public void DB2Connection.removeDB2JavaStoredProcedure(
        String jarId)
    throws java.sql.SQLException
```

Invokes the SQLJ.DB2_REMOVE_JAR stored procedure on a DB2 for z/OS server to delete the definition of a JAR file from the catalog for that server.

Parameter descriptions:

*jarId*

The name of the JAR in the database, in the form *schema.JAR-id* or *JAR-id*. This is the name that you use when you refer to the JAR in SQL statements. If you omit *schema*, the database system uses the SQL authorization ID that is in the CURRENT SCHEMA special register.

This method does not apply to connections to IBM Informix data sources.

**replaceDB2JavaStoredProcedure**

Format:
public void DB2Connection.replaceDB2JavaStoredProcedure(
    java.io.InputStream jarFile,
    int jarFileLength,
    String jarId)
throws java.sql.SQLException

Invokes the SQLJ.DB2_REPLACE_JAR stored procedure on a DB2 for z/OS server to replace the definition of a JAR file in the catalog for that server.

Parameter descriptions:

jarFile
The contents of the JAR file that is to be replaced on the server.

jarFileLength
The length of the JAR file that is to be replace on the server.

jarId
The name of the JAR in the database, in the form schema.JAR-id or JAR-id. This is the name that you use when you refer to the JAR in SQL statements. If you omit schema, the database system uses the SQL authorization ID that is in the CURRENT SCHEMA special register. The owner of the JAR is the authorization ID in the CURRENT SQLID special register.

This method does not apply to connections to IBM Informix data sources.

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 on Linux, UNIX, and Windows systems 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:

```java
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:

```java
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 on Linux, UNIX, and Windows systems 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:

```java
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. `setDB2ClientAccountingToken` applies only to connections to DB2 for z/OS.
setDB2ClientAccountingInformation sets the value in the CURRENT
CLIENT_ACCTNG special register.

Parameter description:

\textit{info}

User-specified accounting information.

The maximum length depends on the data server version. See “Client info
properties support by the IBM Data Server Driver for JDBC and SQLJ” on
page 89 for the maximum lengths.

A Java empty string (""") or a Java null value is valid for this parameter.

**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.

\textbf{setDB2ClientApplicationInformation}

Format:

\begin{verbatim}
public String setDB2ClientApplicationInformation(String info)
    throws java.sql.SQLException
\end{verbatim}

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:

\textit{info}

User-specified application information.

The maximum length depends on the data server version. See “Client info
properties support by the IBM Data Server Driver for JDBC and SQLJ” on
page 89 for the maximum lengths.

A Java empty string (""") or a Java null value is valid for this parameter.

**setDB2ClientDebugInfo**

Formats:

\begin{verbatim}
public void setDB2ClientDebugInfo(String debugInfo)
    throws java.sql.SQLException

public void setDB2ClientDebugInfo(String mgrInfo,
                          String traceInfo)
    throws java.sql.SQLException
\end{verbatim}

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:

\begin{itemize}
  \item The DEBUGSESSION privilege
  \item SYSADM authority
\end{itemize}

Parameter description:

\textit{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:

**Mip:port**
Session manager IP address and port number

**Iip**
Client IP address

**Ppid**
Client process ID

**Ttid**
Client thread ID (optional)

**Cid**
Data connection generated ID

**Llvl**
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.

**Mip: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 on Linux, UNIX, and Windows systems 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.

setDB2ClientUser sets the value in the CLIENT USERID special register.

Parameter description:

user
The user ID for the current client. The maximum length depends on the server.

The maximum length depends on the data server version. See “Client info properties support by the IBM Data Server Driver for JDBC and SQLJ” on page 89 for the maximum lengths.

A Java empty string ("") or a Java null value is valid for this parameter.

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.

setDB2ClientWorkstation sets the value in the CLIENT WRKSTNNAME special register.

Parameter description:

name
The workstation name for the current client.

The maximum length depends on the data server version. See “Client info properties support by the IBM Data Server Driver for JDBC and SQLJ” on page 89 for the maximum lengths.

A Java empty string ("") or a Java null value is valid for this parameter.

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:
```java
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.

**setDB2ProgressiveStreaming**
Format:
```java
public void setDB2ProgressiveStreaming(int newSetting)
throws java.sql.SQLException
```

Sets the progressive streaming setting for all ResultSet objects that are created on the connection.

Parameter descriptions:

**newSetting**
The new progressive streaming setting. Possible values are:

- **DB2BaseDataSource.YES (1)**
  Enable progressive streaming. If the data source does not support progressive streaming, this setting has no effect.

- **DB2BaseDataSource.NO (2)**
  Disable progressive streaming.

**setJccLogWriter**
Formats:
```java
public void setJccLogWriter(PrintWriter logWriter)
throws java.sql.SQLException
```
```java
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.

updateDB2XmlSchema
Format:
public void updateDB2XmlSchema(String[] targetSqlIdSchema,
String[] targetSqlIdName,
String[] sourceSqlIdSchema,
String[] sourceSqlIdName,
String[] xmlSchemaLocations,
boolean dropSourceSchema)
throws SQLException

Updates the contents of an XML schema with the contents of another XML schema in the XML schema repository, and optionally drops the source schema. The schema documents in the target XML schema are replaced with the schema documents from the source XML schema. Before updateDB2XmlSchema can be called, registration of the source and target XML schemas must be completed.

The SQL ALTERIN privilege is required for updating the target XML schema. The SQL DROPIN privilege is required for dropping the source XML schema.

Parameter descriptions:

targetSqlIdSchema
The SQL schema name for a registered XML schema that is to be updated. targetSqlIdSchema is a String value with a maximum length of 128 bytes.
**targetSqlIdName**

The name of the registered XML schema that is to be updated. 
**targetSqlIdName** is a String value with a maximum length of 128 bytes.

**sourceSqlIdSchema**

The SQL schema name for a registered XML schema that is used to update the target XML schema. **sourceSqlIdSchema** is a String value with a maximum length of 128 bytes.

**sourceSqlIdName**

The name of the registered XML schema that is used to update the target XML schema. **sourceSqlIdName** is a String value with a maximum length of 128 bytes.

**dropSourceSchema**

Indicates whether the source XML schema is to be dropped after the target XML schema is updated. **dropSourceSchema** is a boolean value. false is the default.

This method does not apply to connections to IBM Informix data sources.

### Related concepts:

- Chapter 16, “Problem diagnosis with the IBM Data Server Driver for JDBC and SQLJ,” on page 629

### Related tasks:

- “Providing extended client information to the data source with IBM Data Server Driver for JDBC and SQLJ-only methods” on page 87

### Related reference:

- “Common IBM Data Server Driver for JDBC and SQLJ properties for DB2 servers” on page 263
- “IBM Data Server Driver for JDBC and SQLJ properties for Db2 on Linux, UNIX, and Windows systems” on page 278

### 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:

```java
void setProperty-name(data-type property-value)
```

A getXXX method has this form:

```java
data-type getPropertyName()
```
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 96. DB2ConnectionPoolDataSource properties and their data types

<table>
<thead>
<tr>
<th>Property name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements</td>
<td>int</td>
</tr>
</tbody>
</table>

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 on Linux, UNIX, and Windows systems 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 uses the user ID and password of the DB2ConnectionPoolDataSource object. The third form `getDB2TrustedPooledConnection` is for connections that use Kerberos security.

Parameter descriptions:

user
   The data server 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.

Related concepts:

- Chapter 12, “JDBC and SQLJ connection pooling support,” on page 617

Related reference:

- "Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 234

### 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:
   ```java
   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.

**getDriverDB2ConnectLevel**

Identifies the version of Db2 Connect and the IBM® Data Server Driver for JDBC and SQLJ version that an application is using for a connection to a data server.

**DB2Diagnosable interface**

The `com.ibm.db2.jcc.DB2Diagnosable` interface provides a mechanism for getting additional diagnostic information 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.SQLException` 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.SQLException` 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.

**Related tasks:**
- "Handling SQL warnings in an SQLJ application" on page 185
- "Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ" on page 116

**DB2DataSource class**

The `com.ibm.db2.jcc.DB2DataSource` class extends the `DB2BaseDataSource` class, and implements the `javax.sql.DataSource`, `java.io.Serializable`, and `javax.naming.Referenceable` interfaces.

**DB2DataSource methods**

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

**setSpecialRegisters**

Format:
```
public void setSpecialRegisters(java.util.Properties properties) 
   throws java.sql.SQLException
```

For each key and value pair in the `java.util.Properties` object, sets the data server special register that is specified by the key to the corresponding value.

This method does not apply to connections to IBM Informix data servers.

Parameter description:

**properties**
A `java.util.Properties` object that contains key and value pairs, in which each key is the name of a special register, and each value is the value to which you want to set that special register. For example, suppose that `ds` is a previously defined `DataSource` object. Set the property values in the following way:
```
Properties prop = new Properties();
prop.put("CURRENT SCHEMA", "SYSPROC");
prop.put("CURRENT PACKAGESET", "PRODUCTION");
((com.ibm.db2.jcc.DB2BaseDataSource) ds).setSpecialRegisters(prop);
```

Certain special registers can be set through IBM Data Server Driver for JDBC and SQLJ properties. If you set a special register value by setting one of those properties in a `java.util.Properties` object, and then use
setSpecialRegisters to set a value for the same special register, the value that is set through setSpecialRegisters overrides the value that is set through the property. In the following example, CURRENT SCHEMA is set to USER002:

Properties prop = new Properties();
((com.ibm.db2.jcc.DB2BaseDataSource) ds).setCurrentSchema("USER001");
properties.put("CURRENT SCHEMA", "USER002");
((com.ibm.db2.jcc.DB2BaseDataSource) ds).setSpecialRegisters(prop);

DB2Driver class
The com.ibm.db2.jcc.DB2Driver class extends the java.sql.Driver interface.

DB2Driver methods
The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

changeDB2Password
Format:
public static void changeDB2Password (String url,
String userid,
String oldPassword,
String newPassword)
throws java.sql.SQLException

Changes the password for accessing a data server that is specified by the url parameter, for the user that is specified by the userid parameter. This method can change an unexpired or expired password.

changeDB2Password is supported for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity only.

changeDB2Password is not supported for connections to IBM Informix.

Parameter descriptions:

url
The URL for the data server for which a user's password is being changed. The url value uses the syntax for a URL for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

userid
The user whose password is being changed.

oldPassword
The original password for the user.

newPassword
The new password for the user.

Related reference:
"URL format for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity" on page 14

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.

Related concepts:
- “Example of a trace program under the IBM Data Server Driver for JDBC and SQLJ” on page 634

DB2FileReference class

The com.ibm.db2.jcc.DB2FileReference class is an abstract class that defines methods that support insertion of data into tables from file reference variables. This class applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS Version 9 or later.

DB2FileReference fields

The following constants define types codes only for the IBM Data Server Driver for JDBC and SQLJ.

public static final short MAX_FILE_NAME_LENGTH = 255
  The maximum length of the file name for a file reference variable.

DB2FileReference methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getDriverType

Format:
- public int getDriverType()

Returns the server data type of the file reference variable. This type is one of the values in com.ibm.db2.jcc.DB2Types.

getFileEncoding

Format:
public String getFileEncoding()

Returns the encoding of the data in the file for a DB2FileReference object.

**getFileName**

Format:
public String getFileName()

Returns the file name for a DB2FileReference object.

**getFileStream**

Format:
public int getFileStream()

Returns the CCSID of the data in the file for a DB2FileReference object.

**setFileName**

Format:
public String setFileName(String fileName)
    throws java.sql.SQLException

Sets the file name in a DB2FileReference object.

Parameter descriptions:

**fileName**

The name of the input file for the file reference variable. The name must specify an existing HFS file.

### DB2JCCPlugin class

The com.ibm.db2.jcc.DB2JCCPlugin class is an abstract class that defines methods that can be implemented to provide Db2 on Linux, UNIX, and Windows systems plug-in support. This class applies only to Db2 on Linux, UNIX, and Windows systems.

### 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.GSSError

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**

Chapter 7. JDBC and SQLJ reference information 427
**DB2JSONResultSet interface**

The `com.ibm.db2.jcc.DB2JSONResultSet` interface is used to convert the contents of JDBC `ResultSet` objects to JSON documents and snippets.

The `DB2JSONResultSet` interface is available only in IBM Data Server Driver for JDBC and SQLJ version 4.18 or later.

**DB2JSONResultSet methods**

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

- **close**
  
  Format:
  
  ```java
  public void close()
  throws java.sql.SQLException
  ```
  
  Closes a `DB2JSONResultSet` object.

- **getAsciiStream**
  
  Format:
  
  ```java
  public InputStream getAsciiStream()
  throws java.sql.SQLException
  ```
  
  Returns a `java.io.InputStream` object that contains the contents of a `DB2JSONResultSet` object. If the contents of the object contain any non-ASCII characters, an encoding error is returned in the `SQLException`.

- **getCharacterStream**
  
  Format:
  
  ```java
  public Reader getCharacterStream()
  throws java.sql.SQLException
  ```
  
  Returns a `java.io.Reader` object on which applications can call methods to read JSON documents incrementally.

- **getCurrentRow**
  
  Format:
  
  ```java
  public String getCurrentRow()
  throws java.sql.SQLException
  ```
  
  Returns the current row of a `DB2JSONResultSet` object as a JSON snippet.

- **next**
  
  Format:
  
  ```java
  public String next()
  throws java.sql.SQLException
  ```
  
  Moves the cursor position to the next row of a `DB2JSONResultSet` object. Returns `false` if there are no more rows to return.

- **toJSONString**
  
  Format:
  
  ```java
  public String toJSONString()
  throws java.sql.SQLException
  ```
Returns the contents of a DB2JSONResultSet object, starting with the current row, as a String. After this method is executed, the cursor points to the last row of the DB2JSONResultSet object. A subsequent next call returns false.

**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.

### getMaxStringUnitBits

Format:

```java
public int getMaxStringUnitBits (int param)
    throws java.sql.SQLException
```

Returns the maximum number of bits in a string unit for single-byte and double-byte character data types. The value that is returned is:

- 8 For a character column that is defined with OCTETS.
- 16 For a character column that is defined with CODEUNITS16.
- 32 For a character column that is defined with CODEUNITS32.

Parameter descriptions:

- **param**
  - The ordinal position of a parameter in the CALL statement.

This method applies only to connections to Db2 on Linux, UNIX, and Windows systems Version 10.5 or later data servers.

### getParameterMarkerNames

Format:

```java
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 to `DB2BaseDataSource.NOT_SET` or `DB2BaseDataSource.NO`, or if there are no named parameter markers in the SQL statement.

### getProcedureParameterName

Format:

```java
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 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems 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:

```java
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)**

Format:

```java
public DB2Connection getDB2Connection(byte[] cookie, 
   String user, 
   String password, 
   String userRegistry, 
   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:

```java
public DB2Connection getDB2Connection(
    String user,
    String password,
    java.util.Properties properties)
throws java.sql.SQLException
```

```java
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:

```java
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.

Related concepts:

Chapter 12, “JDBC and SQUIP connection pooling support,” on page 617

Related reference:

“DB2ConnectionPoolDataSource class” on page 420

### DB2PoolMonitor class

The `com.ibm.db2.jdbc.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 SQUIP.

```java
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:
```java
public abstract int numberOfConnectionReleaseRefused()
```
Retrieves the number of times that the release of a connection was refused.

**numberOfRequestsBlocked**
Format:
```java
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:
```java
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:
```java
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:
```java
public abstract int removedObjectCount()
```
Retrieves the number of objects that have been deleted from the pool since the pool was created.

**shortestBlockedRequestTime**
Format:
```java
public abstract long shortestBlockedRequestTime()
```
Retrieves the shortest amount of time that a request was blocked, in milliseconds.

**successfulRequestsFromPool**
Format:
```java
public abstract int successfulRequestsFromPool()
```
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.
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

### 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 unassigned.

### DB2PreparedStatement methods
The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

**executeDB2QueryBatch**
Format:
```
public void executeDB2QueryBatch()
```

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.

setDB2BlobFileReference
Format:
public void setDB2BlobFileReference(int parameterIndex,
  com.ibm.db2.jcc.DB2BlobFileReference fileRef)
throws java.sql.SQLException

Assigns a BLOB file reference variable value to a parameter.
This method applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS.
Parameters:

*parameterIndex*
  The index of the parameter marker to which a file reference variable value is assigned.

*fileRef*
  The `com.ibm.db2.jcc.DB2BlobFileReference` value that is assigned to the parameter marker.

**setDB2ClobFileReference**
Format:
```java
public void setDB2ClobFileReference(int parameterIndex,
        com.ibm.db2.jcc.DB2ClobFileReference fileRef)
        throws java.sql.SQLException
```

Assigns a CLOB file reference variable value to a parameter.
This method applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS.

Parameters:

*parameterIndex*
  The index of the parameter marker to which a file reference variable value is assigned.

*fileRef*
  The `com.ibm.db2.jcc.DB2ClobFileReference` value that is assigned to the parameter marker.

**setDB2XmlAsBlobFileReference**
Format:
```java
public void setDB2XmlAsBlobFileReference(int parameterIndex,
        com.ibm.db2.jcc.DB2XmlAsBlobFileReference fileRef)
        throws java.sql.SQLException
```

Assigns a XML AS BLOB file reference variable value to a parameter.
This method applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS.

Parameters:

*parameterIndex*
  The index of the parameter marker to which a file reference variable value is assigned.

*fileRef*
  The `com.ibm.db2.jcc.DB2XmlAsBlobFileReference` value that is assigned to the parameter marker.

**setDB2XmlAsClobFileReference**
Format:
```java
public void setDB2XmlAsClobFileReference(int parameterIndex,
        com.ibm.db2.jcc.DB2XmlAsClobFileReference fileRef)
        throws java.sql.SQLException
```

Assigns a XML AS CLOB file reference variable value to a parameter.
This method applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS.

Parameters:
**parameterIndex**

The index of the parameter marker to which a file reference variable value is assigned.

**fileRef**

The com.ibm.db2.jcc.DB2Xm1AsClobFileReference value that is assigned to the parameter marker.

**setDBTimestamp**

Format:

```java
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:

```java
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:

```java
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:

```java
public void setJccAsciiStreamAtName(String parameterMarkerName,
    java.io.InputStream x)
    throws java.sql.SQLException
```

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
public void setJccBlobAtName(String parameterMarkerName,
   java.io.InputStream x)
   throws java.sql.SQLException
```

```java
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:

```java
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.
The `java.util.Calendar` object that the IBM Data Server Driver for JDBC and SQLJ uses to construct the date.

`setJccDB2BlobFileReferenceAtName`  
Format:  
```java
public void setJccDB2BlobFileReferenceAtName(int parameterMarkerName,
com.ibm.db2.jcc.DB2BlobFileReference fileRef)
throws java.sql.SQLException
```

Assigns a BLOB file reference variable value to a named parameter marker.  
This method applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS.  
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 file reference variable value is assigned.  
- `fileRef`  
The `com.ibm.db2.jcc.DB2BlobFileReference` value that is assigned to the named parameter marker.

`setJccDB2ClobFileReferenceAtName`  
Format:  
```java
public void setJccDB2ClobFileReferenceAtName(int parameterMarkerName,
com.ibm.db2.jcc.DB2ClobFileReference fileRef)
throws java.sql.SQLException
```

Assigns a CLOB file reference variable value to a named parameter marker.  
This method applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS.  
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 file reference variable value is assigned.  
- `fileRef`  
The `com.ibm.db2.jcc.DB2ClobFileReference` value that is assigned to the named parameter marker.

`setJccDB2XmlAsBlobFileReferenceAtName`  
Format:  
```java
public void setJccDB2XmlAsBlobFileReferenceAtName(String parameterMarkerName,
com.ibm.db2.jcc.DB2XmlAsBlobFileReference fileRef)
throws java.sql.SQLException
```

Assigns a XML AS BLOB file reference variable value to a named parameter marker.  
This method applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS Version 9 or later.
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 file reference variable value is assigned.

`fileRef`

The com.ibm.db2.jcc.DB2XmlAsBlobFileReference value that is assigned to the named parameter marker.

`setJccDB2XmlAsClobFileReferenceAtName` Format:

```java
public void setJccDB2XmlAsClobFileReferenceAtName(int parameterMarkerName,
com.ibm.db2.jcc.DB2XmlAsClobFileReference fileRef)
throws java.sql.SQLException
```

Assigns a XML AS CLOB file reference variable value to a named parameter marker.

This method applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS Version 9 or later.

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 file reference variable value is assigned.

`fileRef`

The com.ibm.db2.jcc.DB2XmlAsClobFileReference value that is assigned to the named parameter marker.

`setJccDBTimestampAtName` Format:

```java
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:

```java
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**

Format:

```java
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:

```java
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:

```java
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:
```java
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:
```java
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:
```java
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.
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:

```java
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 length of the data in the Stream or Reader object.

**setJccShortAtName**

Format:

```java
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.
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 on Linux, UNIX, and Windows systems 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:

```java
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.

Related tasks:

“Making batch queries in JDBC applications” on page 41

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 fields

The following fields are defined only for the IBM Data Server Driver for JDBC and SQLJ.

The integer constants in the following table are used in the column descriptor information that getDBRowDescriptor returns. These constants contain information about the column values that are returned by getDBRowAsBytes. All fields are defined as public static int.

<table>
<thead>
<tr>
<th>Field value</th>
<th>Description of returned data</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPRESENTATION_FIXED_STRING (0)</td>
<td>Fixed-length string data</td>
</tr>
<tr>
<td>REPRESENTATION_BIG_ENDIAN (1)</td>
<td>Signed binary format in which the most significant byte is stored in the highest address</td>
</tr>
<tr>
<td>REPRESENTATION_LITTLE_ENDIAN (2)</td>
<td>Signed binary format in which the least significant byte is stored in the highest address</td>
</tr>
<tr>
<td>REPRESENTATION_VARIABLE_STRING (2)</td>
<td>String data that begins with a two-byte length field</td>
</tr>
<tr>
<td>REPRESENTATION_NUL_TERMINATED_STRING (3)</td>
<td>Nul-terminated string data</td>
</tr>
<tr>
<td>REPRESENTATION_FIXED_BYTES (4)</td>
<td>Fixed-length byte string</td>
</tr>
<tr>
<td>REPRESENTATION_VARIABLE_BYTES (5)</td>
<td>Byte string that begins with a two-byte length field</td>
</tr>
<tr>
<td>REPRESENTATION_NUL_TERMINATED_BYTES (7)</td>
<td>Nul-terminated byte data</td>
</tr>
<tr>
<td>REPRESENTATION_FIXED_BINARY (15)</td>
<td>Fixed-length binary string</td>
</tr>
<tr>
<td>REPRESENTATION_VARIABLE_BINARY (16)</td>
<td>Binary string that begins with a two-byte length field</td>
</tr>
<tr>
<td>REPRESENTATION_PACKED_DECIMAL (48)</td>
<td>Nul-terminated binary string</td>
</tr>
<tr>
<td>REPRESENTATION_NUMERIC_CHARACTER (50)</td>
<td>Character-based, fixed-point format</td>
</tr>
<tr>
<td>REPRESENTATION_ZONED_DECIMAL (51)</td>
<td>Zoned-decimal format that is returned by IBM System i® and IBM Z</td>
</tr>
<tr>
<td>REPRESENTATION_COBOL2_ZONED_DECIMAL (53)</td>
<td>Zoned-decimal format that is returned by Windows or UNIX systems</td>
</tr>
<tr>
<td>Field value</td>
<td>Description of returned data</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>REPRESENTATION_HEXDECIMAL_FLOATING_POINT (64)</td>
<td>S/390 hexadecimal floating point format</td>
</tr>
<tr>
<td>REPRESENTATION_DECIMAL_FLOATING_POINT (66)</td>
<td>Decimal floating point format</td>
</tr>
<tr>
<td>REPRESENTATION_IEEE_754_FLOATING_POINT_BYTE_REVERSED (71)</td>
<td>IEEE floating-point format in which the least significant byte is stored in the highest address</td>
</tr>
<tr>
<td>REPRESENTATION_IEEE_754_FLOATING_POINT (72)</td>
<td>IEEE floating-point format in which the most significant byte is stored in the highest address</td>
</tr>
</tbody>
</table>

**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 on Linux, UNIX, and Windows systems.

**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 on Linux, UNIX, and Windows systems.

**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 on Linux, UNIX, and Windows systems.

**getDBRowDataAsBytes**

Format:
```
public Object [] getDBRowDataAsBytes()
    throws SQLException
```

Returns an Object array that represents the data in the current row of an open ResultSet object.

This method does not apply to connections to IBM Informix.
getDBRowDataAsBytes cannot be called if the ResultSet object on which it operates meets any of the following conditions:

- A ResultSet row that is being retrieved has been updated, deleted, or inserted.
- The ResultSet object was created for optimistic locking.

The returned information includes:

- The data in raw byte array format
- The offset to the data for each column

Suppose that obj is an instance of the returned Object array. The format of the Object array is:

obj[0] A byte array that describes the row data.

obj[1] An integer array that contains the offset into obj[0] of each column description. The offsets can be used to determine the length of the data that is returned for each column. That length represents the length of the raw data, and not the defined length of the column.

If a ResultSet object contains a column of any of the following types, the offset value for that column value in obj[1] is -1. -1 indicates that a value for that column is not returned.

- BLOB
- CLOB
- DBCLOB
- XML

The byte array in obj[0] has the following format:

\[rnn\ldots\dd\ldots\nn\dd\ldots\dd\]

There is one \[\nn\dd\ldots\dd\] set for each column in the row.

The following table describes the contents of the row data:

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>A single byte that has one of the following values:</td>
</tr>
<tr>
<td></td>
<td>0           The row data is not valid. One reason for</td>
</tr>
<tr>
<td></td>
<td>invalid data is that the row has not yet been</td>
</tr>
<tr>
<td></td>
<td>fetched.</td>
</tr>
<tr>
<td></td>
<td>1           The row data is valid.</td>
</tr>
<tr>
<td>nn</td>
<td>A two-byte NULL indicator for a column value.</td>
</tr>
<tr>
<td></td>
<td>Possible values are:</td>
</tr>
<tr>
<td></td>
<td>-1 The column value that follows is null.</td>
</tr>
<tr>
<td></td>
<td>0 The column value that follows is not null.</td>
</tr>
<tr>
<td>dd...dd</td>
<td>Raw byte data for a column value.</td>
</tr>
</tbody>
</table>

getDBRowDescriptor

Format:

```java
public int[] getDBRowDescriptor()
    throws SQLException
```

Returns an int array that contains descriptive information about each column of the row data that is returned by getDBRowDataAsBytes.

This method does not apply to connections to IBM Informix.
Suppose that returnedInfo is an instance of the array that is returned by getDBRowDescriptor. The format of the returned array is:

returnedInfo[0]
The number of columns in the row data. Suppose that this value is \( n \).

returnedInfo[1] through returnedInfo[4*n]
\( n \) repeating groups of four integer values. Each group contains descriptive information for a single column. That information is:

<table>
<thead>
<tr>
<th>Column descriptor number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The data type of the column, expressed as an SQLTYPE value. This value is the same as the SQLTYPE value that is returned in an SQLDA.</td>
</tr>
<tr>
<td>2</td>
<td>The CCSID of the column, for a character data type. For a DECIMAL data type, this value is the scale of the column.</td>
</tr>
<tr>
<td>3</td>
<td>The defined length of the column, for all data types except DECIMAL. For a DECIMAL data type, this value is the precision of the column. For varying-length character data types, this value might be greater than the number of returned bytes.</td>
</tr>
<tr>
<td>4</td>
<td>Additional information about the column. Possible values are described in &quot;DB2ResultSet fields&quot; on page 450.</td>
</tr>
</tbody>
</table>

getDBTimestamp
Formats:

```java
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:

```java
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.

**getDBTemporalColumnType**

Format:

```java
public int getDBTemporalColumnType (int column)
throws java.sql.SQLException
```

Returns:

-1 If `column` is not a ROW BEGIN, ROW END or TRANSACTION START ID column.

1 If `column` is a ROW BEGIN column.

2 If `column` is a ROW END column.

3 If `column` is a TRANSACTION START ID column.

Parameter descriptions:

**column**

The ordinal position of a column in the `ResultSet`.

**getMaxStringUnitBits**

Format:

```java
public int getMaxStringUnitBits (int column)
throws java.sql.SQLException
```

Returns the maximum number of bits in a string unit for single-byte and double-byte character data types. The value that is returned is:

8 For a character column that is defined with OCTETS.

16 For a character column that is defined with CODEUNITS16.

32 For a character column that is defined with CODEUNITS32.

Parameter descriptions:

**column**

The ordinal position of a column in the `ResultSet`.
This method applies only to connections to Db2 on Linux, UNIX, and Windows systems Version 10.5 or later data servers.

`isDB2ColumnNameDerived`

Format:

```java
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

```sql
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:

```java
public byte[] getBytes()
```

Converts a `com.ibm.jcc.DB2RowID` object to bytes.

**Related concepts:**

- "ROWIDs in SQLJ with the IBM Data Server Driver for JDBC and SQLJ” on page 170
- "ROWIDs in JDBC with the IBM Data Server Driver for JDBC and SQLJ” on page 65

**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 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.

**Related tasks:**
- “Creating and deploying DataSource objects” on page 24
- “Connecting to a data source using the DataSource interface” on page 20

**Related reference:**
- “Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products” on page 235

**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.

Related tasks:
- “Handling SQL warnings in an SQLJ application” on page 185
- “Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ” on page 116

Related reference:
- [Description of SQLCA fields (DB2 SQL)](#)

**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.

**getAffectedRowCount**
Format: 
public int getAffectedRowCount() throws java.sql.SQLException

Returns the number of rows that are affected by successful execution of an SQL statement. If the SQL statement is INSERT, UPDATE, or DELETE, `getAffectedRowCount` returns the same value that is returned by `java.sql.Statement.getUpdateCount`.

The value that is returned by `getAffectedRowCount` is the same information that is returned by the data server in the SQLCA after successful execution of an SQL statement.

**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 on Linux, UNIX, and Windows systems data servers.

**setDB2ClientProgramId**

Format:

```java
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 on Linux, UNIX, and Windows systems 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:

```java
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:

```java
public intgetIDSSerial()
    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:
```java
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:
```java
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.

**Related reference:**
“DB2PreparedStatement interface” on page 435

**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.

```java
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:**
```java
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:**
```java
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 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems, 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 start method results in an SQLException.

**start**
Format:
```java
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:
```java
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.

**Related concepts:**
[Chapter 18, “System monitoring for the IBM Data Server Driver for JDBC and SQLJ,” on page 643]

**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:

```java
static public DB2TraceManager getTraceManager()
throws java.sql.SQLException
```

Gets an instance of the global log writer.

**setLogWriter**

Formats:

```java
public abstract void setLogWriter(String traceDirectory,
        String baseTraceFileName, int traceLevel)
throws java.sql.SQLException
```

```java
public abstract void setLogWriter(String traceFile,
        boolean fileAppend, int traceLevel)
throws java.sql.SQLException
```

```java
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 on Linux, UNIX, and Windows systems 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_T2ZOS (X'10000')`
- `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:

  \[ \text{TRACEDRDA FLOWS} | \text{TRACECONNECTION CALLS} \]

- 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:

  \[ \sim \text{TRACEDRDA FLOWS} \]
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.

glogfile
Format:
public abstract java.io.PrintWriter getLogWriter()
throws java.sql.SQLException

Returns the PrintWriter for the global log writer, if it is set. Otherwise,
glogfile returns null.

ggetTraceFile
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.

ggetTraceDirectory
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.

ggetTraceLevel
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.

Related reference:
"Properties for the IBM Data Server Driver for JDBC and SQLJ" on page 234

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_T2ZOS (X'10000')
• 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 nth 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 nth 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_T2ZOS (X'10000')`
- `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:

```java
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:

```java
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:

```java
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

DB2Version class
The com.ibm.db2.jcc.DB2Version class provides methods that return information about the IBM Data Server Driver for JDBC and SQLJ version.

DB2Version methods
The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getBuildNumber
Format:
    public static String getBuildNumber()
Returns the last portion of the driver version. For driver version `xx.yy.zzz`, the build number is `zzz`.

`getCompatibleJREVersions`
Format:
`public static String[] getCompatibleJREVersions()`

Returns the Java Runtime Environment versions that are compatible with the driver version.

`getDriverName`
Format:
`public static String getDriverName()`

Returns the name of the driver.

`getDB2ConnectLicenseVersion`
Format:
`public static String getDB2ConnectLicenseVersion()`

Returns the Db2 Connect version for which the IBM Data Server Driver for JDBC and SQLJ client license is valid.

This method is supported only for connections to DB2 for z/OS or Db2 for IBM i data servers.

`getExpirationDateForDriver`
Format:
`public static java.util.Date getExpirationDateForDriver()`

Returns the date on which the client license file for connections to DB2 for z/OS or Db2 for IBM i data servers expires.

This method is supported only for connections to DB2 for z/OS or Db2 for IBM i data servers.

`getExpirationDateForDriverWithLicenseType`
Format:
`public static String getExpirationDateForDriverWithLicenseType()`

Returns a String value that contains:
• The date on which the client license file for connections to DB2 for z/OS or Db2 for IBM i data servers expires.
• The type of license: temporary or permanent

This method is supported only for connections to DB2 for z/OS or Db2 for IBM i data servers.

`getMajorVersion`
Format:
`public static String getMajorVersion()`

Returns the first portion of the driver version. For driver version `xx.yy.zzz`, the major version is `xx`.

`getMinorVersion`
Format:
`public static String getMinorVersion()`
Returns the middle portion of the driver version. For driver version \( xx.yy.zzz \),
the minor version is \( yy \).

**getVersion**

Format:

```java
public static String getVersion()
```

Returns the driver version.

**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:

```java
public Object[] getDB2TrustedXAConnection(String user, 
String password, 
java.util.Properties properties)
throws java.sql.SQLException
```

```java
public Object[] getDB2TrustedXAConnection( 
java.util.Properties properties)
throws java.sql.SQLException
```

```java
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 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems 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.

**Related concepts:**
“Example of a distributed transaction that uses JTA methods” on page 621

**Related tasks:**
“Creating and deploying DataSource objects” on page 24

### DB2Xml interface

The com.ibm.db2.jcc.DB2Xml interface is used for declaring Java objects for use with the data server's XML data type.

### DB2Xml methods

The following method is defined only for the IBM Data Server Driver for JDBC and SQLJ.

**closeDB2Xml**
Format:
```
public void closeDB2Xml()
```
throws SQLException

Releases the resources that are associated with a com.ibm.jcc.DB2Xml object.

**getDB2AsciiStream**
Format:
```
public java.io.InputStream getDB2AsciiStream()
```
throws SQLException

Retrieves data from a DB2Xml object, and converts the data to US-ASCII encoding.

**getDB2BinaryStream**
Format:
```
public java.io.InputStream getDB2BinaryStream()
```
throws SQLException

Retrieves data from a DB2Xml object as a binary stream. The character encoding of the bytes in the binary stream is defined in the XML 1.0 specification.

**getDB2Bytes**
Format:
```
public byte[] getDB2Bytes()
```
throws SQLException

Retrieves data from a DB2Xml object as a byte array. The character encoding of the bytes is defined in the XML 1.0 specification.
getDB2CharacterStream
Format:
public java.io.Reader getDB2CharacterStream()
throws SQLException

Retrieves data from a DB2Xml object as a java.io.Reader object.

getDB2String
Format:
public String getDB2String()
throws SQLException

Retrieves data from a DB2Xml object as a String value.

getDB2XmlAsciiStream
Format:
public InputStream getDB2XmlAsciiStream()
throws SQLException

Retrieves data from a DB2Xml object, converts the data to US-ASCII encoding, and imbeds an XML declaration with an encoding specification for US-ASCII in the returned data.

getDB2XmlBinaryStream
Format:
public java.io.InputStream getDB2XmlBinaryStream(String targetEncoding)
throws SQLException

Retrieves data from a DB2Xml object as a binary stream, converts the data to targetEncoding, and imbeds an XML declaration with an encoding specification for targetEncoding in the returned data.

Parameter:

targetEncoding
A valid encoding name that is listed in the IANA Charset Registry. The encoding names that are supported by the data server are listed in "Mappings of CCSIDs to encoding names for serialized XML output data".

getDB2XmlBytes
Format:
public byte[] getDB2XmlBytes(String targetEncoding)
throws SQLException

Retrieves data from a DB2Xml object as a byte array, converts the data to targetEncoding, and imbeds an XML declaration with an encoding specification for targetEncoding in the returned data.

Parameter:

targetEncoding
A valid encoding name that is listed in the IANA Charset Registry. The encoding names that are supported by the data server are listed in "Mappings of CCSIDs to encoding names for serialized XML output data".

getDB2XmlCharacterStream
Format:
public java.io.Reader getDB2XmlCharacterStream()
throws SQLException
Retrieves data from a DB2Xml object as a java.io.Reader object, converts the data to ISO-10646-UCS-2 encoding, and imbeds an XML declaration with an encoding specification for ISO-10646-UCS-2 in the returned data.

**getDB2XmlString**
Format:
```java
public String getDB2XmlString()
throws SQLException
```

Retrieves data from a DB2Xml object as a String object, converts the data to ISO-10646-UCS-2 encoding, and imbeds an XML declaration with an encoding specification for ISO-10646-UCS-2 in the returned data.

**isDB2XmlClosed**
Format:
```java
public boolean isDB2XmlClosed()
throws SQLException
```

Indicates whether a com.ibm.jcc.DB2Xml object has been closed.

Related concepts:
- “XML data retrieval in SQLJ applications” on page 178
- “XML column updates in SQLJ applications” on page 176
- “XML data retrieval in JDBC applications” on page 98
- “XML column updates in JDBC applications” on page 96

**DB2XmlAsBlobFileReference class**
The com.ibm.db2.jcc.DB2XmlAsBlobFileReference class is subclass of DB2FileReference that is used for creating XML AS BLOB file reference variable objects. This class applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS Version 9 or later.

**DB2XmlAsBlobFileReference constructor**
The following constructor is defined only for the IBM Data Server Driver for JDBC and SQLJ.

```java
public DB2XmlAsBlobFileReference(String fileName)
throws java.sql.SQLException
```

Constructs a DB2XmlAsBlobFileReference object for an XML AS BLOB file reference variable.

Parameter descriptions:
- **fileName**
  The name of the file for the file reference variable. The name must specify the absolute path name for an existing HFS file.

**DB2XmlAsClobFileReference class**
The com.ibm.db2.jcc.DB2XmlAsClobFileReference class is subclass of DB2FileReference that is used for creating XML AS CLOB file reference variable objects. This class applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS Version 9 or later.
DB2XmlAsClobFileReference constructor

The following constructor is defined only for the IBM Data Server Driver for JDBC and SQLJ.

DB2XmlAsClobFileReference

Format:
public DB2XmlAsClobFileReference(String fileName,  
    int fileCcsid)  
    throws java.sql.SQLException
public DB2XmlAsClobFileReference(String fileName,  
    String fileEncoding)  
    throws java.sql.SQLException

Constructs a DB2XmlAsClobFileReference object for an XML AS CLOB file reference variable.

Parameter descriptions:

fileName
The name of the file for the file reference variable. The name must specify the absolute path name for an existing HFS file.

fileCcsid
The CCSID of the data in the file for the file reference variable.

fileEncoding
The encoding scheme of the data in the file for the file reference variable.

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(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 999999999999, 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

Chapter 7. JDBC and SQLJ reference information 477
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[.ffffffffffff]-th:tm`
  - `yyyy-mm-dd hh:mm:ss[.ffffffffffff]-th:tm`
  - `yyyy-mm-dd hh:mm:ss[.ffffffffffff]`
  - `yyyy-mm-dd hh:mm:ss[.ffffffffffff]`
  
  - **yyyy** is a year.
  - **mm** is a month.
  - **dd** is a day.
  - **hh** is hours.
  - **mm** is minutes.
  - **ss** is seconds.
  - **[.ffffffffffff]** 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:**

```java
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[.ffffffffffff]-th:tm`
- `yyyy-mm-dd hh:mm:ss[.ffffffffffff]`

**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 the information on 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 on Linux, UNIX, and Windows systems 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 a SQLException. For example, suppose that progressive streaming is enabled, and you execute statements like this:

```java
... 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**

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 z/OS or Db2 on Linux, UNIX, and Windows systems, or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on Db2 on Linux, UNIX, and Windows systems, the driver 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 on Linux, UNIX, and Windows systems 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);`
Data loss for TIMESTAMP(p) column updates using setString

If you use a setString call to pass an input value to a TIMESTAMP(p) column, it is possible to send a value with a precision of greater than nine to the column.

Before version 3.59 or version 4.9 of the IBM Data Server Driver for JDBC and SQLJ, data loss could occur if the sendDataAsIs property was set to false, and the precision of the input value was greater than nine.

Starting with version 3.59 and version 4.9 of the IBM Data Server Driver for JDBC and SQLJ, data loss does not occur if the TIMESTAMP(p) column is large enough to accommodate the input value.

Special processing for java.sql.Timestamp input data

Before version 3.63 or 4.13 of the IBM Data Server Driver for JDBC and SQLJ, if the target data type is not known, the target data server supports TIMESTAMP WITH TIME ZONE, and the input data type is java.sql.Timestamp, the driver chooses TIMESTAMP WITH TIME ZONE as the target type. Starting with version 3.63 or 4.13, if the target data type is not known, the target data server supports TIMESTAMP WITH TIME ZONE, and the input data type is java.sql.Timestamp, the driver chooses TIMESTAMP WITH TIME ZONE as the target type, except when the input object has a value of 0001-01-01-00:00:00.000000 or 9999-12-31-23:59:59.999999. In those cases, the driver chooses the TIMESTAMP type, without a time zone. Use of the TIMESTAMP data type in those two cases prevents an overflow condition from occurring because of adjustment of the value for the implied time zone. The implied time zone is the time zone of the Java virtual machine (JVM). Starting with version 3.65 or 4.15, the timestamps for which the driver chooses the TIMESTAMP type, without the time zone, are 0001-01-01, with any time, or 9999-12-31, with any time.

Change to result set column name for getColumns

In version 4.12 or earlier of the IBM Data Server Driver for JDBC and SQLJ, the DatabaseMetaData.getColumns method returned a result set that contained a column named SCOPE_CATALOG. In version 4.13 or later of the IBM Data Server Driver for JDBC and SQLJ, the name of that column is SCOPE_CATALOG. If you want the IBM Data Server Driver for JDBC and SQLJ to continue to use the column name SCOPE_CATALOG, set DataSource or Connection property useJDBC41DefinitionForGetColumns to DB2BaseDataSource.NO (2).

Changes to defaults for global configuration properties db2.jcc.maxRefreshInterval, db2.jcc.maxTransportObjects, and db2.jcc.maxTransportObjectWaitTime

The default values for global configuration properties db2.jcc.maxRefreshInterval, db2.jcc.maxTransportObjects, and db2.jcc.maxTransportObjectWaitTime change in version 3.63 and 4.13 of the IBM Data Server Driver for JDBC and SQLJ. The following table lists the old and new defaults.

<table>
<thead>
<tr>
<th>Configuration property</th>
<th>Default before versions 3.63 and 4.13</th>
<th>Default for versions 3.63 and 4.13 or later</th>
</tr>
</thead>
<tbody>
<tr>
<td>db2.jcc.maxRefreshInterval</td>
<td>30 seconds</td>
<td>10 seconds</td>
</tr>
<tr>
<td>db2.jcc.maxTransportObjects</td>
<td>-1 (unlimited)</td>
<td>1000</td>
</tr>
<tr>
<td>db2.jcc.maxTransportObjectWaitTime</td>
<td>-1 (unlimited)</td>
<td>1 second</td>
</tr>
</tbody>
</table>
Changes to default values for Connection and DataSource property maxTransportObjects

The default value for Connection and DataSource properties maxTransportObjects changes in version 3.63 and 4.13 of the IBM Data Server Driver for JDBC and SQLJ. The following table lists the old and new defaults.

<table>
<thead>
<tr>
<th>Connection and DataSource property</th>
<th>Default value before versions 3.63 and 4.13</th>
<th>Default value for versions 3.63 and 4.13 or later</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxTransportObjects</td>
<td>-1 (unlimited)</td>
<td>1000</td>
</tr>
</tbody>
</table>

Changes to default value for Connection and DataSource properties maxRetriesForClientReroute and retryIntervalForClientReroute for connections to a DB2 for z/OS data sharing group (driver versions 3.64, 4.14)

The default value for Connection and DataSource properties maxRetriesForClientReroute and retryIntervalForClientReroute change in version 3.64 and 4.14, of the IBM Data Server Driver for JDBC and SQLJ for connections to a DB2 for z/OS data sharing group. The following table lists the new defaults.

<table>
<thead>
<tr>
<th>Connection and DataSource property</th>
<th>Default value before versions 3.64 and 4.14</th>
<th>Default value for versions 3.64 and 4.14, or later</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxRetriesForClientReroute</td>
<td>Retry for 10 minutes, with a wait time between retries that increases as the length of time from the first retry increases.</td>
<td>5</td>
</tr>
<tr>
<td>retryIntervalForClientReroute</td>
<td>Retry for 10 minutes, with a wait time between retries that increases as the length of time from the first retry increases.</td>
<td>0</td>
</tr>
</tbody>
</table>

Changes to the default value and meaning of a retry for Connection and DataSource property maxRetriesForClientReroute for connections to a DB2 for z/OS data sharing group (driver versions 3.66 and 4.16)

The default value for Connection and DataSource property maxRetriesForClientReroute and the meaning of a retry change in version 3.66 and 4.16 of the IBM Data Server Driver for JDBC and SQLJ for connections to a DB2 for z/OS data sharing group.

The following table shows the old and new defaults.

<table>
<thead>
<tr>
<th>Change for property</th>
<th>Before versions 3.66 and 4.16</th>
<th>Versions 3.66 and 4.16, or later</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meaning of a retry</td>
<td>One attempt to connect to one member of the data sharing group.</td>
<td>One attempt to connect to all members of the data sharing group other than the failed member, and to the group IP address. This change might make it necessary to lower the value of maxRetriesForClientReroute.</td>
</tr>
</tbody>
</table>
**Change for property**

**Before versions 3.66 and 4.16**

For connections to a DB2 for z/OS data server:

- For the first connection to a data sharing group, if maxRetriesForClientReroute and retryIntervalForClientReroute are not set, and the enableSysplexWLB property is set to true, the default is to retry five times with a retry interval of 0.

- For a failover during a subsequent connection to a data sharing group, if maxRetriesForClientReroute and retryIntervalForClientReroute are not set, the enableSysplexWLB property is set to true, and a cached server list or an alternate server is specified, the default is to retry the connection for 10 minutes, with a wait time between retries that increases as the length of time from the first retry increases.

If the enableSysplexWLB property is set to true, the default is 1.

**Versions 3.66 and 4.16, or later**

**Change to the meaning of a retry for connections to a Db2 on Linux, UNIX, and Windows systems Db2 pureScale instance (driver versions 3.67 and 4.17)**

The meaning of a retry for automatic client reroute changes in version 3.67 and 4.17 of the IBM Data Server Driver for JDBC and SQLJ for connections to a Db2 on Linux, UNIX, and Windows systems Db2 pureScale instance.

The following table shows the old and new meanings.

<table>
<thead>
<tr>
<th>Meaning of a retry before versions 3.67 and 4.17</th>
<th>Meaning of a retry for versions 3.67 and 4.17, or later</th>
</tr>
</thead>
<tbody>
<tr>
<td>One attempt to connect to one member of the Db2 pureScale instance.</td>
<td>One attempt to connect to all members of the Db2 pureScale instance. This change might make it necessary to lower the value of maxRetriesForClientReroute.</td>
</tr>
</tbody>
</table>

**Changes to default values for client info properties for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS**

The default values for client info properties for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS change in version 3.64 and 4.14 of the IBM Data Server Driver for JDBC and SQLJ. The following table lists the old and new defaults.

<table>
<thead>
<tr>
<th>Client info property</th>
<th>Default value before versions 3.64 and 4.14</th>
<th>Default value for versions 3.64 and 4.14 or later</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationName</td>
<td>Empty string</td>
<td>The string &quot;db2jcc_application&quot;</td>
</tr>
<tr>
<td>ClientAccountingInformation</td>
<td>Empty string</td>
<td>Empty string</td>
</tr>
<tr>
<td>ClientHostname</td>
<td>Empty string</td>
<td>The string &quot;RRSAF&quot;</td>
</tr>
</tbody>
</table>
### Change to default for global configuration property

**db2.jcc.enableInetAddressGetHostName**

Starting with versions 3.65 and 4.15 of the IBM Data Server Driver for JDBC and SQLJ, the default for `db2.jcc.enableInetAddressGetHostName` is `false`. For versions 3.64 and 4.14 or earlier, the default is `true`.

### Changes to the behavior of the `xmlFormat` property

Starting with version 4.15 of the IBM Data Server Driver for JDBC and SQLJ, the `xmlFormat` property applies only to XML data retrieval, instead of to XML data update and retrieval. In addition, the default behavior has changed to retrieval of XML data in textual XML format, regardless of whether the data server supports binary XML format.

For update of data in XML columns, `xmlFormat` has no effect. If the input data is binary XML data, and the data server does not support binary XML data, the input data is converted to textual XML data. Otherwise, no conversion occurs.

### Changes to the default value of Connection and DataSource property `useCachedCursor`

For connections to Db2 on Linux, UNIX, and Windows systems, the default value of the `useCachedCursor` property has changed.

The default is:

- `false`, if the driver is at one of the following levels:
  - Version 3.67 or 4.17, or later
  - Version 3.64 or 4.14
- `true`, for any other driver versions.

If the driver version is 3.67 or 4.17, or later, or 3.64 or 4.14, and the `deferPrepares` property is set to `true`, the driver behaves as if `useCachedCursor` is set to `false`, regardless of the `useCachedCursor` setting.

### Changes to driver behavior for failure of seamless failover during automatic client reroute

Before versions 3.67 and 4.17 of the IBM Data Server Driver for JDBC and SQLJ, seamless failover error behavior was that after 10 attempts to reconnect to a failover server and execute the SQL statement that failed previously, the driver issued an `SQLException` with SQL error code `-20542`.

Starting with versions 3.67 and 4.17 of the IBM Data Server Driver for JDBC and SQLJ, seamless failover error behavior is that after one attempt to reconnect to a failover server and execute the SQL statement that failed previously, the driver issues an `SQLException` with SQL error code `-4228`.

---

**Table:**

<table>
<thead>
<tr>
<th>Client into property</th>
<th>Default value before versions 3.64 and 4.14</th>
<th>Default value for versions 3.64 and 4.14 or later</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClientUser</td>
<td>Empty string</td>
<td>The user ID that was specified for the connection. If no user ID was specified, the RACF user ID is used.</td>
</tr>
</tbody>
</table>
Changes to trace polling default

The default for db2.jcc.tracePolling is false before Version 3.69 of IBM Data Server Driver for JDBC and SQLJ, and the default is true for Version 3.69 and later.

Related concepts:
"Examples of ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnLabel values" on page 487

IBM Netezza connection through the IBM Data Server Driver for JDBC and SQLJ

IBM Netezza® clients can use the IBM Data Server Driver for JDBC and SQLJ to connect to a database server. Minimal client-side configuration is needed when switching from an nzjdbc3 driver to the IBM Data Server Driver for JDBC and SQLJ.

The IBM Data Server Driver for JDBC and SQLJ readily accepts Netezza configuration settings:

- The Class.forName method will accept the Netezza driver:
  ```java
  Class.forName("org.netezza.Driver");
  ```
- The JDBC Connection URL will accept the "netezza" subprotocol:
  ```java
  jdbc:netezza://host:port/databaseName
  ```
- You can use the org.netezza.datasource.NzDataSource class to construct a DataSource for acquiring connections to the server.

The IBM Data Server Driver for JDBC and SQLJ uses a default port number of 50,000 when connecting an IBM Netezza client.

Compatible properties

Certain Netezza driver properties can be used to establish connection settings with the IBM Data Server Driver for JDBC and SQLJ. The following Netezza driver properties will be recognized by the IBM Data Server Driver for JDBC and SQLJ. They can be used in place of corresponding IBM Data Server Driver for JDBC and SQLJ properties with similar functionality if specified in camel case, all lowercase, or all uppercase letters.

Table 97. The following table lists several common Netezza driver properties that can be used in Netezza client applications instead of similar IBM Data Server Driver for JDBC and SQLJ properties.

<table>
<thead>
<tr>
<th>Netezza Driver Property</th>
<th>IBM Data Server Driver for JDBC and SQLJ Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>applicationName</td>
<td>clientProgramName</td>
</tr>
<tr>
<td>autocommit</td>
<td>autoCommit</td>
</tr>
<tr>
<td>batchSize</td>
<td>fetchSize</td>
</tr>
<tr>
<td>caCertFile</td>
<td>sslCertLocation</td>
</tr>
<tr>
<td>clientHostName</td>
<td>clientWorkstation</td>
</tr>
<tr>
<td>clientUser</td>
<td>clientUser</td>
</tr>
<tr>
<td>database</td>
<td>databaseName</td>
</tr>
<tr>
<td>description</td>
<td>description</td>
</tr>
<tr>
<td>host</td>
<td>serverName</td>
</tr>
</tbody>
</table>
Table 97. The following table lists several common Netezza driver properties that can be used in Netezza client applications instead of similar IBM Data Server Driver for JDBC and SQLJ properties. (continued)

<table>
<thead>
<tr>
<th>Netezza Driver Property</th>
<th>IBM Data Server Driver for JDBC and SQLJ Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>logDirPath</td>
<td>traceDirectory</td>
</tr>
<tr>
<td>loginTimeout</td>
<td>loginTimeout</td>
</tr>
<tr>
<td>name</td>
<td>dataSourceName</td>
</tr>
<tr>
<td>password</td>
<td>password</td>
</tr>
<tr>
<td>port</td>
<td>portNumber</td>
</tr>
<tr>
<td>readonly</td>
<td>readOnly</td>
</tr>
<tr>
<td>schema</td>
<td>currentSchema</td>
</tr>
<tr>
<td>securityLevel</td>
<td>sslConnection</td>
</tr>
</tbody>
</table>

**Security specifications**

Set the Netezza securityLevel property to "preferredSecured" or "onlySecured" if you would like to enable SSL connections in the application. Any other value for securityLevel will disable SSL connections.

**Incompatibilities**

- The IBM Data Server Driver for JDBC and SQLJ will not throw an error if you use a Netezza driver property other than the aforementioned compatible properties.
- The IBM Data Server Driver for JDBC and SQLJ will use empty functions in the case of unsupported application programming interfaces.
- The Netezza logLevel property is not supported with the IBM Data Server Driver for JDBC and SQLJ. Implement a driver logging mechanism to evaluate IBM Data Server Driver for JDBC and SQLJ traceLevel settings to learn which trace settings best match your needs.

**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:

```sql
CREATE TABLE MYTABLE(INTCOL INT)
```

**Example:** The following query contains an AS CLAUSE, which defines a label for a column in the result set:

```sql
SELECT MYCOL AS MYLABEL FROM MYTABLE
```
The following table lists the `ResultSetMetaData.getColumnName` and `ResultSetMetaData.getColumnLabel` values that are returned for the query:

### Table 98. `ResultSetMetaData.getColumnName` and `ResultSetMetaData.getColumnLabel` before and after IBM Data Server Driver for JDBC and SQLJ Version 4.0 for a query with an AS CLAUSE

<table>
<thead>
<tr>
<th>Target data source</th>
<th>Behavior before IBM Data Server Driver for JDBC and SQLJ Version 4.0</th>
<th>Behavior for IBM Data Server Driver for JDBC and SQLJ Version 4.0 and later</th>
</tr>
</thead>
<tbody>
<tr>
<td>getColumnName</td>
<td>getColumnLabel</td>
<td>getColumnName</td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>Db2 on Linux, UNIX, and Windows systems</td>
<td>MYLABEL</td>
<td>MYLABEL</td>
</tr>
<tr>
<td>IBM Informix</td>
<td>MYLABEL</td>
<td>MYLABEL</td>
</tr>
<tr>
<td>DB2 for z/OS Version 8 or later, and DB2 for IBM i V5R3 and later</td>
<td>MYLABEL</td>
<td>MYLABEL</td>
</tr>
<tr>
<td>DB2 for z/OS Version 7, and DB2 for IBM i V5R2</td>
<td>MYLABEL</td>
<td>MYLABEL</td>
</tr>
</tbody>
</table>

**Example**: The following query contains no AS clause:

```sql
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 IBM i data source, a `LABEL ON` statement is used to define a label for a column:

```sql
LABEL ON COLUMN MYTABLE.MYCOL IS 'LABELONCOL'
```

The following query contains an AS CLAUSE, which defines a label for a column in the ResultSet:

```sql
SELECT MYCOL AS MYLABEL FROM MYTABLE
```

The following table lists the `ResultSetMetaData.getColumnName` and `ResultSetMetaData.getColumnLabel` values that are returned for the query:

### Table 99. `ResultSetMetaData.getColumnName` and `ResultSetMetaData.getColumnLabel` 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

<table>
<thead>
<tr>
<th>Target data source</th>
<th>Behavior before IBM Data Server Driver for JDBC and SQLJ Version 4.0</th>
<th>Behavior for IBM Data Server Driver for JDBC and SQLJ Version 4.0 and later</th>
</tr>
</thead>
<tbody>
<tr>
<td>getColumnName</td>
<td>getColumnLabel</td>
<td>getColumnName</td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>DB2 for z/OS Version 8 or later, and DB2 for IBM i V5R3 and later</td>
<td>MYLABEL</td>
<td>LABELONCOL</td>
</tr>
<tr>
<td>DB2 for z/OS Version 7, and DB2 for IBM i V5R2</td>
<td>MYLABEL</td>
<td>LABELONCOL</td>
</tr>
</tbody>
</table>

**Example**: On a DB2 for z/OS or Db2 for IBM i data source, a `LABEL ON` statement is used to define a label for a column:
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 100. 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

<table>
<thead>
<tr>
<th>Target data source</th>
<th>Behavior before IBM Data Server Driver for JDBC and SQLJ Version 4.0</th>
<th>Behavior for IBM Data Server Driver for JDBC and SQLJ Version 4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>getColumnLabel value</td>
<td>getColumnLabel value</td>
</tr>
<tr>
<td>DB2 for z/OS Version 8 or later, and Db2 for IBM i V5R3 and later</td>
<td>MYCOL</td>
<td>LABELONCOL</td>
</tr>
<tr>
<td>DB2 for z/OS Version 7, and Db2 for IBM i V5R2</td>
<td>MYCOL</td>
<td>MYLABEL</td>
</tr>
</tbody>
</table>

**Error codes issued by the IBM Data Server Driver for JDBC and SQLJ**

Warning codes in the ranges +4200 to +4299, and +4450 to +4499, and error codes in the ranges -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=
```

The IBM Data Server Driver for JDBC and SQLJ issues the following warning codes:

**+4204**  **Message text:** 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 on Linux, UNIX, and Windows systems servers. It is issued only when a cursor operation, such as a ResultSet.next or ResultSet.previous call, returns false.

**SQLSTATE:** 02506

**+4222**  **Message text:** 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 Message text: text-from-getMessage

Explanation: A warning condition occurred during initialization.

User response: Call SQLException.getMessage to retrieve specific information about the problem.

+4225 Message text and explanation: 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 Message text and explanation: 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 Message text: 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 Message text: Feature not supported: feature-name

+4460 Message text: 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 Message text: 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 Message text: text-from-getMessage

Explanation: A required value is missing.

User response: Call SQLException.getMessage to retrieve specific information about the problem.

+4470 Message text: 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 Message text: 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.

**Message text**: `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.

**Message text**: `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.

The IBM Data Server Driver for JDBC and SQLJ issues the following error codes:

**-1224 Message text**: The database manager is not able to accept new requests, has terminated all requests in progress, or has terminated the specified request because of an error or a forced interrupt.

**Explanation**: For connections to Db2 on Linux, UNIX, and Windows systems data servers, see SQL1224N for details.

For connections to Db2 for z/OS data servers, this error indicates the server thread that is associated with the application abnormally terminated. Server diagnostics that are related to the application need to be collected. The application can be identified by its unique application ID. Db2 for z/OS externalizes the application ID in messages and traces as the connection correlation token (CRRTKN) and logical unit of work ID (LUWID). Message DSNL027I is generated on the z/OS console when a Db2 for z/OS thread abnormally terminates. DSNL027I provides a reason code for the failure. In most cases, Db2 for z/OS generates a z/OS SVC dump, which is needed to solve the problem.

**SQLSTATE**: 2D521

**-4200 Message text**: 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.

**SQLSTATE**: 2D521

**-4201 Message text**: 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.

**SQLSTATE**: 2D521

**-4203 Message text**: Error executing function. Server returned `rc`. 

Chapter 7. JDBC and SQLJ reference information 491
**Explanation:** 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** **Message text:** Timeout getting a transport object from pool.
  **SQLSTATE:** 57033

- **4211** **Message text:** Timeout getting an object from pool.
  **SQLSTATE:** 57033

- **4212** **Message text:** Sysplex member unavailable.
- **4213** **Message text:** Timeout.
  **SQLSTATE:** 57033

- **4214** **Message text:** text-from-getMessage
  **Explanation:** Authorization failed.
  **User response:** Call SQLException.getMessage to retrieve specific information about the problem.
  **SQLSTATE:** 28000

- **4220** **Message text:** text-from-getMessage
  **Explanation:** An error occurred during character conversion.
  **User response:** Call SQLException.getMessage to retrieve specific information about the problem.

- **4221** **Message text:** text-from-getMessage
  **Explanation:** An error occurred during encryption or decryption.
  **User response:** Call SQLException.getMessage to retrieve specific information about the problem.

- **4222** **Message text:** 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** **Message text:** text-from-getMessage
  **Explanation:** An error occurred during initialization.
  **User response:** Call SQLException.getMessage to retrieve specific information about the problem.

- **4224** **Message text:** text-from-getMessage
  **Explanation:** An error occurred during resource cleanup.
  **User response:** Call SQLException.getMessage to retrieve specific information about the problem.

- **4225** **Message text:** 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 Message text: text-from-getMessage
Explanation: An error occurred during customization or bind.
User response: Call SQLException.getMessage to retrieve specific information about the problem.

-4227 Message text: text-from-getMessage
Explanation: An error occurred during reset.
User response: Call SQLException.getMessage to retrieve specific information about the problem.

-4228 Message text: 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 Message text: text-from-getMessage
Explanation: An error occurred during a batch execution.
User response: Call SQLException.getMessage to retrieve specific information about the problem.

-4231 Message text: 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 Message text: Feature not supported: feature-name
SQLSTATE: 0A504

-4460 Message text: 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 Message text: 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.

SQLSTATE: 42815

-4462 Message text: text-from-getMessage
Explanation: A required value is missing.
User response: Call SQLException.getMessage to retrieve specific information about the problem.

-4463 Message text: text-from-getMessage
Explanation: The specified value has a syntax error.
User response: Call SQLException.getMessage to retrieve specific information about the problem.

SQLSTATE: 42601
-4470 Message text: 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 Message text: 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 Message text: 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 Message text: 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 Message text: 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.

-4475 Message text: 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 Message text: 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 Message text: 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 Message text: 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 Message text: A connection failed but has been re-established. Special
register settings have been replayed if necessary. Host name or IP address of the connection: *host-name*. Service name or port number of the connection: *service-name*. Reason code: *reason-code*. Failure code: *failure-code*. Error code: *error-code*.

**Explanation:** The connection has been reestablished. In some cases, the network connection or transport to the server is not established until the next use. After the connection is reestablished, all session resources are set to their initial default values. The application is rolled back to the previous commit point. The reason code indicates which special register values are applied to the new connection. Possible values for the reason code are:

1. All special register settings were returned to their values at the point of failure. The connection was reestablished within the current group.
2. All special register settings were returned to their values at the previous commit point. The connection was reestablished within the current group.
3. All special registers were returned to their settings at the point of failure. The connection was reestablished in a new group.
4. All special register settings were returned to their values at the previous commit point. The connection was reestablished in a new group.

*failure-code* indicates the error that caused the connection to fail:

1. A communication failure occurred.
2. The data server closed the connection.
3. An SQL error occurred.
4. The client closed the connection.

*error-code* depends on the value of *failure-code*:

**Failure code: 1 or 2**

**Error code:** The Java SocketException message that was returned.

**Failure code: 3**

**Error code:** The SQL error code that was returned by the SQL statement that caused the connection to fail.

**Failure code: 4**

**Error code:** The following value:

2. The driver received an interrupt or cancel request.

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 **Message text:** 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.
SQLSTATE: 08001 or 58009

-20542 Message text: The statement was not executed because the connection to the database server was dropped, and the automatic client reroute (ACR) feature failed to successfully re-execute the statement.

Explanation: See [SQL20542N](#)

User response: Call SQLException.getMessage to retrieve specific information about the problem.

SQLSTATE: 54068


Explanation: See [SQL30108N](#)

User response: Call SQLException.getMessage to retrieve specific information about the problem.

SQLSTATE: 08506

-99999 Message text: The IBM Data Server Driver for JDBC and SQLJ issued an error that does not yet have an error code.

Related tasks:
- “Handling SQL errors in an SQLJ application” on page 185
- “Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ” on page 116

### 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 101. SQLSTATEs returned by the IBM Data Server Driver for JDBC and SQLJ**

<table>
<thead>
<tr>
<th>SQLSTATE class</th>
<th>SQLSTATE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01xxx</td>
<td>Warning</td>
<td></td>
</tr>
<tr>
<td>02xxx</td>
<td>No data</td>
<td></td>
</tr>
<tr>
<td>02xxx</td>
<td>02501</td>
<td>The cursor position is not valid for a FETCH of the current row.</td>
</tr>
<tr>
<td>02xxx</td>
<td>02506</td>
<td>Tolerable error</td>
</tr>
<tr>
<td>08xxx</td>
<td>Connection exception</td>
<td></td>
</tr>
<tr>
<td>08xxx</td>
<td>08001</td>
<td>The application requester is unable to establish the connection.</td>
</tr>
<tr>
<td>08xxx</td>
<td>08003</td>
<td>A connection does not exist</td>
</tr>
<tr>
<td>08xxx</td>
<td>08004</td>
<td>The application server rejected establishment of the connection</td>
</tr>
<tr>
<td>08xxx</td>
<td>08506</td>
<td>Client reroute exception</td>
</tr>
</tbody>
</table>
Table 101. SQLSTATEs returned by the IBM Data Server Driver for JDBC and SQLJ (continued)

<table>
<thead>
<tr>
<th>SQLSTATE class</th>
<th>SQLSTATE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0Axxx</td>
<td>0A502</td>
<td>The action or operation is not enabled for this database instance</td>
</tr>
<tr>
<td>0Axxx</td>
<td>0A504</td>
<td>The feature is not supported by the driver</td>
</tr>
<tr>
<td>22xxx</td>
<td>22007</td>
<td>The string representation of a datetime value is invalid</td>
</tr>
<tr>
<td>22xxx</td>
<td>22021</td>
<td>A character is not in the coded character set</td>
</tr>
<tr>
<td>23xxx</td>
<td>23502</td>
<td>A value that is inserted into a column or updates a column is null, but the column cannot contain null values.</td>
</tr>
<tr>
<td>24xxx</td>
<td>24501</td>
<td>The identified cursor is not open</td>
</tr>
<tr>
<td>28xxx</td>
<td>28000</td>
<td>Authorization name is invalid.</td>
</tr>
<tr>
<td>2Dxxx</td>
<td>2D521</td>
<td>SQL COMMIT or ROLLBACK are invalid in the current operating environment.</td>
</tr>
<tr>
<td>34xxx</td>
<td>34000</td>
<td>Cursor name is invalid.</td>
</tr>
<tr>
<td>3Bxxx</td>
<td>3B503</td>
<td>A SAVEPOINT, RELEASE SAVEPOINT, or ROLLBACK TO SAVEPOINT statement is not allowed in a trigger or global transaction.</td>
</tr>
<tr>
<td>40xxx</td>
<td></td>
<td>Transaction rollback</td>
</tr>
<tr>
<td>42xxx</td>
<td>42601</td>
<td>A character, token, or clause is invalid or missing</td>
</tr>
<tr>
<td>42xxx</td>
<td>42734</td>
<td>A duplicate parameter name, SQL variable name, cursor name, condition name, or label was detected.</td>
</tr>
<tr>
<td>42xxx</td>
<td>42807</td>
<td>The INSERT, UPDATE, or DELETE is not permitted on this object</td>
</tr>
<tr>
<td>42xxx</td>
<td>42808</td>
<td>A column identified in the insert or update operation is not updateable</td>
</tr>
<tr>
<td>42xxx</td>
<td>42815</td>
<td>The data type, length, scale, value, or CCSID is invalid</td>
</tr>
<tr>
<td>42xxx</td>
<td>42820</td>
<td>A numeric constant is too long, or it has a value that is not within the range of its data type</td>
</tr>
<tr>
<td>42xxx</td>
<td>42968</td>
<td>The connection failed because there is no current software license</td>
</tr>
<tr>
<td>57xxx</td>
<td></td>
<td>Resource not available or operator intervention</td>
</tr>
<tr>
<td>57xxx</td>
<td>57033</td>
<td>A deadlock or timeout occurred without automatic rollback</td>
</tr>
<tr>
<td>58xxx</td>
<td></td>
<td>System error</td>
</tr>
</tbody>
</table>
Table 101. SQLSTATEs returned by the IBM Data Server Driver for JDBC and SQLJ (continued)

<table>
<thead>
<tr>
<th>SQLSTATE class</th>
<th>SQLSTATE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>58xxx</td>
<td>58008</td>
<td>Execution failed due to a distribution protocol error that will not affect the successful execution of subsequent DDM commands or SQL statements</td>
</tr>
<tr>
<td>58xxx</td>
<td>58009</td>
<td>Execution failed due to a distribution protocol error that caused deallocation of the conversation</td>
</tr>
<tr>
<td>58xxx</td>
<td>58012</td>
<td>The bind process with the specified package name and consistency token is not active</td>
</tr>
<tr>
<td>58xxx</td>
<td>58014</td>
<td>The DDM command is not supported</td>
</tr>
<tr>
<td>58xxx</td>
<td>58015</td>
<td>The DDM object is not supported</td>
</tr>
<tr>
<td>58xxx</td>
<td>58016</td>
<td>The DDM parameter is not supported</td>
</tr>
<tr>
<td>58xxx</td>
<td>58017</td>
<td>The DDM parameter value is not supported</td>
</tr>
</tbody>
</table>

Related tasks:
- “Handling SQL errors in an SQLJ application” on page 185
- “Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ” on page 116

Commands for SQLJ program preparation

To prepare SQLJ programs for execution, you use commands to translate SQLJ source code into Java source code, compile the Java source code, create and customize SQLJ serialized profiles, and bind DB2 packages.

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

```text
sqlj -help -dir=directory -d=directory -props=properties-file

-compile=true -linemap=NO -smap=NO -encoding=encoding -db2optimize

-compile=false -linemap=YES -smap=YES
```
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_SIProfileKeys.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 that enables SQLJ context caching in a WebSphere Application Server environment for applications that run against DB2 data servers.

-db2optimize causes a user-defined context to extend a custom driver class, which enables context caching and connection caching in WebSphere Application Server.

Because context caching is enabled by using an instance of IBM Data Server Driver for JDBC and SQLJ class sqlj.runtime.ref.DefaultContext, db2jcc.jar must be in the CLASSPATH when you translate and compile the Java application.

You cannot use connection sharing in WebSphere Application Server if you use context caching.

Important: Context caching that is enabled by the -db2optimize option can provide performance benefits over connection pooling and statement pooling that is provided by WebSphere Application Server. However, context caching can result in significant resource consumption in the application server, and might have unintended side effects if it is not used correctly. For example, if two applications use an SQLJ profile with the same name, they might overwrite each other, because both use sqlj.runtime.ref.DefaultContext. Use context caching only if:

- The system is not storage-constrained.
• Cached statements are often reused on the same Connection.
• All applications have distinct names for their SQLJ profiles.

-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-Xmne2M

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:
If the SQLJ translator invokes the Java compiler, the class files that the compiler generates.

**Examples**

```
sqlj -encoding=UTF8 -C-O MyApp.sqlj
```

**db2sqljcustomize - SQLJ profile customizer**

`db2sqljcustomize` processes an SQLJ profile, which contains embedded SQL statements.

By default, `db2sqljcustomize` produces four DB2 packages: one for each isolation level. `db2sqljcustomize` augments the profile with information specific to DB2 for use at run time.

**Authorization**

The privilege set of the process must include one of the following authorities:

- SYSDMB authority
- DBADM authority
- If the package does not exist, the BINDADD privilege, and one of the following privileges:
  - CREATEIN privilege
  - PACKADM authority on the collection or on all collections
- If the package exists, the BIND privilege on the package

**Command syntax**

```
db2sqljcustomize [options]
```

- `-url` **jdbc:**db2://server/database[:port][;property=value]
- `-datasource` JNDI-name
- `-user` user-ID
- `-password` password
- `-automaticbind` YES | NO
- `-pkgversion` AUTO | version-id
- `-bindoptions"options-string"
- `-storebindoptions`
- `-collection` collection-name
Command parameters

-help
   Specifies that the SQLJ customizer describes each of the options that the
   customizer supports. If any other options are specified with -help, they are
   ignored.

-url
   Specifies the URL for the data server for which the profile is to be bound. The
   URL can be a URL for IBM Data Server Driver for JDBC and SQLJ type 4
   connectivity or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.
   A connection is established to the data source that this URL represents if the
   -automaticbind or -onlinecheck option is specified as YES or defaults to YES.
   The variable parts of the -url value are:
server
The domain name or IP address of the z/OS system on which the DB2 subsystem resides.

port
The TCP/IP server port number that is assigned to the DB2 subsystem. The default is 446.

-url
Specifies the URL for the data source for which the profile is to be customized. A connection is established to the data source that this URL represents if the -automaticbind or -onlinecheck option is specified as YES or defaults to YES. 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.

port
The TCP/IP server port number that is assigned to the database server. The default is 446.

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 SYSPROC.DUMMY;
```

If the connection is to a Db2 on Linux, UNIX, and Windows systems server, database is the database name that is defined during installation.

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 for the JDBC connection.

property=value;
A property for the JDBC connection.

-datasource JNDI-name
Specifies the logical name of a DataSource object that was registered with JNDI. The DataSource object represents the data source for which the profile is to be customized. A connection is established to the data source if the -automaticbind or -onlinecheck option is specified as YES or defaults to YES. Specifying -datasource is an alternative to specifying -url. The DataSource object must represent a connection that uses IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

-user user-ID
Specifies the user ID to be used to connect to the data source for online checking or binding a package. You must specify -user if you specify -url. You must specify -user if you specify -datasource, and the DataSource object that JNDI-name represents does not contain a user ID.
-password password
Specifies the password to be used to connect to the data source for online checking or binding a package. You must specify -password if you specify -url. You must specify -password if you specify -datasource, and the DataSource object that JNDI-name represents does not contain a password.

-automaticbind YES|NO
Specifies whether the customizer binds DB2 packages at the data source that is specified by the -url parameter.
The default is YES.
The number of packages and the isolation levels of those packages are controlled by the -rootpkgname and -singlepkgname options.
Before the bind operation can work, the following conditions need to be met:
• TCP/IP and DRDA must be installed at the target data source.
• Valid -url, -username, and -password values must be specified.
• The -username value must have authorization to bind a package at the target data source.

-pkgversion AUTO|version-id
Specifies the package version that is to be used when packages are bound at the server for the serialized profile that is being customized. db2sqljcustomize stores the version ID in the serialized profile and in the DB2 package. Run-time version verification is based on the consistency token, not the version name. To automatically generate a version name that is based on the consistency token, specify -pkgversion AUTO.
The default is that there is no version.

-bindoptions options-string
Specifies a list of options, separated by spaces. These options have the same function as DB2 precompile and bind options with the same names. If you are preparing your program to run on a DB2 for z/OS system, specify z/OS options. If you are preparing your program to run on a Db2 on Linux, UNIX, and Windows systems system, specify Db2 on Linux, UNIX, and Windows systems options.

Notes on bind options:
• Specify ISOLATION only if you also specify the -singlepkgname option.

Important: Specify only those program preparation options that are appropriate for the data source at which you are binding a package. Some values and defaults for the IBM Data Server Driver for JDBC and SQLJ are different from the values and defaults for DB2.

-storebindoptions
Specifies that values for the -bindoptions and -staticpositioned parameters are stored in the serialized profile. If db2sqljbind is invoked without the -bindoptions or -staticpositioned parameter, the values that are stored in the serialized profile are used during the bind operation. When multiple serialized profiles are specified for one invocation of db2sqljcustomize, the parameter values are stored in each serialized profile. The stored values are displayed in the output from the db2sqljprint utility.

-collection collection-name
The qualifier for the packages that db2sqljcustomize binds. db2sqljcustomize stores this value in the customized serialized profile, and it is used when the
associated packages are bound. If you do not specify this parameter, db2sqljcustomize uses a collection ID of NULLID.

**-onlinecheck YES|NO**
Specifies whether online checking of data types in the SQLJ program is to be performed. The -url or -datasource option determines the data source that is to be used for online checking. The default is YES if the -url or -datasource parameter is specified. Otherwise, the default is NO.

**-qualifier qualifier-name**
Specifies the qualifier that is to be used for unqualified objects in the SQLJ program during online checking. This value is not used as the qualifier when the packages are bound.

**-rootpkgname|-singlepkgname**
Specifies the names for the packages that are associated with the program. If -automaticbind is NO, these package names are used when db2sqljbind runs. The meanings of the parameters are:

**-rootpkgname package-name-stem**
Specifies that the customizer creates four packages, one for each of the four DB2 isolation levels. The names for the four packages are:

- `package-name-stem1`
  For isolation level UR
- `package-name-stem2`
  For isolation level CS
- `package-name-stem3`
  For isolation level RS
- `package-name-stem4`
  For isolation level RR

If -longpkgname is not specified, `package-name-stem` must be an alphanumeric string of seven or fewer bytes.

If -longpkgname is specified, `package-name-stem` must be an alphanumeric string of 127 or fewer bytes.

**-singlepkgname package-name**
Specifies that the customizer creates one package, with the name `package-name`. If you specify this option, your program can run at only one isolation level. You specify the isolation level for the package by specifying the ISOLATION option in the -bindoptions options string.

If -longpkgname is not specified, `package-name` must be an alphanumeric string of eight or fewer bytes.

If -longpkgname is specified, `package-name` must be an alphanumeric string of 128 or fewer bytes.

Using the -singlepkgname option is not recommended.

**Recommendation:** If the target data source is DB2 for z/OS, use uppercase characters for the `package-name-stem` or `package-name` value. DB2 for z/OS systems that are defined with certain CCSID values cannot tolerate lowercase characters in package names or collection names.

If you do not specify -rootpkgname or -singlepkgname, db2sqljcustomize generates four package names that are based on the serialized profile name. A serialized profile name is of the following form:
program-name_SJProfileIDNumber.ser

The four generated package names are of the following form:
*Bytes-from-program-nameIDNumberPkgIsolation*

Table 102 shows the parts of a generated package name and the number of bytes for each part.

The maximum length of a package name is *maxlen*. *maxlen* is 8 if -longpkgname is not specified. *maxlen* is 128 if -longpkgname is specified.

<table>
<thead>
<tr>
<th>Package name part</th>
<th>Number of bytes</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes-from-program-name</td>
<td>$m=\min(\text{Length}(\text{program-name}), maxlen-1-\text{Length}(\text{IDNumber}))$</td>
<td>First $m$ bytes of program-name, in uppercase</td>
</tr>
<tr>
<td>IDNumber</td>
<td>Length(IDNumber)</td>
<td>IDNumber</td>
</tr>
<tr>
<td>PkgIsolation</td>
<td>1</td>
<td>1, 2, 3, or 4. This value represents the transaction isolation level for the package. See Table 103</td>
</tr>
</tbody>
</table>

Table 103 shows the values of the PkgIsolation portion of a package name that is generated by db2sqljcustomize.

<table>
<thead>
<tr>
<th>PkgNumber value</th>
<th>Isolation level for package</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Uncommitted read (UR)</td>
</tr>
<tr>
<td>2</td>
<td>Cursor stability (CS)</td>
</tr>
<tr>
<td>3</td>
<td>Read stability (RS)</td>
</tr>
<tr>
<td>4</td>
<td>Repeatable read (RR)</td>
</tr>
</tbody>
</table>

Example: Suppose that a profile name is ThisIsMyProg_SJProfile111.ser. The db2sqljcustomize option -longpkgname is not specified. Therefore, *Bytes-from-program-name* is the first four bytes of ThisIsMyProg, translated to uppercase, or THIS. IDNumber is 111. The four package names are:

THIS1111
THIS1112
THIS1113
THIS1114

Example: Suppose that a profile name is ThisIsMyProg_SJProfile111.ser. The db2sqljcustomize option -longpkgname is specified. Therefore, *Bytes-from-program-name* is ThisIsMyProg, translated to uppercase, or THISISMYPROG. IDNumber is 111. The four package names are:

THISISMYPROG1111
THISISMYPROG1112
THISISMYPROG1113
THISISMYPROG1114

Example: Suppose that a profile name is A_SJProfile0.ser. *Bytes-from-program-name* is A. IDNumber is 0. Therefore, the four package names are:

A01
A02
A03
A04
Letting db2sqljcustomize generate package names is not recommended. If any generated package names are the same as the names of existing packages, db2sqljcustomize overwrites the existing packages. To ensure uniqueness of package names, specify -rootpkgname.

-longpkgname
Specifies that the names of the DB2 packages that db2sqljcustomize generates can be up to 128 bytes. Use this option only if you are binding packages at a server that supports long package names. If you specify -singlepkgname or -rootpkgname, you must also specify -longpkgname under the following conditions:

- The argument of -singlepkgname is longer than eight bytes.
- The argument of -rootpkgname is longer than seven bytes.

-staticpositioned NO|YES
For iterators that are declared in the same source file as positioned UPDATE statements that use the iterators, specifies whether the positioned UPDATEs are executed as statically bound statements. The default is NO. NO means that the positioned UPDATEs are executed as dynamically prepared statements.

-zosDescProcParms
Specifies that db2sqljcustomize queries the DB2 catalog at the target database source to determine the SQL parameter data types that correspond to the host variables in CALL statements.

-zosDescProcParms applies only to programs that run on DB2 for z/OS data servers.

If -zosDescProcParms is specified, and the authorization ID under which db2sqljcustomize runs does not have read access to the SYSIBM.SYSROUTINES catalog table, db2sqljcustomize returns an error and uses the host variable data types in the CALL statements to determine the SQL data types.

Specification of -zosDescProcParms can lead to more efficient storage usage at run time. If SQL data type information is available, SQLJ has information about the length and precision of INOUT and OUT parameters, so it allocates only the amount of memory that is needed for those parameters. Availability of SQL data type information can have the biggest impact on storage usage for character INOUT parameters, LOB OUT parameters, and decimal OUT parameters.

When -zosDescProcParms is specified, the DB2 data server uses the specified or default value of -zosProcedurePath to resolve unqualified names of stored procedures for which SQL data type information is requested.

If -zosDescProcParms is not specified, db2sqljcustomize uses the host variable data types in the CALL statements to determine the SQL data types. If db2sqljcustomize determines the wrong SQL data type, an SQL error might occur at run time. For example, if the Java host variable type is String, and the corresponding stored procedure parameter type is VARCHAR FOR BIT DATA, an SQL run-time error such as -4220 might occur.

-zosProcedurePath procedure-path
Specifies a list of schema names that DB2 for z/OS uses to resolve unqualified stored procedure names during online checking of an SQLJ program.

-zosProcedurePath applies to programs that are to be run on DB2 for z/OS database servers only.
The list is a String value that is a comma-separated list of schema names that is enclosed in double quotation marks. The DB2 database server inserts that list into the SQL path for resolution of unqualified stored procedure names. The SQL path is:
SYSIBM, SYSFUN, SYSPROC, procedure-path, qualifier-name, user-ID

**qualifier-name** is the value of the -qualifier parameter, and **user-ID** is the value of the -user parameter.

The DB2 data server tries the schema names in the SQL path from left to right until it finds a match with the name of a stored procedure that exists on that database server. If the data server finds a match, it obtains the information about the parameters for that stored procedure from the DB2 catalog. If the DB2 data server does not find a match, SQLJ sets the parameter data without any DB2 catalog information.

If -zosProcedurePath is not specified, the DB2 data server uses this SQL path:
SYSIBM, SYSFUN, SYSPROC, qualifier-name, user-ID

If the -qualifier parameter is not specified, the SQL path does not include qualifier-name.

**-genDBRM**
Specifies that db2sqljcustomize generates database request modules (DBRMs). Those DBRMs can be used to create DB2 for z/OS packages.

- **-genDBRM** applies to programs that are to be run on DB2 for z/OS database servers only.

If -genDBRM and -automaticbind NO are specified, db2sqljcustomize creates the DBRMs but does not bind them into DB2 packages. If -genDBRM and -automaticbind YES are specified, db2sqljcustomize creates the DBRMs and binds them into DB2 packages. One DBRM is created for each DB2 isolation level. The naming convention for the generated DBRM files is the same as the naming convention for packages. For example, if -rootpkgnname SQLJSA0 is specified, and -genDBRM is also specified, the names of the four DBRM files are:
- SQLJSA01
- SQLJSA02
- SQLJSA03
- SQLJSA04

**-hostLangJAVA**
When -genDBRM is specified, the **-hostLangJAVA** option specifies that the db2sqljcustomize process should create DBRM files that identify their host language as 'J'. When these DBRMs are bound into DB2 packages, the corresponding HOSTLANG column in the SYSIBM.SYSPACKAGE table is set to 'J'. This identifies packages that were generated with the Java language. This option is only available for DB2 for z/OS v11 and above.

**-DBRMDir directory-name**
When -genDBRM is specified, -DBRMDir specifies the local directory into which db2sqljcustomize puts the generated DBRM files. The default is the current directory.

-DBRMDir applies to programs that are to be run on DB2 for z/OS database servers only.
-tracefile file-name
  Enables tracing and identifies the output file for trace information. This option should be specified only under the direction of IBM Software Support.

-tracelevel
  If -tracefile is specified, indicates what to trace while db2sqljcustomize runs. The default is TRACE_SQLJ. This option should be specified only under the direction of IBM Software Support.

serialized-profile-name=file-name.grp
  Specifies the names of one or more serialized profiles that are to be customized. The specified serialized profile must be in a directory that is named in the CLASSPATH environment variable.

A serialized profile name is of the following form:
  program-name_SJProfileIDNumber.ser

You can specify the serialized profile name with or without the .ser extension.

program-name is the name of the SQLJ source program, without the extension .sqlj. n is an integer between 0 and m-1, where m is the number of serialized profiles that the SQLJ translator generated from the SQLJ source program.

You can specify serialized profile names in one of the following ways:
  • List the names in the db2sqljcustomize command. Multiple serialized profile names must be separated by spaces.
  • Specify the serialized profile names, one on each line, in a file with the name file-name.grp, and specify file-name.grp in the db2sqljcustomize command.

If you specify more than one serialized profile name, and if you specify or use the default value of -automaticbind YES, db2sqljcustomize binds a single DB2 package from the profiles. When you use db2sqljcustomize to create a single DB2 package from multiple serialized profiles, you must also specify the -rootpkgnamename or -singlepkgnamename option.

If you specify more than one serialized profile name, and you specify -automaticbind NO, if you want to bind the serialized profiles into a single DB2 package when you run db2sqljbind, you need to specify the same list of serialized profile names, in the same order, in db2sqljcustomize and db2sqljbind.

If you specify one or more file-name.grp files, and you specify -automaticbind NO, when you run db2sqljbind, you must specify that same list of files, and in the same order in which the files were customized.

You cannot run db2sqljcustomize on individual files, and then group those files when you run db2sqljbind.

Output

When db2sqljcustomize runs, it creates a customized serialized profile. It also creates DB2 packages, if the automaticbind value is YES.

Examples

db2sqljcustomize -user richler -password mordecai
-url jdbc:db2:/server:50000/sample -collection duddy
-bindoptions "EXPLAIN YES" pgmname_SJProfile0.ser

The following example demonstrates passing the hostLangJAVA option to the customizer, and binding a package on a DB2 for z/OS server:
**Usage notes**

*Online checking is always recommended:* It is highly recommended that you use online checking when you customize your serialized profiles. Online checking determines information about the data types and lengths of DB2 host variables, and is especially important for the following items:

- **Predicates with `java.lang.String` host variables and `CHAR` columns**
  Unlike character variables in other host languages, Java `String` host variables are not declared with a length attribute. To optimize a query properly that contains character host variables, DB2 needs the length of the host variables. For example, suppose that a query has a predicate in which a `String` host variable is compared to a `CHAR` column, and an index is defined on the `CHAR` column. If DB2 cannot determine the length of the host variable, it might do a table space scan instead of an index scan. Online checking avoids this problem by providing the lengths of the corresponding character columns.

- **Predicates with `java.lang.String` host variables and `GRAPHIC` columns**
  Without online checking, DB2 might issue a bind error (SQLCODE -134) when it encounters a predicate in which a `String` host variable is compared to a `GRAPHIC` column.

- **Column names in the result table of an SQLJ SELECT statement at a remote server:**
  Without online checking, the driver cannot determine the column names for the result table of a remote SELECT.

*Customizing multiple serialized profiles together:* Multiple serialized profiles can be customized together to create a single DB2 package. If you do this, and if you specify `-staticpositioned YES`, any positioned UPDATE or DELETE statement that references a cursor that is declared *earlier in the package* executes statically, even if the UPDATE or DELETE statement is in a different source file from the cursor declaration. If you want `-staticpositioned YES` behavior when your program consists of multiple source files, you need to order the profiles in the `db2sqljcustomize` command to cause cursor declarations to be ahead of positioned UPDATE or DELETE statements in the package. To do that, list profiles that contain SELECT statements that assign result tables to iterators before profiles that contain the positioned UPDATE or DELETE statements that reference those iterators.

*Using a customized serialized profile at one data source that was customized at another data source:* You can run `db2sqljcustomize` to produce a customized serialized profile for an SQLJ program at one data source, and then use that profile at another data source. You do this by running `db2sqljbind` multiple times on customized serialized profiles that you created by running `db2sqljcustomize` once. When you run the programs at these data sources, the DB2 objects that the programs access must be identical at every data source. For example, tables at all data sources must have the same encoding schemes and the same columns with the same data types.

*Using the `-collection` parameter:* `db2sqljcustomize` stores the DB2 collection name in each customized serialized profile that it produces. When an SQLJ program is executed, the driver uses the collection name that is stored in the customized serialized profile to search for packages to execute. The name that is stored in the
customized serialized profile is determined by the value of the -collection parameter. Only one collection ID can be stored in the serialized profile. However, you can bind the same serialized profile into multiple package collections by specifying the COLLECTION option in the -bindoptions parameter. To execute a package that is in a collection other than the collection that is specified in the serialized profile, include a SET CURRENT PACKAGESET statement in the program.

**Using the VERSION parameter:** Use the VERSION parameter to bind two or more versions of a package for the same SQLJ program into the same collection. You might do this if you have changed an SQLJ source program, and you want to run the old and new versions of the program.

To maintain two versions of a package, follow these steps:
1. Change the code in your source program.
2. Translate the source program to create a new serialized profile. Ensure that you do not overwrite your original serialized profile.
3. Run `db2sqljcustomize` to customize the serialized profile and create DB2 packages with the same package names and in the same collection as the original packages. Do this by using the same values for -rootpkgname and -collection when you bind the new packages that you used when you created the original packages. Specify the VERSION option in the -bindoptions parameter to put a version ID in the new customized serialized profile and in the new packages.

   It is essential that you specify the VERSION option when you perform this step. If you do not, you overwrite your original packages.

When you run the old version of the program, DB2 loads the old versions of the packages. When you run the new version of the program, DB2 loads the new versions of the packages.

**Binding packages and plans on DB2 for z/OS:** You can use the `db2sqljcustomize` -genDBRM parameter to create DBRMs on your local system. You can then transfer those DBRMs to a DB2 for z/OS system, and bind them into packages there. If you plan to use this technique, you need to transfer the DBRM files to the z/OS system as binary files, to a partitioned data set with record format FB and record length 80. When you bind the packages, you need to specify the following bind option values:

- **ENCODING(EBCDIC)**
  - The IBM Data Server Driver for JDBC and SQLJ on DB2 for z/OS requires EBCDIC encoding for your packages.

- **DYNAMICRULES(BIND)**
  - This option ensures consistent authorization rules when SQLJ uses dynamic SQL. SQLJ uses dynamic SQL for positioned UPDATE or DELETE operations that involve multiple SQLJ programs.

**EXTENDEDINDICATOR bind option behavior:** If the EXTENDEDINDICATOR bind option is not specified in the -bindoptions options string, and the target data server supports extended indicators, bind operations use EXTENDEDINDICATOR(YES). If EXTENDEDINDICATOR(NO) is explicitly specified, and the application contains extended indicator syntax, unexpected behavior can occur because the IBM Data Server Driver for JDBC and SQLJ treats extended indicators as NULL values.

**Related reference:**

512  Application Programming Guide and Reference for Java
**db2sqljbind - SQLJ profile binder**

db2sqljbind binds DB2 packages for a serialized profile that was previously customized with the db2sqljcustomize command.

Applications that run with the IBM Data Server Driver for JDBC and SQLJ require packages but no plans. If the db2sqljcustomize -automaticbind option is specified as YES or defaults to YES, db2sqljcustomize binds packages for you at the data source that you specify in the -url parameter. However, if -automaticbind is NO, if a bind fails when db2sqljcustomize runs, or if you want to create identical packages at multiple locations for the same serialized profile, you can use the db2sqljbind command to bind packages.

**Authorization**

The privilege set of the process must include one of the following authorities:
- SYSADM authority
- DBADM authority
- If the package does not exist, the BINDADD privilege, and one of the following privileges:
  - CREATEIN privilege
  - PACKADM authority on the collection or on all collections
- If the package exists, the BIND privilege on the package

**Command syntax**

```
db2sqljbind -help
```

```
db2sqljbind -url jdbc:db2://server/database:port;property=value;
```

```
db2sqljbind -user user-ID
```

```
db2sqljbind -password password
```

```
db2sqljbind -bindoptions "options-string"
```

```
db2sqljbind -staticpositioned NO
```

```
db2sqljbind -staticpositioned YES
```

```
db2sqljbind -genDBRM
```

```
db2sqljbind -DBRMDir directory-name
```
Command parameters

-help
Specifies that db2sqljbind describes each of the options that it supports. If any other options are specified with -help, they are ignored.

-url
Specifies the URL for the data server for which the profile is to be bound. The URL can be a URL for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity or IBM Data Server Driver for JDBC and SQLJ type 2 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.

port
The TCP/IP server port number that is assigned to the database server. The default is 446.

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:

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

If the connection is to a Db2 on Linux, UNIX, and Windows systems server, database is the database name that is defined during installation.
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 for the JDBC connection.

**-datasource JNDI-name**
Specifies the logical name of a DataSource object that was registered with JNDI. The DataSource object represents the data source for which the profile is to be customized. A connection is established to the data source if the -automaticbind or -onlinecheck option is specified as YES or defaults to YES. Specifying -datasource is an alternative to specifying -url. The DataSource object must represent a connection that uses IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

**-user user-ID**
Specifies the user ID to be used to connect to the data source for binding the package.

**-password password**
Specifies the password to be used to connect to the data source for binding the package.

**-bindoptions options-string**
Specifies a list of options, separated by spaces. These options have the same function as DB2 precompile and bind options with the same names. If you are preparing your program to run on a DB2 for z/OS system, specify z/OS options. If you are preparing your program to run on a Db2 on Linux, UNIX, and Windows systems system, specify Db2 on Linux, UNIX, and Windows systems options.

**Notes on bind options:**
- Specify VERSION only if the following conditions are true:
  - If you are binding a package at a Db2 on Linux, UNIX, and Windows systems system, the system is at Version 8 or later.
  - You rerun the translator on a program before you bind the associated package with a new VERSION value.

**Important:** Specify only those program preparation options that are appropriate for the data source at which you are binding a package. Some values and defaults for the IBM Data Server Driver for JDBC and SQLJ are different from the values and defaults for DB2.

**-staticpositioned NO|YES**
For iterators that are declared in the same source file as positioned UPDATE statements that use the iterators, specifies whether the positioned UPDATEs are executed as statically bound statements. The default is NO. NO means that the positioned UPDATEs are executed as dynamically prepared statements. This value must be the same as the -staticpositioned value for the previous db2sqljcustomize invocation for the serialized profile.

**-genDBRM**
Specifies that db2sqljbind generates database request modules (DBRMs) from the serialized profile, and that db2sqljbind does not perform remote bind operations.

- genDBRM applies to programs that are to be run on DB2 for z/OS database servers only.
-DBRMDir  *directory-name*

When -genDBRM is specified, -DBRMDir specifies the local directory into which db2sqljbind puts the generated DBRM files. The default is the current directory.

-DBRMDir applies to programs that are to be run on DB2 for z/OS database servers only.

-tracefile  *file-name*

Enables tracing and identifies the output file for trace information. This option should be specified only under the direction of IBM Software Support.

-tracelevel

If -tracefile is specified, indicates what to trace while db2sqljcustomize runs. The default is TRACE_SQLJ. This option should be specified only under the direction of IBM Software Support.

serialized-profile-name|file-name.grp

Specifies the names of one or more serialized profiles from which the package is bound. A serialized profile name is of the following form:

```
program-name_SJProfileIDNumber.ser
```

*program-name* is the name of the SQLJ source program, without the extension .sqlj. *n* is an integer between 0 and *m*-1, where *m* is the number of serialized profiles that the SQLJ translator generated from the SQLJ source program.

You can specify serialized profile names in one of the following ways:

- List the names in the db2sqljcustomize command. Multiple serialized profile names must be separated by spaces.
- Specify the serialized profile names, one on each line, in a file with the name *file-name.grp*, and specify *file-name.grp* in the db2sqljbind command.

If you specify more than one serialized profile name to bind a single DB2 package from several serialized profiles, you must have specified the same serialized profile names, in the same order, when you ran db2sqljcustomize.

If you specify one or more *file-name.grp* files, you must have run db2sqljcustomize once with that same list of files. The order in which you specify the files in db2sqljbind must be the same as the order in db2sqljcustomize.

You cannot run db2sqljcustomize on individual files, and then group those files when you run db2sqljbind.

**Examples**

```
db2sqljbind -user richler -password mordecai
-ur1 jdbc:db2://server:50000/sample -bindoptions "EXPLAIN YES"
pgmname_SJProfile0.ser
```

**Usage notes**

*Package names produced by db2sqljbind:* The names of the packages that are created by db2sqljbind are the names that you specified using the-rootpkgname or -singlepkgname parameter when you ran db2sqljcustomize. If you did not specify -rootpkgname or -singlepkgname, the package names are the first seven bytes of the profile name, appended with the isolation level character.

*DYNAMICRULES value for db2sqljbind:* The DYNAMICRULES bind option determines a number of run-time attributes for the DB2 package. Two of those
attributes are the authorization ID that is used to check authorization, and the
qualifier that is used for unqualified objects. To ensure the correct authorization for
dynamically executed positioned UPDATE and DELETE statements in SQLJ
programs, db2sqljbind always binds the DB2 packages with the
DYNAMICRULES(BIND) option. You cannot modify this option. The
DYNAMICRULES(BIND) option causes the SET CURRENT SQLID statement to
have no impact on an SQLJ program, because those statements affect only dynamic
statements that are bound with DYNAMICRULES values other than BIND.

With DYNAMICRULES(BIND), unqualified table, view, index, and alias names in
dynamic SQL statements are implicitly qualified with value of the bind option
QUALIFIER. If you do not specify QUALIFIER, DB2 uses the authorization ID of
the package owner as the implicit qualifier. If this behavior is not suitable for your
program, you can use one of the following techniques to set the correct qualifier:

- Force positioned UPDATE and DELETE statements to execute statically. You can
  use the -staticpositioned YES option of db2sqljcustomize or db2sqljbind to do
  this if the cursor (iterator) for a positioned UPDATE or DELETE statement is in
  the same package as the positioned UPDATE or DELETE statement.
- Fully qualify DB2 table names in positioned UPDATE and positioned DELETE
  statements.

**EXTENDEDINDICATOR bind option behavior:** If the EXTENDEDINDICATOR
bind option is not specified in the -bindoptions options string, and the target data
server supports extended indicators, bind operations use
EXTENDEDINDICATOR(YES). If EXTENDEDINDICATOR(NO) is explicitly
specified, and the application contains extended indicator syntax, unexpected
behavior can occur because the IBM Data Server Driver for JDBC and SQLJ treats
extended indicators as NULL values.

Related reference:

- BIND and REBIND options for packages and plans (DB2 Commands)

**db2sqljprint - SQLJ profile printer**

db2sqljprint prints the contents of the customized version of a profile as plain text.

**Authorization**

None

**Command syntax**

```
  db2sqljprint <filename>
```

**Command parameters**

- **filename**
  Specifies the relative or absolute name of an SQLJ profile file. When an
  SQLJ file is translated into a Java source file, information about the SQL
  operations it contains is stored in SQLJ-generated resource files called
  profiles. Profiles are identified by the suffix _SJProfileN (where N is an
  integer) following the name of the original input file. They have a .ser
  extension. Profile names can be specified with or without the .ser
  extension.
Examples

db2sqljprint pgmname_SJProfile0.ser
Chapter 8. Installing the IBM Data Server Driver for JDBC and SQLJ on DB2 for z/OS

After you install DB2 for z/OS or migrate to the current version of DB2 for z/OS, you need to install the current version of the IBM Data Server Driver for JDBC and SQLJ.

Installing the IBM Data Server Driver for JDBC and SQLJ as part of a DB2 installation

Installation of the IBM Data Server Driver for JDBC and SQLJ on your DB2 subsystem allows you to build and run Java database applications.

Before you begin

Prerequisites for the IBM Data Server Driver for JDBC and SQLJ:

- Java 2 Technology Edition, V5 or later.
  - JDBC 4.0 methods require Java 2 Technology Edition, V6 or later.
    - The IBM Data Server Driver for JDBC and SQLJ supports 31-bit or 64-bit Java applications. 64-bit Java stored procedures and user-defined functions are not supported.
    - For applications that require a 64-bit JVM, you need to install the IBM 64-bit SDK for z/OS, Java 2 Technology Edition, V5 or later.
- TCP/IP
  - TCP/IP is required on the client and all database servers to which you connect.
- DB2 for z/OS distributed data facility (DDF) and TCP/IP support.
- Unicode support for OS/390® and z/OS servers.

Procedure

To install the IBM Data Server Driver for JDBC and SQLJ, follow these steps:

1. When you allocate and load the DB2 for z/OS libraries, include the steps that allocate and load the IBM Data Server Driver for JDBC and SQLJ libraries.
2. On DB2 for z/OS, set subsystem parameter DESCSTAT to YES. DESCSTAT corresponds to installation field DESCRIBE FOR STATIC on panel DSNTIPF. This step is necessary for SQLJ support.
3. In z/OS UNIX System Services, edit your .profile file to customize the environment variable settings. You use this step to set the libraries, paths, and files that the IBM Data Server Driver for JDBC and SQLJ uses. You also use this step to indicate the versions of JDBC and SQLJ support that you need.
4. Optional: Customize the IBM Data Server Driver for JDBC and SQLJ configuration properties.
5. On DB2 for z/OS, enable the DB2 supplied stored procedures and define the tables that are used by the IBM Data Server Driver for JDBC and SQLJ.
6. In z/OS UNIX System Services, run the DB2Binder utility to bind the packages for the IBM Data Server Driver for JDBC and SQLJ.
7. Verify the installation by running a simple JDBC application.

Related tasks:
Jobs for loading the IBM Data Server Driver for JDBC and SQLJ libraries

When you install DB2 for z/OS, allocate the HFS or zFS directory structure, and use SMP/E to load the IBM Data Server Driver for JDBC and SQLJ libraries.

The following jobs perform those functions.

**DSNISMKD**
Invokes the DSNMKDIR EXEC to allocate the HFS or zFS directory structures.

**DSNDDEF1**
Includes steps to define DDDEFs for the IBM Data Server Driver for JDBC and SQLJ libraries.

**DSNRECV3**
Includes steps that perform the SMP/E RECEIVE function for the IBM Data Server Driver for JDBC and SQLJ libraries.

**DSNAPPL1**
Includes the steps that perform the SMP/E APPLY CHECK and APPLY functions for the IBM Data Server Driver for JDBC and SQLJ libraries.

**DSNACEP1**
Includes the steps that perform the SMP/E ACCEPT CHECK and ACCEPT functions for the IBM Data Server Driver for JDBC and SQLJ libraries.

See *IBM Db2 for z/OS Program Directory* for information on allocating and loading DB2 data sets.

Environment variables for the IBM Data Server Driver for JDBC and SQLJ

If you set specific environment variables, the operating system can locate the IBM Data Server Driver for JDBC and SQLJ.

The environment variables that you must set are:

**STEPLIB**
Modify STEPLIB to include the SDSNEXIT, SDSNLOAD, and SDSNLOD2 data sets. For example:

```
```

**PATH**
Modify PATH to include the directory that contains the shell scripts that invoke IBM Data Server Driver for JDBC and SQLJ program preparation and debugging functions.

For example, if the IBM Data Server Driver for JDBC and SQLJ is installed in `/usr/lpp/db2a10/jdbc`, modify PATH as follows:

```
export PATH=/usr/lpp/db2a10/jdbc/bin:$PATH
```
LIBPATH
The IBM Data Server Driver for JDBC and SQLJ contains the following dynamic load libraries (DLLs):

- libdb2jct2zos.so
- libdb2jct2zos_64.so

Those DLLs contain the native (C or C++) implementation of the IBM Data Server Driver for JDBC and SQLJ. The driver uses this code when you use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

Modify LIBPATH to include the directory that contains these DLLs.

For example, if the IBM Data Server Driver for JDBC and SQLJ is installed in /usr/lpp/db2a10/jdbc, modify LIBPATH as follows:

```
export LIBPATH=/usr/lpp/db2a10/jdbc/lib:$LIBPATH
```

CLASSPATH
The IBM Data Server Driver for JDBC and SQLJ contains the following class files:

- **db2jcc.jar** or **db2jcc4.jar**
  Contains all JDBC classes and the SQLJ runtime classes for the IBM Data Server Driver for JDBC and SQLJ.

  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 later functions, as well as JDBC 3.0 and earlier functions**.

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

- **sqlj.zip** or **sqlj4.zip**
  Contains the classes that are needed to prepare SQLJ applications for execution under the IBM Data Server Driver for JDBC and 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 later functions, as well as JDBC 3.0 and earlier functions**.

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

- **db2jcc_license_cisuz.jar**
  A license file that permits access to the DB2 server.

Modify your CLASSPATH to include these files. If the IBM Data Server Driver for JDBC and SQLJ is installed in /usr/lpp/db2a10/jdbc, modify CLASSPATH as follows:

For JDBC 3.0 and earlier support:

```
export CLASSPATH=/usr/lpp/db2a10/jdbc/classes/db2jcc.jar:
/usr/lpp/db2a10/jdbc/classes/db2jcc_javax.jar:
/usr/lpp/db2a10/jdbc/classes/sqlj.zip:
/usr/lpp/db2a10/jdbc/classes/db2jcc_license_cisuz.jar:
$CLASSPATH
```

For JDBC 4.0 and later, and JDBC 3.0 and earlier support:

```bash
export CLASSPATH=/usr/lpp/db2a10_jdbc/classes/db2jcc4.jar:/$
/usr/lpp/db2a10/jdbc/classes/db2jcc_javax.jar:/$
/usr/lpp/db2a10/jdbc/classes/sqlj4.zip:/$
/usr/lpp/db2a10/jdbc/classes/db2jcc_license_cisuz.jar:/$
$CLASSPATH
```

If you use Java stored procedures, you need to set additional environment variables in a JAVAENV data set.

**Related concepts:**
- "WLM application environment values for Java routines" on page 191
- "Runtime environment for Java routines" on page 193

### 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
```

IBM Data Server Driver for JDBC and SQLJ configuration property names are case-sensitive.

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 `Dproperty=value` for each configuration property when you execute the java command.

  For Java stored procedures or user-defined functions, you can set the configuration properties by specifying `Dproperty=value` for each configuration property in a file whose name you specify in the JVMPROPS option. You specify the JVMPROPS options in the ENVAR option of the Language Environment options string. The Language Environment options string is in a data set that is specified by the JAVAENV DD statement in the WLM address space startup procedure.

- 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.

  For Java stored procedures or user-defined functions, you can set the configuration properties by specifying the `Ddb2.jcc.propertiesFile=path/properties-file-name` option in a file whose name you specify in the JVMPROPS option. You specify the JVMPROPS options in the ENVAR option of the
Language Environment options string. The Language Environment options string is in a data set that is specified by the JAVAENV DD statement in the WLM address space startup procedure.

- 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 a stand-alone file, the contents are automatically converted to Unicode. If you include DB2JccConfiguration.properties in a JAR file, you need to convert the contents to Unicode before you put them in the JAR file.

If DB2JccConfiguration.properties is a stand-alone file, the path for DB2JccConfiguration.properties must be in the following places:

- For stand-alone Java applications: Include the directory that contains DB2JccConfiguration.properties in the CLASSPATH concatenation.
- For Java stored procedures or user-defined functions: Include the directory that contains DB2JccConfiguration.properties in the CLASSPATH concatenation in the ENVAR option of the Language Environment options string. The Language Environment options string is in a data set that is specified by the JAVAENV DD statement in the WLM address space startup procedure.

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

Example: Putting DB2JccConfiguration.properties in a JAR file: Suppose that your configuration properties are in a file that is in EBCDIC code page 1047. To put the properties file into a JAR file, follow these steps:

1. Rename DB2JccConfiguration.properties to another name, such as EBCDICVersion.properties.
2. Run the iconv shell utility on the z/OS UNIX System Services command line to convert the file contents to Unicode. For example, to convert EBCDICVersion.properties to a Unicode file named DB2JccConfiguration.properties, issue this command:
   ```shell
   iconv -f ibm-1047 -t utf-8 EBCDICVersion.properties
   > DB2JccConfiguration.properties
   ```
3. Execute the jar command to add the Unicode file to the JAR file. In the JAR file, the configuration properties file must be named DB2JccConfiguration.properties. For example:
   ```shell
   jar -cvf jdbcProperties.jar DB2JccConfiguration.properties
   ```

Related reference:

- “Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 234
- “IBM Data Server Driver for JDBC and SQLJ configuration properties” on page 295

Enabling the DB2-supplied stored procedures used by the IBM Data Server Driver for JDBC and SQLJ

The IBM Data Server Driver for JDBC and SQLJ requires a set of stored procedures to make certain methods work on DB2 for z/OS.
Before you begin

WLM must be installed on the z/OS system, and a WLM environment must be set up for running the DB2-supplied stored procedures that are required by the IBM Data Server Driver for JDBC and SQLJ.

About this task

The stored procedures that you need to install are:
- SQLCOLPRIVILEGES
- SQLCOLUMNNS
- SQLFOREIGNKEYS
- SQLFUNCTIONCOLS
- SQLFUNCTIONS
- SQLGETTYPEINFO
- SQLPRIMARYKEYS
- SQLPROCEDURECOLS
- SQLPROcedures
- SQLPSEUDOCOLUMNNS
- SQLSPECIALCOLUMNNS
- SQLSTATISTICS
- SQLTABLEPRIVILEGES
- SQLTABLES
- SQLUDTS
- SQLCAMESSAGE

Procedure

Follow these steps to install the stored procedures:

1. Set up a WLM environment for running the stored procedures.
   - Define WLM application environment DSNWLM_GENERAL to WLM. Application environment DSNWLM_GENERAL and its associated JCL startup procedure, DSNWLMG, are created during DB2 installation. DSNWLM_GENERAL is designed for running the DB2-supplied stored procedures that support JDBC.

2. Define the stored procedures to DB2, and bind the stored procedure packages.
   - To perform those tasks, use job DSNTIJRT.

Values for the WLM environment for IBM Data Server Driver for JDBC and SQLJ stored procedures

You need to define an application environment for DB2-supplied stored procedures that support the IBM Data Server Driver for JDBC and SQLJ to WLM.

The following example shows a WLM Definition Menu for an application environment for DB2-supplied stored procedures that support the IBM Data Server Driver for JDBC and SQLJ.
Definition

Specify the name of the WLM application environment that you are setting up for stored procedures. The Definition name value needs to match the APPLENV value in the WLM address space startup procedure.

Description

Specify any value.

Options

Specify 9 (Application Environments).

The following example shows a WLM Create an Application Environment menu with values for the application environment that is used by DB2-supplied stored procedures that support the IBM Data Server Driver for JDBC and SQLJ.

Subsystem Type

Specify DB2.

Procedure Name

Specify a name that matches the name of the JCL startup procedure for the stored procedure address spaces that are associated with this application environment.

Start Parameters

If the DB2 subsystem in which the stored procedure runs is not in a sysplex, specify a DB2SSN value that matches the name of that DB2 subsystem. If the same JCL is used for multiple DB2 subsystems, specify DB2SSN=\IWMS\SSNM.
Specify a NUMTCB value between 40 and 60. Specify an APPLENV value that matches the value that you specify in the WLM address space startup procedure and on the CREATE PROCEDURE statements for the stored procedures.

Limit on starting server address spaces for a subsystem instance
Specify 1 (no limit).

Creation of IBM Data Server Driver for JDBC and SQLJ stored procedures
DB2 provides a JCL job that includes statements that you can use to define the DB2-supplied stored procedures for JDBC and to bind the stored procedure packages.

Job DSNTIJRT is customized by the DB2 installation process to specify WLM environment DSNWLM_GENERAL, which is designed to work with the DB2-supplied stored procedures that support JDBC methods. In general, you run job DSNTIJRT when you install or migration your DB2 subsystem.

DB2Binder utility
The DB2Binder utility binds the DB2 packages that are used at the data server by the IBM Data Server Driver for JDBC and SQLJ, and grants EXECUTE authority on the packages to PUBLIC. Optionally, the DB2Binder utility can rebind DB2 packages that are not part of the IBM Data Server Driver for JDBC and SQLJ.

DB2Binder syntax

```
```

```
```

```
```

```
```

```
```
DB2Binder option descriptions

-url
Specifies the data source at which the IBM Data Server Driver for JDBC and SQLJ packages are to be bound. The variable parts of the -url value are:

  server
  The domain name or IP address of the operating system on which the data server resides.

  port
  The TCP/IP server port number that is assigned to the data server. The default is 446.

  database
  The location name for the data server, as defined in the SYSIBM.LOCTAB catalog table.

-user
Specifies the user ID under which the packages are to be bound. This user must have BIND authority on the packages.

-action
Specifies the action to perform on the packages.

  add
  Indicates that a package can be created only if it does not already exist. Add is the default.

  replace
  Indicates that a package can be created even if a package with the same name already exists. The new package replaces the old package.

  rebind
  Indicates that the existing package should be rebound. This option does not apply to IBM Data Server Driver for JDBC and SQLJ packages.
  If -action rebind and -bindOptions are specified, -generic must also be specified.

  drop
  Indicates that packages should be dropped:
    - For IBM Data Server Driver for JDBC and SQLJ packages, -action drop indicates that some or all IBM Data Server Driver for JDBC and SQLJ packages should be dropped. The number of packages depends on the -size parameter.
    - For user packages, -action drop indicates that the specified package should be dropped.

  -action drop applies only if the target data server is DB2 for z/OS.

-size
Controls the number of Statement, PreparedStatement, or CallableStatement objects that can be open concurrently, or the number of IBM Data Server Driver for JDBC and SQLJ packages that are dropped.

  The meaning of the -size parameter depends on the -action parameter:
• If the value of -action is add or replace, the value of -size is an integer that is used to calculate the number of DB2 packages that the IBM Data Server Driver for JDBC and SQLJ binds. If the value of -size is integer, the total number of packages is:

\[
\text{number-of-isolation-levels} \times \\
\text{number-of-holdability-values} \times \\
\text{integer} + \\
\text{number-of-packages-for-static-SQL} = 4 \times 2 \times \text{integer} + 1
\]

The default -size value for -action add or -action replace is 3.

In most cases, the default of 3 is adequate. If your applications throw SQLExceptions with -805 SQLCODEs, check that the applications close all unused resources. If they do, increase the -size value.

If the value of -action is replace, and the value of -size results in fewer packages than already exist, no packages are dropped.

• If the value of -action is drop, the value of -size is the number of packages that are dropped. If -size is not specified, all IBM Data Server Driver for JDBC and SQLJ packages are dropped.

• If the value of -action is rebind, -size is ignored.

**-collection**

Specifies the collection ID for IBM Data Server Driver for JDBC and SQLJ or user packages. The default is NULLID. DB2Binder translates this value to uppercase.

You can create multiple instances of the IBM Data Server Driver for JDBC and SQLJ packages on a single data server by running `com.ibm.db2.jcc.DB2Binder` multiple times, and specifying a different value for -collection each time. At run time, you select a copy of the IBM Data Server Driver for JDBC and SQLJ by setting the currentPackageSet property to a value that matches a -collection value.

**-tracelevel**

Specifies what to trace while DB2Binder runs. For a list of the valid values, see “DB2TraceManager class” on page 462.

**-reopt**

Specifies whether data servers determine access paths at run time. This option is not sent to the data server if it is not specified. In that case, the data server determines the reoptimization behavior.

-reopt applies to connections to DB2 for z/OS Version 8 or later, or Db2 on Linux, UNIX, and Windows systems Version 9.1 or later.

- none  Specifies that access paths are not determined at run time.
- always  Specifies that access paths are determined each time a statement is run.
- once  Specifies that DB2 determines and caches the access path for a dynamic statement only once at run time. DB2 uses this access path until the prepared statement is invalidated, or until the statement is removed from the dynamic statement cache and needs to be prepared again.
- auto  Specifies that access paths are automatically determined by the data server. auto is valid only for connections to DB2 for z/OS data servers.
-blocking
Specifies the type of row blocking for cursors.

ALL For cursors that are specified with the FOR READ ONLY clause or are not specified as FOR UPDATE, blocking occurs.

UNAMBIG
For cursors that are specified with the FOR READ ONLY clause, blocking occurs.

Cursors that are not declared with the FOR READ ONLY or FOR UPDATE clause which are not ambiguous and are read-only will be blocked. Ambiguous cursors will not be blocked

NO Blocking does not occur for any cursor.

For the definition of a read-only cursor and an ambiguous cursor, refer to "DECLARE CURSOR".

-optprofile
Specifies an optimization profile that is used for optimization of data change statements in the packages. This profile is an XML file that must exist on the target server. If -optprofile is not specified, and the CURRENT OPTIMIZATION PROFILE special register is set, the value of CURRENT OPTIMIZATION PROFILE is used. If -optprofile is not specified, and CURRENT OPTIMIZATION PROFILE is not set, no optimization profile is used.

-optprofile is valid only for connections to Db2 on Linux, UNIX, and Windows systems data servers.

-owner
Specifies the authorization ID of the owner of the packages. The default value is set by the data server.

-owner applies only to IBM Data Server Driver for JDBC and SQLJ packages.

-sqlid
Specifies a value to which the CURRENT SQLID special register is set before DB2Binder executes GRANT operations on the IBM Data Server Driver for JDBC and SQLJ packages. If the primary authorization ID does not have a sufficient level of authority to grant privileges on the packages, and the primary authorization ID has an associated secondary authorization ID that has those privileges, set -sqlid to the secondary authorization ID.

-sqlid is valid only for connections to DB2 for z/OS data servers.

-generic
Specifies that DB2Binder rebinds a user package instead of the IBM Data Server Driver for JDBC and SQLJ packages. If -generic is specified, -action rebind and -package must also be specified.

-package
Specifies the name of the package that is to be rebound. This option applies only to user packages. If -package is specified, -action rebind and -generic must also be specified.

-version
Specifies the version ID of the package that is to be rebound. If -version is specified, -action rebind, -package, and -generic must also be specified.

-keepdynamic
Specifies whether the data server keeps already prepared dynamic SQL
statements in the dynamic statement cache after commit points so that those
prepared statements can be reused. -keepdynamic applies only to connections
to DB2 for z/OS.

Possible values are:

- **no**  The data server does not keep already prepared dynamic SQL
  statements in the dynamic statement cache after commit points.
- **yes**  The data server keeps already prepared dynamic SQL statements in the
dynamic statement cache after commit points.

There is no default value for -keepdynamic. If you do not send a value to the
data server, the setting at the data server determines whether dynamic
statement caching is in effect. Dynamic statement caching occurs only if the
EDM dynamic statement cache is enabled on the data server. The CACHEDYN
subsystem parameter must be set to YES to enable the dynamic statement
cache.

**-release**

Specifies when to release data server resources that a program uses. -release
applies only to connections to DB2 for z/OS. Possible values are:

- **deallocation**
  Specifies that resources are released when a program terminates.
  -release deallocate is the default for DB2 for z/OS Version 10 and later.

- **commit**
  Specifies that resources are released at commit points. -release commit
  is the default for DB2 for z/OS Version 9 and earlier.

**-bindOptions**

Specifies a string that is enclosed in quotation marks. The contents of that
string are one or more parameter and value pairs that represent options for
rebinding a user package. All items in the string are delimited with spaces:

"parm1=value1 parm2=value2 ... parmn=valuen"

-bindOptions does not apply to IBM Data Server Driver for JDBC and SQLJ
packages that are bound on Db2 on Linux, UNIX, and Windows systems data
servers.

You can specify the following -bindOptions values only when you rebind user
packages:

**bindObjectExistenceRequired**

Specifies whether the data server issues an error and does not rebind
the package, if all objects or needed privileges do not exist at rebind
time. Possible values are:

- **true**  This option corresponds to the SQLERROR(NOPACKAGE)
  bind option.
- **false**  This option corresponds to the SQLERROR(CONTINUE) bind
  option.

**degreeIOParallelism**

Specifies whether to attempt to run static queries using parallel
processing to maximize performance. Possible values are:

- **1**  No parallel processing.
  This option corresponds to the DEGREE(1) bind option.
- **-1**  Allow parallel processing.
This option corresponds to the DEGREE(ANY) bind option.

**packageAuthorizationRules**

Determines the values that apply at run time for the following dynamic SQL attributes:

- The authorization ID that is used to check authorization
- The qualifier that is used for unqualified objects
- The source for application programming options that the data server uses to parse and semantically verify dynamic SQL statements
- Whether dynamic SQL statements can include GRANT, REVOKE, ALTER, CREATE, DROP, and RENAME statements

Possible values are:

0  Use run behavior. This is the default.
   This option corresponds to the DYNAMICRULES(RUN) bind option.

1  Use bind behavior.
   This option corresponds to the DYNAMICRULES(BIND) bind option.

2  When the package is run as or runs under a stored procedure or user-defined function package, the data server processes dynamic SQL statements using invoke behavior. Otherwise, the data server processes dynamic SQL statements using run behavior.
   This option corresponds to the DYNAMICRULES(INVOKERUN) bind option.

3  When the package is run as or runs under a stored procedure or user-defined function package, the data server processes dynamic SQL statements using invoke behavior. Otherwise, the data server processes dynamic SQL statements using bind behavior.
   This option corresponds to the DYNAMICRULES(INVOKEBIND) bind option.

4  When the package is run as or runs under a stored procedure or user-defined function package, the data server processes dynamic SQL statements using define behavior. Otherwise, the data server processes dynamic SQL statements using run behavior.
   This option corresponds to the DYNAMICRULES(DEFINERUN) bind option.

5  When the package is run as or runs under a stored procedure or user-defined function package, the data server processes dynamic SQL statements using define behavior. Otherwise, the data server processes dynamic SQL statements using bind behavior.
   This option corresponds to the DYNAMICRULES(DEFINEBIND) bind option.

**packageOwnerIdentifier**

Specifies the authorization ID of the owner of the packages.
isolationLevel
Specifies how far to isolate an application from the effects of other running applications. Possible values are:

1 Uncommitted read
This option corresponds to the ISOLATION(UR) bind option.

2 Cursor stability
This option corresponds to the ISOLATION(CS) bind option.

3 Read stability
This option corresponds to the ISOLATION(RS) bind option.

4 Repeatable read
This option corresponds to the ISOLATION(RR) bind option.

releasePackageResourcesAtCommit
Specifies when to release resources that a program uses at each commit point. Possible values are:

true This option corresponds to the RELEASE(COMMIT) bind option.
false This option corresponds to the RELEASE(DEALLOCATE) bind option.

If -action rebind and -bindOptions are specified, -generic must also be specified.

-verbose
Specifies that the DB2Binder utility displays detailed information about the bind process.

-help
Specifies that the DB2Binder utility describes each of the options that it supports. If any other options are specified with -help, they are ignored.

DB2Binder return codes when the target operating system is not Windows

If the target data source for DB2Binder is not on the Windows operating system, DB2Binder returns one of the following return codes.

Table 104. DB2Binder return codes when the target operating system is not Windows

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>1</td>
<td>An error occurred during DB2Binder execution.</td>
</tr>
</tbody>
</table>

DB2Binder return codes when the target operating system is Windows

If the target data source for DB2Binder is on the Windows operating system, DB2Binder returns one of the following return codes.
Table 105. DB2Binder return codes when the target operating system is Windows

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>-100</td>
<td>No bind options were specified.</td>
</tr>
<tr>
<td>-101</td>
<td>-url value was not specified.</td>
</tr>
<tr>
<td>-102</td>
<td>-user value was not specified.</td>
</tr>
<tr>
<td>-103</td>
<td>-password value was not specified.</td>
</tr>
<tr>
<td>-200</td>
<td>No valid bind options were specified.</td>
</tr>
<tr>
<td>-114</td>
<td>The -package option was not specified, but the -generic option was specified.</td>
</tr>
<tr>
<td>-201</td>
<td>-url value is invalid.</td>
</tr>
<tr>
<td>-204</td>
<td>-action value is invalid.</td>
</tr>
<tr>
<td>-205</td>
<td>-blocking value is invalid.</td>
</tr>
<tr>
<td>-206</td>
<td>-collection value is invalid.</td>
</tr>
<tr>
<td>-207</td>
<td>-dbprotocol value is invalid.</td>
</tr>
<tr>
<td>-208</td>
<td>-keepdynamic value is invalid.</td>
</tr>
<tr>
<td>-210</td>
<td>-reopt value is invalid.</td>
</tr>
<tr>
<td>-211</td>
<td>-size value is invalid.</td>
</tr>
<tr>
<td>-212</td>
<td>-tracelevel value is invalid.</td>
</tr>
<tr>
<td>-307</td>
<td>-dbprotocol value is not supported by the target data server.</td>
</tr>
<tr>
<td>-308</td>
<td>-keepdynamic value is not supported by the target data server.</td>
</tr>
<tr>
<td>-310</td>
<td>-reopt value is not supported by the target data server.</td>
</tr>
<tr>
<td>-313</td>
<td>-optprofile value is not supported by the target data server.</td>
</tr>
<tr>
<td>-401</td>
<td>The Binder class was not found.</td>
</tr>
<tr>
<td>-402</td>
<td>Connection to the data server failed.</td>
</tr>
<tr>
<td>-403</td>
<td>DatabaseMetaData retrieval for the data server failed.</td>
</tr>
<tr>
<td>-501</td>
<td>No more packages are available in the cluster.</td>
</tr>
<tr>
<td>-502</td>
<td>An existing package is not valid.</td>
</tr>
<tr>
<td>-503</td>
<td>The bind process returned an error.</td>
</tr>
<tr>
<td>-999</td>
<td>An error occurred during processing of an undocumented bind option.</td>
</tr>
</tbody>
</table>

**DB2LobTableCreator utility**

The DB2LobTableCreator utility creates tables on a DB2 for z/OS database server. Those tables are required by JDBC or SQLJ applications that use LOB locators to access data in DBCLOB or CLOB columns.

**DB2LobTableCreator syntax**

```bash
java -com.ibm.db2.jcc.DB2LobTableCreator -url jdbc:db2://server:port/database -user user-ID -password password
```

Chapter 8. Installing the IBM Data Server Driver for JDBC and SQLJ on DB2 for z/OS  533


**DB2LobTableCreator option descriptions**

- **-url**
  Specifies the data source at which DB2LobTableCreator is to run. The variable parts of the -url value are:

  - **jdbc:db2:**
    Indicates that the connection is to a server in the DB2 family.

  - **server**
    The domain name or IP address of the database server.

  - **port**
    The TCP/IP server port number that is assigned to the database server. This is an integer between 0 and 65535. The default is 446.

  - **database**
    A name for the database server. The 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:
    ```sql
    SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
    ```

- **-user**
  Specifies the user ID under which DB2LobTableCreator is to run. This user must have authority to create tables in the DSNATPDB database.

- **-password**
  Specifies the password for the user ID.

- **-help**
  Specifies that the DB2LobTableCreator utility describes each of the options that it supports. If any other options are specified with -help, they are ignored.

**Verify the installation of the IBM Data Server Driver for JDBC and SQLJ**

To verify the installation of the IBM Data Server Driver for JDBC and SQLJ, compile and run any simple JDBC application.

For example, you can compile and run this program to verify your installation:

```java
/**
 * File: TestJDBCSelect.java
 *
 * Purpose: Verify IBM Data Server Driver for JDBC and SQLJ installation.
 * This program uses IBM Data Server Driver for JDBC and SQLJ
 * type 2 connectivity on DB2 for z/OS.
 *
 * Authorization: This program requires SELECT authority on
 * DB2 catalog table SYSIBM.SYSTABLES.
 *
 * Flow:
 * - Load the IBM Data Server Driver for JDBC and SQLJ.
 * - Get the driver version and display it.
 * - Establish a connection to the local DB2 for z/OS server.
 * - Get the DB2 version and display it.
 * - Execute a query against SYSIBM.SYSTABLES.
 * - Clean up by closing all open objects.
 */

import java.sql.*;

public class TestJDBCSelect {

```
public static void main(String[] args)
{
    try
    {
        // Load the driver and get the version
        System.out.println("Loading IBM Data Server Driver for JDBC and SQLJ");
        Class.forName("com.ibm.db2.jcc.DB2Driver");

        // Connect to the local DB2 for z/OS server
        System.out.println("Establishing connection to local server");
        Connection conn = DriverManager.getConnection("jdbc:db2:");
        System.out.println("Successful connect");
        conn.setAutoCommit(false);

        // Use DatabaseMetaData to determine the DB2 version
        System.out.println("Acquiring DatabaseMetaData");
        DatabaseMetaData dbmd = conn.getMetaData();
        System.out.println("DB2 version: "+dbmd.getDatabaseProductVersion());

        // Create a Statement object for executing a query
        System.out.println("Creating Statement");
        Statement stmt = conn.createStatement();
        System.out.println("successful creation of Statement");

        // Execute the query and retrieve the ResultSet object
        String sqlText = "SELECT CREATOR, " +
                        "NAME " +
                        "FROM SYSCHEMA.SYSTABLES " +
                        "ORDER BY CREATOR, NAME";
        System.out.println("Preparing to execute SELECT");
        ResultSet results = stmt.executeQuery(sqlText);
        System.out.println("Successful execution of SELECT");

        // Retrieve and display the rows from the ResultSet
        System.out.println("Preparing to fetch from ResultSet");
        int recCnt = 0;
        while(results.next())
        {
            String creator = results.getString("CREATOR");
            String name = results.getString("NAME");
            System.out.println("CREATOR: <" + creator + "> NAME: <" + name + "+">");
            recCnt++;
            if(recCnt == 10) break;
        }
        System.out.println("Successful processing of ResultSet");

        // Close the ResultSet, Statement, and Connection objects
        System.out.println("Preparing to close ResultSet");
        results.close();
        System.out.println("Successful close of ResultSet");
        System.out.println("Preparing to close Statement");
        stmt.close();
        System.out.println("Successful close of Statement");
        System.out.println("Preparing to rollback Connection");
        conn.rollback();
        System.out.println("Successful rollback");
        System.out.println("Preparing to close Connection");
        conn.close();
        System.out.println("Successful close of Connection");
    }
}
Upgrading the IBM Data Server Driver for JDBC and SQLJ to a new version

Upgrading to a new version of the IBM Data Server Driver for JDBC and SQLJ is similar to installing the IBM Data Server Driver for JDBC and SQLJ for the first time. However, you need to adjust your application programs to work with the new version of the driver.

Before you begin

You should have already completed these steps when you installed the earlier version of the IBM Data Server Driver for JDBC and SQLJ:

1. On DB2 for z/OS, set subsystem parameter DESCSTAT to YES. DESCSTAT corresponds to installation field DESCRIBE FOR STATIC on panel DSNTIPF. This step is necessary for SQLJ support.
2. On DB2 for z/OS, enable the DB2 supplied stored procedures and define the tables that are used by the IBM Data Server Driver for JDBC and SQLJ.

About this task

Procedure

To upgrade the IBM Data Server Driver for JDBC and SQLJ to a new version, follow these steps:

1. In z/OS UNIX System Services, edit your .profile file to customize the environment variable settings. You use this step to set the libraries, paths, and files that the IBM Data Server Driver for JDBC and SQLJ uses.
2. Optional: Customize the IBM Data Server Driver for JDBC and SQLJ configuration properties.
3. In z/OS UNIX System Services, run the DB2Binder utility to bind the packages for the IBM Data Server Driver for JDBC and SQLJ.
4. Modify your applications to account for differences between the driver versions.
5. Verify the installation by running a simple JDBC application.

Related concepts:

- “Verify the installation of the IBM Data Server Driver for JDBC and SQLJ” on page 534
- “Environment variables for the IBM Data Server Driver for JDBC and SQLJ” on page 520
- “Runtime environment for Java routines” on page 193
Installing the z/OS Application Connectivity to DB2 for z/OS feature

z/OS Application Connectivity to DB2 for z/OS is a DB2 for z/OS feature. It allows IBM Data Server Driver for JDBC and SQLJ type 4 connectivity from clients that do not have DB2 for z/OS installed to DB2 for z/OS or DB2 servers.

Before you begin

Prerequisites for the IBM Data Server Driver for JDBC and SQLJ:

- Java 2 Technology Edition, V5 or later
  - The IBM Data Server Driver for JDBC and SQLJ supports 31-bit or 64-bit Java applications.
  - If your applications require a 64-bit JVM, you need to install the IBM 64-bit SDK for z/OS, Java 2 Technology Edition, V5 or later.
- TCP/IP
  - TCP/IP is required on the client and all database servers to which you connect.
- DB2 for z/OS distributed data facility (DDF) and TCP/IP support.
- Unicode support for OS/390 and z/OS servers.

About this task

To install the z/OS Application Connectivity to DB2 for z/OS, follow this process. Unless otherwise noted, all steps apply to the z/OS system on which you are installing z/OS Application Connectivity to DB2 for z/OS.

Procedure

To install the z/OS Application Connectivity to DB2 for z/OS feature:

1. Allocate and load the z/OS Application Connectivity to DB2 for z/OS libraries.
2. On all DB2 for z/OS servers to which you plan to connect, set subsystem parameter DESCSTAT to YES. DESCSTAT corresponds to installation field DESCRIBE FOR STATIC on panel DSNTIPF.
   - This step is necessary for SQLJ support.
3. In z/OS UNIX System Services, edit your .profile file to customize the environment variable settings. You use this step to set the libraries, paths, and files that the IBM Data Server Driver for JDBC and SQLJ uses.
4. On all DB2 for z/OS servers to which you plan to connect, enable the DB2-supplied stored procedures that are used by the IBM Data Server Driver for JDBC and SQLJ.

5. In z/OS UNIX System Services, run the DB2Binder utility against the z/OS system on which you are installing z/OS Application Connectivity to DB2 for z/OS to bind the packages for the IBM Data Server Driver for JDBC and SQLJ at all DB2 for z/OS servers to which you plan to connect. You need to run DB2Binder once for each server.

6. Verify the installation by running a simple JDBC application.

Related concepts:
- “Verify the installation of the IBM Data Server Driver for JDBC and SQLJ” on page 534
- “Environment variables for the IBM Data Server Driver for JDBC and SQLJ” on page 520
- “Runtime environment for Java routines” on page 193
- “Customization of IBM Data Server Driver for JDBC and SQLJ configuration properties” on page 522

Related tasks:
- “Enabling the DB2-supplied stored procedures used by the IBM Data Server Driver for JDBC and SQLJ” on page 523
- Connecting distributed database systems (DB2 Installation and Migration)
- Connecting systems with TCP/IP (DB2 Installation and Migration)

Related reference:
- “IBM Data Server Driver for JDBC and SQLJ configuration properties” on page 295
- “DB2LobTableCreator utility” on page 533
- “DB2Binder utility” on page 526

Jobs for loading the z/OS Application Connectivity to DB2 for z/OS libraries

To allocate the HFS or zFS directory structure and to use SMP/E to load the z/OS Application Connectivity to DB2 for z/OS libraries, you need to run jobs that are provided by DB2.

Those jobs are:

**DDAALA**
- Creates the SMP/E consolidate software inventory (CSI) file. DDAALA is required only if the SMP/E target and distribution zones are not created and allocated to the SMP/E global zone.

**DDAALB**
- Creates the z/OS Application Connectivity to DB2 for z/OS target and distribution zones. Also creates DDDEFs for SMP/E data sets. DDAALB is required only if the SMP/E target and distribution zones are not created and allocated to the SMP/E global zone.
DDAALLOC
Creates the z/OS Application Connectivity to DB2 for z/OS target and distribution libraries and defines them in the SMP/E target and distribution zones.

DDADDDEF
Creates DDDEFs for the z/OS Application Connectivity to DB2 for z/OS target and distribution libraries.

DDAISMKD
Invokes the DDAMKDIR EXEC to allocate the HFS or zFS directory structure for the z/OS Application Connectivity to DB2 for z/OS.

DDARECEV
Performs the SMP/E RECEIVE function for the z/OS Application Connectivity to DB2 for z/OS libraries.

DDAAPPLY
Performs the SMP/E APPLY CHECK and APPLY functions for the z/OS Application Connectivity to DB2 for z/OS libraries.

DDAACCEP
Performs the SMP/E ACCEPT CHECK and ACCEPT functions for the z/OS Application Connectivity to DB2 for z/OS libraries.

See z/OS Application Connectivity to DB2 for z/OS Program Directory for information on allocating and loading z/OS Application Connectivity to DB2 for z/OS data sets.

Environment variables for the z/OS Application Connectivity to DB2 for z/OS feature

You need to set environment variables so that the operating system can locate the z/OS Application Connectivity to DB2 for z/OS feature.

The environment variables that you must set are:

PATH
Modify PATH to include the directory that contains the shell scripts that invoke IBM Data Server Driver for JDBC and SQLJ program preparation and debugging functions. If z/OS Application Connectivity to DB2 for z/OS is installed in /usr/lpp/jcct4v3, modify PATH as follows:

export PATH=/usr/lpp/jcct4v3/bin:$PATH

CLASSPATH
z/OS Application Connectivity to DB2 for z/OS contains the following class files:

db2jcc.jar or db2jcc4.jar
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 or 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 or later functions, and JDBC 3.0 or earlier functions.

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

sqlj.zip or sqlj4.zip
Include sqlj.zip in the CLASSPATH if you plan to prepare SQLJ.
applications that include only JDBC 3.0 or earlier functions. Include sqlj4.zip in the CLASSPATH if you plan to prepare SQLJ applications that include JDBC 4.0 or later functions, and JDBC 3.0 or earlier functions.

**Important:** Include sqlj.zip or sqlj4.zip in the CLASSPATH. Do not include both files. Do not include db2jcc.jar with sqlj4.zip, or db2jcc4.jar with sqlj.zip.

*db2jcc_license_cisuz.jar*
A license file that permits access to DB2 for z/OS servers.

Modify your CLASSPATH to include these files. If z/OS Application Connectivity to DB2 for z/OS is installed in /usr/lpp/jcct4v3, modify CLASSPATH as follows:

```bash
export CLASSPATH=/usr/lpp/jcct4v3/classes/db2jcc.jar:
/usr/lpp/jcct4v3/classes/db2jcc_javax.jar:
/usr/lpp/jcct4v3/classes/sqlj.zip:
/usr/lpp/jcct4v3/classes/db2jcc_license_cisuz.jar:
$CLASSPATH
```

540 Application Programming Guide and Reference for Java
Chapter 9. Installing the IBM Data Server Driver for JDBC and SQLJ on Db2 on Linux, UNIX, and Windows systems

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

Before you begin

Before you install the IBM Data Server Driver for JDBC and SQLJ, you need the following software.

- An SDK for Java, 1.4.2 or later.
  
  For all DB2 products except the IBM Data Server Runtime Client and the IBM Data Server Driver Package, the Db2 on Linux, UNIX, and Windows systems installation process automatically installs the SDK for Java, Version 5.
  
  If you want to use JDBC 4.0 functions, you need to install an SDK for Java, 6 or later.
  
  If you want to use JDBC 4.1 functions, you need to install an SDK for Java, 7 or later.
  
  If you plan to run JDBC or SQLJ applications on your system, but not to prepare them, you need a Java run-time environment only.

Important: Support for the SDK for Java 1.4.2 is deprecated for Java routines, and might be discontinued in a future release.

- JVM native threads support
  
  Any JVMs that run Java applications that access DB2 databases must include native threads support. You can specify native threads as the default thread support for some JVMs by setting the THREADS_FLAG environment variable to "native". Refer to the documentation for your Java environment for instructions on making native threads the default on your system.

- Java support for HP-UX clients and servers
  
  **HP-UX servers:** The IBM Data Server Driver for JDBC and SQLJ does not support databases that are in the HP-UX default character set, Roman8. Therefore, when you create a database on an HP-UX server that you plan to access with the IBM Data Server Driver for JDBC and SQLJ, you need to create the database with a different character set.
  
  **HP-UX clients and servers:** The Java environment on an HP-UX system requires special setup to run stored procedures under the IBM Data Server Driver for JDBC and SQLJ.

About this task

**Restriction:** If you install the IBM Data Server Driver for JDBC and SQLJ on a Windows 64-bit operating system, you cannot use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to connect to a Db2 on Linux, UNIX, and Windows systems instance from a 32-bit Java application.

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

1. During the Db2 on Linux, UNIX, and Windows systems installation process, select Java support on UNIX or Linux, or JDBC support on Windows. These selections are defaults. If you have already installed Db2 on Linux, UNIX, and Windows systems without JDBC support, you can run the installation process in Custom mode to add JDBC support.

Selection of Java support or JDBC support causes the installation process to perform the following actions:

a. Installs the IBM Data Server Driver for JDBC and SQLJ class files.
   The files are placed in the sqllib\java directory for Windows systems, or the sqllib/java directory for UNIX or Linux systems.
   The files names are:

   **db2jcc.jar or db2jcc4.jar**
   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 or 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 or later functions, and JDBC 3.0 or earlier functions.

   **sqlj.zip or sqlj4.zip**
   Include sqlj.zip in the CLASSPATH if you plan to prepare SQLJ applications that include only JDBC 3.0 or earlier functions.

   Include sqlj4.zip in the CLASSPATH if you plan to prepare SQLJ applications that include JDBC 4.0 or later functions, and JDBC 3.0 or earlier functions.

b. Modifies the CLASSPATH to include the IBM Data Server Driver for JDBC and SQLJ class files.

   **Important:** This step is performed automatically only for the db2jcc.jar and sqlj.zip file. If you are using the db2jcc4.jar file or the sqlj4.zip file, you must modify the CLASSPATH manually. Change db2jcc.jar to db2jcc4.jar or sqlj.zip to sqlj4.zip in the CLASSPATH.

   You also need to make this change in every DB2 command line window that you open.

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

   **Important:** Include sqlj.zip or sqlj4.zip in the CLASSPATH. Do not include both files. Do not include db2jcc.jar with sqlj4.zip, or db2jcc4.jar with sqlj.zip.

c. If IBM Data Server Driver for JDBC and SQLJ client license files exist, the installation process installs them and modifies the CLASSPATH to include them.

   The files are placed in the sqllib\java directory for Windows systems, or the sqllib/java directory for UNIX or Linux systems. The file names are:
### Table 106. IBM Data Server Driver for JDBC and SQLJ license files

<table>
<thead>
<tr>
<th>License file</th>
<th>Server to which license file permits a connection</th>
<th>Product that includes license file</th>
</tr>
</thead>
<tbody>
<tr>
<td>db2jcc_license_cisuz.jar</td>
<td>DB2 for z/OS</td>
<td>All Db2 Connect products</td>
</tr>
<tr>
<td></td>
<td>Db2 for IBM i</td>
<td></td>
</tr>
</tbody>
</table>

Client license files are not required for connections to Db2 on Linux, UNIX, and Windows systems, Cloudscape, or IBM Informix databases from the IBM Data Server Driver for JDBC and SQLJ version 3.50 or later.

d. Installs IBM Data Server Driver for JDBC and SQLJ native libraries for support of IBM Data Server Driver for JDBC and SQLJ type 2 connectivity. The files are placed in the sqllib\bin directory for Windows systems, or the sqllib/lib directory for UNIX or Linux systems.

The file names are:

- **libdb2jcct2.so**
  - For AIX®, HP-UX on IPF, Linux, and Solaris

- **db2jct2.dll**
  - For Windows

As an alternative to installing the IBM Data Server Driver for JDBC and SQLJ class files during installation, you can download the class files, and follow the steps above to configure the driver. You cannot download the IBM Data Server Driver for JDBC and SQLJ type 2 connectivity native libraries. To download the IBM Data Server Driver for JDBC and SQLJ class files, follow these steps:


2. Under Downloads and fixes, select View IBM Data Server Client Packages...

3. In the Refine my fix list window, select Show me more options.

4. 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.

5. On the Identify fixes page, type "Data Server Driver for JDBC" in the Text field.

6. On the Select fixes page, select the latest version of the IBM Data Server Driver for JDBC and SQLJ.

7. On the Download options page, select the options that are appropriate for you.

- Extract the zip file into an empty directory.
  - The zip file contains the following files:
    - **db2jcc.jar**
    - **db2jcc4.jar**
    - **sqlj.zip**
    - **sqlj4.zip**

- Copy the files to the locations that are specified in step 1 on page 542 above.

After you have downloaded the IBM Data Server Driver for JDBC and SQLJ class files, you need to follow the entire procedure that is described in this topic to install the driver.
2. Customize the driver-wide configuration properties, if any of the defaults are inappropriate.

3. Configure TCP/IP.
   Servers must be configured for TCP/IP communication in the following cases:
   • JDBC or SQLJ applications that use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.
   • JDBC or SQLJ applications that use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity, and specify server and port in the connection URL.
   Ensure that the TCP/IP listener is running. To activate the TCP/IP listener:
   a. Set the environment variable DB2COMM to TCPIP:
      ```
      db2set DB2COMM=TCPIP
      ```
   b. Update the database manager configuration file with the TCP/IP service name as specified in the services file:
      ```
      db2 update dbm cfg using SVCENAME TCP/IP-service-name
      ```
      The port number used for applets and SQLJ programs needs to be the same as the TCP/IP SVCENAME number used in the database manager configuration file.
   c. Execute the `db2stop` and `db2start` commands for the service name setting to take effect.

4. On Db2 on Linux, UNIX, and Windows systems servers on which you plan to run Java stored procedures or user-defined functions, update the database manager configuration to include the path where the SDK for Java is located. You can do this by entering commands similar to these on the server command line:
   • For database systems on UNIX or Linux:
     ```
     db2 update dbm cfg using JDK_PATH /home/db2inst/jdk15
     ```
     `/home/db2inst/jdk15` is the path where the SDK for Java is installed.
   • For database systems on Windows:
     ```
     db2 update dbm cfg using JDK_PATH c:\Program Files\jdk15
     ```
     `c:\Program Files\jdk15` is the path where the SDK for Java is installed.
   To verify the correct value for the JDK_PATH field in the DB2 database manager configuration, enter the following command on the database server:
   ```
   db2 get dbm cfg
   ```
   You might want to redirect the output to a file for easier viewing. The JDK_PATH field appears near the beginning of the output.

5. If you plan to call SQL procedures that are on Db2 on Linux, UNIX, and Windows systems servers from Java programs, and the date and time format that is associated with the territory code of the database servers is not the USA format, take the following actions:
   a. Set the DB2_SQLROUTINE_PREPOPTS registry variable on the database servers to indicate that the default datetime format is ISO:
      ```
      db2set DB2_SQLROUTINE_PREPOPTS="DATETIME ISO"
      ```
   b. Redefine any existing SQL procedures that you plan to call from Java programs.
   These steps are necessary to ensure that the calling application receives date and time values correctly.
6. If you plan to access DB2 for z/OS database servers with your Java applications, follow the instructions in "Special setup for accessing DB2 for z/OS servers from Java programs" in Developing Java Applications.

Special setup for accessing DB2 for z/OS servers from Java programs

If you plan to write JDBC or SQLJ applications that access DB2 for z/OS database servers, your IBM Data Server Driver for JDBC and SQLJ installation process requires additional steps.

Procedure

Follow these steps to allow connectivity to DB2 for z/OS servers:

1. If you plan to connect to any DB2 for z/OS data servers, install these PTFs on those database servers.

   Table 107. PTFs for DB2 for z/OS data servers

<table>
<thead>
<tr>
<th>DB2 for z/OS</th>
<th>PTF or APAR numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version 8</td>
<td>UQ93890, UK21849</td>
</tr>
<tr>
<td>Version 9</td>
<td>UK38087</td>
</tr>
</tbody>
</table>

2. Run the com.ibm.db2.jcc.DB2Binder utility to bind the DB2 packages that are used at the server by the IBM Data Server Driver for JDBC and SQLJ.

3. On DB2 for z/OS database servers, customize and run job DSNTIJMS. DSNTIJMS is located in data set prefix.SDSNSAMP. It performs the following functions:
   - Creates the following stored procedures to support DatabaseMetaData methods, tracing, and error message formatting.
     - SQLCOLPRIVILEGES
     - SQLCOLUMNS
     - SQLFOREIGNKEYS
     - SQLFUNCTIONS
     - SQLFUNCTIONCOLUMNS
     - SQLGETTYPEINFO
     - SQLPRIMARYKEYS
     - SQLPROCEDURECOLUMNS
     - SQLPROCEDURES
     - SQLPSEUDOCOLUMNS (DB2 for z/OS Version 10 or later)
     - SQLSPECIALCOLUMNS
     - SQLSTATISTICS
     - SQLTABLEPRIVILEGES
     - SQLTABLES
     - SQLUDTS
     - SQLCAMESSAGE
   - Creates the following tables to support efficient storing of data in CLOB or DBCLOB columns and the use of LOB locators for CLOB or DBCLOB retrieval:
     - SYSIBM.SYSDUMMYU
     - SYSIBM.SYSDUMMYA
     - SYSIBM.SYSDUMMYE

   An alternative way to create those tables is to run the com.ibm.db2.jcc.DB2LobTableCreator utility on the client, against each of the DB2 for z/OS servers.
4. Enable Unicode support for OS/390 and z/OS servers.

If any SQLJ or JDBC programs will use IBM Data Server Driver for JDBC and
SQLJ type 4 connectivity to connect to a DB2 for z/OS Version 7 server, the
OS/390 or z/OS operating system must support the Unicode UTF-8 encoding
scheme. This support requires OS/390 Version 2 Release 9 with APAR
OW44581, or a later release of OS/390 or z/OS, plus the OS/390 R8/R9/R10
Support for Unicode. Information APARs II13048 and II13049 contain additional
information.
Chapter 10. 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 Connection or DataSource property, or the db2.jcc.securityMechanism global configuration property.

You can set the securityMechanism 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.

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, an error occurs. In addition, any other mismatch in security mechanism support between the requester and the server results in an error.

Table 108. Security mechanisms supported by the IBM Data Server Driver for JDBC and SQLJ

<table>
<thead>
<tr>
<th>Security mechanism</th>
<th>securityMechanism property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>User ID and password</td>
<td>DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY</td>
</tr>
<tr>
<td>User ID only</td>
<td>DB2BaseDataSource.USER_ONLY_SECURITY</td>
</tr>
<tr>
<td>User ID and encrypted password</td>
<td>DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY</td>
</tr>
<tr>
<td>Encrypted user ID and encrypted password</td>
<td>DB2BaseDataSource.ENCRYPTED_USER_AND_PASSWORD_SECURITY</td>
</tr>
<tr>
<td>Encrypted user ID and encrypted security-sensitive data</td>
<td>DB2BaseDataSource.ENCRYPTED_USER_AND_DATA_SECURITY</td>
</tr>
<tr>
<td>Encrypted user ID, encrypted password, and encrypted security-sensitive data</td>
<td>DB2BaseDataSource.ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY</td>
</tr>
<tr>
<td>Kerberos</td>
<td>DB2BaseDataSource.KERBEROS_SECURITY</td>
</tr>
<tr>
<td>Plugin</td>
<td>DB2BaseDataSource.PLUGIN_SECURITY</td>
</tr>
<tr>
<td>Certificate authentication</td>
<td>DB2BaseDataSource.TLS_CLIENT_CERTIFICATE_SECURITY</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1998, 2017
Table 108. Security mechanisms supported by the IBM Data Server Driver for JDBC and SQLJ (continued)

<table>
<thead>
<tr>
<th>Security mechanism</th>
<th>securityMechanism property value</th>
</tr>
</thead>
</table>

Note:
1. DRDA encryption is not intended to provide confidentiality and integrity of passwords or data over a network that is not secure, such as the Internet. DRDA encryption uses an anonymous key exchange, Diffie-Hellman, which does not provide authentication of the server or the client. DRDA encryption is vulnerable to man-in-the-middle attacks.

Related concepts:
- “User ID and password security under the IBM Data Server Driver for JDBC and SQLJ”
- “User ID-only security under the IBM Data Server Driver for JDBC and SQLJ” on page 551
- “Kerberos security under the IBM Data Server Driver for JDBC and SQLJ” on page 554
- “Encrypted password, user ID, or data security under the IBM Data Server Driver for JDBC and SQLJ” on page 552

Related reference:
- “Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 234

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:

```java
import java.sql.*;  // JDBC base
...
String id = "dbadm";  // Set user ID
String pw = "dbadm";  // Set password
String url = "jdbc:db2://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:

```java
import java.sql.*;  // JDBC base
...
String url = 
  "jdbc:db2://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:

```java
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:db2://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:

```java
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:

```java
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( // Set security mechanism to
        com.ibm.db2.jcc.DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY); // user ID and password
```

**IBM Data Server Driver for JDBC and SQLJ type 2 connectivity with no user ID or password:** For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity, if you use user ID and password security, but you do not specify a user ID and password, the database system uses the external security environment, such as the RACF security environment, that was previously established for the user. For a CICS connection, you cannot specify a user ID or password.
**Valid characters in passwords:** All characters in the ASCII range X'20' (decimal 32) through X'7E' (decimal 126) are valid in passwords, except for the following characters:

- X'20' (space) at the end of a password. The IBM Data Server Driver for JDBC and SQLJ strips space characters at the end of a password.
- X'3B' (semicolon)
- Any characters that cannot be converted to EBCDIC characters, if passwords in plain text are sent to a data server.

**RACF password phrase security:** If you are connecting to a DB2 for z/OS that is configured for RACF protection, and the RACF version supports RACF password phrases, you can supply a RACF password phrase for the password property value, instead of a simple password. A password phrase must conform to the following rules:

- A password phrase is a character string that can consist of mixed-case letters, numbers, and special characters, including blanks.
- The length of the password phrase can be 9 to 100 characters, or 14 to 100 characters.
- Password phrases of between 9 and 13 characters are allowed when the new-password-phrase exit (ICHPWX11) is installed on the z/OS system, and the exit allows password phrases of fewer than 14 characters.
- A password phrase must not contain the user ID, as sequential uppercase or sequential lowercase characters.
- A password phrase must contain at least two alphabetic characters (A through Z or a through z).
- A password phrase must contain at least two non-alphabetic characters (numerics, punctuation, or special characters).
- A password phrase must not contain more than two consecutive characters that are identical.
- If a single quotation mark (') is part of the password phrase, the single quotation mark must be represented as two consecutive single quotation marks ('').

The following example uses a password phrase for a connection:

```java
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", "a*blcD12345 678"); // Set password phrase 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:db2://mvs1.sj.ibm.com:5021/san_jose"; // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties); // Create connection
```

**Related tasks:**

- “Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ” on page 13
- “Creating and deploying DataSource objects” on page 24
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:

```java
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:db2://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:

```java
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
```

Related tasks:

"Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ" on page 13
Encrypted password, user ID, or data security under the IBM Data Server Driver for JDBC and SQL/J

IBM Data Server Driver for JDBC and SQL/J supports encryption of user IDs, passwords, or data when Java applications access data servers.

Those security mechanisms use DRDA encryption. DRDA encryption is not intended to provide confidentiality and integrity of passwords or data over a network that is not secure, such as the Internet. DRDA encryption uses an anonymous key exchange, Diffie-Hellman, which does not provide authentication of the server or the client. DRDA encryption is vulnerable to man-in-the-middle attacks.

The IBM Data Server Driver for JDBC and SQL/J supports 56-bit DES (weak) encryption or 256-bit AES (strong) encryption. AES encryption is available with IBM Data Server Driver for JDBC and SQL/J type 4 connectivity only. You set the encryptionAlgorithm driver property to choose between 56-bit DES encryption (encryptionAlgorithm value of 1) and 256-bit AES encryption (encryptionAlgorithm value of 2). 256-bit AES encryption is used for a connection only if the database server supports it and is configured to use it.

If you use encrypted password security, encrypted user ID security, or encrypted user ID and encrypted password security from a DB2 for z/OS client, the Java Cryptography Extension, IBMJCE for z/OS needs to be enabled on the client. The Java Cryptography Extension is part of the IBM Developer Kit for z/OS, Java 2 Technology Edition. For information on how to enable IBMJCE, go to this URL on the web:  


For AES encryption, you need to get the unrestricted policy file for JCE. It is available at the following URL:  


Connections to Db2 for IBM 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 (ibmjcepovidre.jar) must be installed on your client. The IBM JCE is part of the IBM SDK for Java, Version 1.4.2 or later.

You can also use encrypted security-sensitive data in addition to encrypted user ID security or encrypted user ID and encrypted password security. You specify encryption of security-sensitive data through the ENCRYPTED_USER_AND_DATA_SECURITY or ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY securityMechanism value. ENCRYPTED_USER_AND_DATA_SECURITY is valid for connections to DB2 for z/OS servers only.

DB2 for z/OS or Db2 on Linux, UNIX, and Windows systems database servers encrypt the following data when you specify encryption of security-sensitive data:

• SQL statements that are being prepared, executed, or bound into a package
• Input and output parameter information
Before you can use encrypted security-sensitive data, the z/OS Integrated Cryptographic Services Facility needs to be installed and enabled on the z/OS operating system.

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 encrypted user ID and encrypted password security mechanism, with AES encryption:

```java
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", new String("" +
  com.ibm.db2.jcc.DB2BaseDataSource.ENCRYPTED_USER_AND_PASSWORD_SECURITY +
  "); // Set security mechanism to
  // user ID and encrypted password
properties.put("encryptionAlgorithm", "2"); // Request AES security
String url = "jdbc:db2://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:

```java
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_USER_AND_PASSWORD_SECURITY);
```
// Set security mechanism to
// User ID and encrypted password
ds.setEncryptionAlgorithm(2); // Request AES encryption

**Valid characters in passwords:** All characters in the ASCII range X'20' (decimal 32) through X'7E' (decimal 126) are valid in passwords, except for the following characters:
- X'20' (space) at the end of a password. The IBM Data Server Driver for JDBC and SQLJ strips space characters at the end of a password.
- X'3B' (semicolon)
- Any characters that cannot be converted to EBCDIC characters, if passwords in plain text are sent to a data server.

**RACF password phrase security:** If you are connecting to a DB2 for z/OS that is configured for RACF protection, and the RACF version supports RACF password phrases, you can supply a RACF password phrase for the password property value, instead of a simple password. A password phrase must conform to the following rules:
- A password phrase is a character string that can consist of mixed-case letters, numbers, and special characters, including blanks.
- The length of the password phrase can be 9 to 100 characters, or 14 to 100 characters.
- Password phrases of between 9 and 13 characters are allowed when the new-password-phrase exit (ICHPWX11) is installed on the z/OS system, and the exit allows password phrases of fewer than 14 characters.
- A password phrase must not contain the user ID, as sequential uppercase or sequential lowercase characters.
- A password phrase must contain at least two alphabetic characters (A through Z or a through z).
- A password phrase must contain at least two non-alphabetic characters (numeric, punctuation, or special characters).
- A password phrase must not contain more than two consecutive characters that are identical.
- If a single quotation mark (') is part of the password phrase, the single quotation mark must be represented as two consecutive single quotation marks ("').

**Related tasks:**
- "Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ" on page 13
- "Creating and deploying DataSource objects" on page 24
- "Connecting to a data source using the DataSource interface" on page 20

**Related reference:**
- "Properties for the IBM Data Server Driver for JDBC and SQLJ" on page 234

---

**Kerberos security under the IBM Data Server Driver for JDBC and SQLJ**

JDBC support for Kerberos security is available for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity only.

To enable JDBC support for Kerberos security, you also need to enable the following components of your software development kit (SDK) for Java:
- Java Cryptography Extension
See the documentation for your SDK for Java for information on how to enable these components.

There are three ways to specify Kerberos security for a connection:
- With a user ID and password
- Without a user ID or password
- With a delegated credential

**Kerberos security with a user ID and password**

For this case, Kerberos uses the specified user ID and password to obtain a ticket-granting ticket (TGT) that lets you authenticate to the database server.

You need to set the user, password, kerberosServerPrincipal, and securityMechanism properties. Set the securityMechanism property to com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY (11). The kerberosServerPrincipal property specifies the principal name that the database server registers with a Kerberos Key Distribution Center (KDC).

**For the DriverManager interface:** Set the user ID, password, Kerberos server, and security mechanism by setting the user, password, kerberosServerPrincipal, 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 Kerberos security mechanism with a user ID and password:

```java
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("password", "db2adm"); // Set password for the connection
properties.put("kerberosServerPrincipal", "sample/srvlsj.ibm.com@SRVLSJ.SJ.IBM.COM"); // Set the Kerberos server
properties.put("securityMechanism",
    new String("" +
        com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY + "))); // Set security mechanism to Kerberos

String url = "jdbc:db2://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, set the Kerberos server and security mechanism by invoking the DataSource.setKerberosServerPrincipal and DataSource.setSecurityMechanism methods after you create the DataSource object. For example:

```java
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 the DataSource object
db2ds.setDriverType(4); // Set the driver type
```
db2ds.setDatabaseName("san_jose"); // Set the location
db2ds.setUser("db2adm"); // Set the user
db2ds.setPassword("db2adm"); // Set the password
db2ds.setServerName("mvs1.sj.ibm.com"); // Set the server name
db2ds.setPortNumber(5021); // Set the port number
db2ds.setKerberosServerPrincipal("sample/srvlsj.ibm.com@SRVLSJ.SJ.IBM.COM"); // Set the Kerberos server
db2ds.setSecurityMechanism(com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY); // Set security mechanism to Kerberos

Kerberos security with no user ID or password

For this case, the Kerberos default credentials cache must contain a ticket-granting ticket (TGT) that lets you authenticate to the database server.

You need to set the kerberosServerPrincipal and securityMechanism properties. Set the securityMechanism property to com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY (11).

For the DriverManager interface: Set the Kerberos server and security mechanism by setting the kerberosServerPrincipal 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 Kerberos security mechanism without a user ID and password:

```java
import java.sql.*; // JDBC base
import com.ibm.db2.jcc.*; // IBM Data Server Driver for JDBC
... Properties properties = new Properties(); // Create a Properties object
properties.put("kerberosServerPrincipal", "sample/srvlsj.ibm.com@SRVLSJ.SJ.IBM.COM"); // Set the Kerberos server
properties.put("securityMechanism", new String("" + com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY + "))); // Set security mechanism to Kerberos
String url = "jdbc:db2://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, set the Kerberos server and security mechanism by invoking the DataSource.setKerberosServerPrincipal and DataSource.setSecurityMechanism methods after you create the DataSource object. For example:

```java
import java.sql.*; // JDBC base
import com.ibm.db2.jcc.*; // IBM Data Server Driver for JDBC
... DB2SimpleDataSource db2ds = new com.ibm.db2.jcc.DB2SimpleDataSource(); // Create the DataSource 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.setKerberosServerPrincipal("sample/srvlsj.ibm.com@SRVLSJ.SJ.IBM.COM");
    // Set the Kerberos server

db2ds.setSecurityMechanism(com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY);
    // Set security mechanism to
    // Kerberos

Kerberos security with a delegated credential from another principal

For this case, you authenticate to the database server using a delegated credential that another principal passes to you.

You need to set the kerberosServerPrincipal, gssCredential, and securityMechanism properties. Set the securityMechanism property to com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY (11).

For the DriverManager interface: Set the Kerberos server, delegated credential, and security mechanism by setting the kerberosServerPrincipal, and securityMechanism properties in a Properties object. Then invoke the form of the getConnection method that includes the Properties object as a parameter. For example, use code like this to set the Kerberos security mechanism without a user ID and password:

```java
import java.sql.*;       // JDBC base
import com.ibm.db2.jcc.*; // IBM Data Server Driver for JDBC
...

Properties properties = new Properties(); // Create a Properties object
properties.put("kerberosServerPrincipal", "sample/srvlsj.ibm.com@SRVLSJ.SJ.IBM.COM");
    // Set the Kerberos server
properties.put("gssCredential",delegatedCredential);
    // Set the delegated credential
properties.put("securityMechanism",
    new String("" +
        com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY + "));
    // Set security mechanism to
    // Kerberos
String url = "jdbc:db2://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, set the Kerberos server, delegated credential, and security mechanism by invoking the DataSource.setKerberosServerPrincipal, DataSource.setGssCredential, and DataSource.setSecurityMechanism methods after you create the DataSource object. For example:

```java
DB2SimpleDataSource db2ds = new com.ibm.db2.jcc.DB2SimpleDataSource();
    // Create the DataSource 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.setKerberosServerPrincipal("sample/srvlsj.ibm.com@SRVLSJ.SJ.IBM.COM");
    // Set the Kerberos server

db2ds.setGssCredential(delegatedCredential);
    // Set the delegated credential

db2ds.setSecurityMechanism(
```
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 on Linux, UNIX, and Windows systems 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, 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 database 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 DB2XDataSource 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:

```java
// 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:

```java
// The first item that was obtained from the previous getDB2TrustedPooledConnection
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 password for the new connection
// Use null when authentication is not required
String newPassword = null;
// 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,newPassword,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 on Linux, UNIX, and Windows systems 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.

Connections to all supported data servers can use server authentication. For server authentication, the server sends a certificate to the client, and the client confirms the identity of the server. Connections to DB2 for z/OS data servers can also use client authentication. For client authentication, the client sends a certificate to the server, and the server confirms the identity of the client. Client authentication can be used with SSL encryption or without SSL encryption.

To use SSL connections, you need to:
- Configure connections to the data server to use SSL.
- Configure your Java Runtime Environment to use SSL.

---

**Related tasks:**
- “Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ” on page 13
- “Creating and deploying DataSource objects” on page 24
- “Connecting to a data source using the DataSource interface” on page 20

**Related reference:**
- “Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 234

**Related concepts:**
- “IBM Data Server Driver for JDBC and SQLJ support for certificate authentication” on page 565

---

560    Application Programming Guide and Reference for Java
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 you begin

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.

Procedure

1. Set DB2BaseDataSource.sslConnection on a Connection or DataSource instance. Alternatively, set the db2.jcc.override.sslConnection or db2.jcc.sslConnection configuration parameter on the driver instance.

2. Optional: Set the location of the truststore and the truststore password. The truststore location can be set without the password, but it is best to set both values.
   a. Set DB2BaseDataSource.sslTrustStoreLocation on a Connection or DataSource instance to identify the location of the truststore. Alternatively, set the db2.jcc.override.sslTrustStoreLocation or db2.jcc.sslTrustStoreLocation configuration parameter on the driver instance. 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.
   b. Optional: Set DB2BaseDataSource.sslTrustStorePassword on a Connection or DataSource instance to identify the truststore password. Alternatively, set the db2.jcc.override.sslTrustStorePassword or db2.jcc.sslTrustStorePassword configuration parameter on the driver instance. 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.

3. Optional: Set DB2BaseDataSource.sslCipherSuites on a Connection or DataSource instance, if you do not want to use the default cipher suites that are enabled in the JRE (Java Runtime Environment). The driver enables only the cipher suites that you set.

4. Optional: Set DB2BaseDataSource.sslCertLocation on a Connection or DataSource to specify the location of a trusted certificate file. Alternatively, you can set the db2.jcc.override.sslCertLocation or db2.jcc.sslCertLocation configuration parameter on the driver instance.

If sslConnection property is set to true, and both sslCertLocation and sslTrustStoreLocation properties are configured, IBM Data Server Driver for JDBC and SQLJ gives precedence to the sslCertLocation property and ignores the sslTrustStoreLocation configuration. Therefore, only the certificate that is supplied through sslCertLocation is considered trusted for an SSL connection with a DB2 server.
Example

The following example demonstrates how to set the sslConnection property on a Connection instance:

```java
define properties = new java.util.Properties();
properties.put("user", "xxxx");
properties.put("password", "yyyy");
properties.put("sslConnection", "true");
define java.sql.Connection con =
define 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. An example procedure is provided. However, the procedure might be different depending on the Java Runtime Environment that you use.

Before you begin

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 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.

Procedure

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.
   
   **Example:** Suppose that the server certificate is stored in a file named cacerts. Issue the following keytool utility statement to read the certificate from file jcc.cacert, and store it in a truststore named cacerts.
   
   ```sh
define keytool -import -file jcc.cacert -keystore cacerts
```

   **Example:** Suppose that the server certificate is stored in a file named mydbserver.arm. Issue the following keytool utility statement to read the certificate from file mydbserver.arm, and store it in a truststore named mynewdbclient.jks.
   
   ```sh
define keytool -import -trustcacerts -alias myalias -file mydbserver.arm -keystore mynewdbclient.jks
```

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:
   
   ```sh
define 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 IBMJCE provider 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 IBMJCE provider and the IBMJCEFIPS 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 on page 564
    
  
  - **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:** If you need to run in FIPS-compliant mode, and you enabled FIPS mode in the IBMJSSE2 provider, use a java.security file similar to this example:

```java
# 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.IMBSASL
```

**Example:** If you need to run in FIPS-compliant mode, and you are using the IBMJSSE provider, use a java.security file similar to this example:

```java
# 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
```
Example: If you are using the SunJSSE provider, use a java.security file similar to this example:

```
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. This step is not necessary if you are using the SunJSSE provider and the Java Runtime Environment, 7 or later.

The format of SSL socket factory provider entries is shown:

```
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: When you enable FIPS mode in the IBMJSSE2 provider, include SSL socket factory provider entries in the java.security file:

```
# Set the SSL socket factory provider
ssl.SocketFactory.provider=com.ibm.jsse2.SSLSocketFactoryImpl
ssl.ServerSocketFactory.provider=com.ibm.jsse2.SSLServerSocketFactoryImpl
```

Example: When you enable FIPS mode in the IBMJSSE provider, include SSL socket factory provider entries in the java.security file:

```
# Set the SSL socket factory provider
ssl.SocketFactory.provider=com.ibm.fips.jsse.JSSESocketFactory
ssl.ServerSocketFactory.provider=com.ibm.fips.jsse.JSSEServerSocketFactory
```

Example: When you use the SunJSSE provider, and the Java Runtime Environment, 6 or earlier, include SSL socket factory provider entries:

```
# 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 562.

If the IBM Data Server Driver for JDBC and SQLJ property

```
DB2BaseDataSource.sslTrustStoreLocation,
```

```
db2.jcc.override.sslTrustStoreLocation, or db2.jcc.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,
db2.jcc.override.sslTrustStorePassword, or db2.jcc.sslTrustStorePassword
is set, its value overrides the javax.net.ssl.trustStore property value.

If you are using a trusted certificate file (if DB2BaseDataSource.sslCertLocation,
db2.jcc.override.sslCertLocation, or db2.jcc.sslCertLocation is set), the
javax.net.ssl.trustStore and javax.net.ssl.trustStorePassword values are not used.

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 that is named MySSL.java, which accesses a
data source by using an SSL connection. If you defined a truststore named
cacerts, then the following command sets the truststore name when you run the
application.

java -Djavax.net.ssl.trustStore=cacerts
MySSL

6. To enable the Common Access Card (IBM CAC) provider, overwrite the default
truststore and keystore definitions:

-Djavax.net.ssl.trustStoreType=Windows-ROOT
-Djavax.net.ssl.keyStoreType=Windows-MY

IBM Data Server Driver for JDBC and SQLJ support for certificate
authentication

The IBM Data Server Driver for JDBC and SQLJ provides support for client
support for certificate authentication for connections to DB2 for z/OS Version 10 or
later data servers.

Client certificate authentication security on a DB2 for z/OS data server supports
the use of digital certificates for mutual authentication by requesters and servers.
By using z/OS digital certificates, the Secure Socket Layer (SSL) protocol supports
server and client authentication during the handshake phase. A data server can
validate the certificates of a client at the server, which prevents the client from
obtaining a secure connection without an installation-approved certificate. The
authentication of the remote client's digital certificate is performed by Application
Transparent Transport Layer Security (AT-TLS) that is provided with the z/OS
Communications Server TCP/IP stack.

The IBM Data Server Driver for JDBC and SQLJ supports certificate authentication
for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity only.

You enable IBM Data Server Driver for JDBC and SQLJ certificate authentication by
specifying DB2BaseDataSource.TLS_CLIENT_CERTIFICATE_SECURITY as the value of
the securityMechanism Connection or DataSource property. If the target data server
supports client certificate authentication, and the mutual authentication succeeds,
the driver passes a valid Connection object to the application. If the data server
does not support client certificate authentication, or the connection was not
authenticated using AT-TLS and SSL, the driver throws DisconnectException.

You can use certificate authentication with or without a user ID or a password. If
the application does not provide a user ID or password, authentication is
performed at the network layer only. If a user ID or password is provided,
authentication is performed at the network layer and the data server layer.
To use SSL encryption with certificate authentication, you can set the sslConnection or DataSource property or the db2.jcc.override.sslConnection or db2.jcc.sslConnection configuration property to true.

The following example demonstrates how to enable certificate authentication and user ID and password security in a JDBC application.

```java
com.ibm.db2.jcc.DB2SimpleDataSource dataSource = new com.ibm.db2.jcc.DB2SimpleDataSource();
// Specify certificate authentication
dataSource.setSecurityMechanism(com.ibm.db2.jcc.DB2BaseDataSource.TLS_CLIENT_CERTIFICATE_SECURITY);
// Set a user ID and password to be passed to the data server
((com.ibm.db2.jcc.DB2BaseDataSource)dataSource).setUser("sysadm");
dataSource.setPassword("password");
// Identify the SSL truststore, keystore and their passwords
System.setProperty("javax.net.ssl.trustStore","c:/temp/SSL/cacerts");
System.setProperty("javax.net.ssl.trustStorePassword","password");
System.setProperty("javax.net.ssl.keyStore","c:/temp/SSL/myKS");
System.setProperty("javax.net.ssl.keyStorePassword","123456");
...
// Create a connection
con = dataSource.getConnection();
```

**Related information:**

- [DB2 for z/OS: Configuring TLS/SSL for Secure Client-Server Communications (IBM Redbooks)](https://www.ibm.com/redbooks/)

---

**Security for preparing SQLJ applications with the IBM Data Server Driver for JDBC and SQLJ**

You can provide security during SQLJ application preparation by allowing users to customize applications only and limiting access to a specific set of tables during customization. If the target data server is DB2 for z/OS, you can also provide security by allowing customers to prepare but not execute applications.

### Allowing users to customize only

You can use one of the following techniques to allow a set of users to customize SQLJ applications, but not to bind or run those applications:

- **Create a database system for customization only (recommended solution):**
  
  Follow these steps:

  1. Create a new DB2 subsystem. This is the customization-only system.
  2. On the customization-only system, define all the tables and views that are accessed by the SQLJ applications. The table or view definitions must be the same as the definitions on the DB2 subsystem where the application will be bound and will run (the bind-and-run system). Executing the DESCRIBE statement on the tables or views must give the same results on the customization-only system and the bind-and-run system.
  3. On the customization-only system, grant the necessary table or view privileges to users who will customize SQLJ applications.
  4. On the customization-only system, users run the sqlj command with the \(-\text{compile}=\text{true}\) option to create Java byte codes and serialized profiles for their programs. Then they run the db2sqljcustomize command with the \(-\text{automaticbind}=\text{NO}\) option to create customized serialized profiles.
  5. Copy the java byte code files and customized serialized profiles to the bind-and-run system.
6. A user with authority to bind packages on the bind-and-run system runs the 
db2sqljbind command on the customized serialized profiles that were copied 
from the customization-only system.

- **Use a stored procedure to do customization:** Write a Java stored procedure that 
customizes serialized profiles and binds packages for SQLJ applications on 
behalf of the end user. This Java stored procedure needs to use a JDBC driver 
package that was bound with one of the DYNAMICRULES options that causes 
dynamic SQL to be performed under a different user ID from the end user’s 
authorization ID. For example, you might use the DYNAMICRULES option 
DEFINEBIND or DEFINERUN to execute dynamic SQL under the authorization 
ID of the creator of the Java stored procedure. You need to grant EXECUTE 
authority on the stored procedure to users who need to do SQLJ customization. 
The stored does the following things:

  1. Receives the compiled SQLJ program and serialized profiles in BLOB input 
     parameters
  2. Copies the input parameters to its file system
  3. Runs db2sqljcustomize to customize the serialized profiles and bind the 
     packages for the SQLJ program
  4. Returns the customized serialized profiles in output parameters

- **Use a stand-alone program to do customization:** This technique involves 
writing a program that performs the same steps as a Java stored procedure that 
customizes serialized profiles and binds packages for SQLJ applications on 
behalf of the end user. However, instead of running the program as a stored 
procedure, you run the program as a stand-alone program under a library 
server.

**Allowing users to customize and bind only**

If the target data server is DB2 for z/OS Version 10 or later, you can allow users to 
customize and bind SQLJ applications, but not to execute the SQL statements in 
them, by granting those users the EXPLAIN privilege.

**Restricting table access during customization**

When you customize serialized profiles, you should do online checking, to give the 
application program information about the data types and lengths of table columns 
that the program accesses. By default, customization includes online checking.

Online checking requires that the user who customizes a serialized profile has 
authorization to execute PREPARE and DESCRIBE statements against SQL 
statements in the SQLJ program. That authorization includes the SELECT privilege 
on tables and views that are accessed by the SQL statements. If SQL statements 
contain unqualified table names, the qualifier that is used during online checking 
is the value of the db2sqljcustomize -qualifier parameter. Therefore, for online 
checking of tables and views with unqualified names in an SQLJ application, you 
can grant the SELECT privilege only on tables and views with a qualifier that 
matches the value of the -qualifier parameter.
Chapter 11. Java client support for high availability on IBM data servers

Client applications that connect to Db2 on Linux, UNIX, and Windows systems, 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 on Linux, UNIX, and Windows systems Db2 pureScale instance.

The following table provides links to server-side information about these features.

<table>
<thead>
<tr>
<th>Data server</th>
<th>Related topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Db2 on Linux, UNIX, and Windows systems</td>
<td>- Db2 pureScale: <a href="#">Db2 pureScale Feature roadmap</a></td>
</tr>
<tr>
<td></td>
<td>- Automatic client reroute: <a href="#">Db2 automatic client reroute roadmap</a></td>
</tr>
<tr>
<td>IBM Informix</td>
<td>Manage Cluster Connections with the Connection Manager</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Communicating with data sharing groups (DB2 Data Sharing Planning and Administration)</td>
</tr>
</tbody>
</table>

**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.

© Copyright IBM Corp. 1998, 2017
Java client support for high availability for connections to Db2 on Linux, UNIX, and Windows systems servers

Db2 on Linux, UNIX, and Windows systems 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, high availability support requires IBM Data Server Driver for JDBC and SQLJ version 3.58 or 4.8, or later.

High availability support for connections to Db2 on Linux, UNIX, and Windows systems 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.

Automatic client reroute capability for Java applications is available with IBM Data Server Driver for JDBC and SQLJ type 4 connectivity or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

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 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.
You can configure automatic client reroute to include the following additional capabilities:

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.

Alternate group support
Alternate group support allows the IBM Data Server Driver for JDBC and SQLJ to execute an application workload on one group at a time. A group can be a primary group or one of several alternate groups. Each group is a Db2 on Linux, UNIX, and Windows systems instance, which can be single-member instance, or a multiple-member instance, such a Db2 pureScale instance.

Workload balancing
Workload balancing can improve availability of a Db2 pureScale instance.

Workload balancing for Java applications is available with IBM Data Server Driver for JDBC and SQLJ type 4 connectivity only.

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 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems.

Configuration of Db2 on Linux, UNIX, and Windows systems automatic client reroute support for Java clients
For connections to Db2 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems 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:
   - Automatic client reroute is not enabled for default connections (jdbc:default:connection).
   - You cannot set the clientRerouteServerListJNDIName property or the clientRerouteServerListJNDICorext properties for a DriverManager connection.
   - Alternate server information is shared between DriverManager connections only if you create the connections with the same URL and properties.

2. 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:

<table>
<thead>
<tr>
<th>Property name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.db2.jcc.DB2ClientRerouteServerList.alternateServerName</td>
<td>String[]</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2ClientRerouteServerList.alternatePortNumber</td>
<td>int[]</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryServerName</td>
<td>String[]</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryPortNumber</td>
<td>int[]</td>
</tr>
</tbody>
</table>

      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).

Related tasks:

- “Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ” on page 13
- “Connecting to a data source using the DataSource interface” on page 20

Related reference:

- “Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products” on page 235

Example of enabling Db2 on Linux, UNIX, and Windows systems automatic client reroute support in Java applications

Java client setup for Db2 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems automatic client reroute support.

Suppose that your installation has a primary server and an alternate server with the following server names and port numbers:

<table>
<thead>
<tr>
<th>Server name</th>
<th>Port number</th>
</tr>
</thead>
<tbody>
<tr>
<td>srv1.sj.ibm.com</td>
<td>50000</td>
</tr>
<tr>
<td>srv3.sj.ibm.com</td>
<td>50002</td>
</tr>
</tbody>
</table>

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.

```java
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.

```java
// 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");
```
Configuration of Db2 on Linux, UNIX, and Windows systems workload balancing support for Java clients

To configure a IBM Data Server Driver for JDBC and SQLJ client application that connects to a Db2 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems workload balancing for Java applications.

**Table 110. Basic settings to enable workload support in Java applications**

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableSysplexWLB property</td>
<td>true</td>
</tr>
<tr>
<td>maxTransportObjects property</td>
<td>The maximum number of connections that the requester can make to the Db2 pureScale instance</td>
</tr>
</tbody>
</table>
Table 10. Basic settings to enable workload support in Java applications (continued)

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection address: server</td>
<td>The IP address of a member of a Db2 pureScale instance.</td>
</tr>
<tr>
<td>Connection address: port</td>
<td>The SQL port number for the Db2 pureScale instance.</td>
</tr>
<tr>
<td>Connection address: database</td>
<td>The database name</td>
</tr>
</tbody>
</table>

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 on Linux, UNIX, and Windows systems 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. Configuration properties for fine-tuning Db2 on Linux, UNIX, and Windows systems workload balancing support for connections from the IBM Data Server Driver for JDBC and SQLJ

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ configuration property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db2.jcc.maxRefreshInterval</td>
<td>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 10. The minimum valid value is 1.</td>
</tr>
<tr>
<td>db2.jcc.maxTransportObjectIdleTime</td>
<td>Specifies the maximum elapsed time in number of seconds before an idle transport is dropped. The default is 10. The minimum supported value is 0.</td>
</tr>
<tr>
<td>db2.jcc.maxTransportObjectWaitTime</td>
<td>Specifies the number of seconds that the client will wait for a transport to become available. The default is 1. The minimum supported value is 0.</td>
</tr>
<tr>
<td>db2.jcc.minTransportObjects</td>
<td>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.</td>
</tr>
</tbody>
</table>

Example of enabling Db2 on Linux, UNIX, and Windows systems workload balancing support in Java applications

Java client setup for Db2 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems 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
      java com.ibm.db2.jcc.DB2Jcc -version
      ```
   b. Find a line in the output like this, and check that \textit{nnn} is 3.58 or later.
   c. \textit{[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:
      
      - \textit{enableSysplexWLB}
      - \textit{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:

   \begin{table}[h]
   \centering
   \caption{Example of property settings for workload balancing for Db2 on Linux, UNIX, and Windows systems}
   \begin{tabular}{|l|l|}
   \hline
   Property & Setting \\
   \hline
   enableSysplexWLB & true \\
   maxTransportObjects & 80 \\
   db2.jcc.maxRefreshInterval & 10 \\
   \hline
   \end{tabular}
   \end{table}

   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:

   \textit{db2.jcc.maxTransportObjects=1000}

4. \textbf{Optional}: Specify alternate server names in the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties. This step is not necessary for enabling workload balancing. However, specifying an alternate server list is useful to ensure that the first connection is successful if the primary server is unavailable.

\section*{Operation of automatic client reroute for connections to Db2 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems 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 once 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 on Linux, UNIX, and Windows systems server when maxRetriesForClientReroute and retryIntervalForClientReroute are not set:

Suppose that the following properties are set for a connection to a database:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.NO (2)</td>
</tr>
<tr>
<td>serverName</td>
<td>host1</td>
</tr>
<tr>
<td>Property</td>
<td>Value</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>portNumber</td>
<td>port1</td>
</tr>
<tr>
<td>clientRerouteAlternateServerName</td>
<td>host2</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>port2</td>
</tr>
</tbody>
</table>

The following steps demonstrate an automatic client reroute scenario for a connection to a Db2 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems 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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.NO (2)</td>
</tr>
<tr>
<td>serverName</td>
<td>host1</td>
</tr>
<tr>
<td>portNumber</td>
<td>port1</td>
</tr>
<tr>
<td>db2.jcc.outputDirectory</td>
<td>/home/tmp</td>
</tr>
</tbody>
</table>

The following steps demonstrate an automatic client reroute scenario for a connection to a Db2 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems server when maxRetriesForClientReroute and retryIntervalForClientReroute are set for multiple retries: Suppose that the following properties are set for a connection to a database:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.NO (2)</td>
</tr>
<tr>
<td>serverName</td>
<td>host1</td>
</tr>
<tr>
<td>portNumber</td>
<td>port1</td>
</tr>
<tr>
<td>clientRerouteAlternateServerName</td>
<td>host2</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>port2</td>
</tr>
<tr>
<td>maxRetriesForClientReroute</td>
<td>3</td>
</tr>
<tr>
<td>retryIntervalForClientReroute</td>
<td>2</td>
</tr>
</tbody>
</table>

The following steps demonstrate an automatic client reroute scenario for a connection to a Db2 on Linux, UNIX, and Windows systems 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.

Related concepts:
“Example of enabling client affinities in Java clients for Db2 on Linux, UNIX, and Windows systems connections” on page 590
“Configuration of Db2 on Linux, UNIX, and Windows systems automatic client reroute support for Java clients” on page 571

Related reference:
“Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products” on page 235

Operation of alternate group support for connections to Db2 on Linux, UNIX, and Windows systems

Alternate group support is a high-availability feature that allows the IBM Data Server Driver for JDBC and SQLJ to execute an application workload on one group at a time. A group can be a primary group or one of several alternate groups.

Each group is a Db2 on Linux, UNIX, and Windows systems instance, which can be single-member instance, or a multiple-member instance, such a Db2 pureScale instance.

You can control whether seamless failover behavior is in effect for alternate group support by setting the enableAlternateGroupSeamlessACR Connection or DataSource property.

You enable alternate group support by providing the addresses of the alternate groups in the alternateGroupServerName, alternateGroupPortNumber, and alternateGroupDatabaseName Connection or DataSource properties. You provide the address of the primary group in the serverName, portNumber, and databaseName Connection or DataSource properties.

In an HADR environment, you enable alternate group support by providing the addresses of the standby clusters in the alternateGroupServerName, alternateGroupPortNumber, and alternateGroupDatabaseName Connection or DataSource properties. The address for the primary connection is the address of the primary cluster.
To avoid unpredictable client reroute behavior, follow these practices:

- Do not specify the alternateGroupServerName, alternateGroupPortNumber, and alternateGroupDatabaseName properties, and also execute the Db2 on Linux, UNIX, and Windows systems command UPDATE ALTERNATE SERVER FOR DATABASE on the data server.
- When you set up a Db2 pureScale instance that consists of two HADR clusters, execute the command UPDATE ALTERNATE SERVER FOR DATABASE command, but do not specify the alternateGroupServerName, alternateGroupPortNumber, and alternateGroupDatabaseName properties.
- When you set up a Db2 pureScale instance that consists of more than two HADR clusters, specify the alternateGroupServerName, alternateGroupPortNumber, and alternateGroupDatabaseName properties, but do not execute the command UPDATE ALTERNATE SERVER FOR DATABASE command.

Alternate group support allows failover from the primary group to any alternate group. After a connection on a DataSource instance fails over to an alternate group, subsequent connections on the DataSource instance connect directly to that alternate group. If an alternate group becomes unavailable during a connection, the connection can fail over to any other alternate group. A group is unavailable if any of the following conditions are true:

- The IBM Data Server Driver for JDBC and SQLJ receives a communication failure from all members of the group.
- The members of the group return SQL errors to the IBM Data Server Driver for JDBC and SQLJ to indicate that the driver should drop the connection to that group. For example, if the role of a primary HADR database changes to a standby role, the database server might return an SQL error to the driver that indicates that the request cannot be issued on an HADR standby database. The driver interprets that SQL error as a group failure.

Alternate group support operates in the following way:

- For the first connection, the process is as follows:
  1. After the first connection to the primary group fails, the driver attempts to connect the application to the alternate group that is specified by the first set of values in the alternateGroupServerName, alternateGroupPortNumber, and alternateGroupDatabaseName properties.
  2. The driver attempts to connect to the alternate group. The number of attempts and the amount of time between retries depends on whether a cached server list exists on the client, and whether maxRetriesForClientReroute and retryIntervalForClientReroute are set:
     - If a cached server list does not exist, and maxRetriesForClientReroute and retryIntervalForClientReroute are not set, the driver makes at most five attempts to connect to the alternate group, with no time between retries.
     - If a cached server list exists, and maxRetriesForClientReroute and retryIntervalForClientReroute are not set, the driver retries the connection for up to two minutes.
     - If maxRetriesForClientReroute and retryIntervalForClientReroute are set, the driver retries the connection the number of times that is specified by maxRetriesForClientReroute, with an interval between retries that is specified by retryIntervalForClientReroute.
  3. If all attempts to connect to the first alternate group fail, the driver attempts to connect the application to the alternate group that is specified by the next set of values in the alternateGroupServerName, alternateGroupPortNumber,
and alternateGroupDatabaseName properties. The driver uses the same rules for the number of retries and interval between retries as it uses for the connection to the first alternate group.

The driver continues this process until all alternate groups have been tried, or a connection has been established.

4. If a connection is not established after all alternate group members have been tried, the driver returns SQL error -4499 to the application.

* After a connection has been established, the process is as follows:
  1. If an error occurs for which automatic client reroute can be performed, the IBM Data Server Driver for JDBC and SQLJ attempts to reconnect to the same group. The amount of time that the driver spends on retries or the number of retries depends on the maxRetriesForClientReroute property setting. However, if the data server returns an SQL error that the driver interprets as a group failure, such as SQL error -1776, the driver does not retry the connection to the same group. SQL error -1776 indicates that the driver is attempting to connect to a standby instance.
  2. If reconnection to the same group is unsuccessful, the driver attempts to connect to each alternate group that is specified by the alternateGroupServerName, alternateGroupPortNumber, and alternateGroupDatabaseName properties, starting with the next alternate group in the list after the one that failed. The amount of time that the driver spends on retries or the number of retries depends on the maxRetriesForClientReroute property setting. If the last group in the alternate server list has been tried, the driver attempts to connect to the first server in the alternate group list, and continues through the list again.
  3. If a connection is not established, the driver returns SQL error -4499 to the application.

maxRetriesForClientReroute controls the amount of time that the driver spends on retries and the number of retries in the following way:

- If maxRetriesForClientReroute is not set, the driver tries to connect to each group in the list until one of the following events occurs:
  - The connection succeeds.
  - The connection has been retried for two minutes.
  - The data server returns SQL error -1776.

If the connection is unsuccessful after two minutes, or if the data server returns SQL error -1776, the driver tries the connection to the next alternate server. This process continues until (2*number-of-groups) minutes have elapsed.

- If maxRetriesForClientReroute is set, the driver tries to connect to each group in the list until one of the following events occurs:
  - The connection succeeds.
  - The driver has attempted to connect to all members of the group the number of times that is specified by maxRetriesForClientReroute.
  - The data server returns SQL error -1776.

If the connection is unsuccessful after the number of retries exceeds the value that is specified by maxRetriesForClientReroute, or if the data server returns SQL error -1776, the driver tries the connection to the next alternate server. This process continues until (maxRetriesForClientReroute*number-of-groups) attempts have occurred.
Examples

Suppose that three groups are defined as shown in the following table:

<table>
<thead>
<tr>
<th>Group</th>
<th>Server type</th>
<th>Members in group</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Primary</td>
<td>A1, A2</td>
</tr>
<tr>
<td>B</td>
<td>Alternate</td>
<td>B1</td>
</tr>
<tr>
<td>C</td>
<td>Alternate</td>
<td>C1, C2, C3°</td>
</tr>
</tbody>
</table>

The address of A1 is the primary group address. The addresses of B1 and C1 are the alternate group addresses. Workload balancing and automatic client reroute are enabled.

The following example shows how properties are set on a DataSource object to enable alternate group support and define the alternate group list:

```java
com.ibm.db2.jcc.DB2SimpleDataSource ds = new com.ibm.db2.jcc.DB2SimpleDataSource();
... // Set other properties
ds.setDatabaseName("mydb2a"); // Set primary group database
    // for group A
ds.setServerName("myservera.ibm.com"); // Set primary server name
    // for group A
ds.setPortNumber(5912); // Set primary port number
    // for group A
ds.setAlternateGroupDatabaseName("mydb2b,mydb2c"); // Set alternate group databases
    // for groups B and C
ds.setAlternateGroupServerName("myserverb.ibm.com,myserverc.ibm.com"); // Set alternate group server names
    // for groups B and C
ds.setAlternateGroupPortNumber(5912,5912); // Set alternate group port numbers
    // for groups B and C
```

Example: Alternate group failover when maxRetriesForClientReroute is not set:
The following steps demonstrate an alternate group scenario when maxRetriesForClientReroute is not set:

1. The driver connects to group A.
2. While A1 is executing an SQL statement, A1 returns a communication error to the driver.
3. The driver unsuccessfully retries the connection to the members in group A for two minutes.
4. The driver tries the connection to group B (server B1), which is the first server in the alternate group list. The driver tries to connect for two minutes. The connection is unsuccessful.
5. The driver tries the connection to group C, which is the next server in the alternate group list.
6. The connection to group C succeeds.
7. SQL work is performed on the connection to group C.
8. While C2 is executing an SQL statement, C2 returns SQL error -1776 to the driver, indicating that it has switched to a standby role.
9. The driver retries the connection to the members in group A, which has taken over the primary role. The connection to group A succeeds.
10. While A1 is executing an SQL statement, A1 returns a communication error to
    the driver.
11. The driver unsuccessfully retries the connection to the members in group A
    for two minutes.
12. The driver unsuccessfully tries the connection to group B (server B1) for two
    minutes.
13. The driver unsuccessfully retries the connection to the members in group C
    for two minutes.
14. The driver has retried for six minutes (three groups * two minutes per group)
    so it returns an SQLException with SQL error -4499 to the application.

Example: Alternate group failover when maxRetriesForClientReroute is set: The
following steps demonstrate an alternate group scenario when
maxRetriesForClientReroute is set to 2:
1. The driver connects to group A.
2. While A1 is executing an SQL statement, A1 returns a communication error to
   the driver.
3. The driver unsuccessfully retries the connection to the members in group A
   twice.
4. The driver tries the connection to group B (server B1), which is the first server
   in the alternate group list. The driver tries to connect twice. The connection is
   unsuccessful.
5. The driver tries the connection to group C (server C1), which is the next
   server in the alternate group list.
6. The connection to group C succeeds.
7. SQL work is performed on the connection to group C.
8. While C2 is executing an SQL statement, C2 returns SQL error -1776 to the
   driver, indicating that it has switched to a standby role.
9. The driver retries the connection to the members in group A, which has taken
   over the primary role. The connection succeeds.
10. While A1 is executing an SQL statement, A1 returns a communication error to
    the driver.
11. The driver unsuccessfully retries the connection to the members in group A
    twice.
12. The driver unsuccessfully tries the connection to group B (server B1) twice.
13. The driver unsuccessfully retries the connection to the members in group C
    twice.
14. The driver has retried six times (three groups * two retries per group) so it
    returns an SQLException with SQL error -4499 to the application.

Operation of workload balancing for connections to Db2 on
Linux, UNIX, and Windows systems

Workload balancing (also called transaction-level workload balancing) for
connections to Db2 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems 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. When the first SQL statement in a transaction runs, if the IBM Data Server Driver for JDBC and SQLJ receives a communication failure because the data server drops the connection or the blockingReadConnectionTimeout value was exceeded, the driver retries the SQL statement 10 times before it reports an error. On every retry, the driver closes the existing transport, obtains a new transport and then executes the transaction. During these retries, if the maxRetriesForClientReroute and retryIntervalForClientReroute properties are set, their values apply only to the process of obtaining a new transport during each retry.

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 on Linux, UNIX, and Windows systems servers**

Failover for automatic client reroute can be seamless or non-seamless. If failover for connections to Db2 on Linux, UNIX, and Windows systems 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 on Linux, UNIX, and Windows systems 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.

**Related reference:**

"Error codes issued by the IBM Data Server Driver for JDBC and SQLJ” on page 489

**Client affinities for Db2 on Linux, UNIX, and Windows systems**

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 on Linux, UNIX, and Windows systems 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 11.3. Property settings to enable client affinities for Java applications

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.YES (1)</td>
</tr>
<tr>
<td>clientRerouteAlternateServerName</td>
<td>A comma-separated list of the primary server and alternate servers</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>A comma-separated list of the port numbers for the primary server and alternate servers</td>
</tr>
<tr>
<td>enableSeamlessFailover</td>
<td>DB2BaseDataSource.YES (1) for seamless failover; DB2BaseDataSource.NO (2) or enableSeamlessFailover not specified for no seamless failover</td>
</tr>
<tr>
<td>maxRetriesForClientReroute</td>
<td>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.</td>
</tr>
<tr>
<td>retryIntervalForClientReroute</td>
<td>The number of seconds to wait between retries. The default is no wait.</td>
</tr>
<tr>
<td>affinityFailbackInterval</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Related reference:**

“Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products” on page 235
Example of enabling client affinities in Java clients for Db2 on Linux, UNIX, and Windows systems 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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.YES (1)</td>
</tr>
<tr>
<td>clientRerouteAlternateServername</td>
<td>host1,host2,host3</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>port1,port2,port3</td>
</tr>
<tr>
<td>maxRetriesForClientReroute</td>
<td>3</td>
</tr>
<tr>
<td>retryIntervalForClientReroute</td>
<td>2</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.YES (1)</td>
</tr>
<tr>
<td>clientRerouteAlternateServername</td>
<td>host1,host2,host3</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>port1,port2,port3</td>
</tr>
<tr>
<td>maxRetriesForClientReroute</td>
<td>3</td>
</tr>
<tr>
<td>retryIntervalForClientReroute</td>
<td>2</td>
</tr>
<tr>
<td>affinityFailbackInterval</td>
<td>300</td>
</tr>
</tbody>
</table>

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 elapsed. 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.

Related concepts:
"Configuration of client affinities for Java clients for Db2 on Linux, UNIX, and Windows systems connections" on page 589

Related reference:
"Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products" on page 235
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 clientRerouteServerListJNDINaming 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>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableSysplexWLB property</td>
<td>true</td>
</tr>
<tr>
<td>maxTransportObjects property</td>
<td>The maximum number of connections that the requester can make to the high-availability cluster</td>
</tr>
<tr>
<td>Connection address: server</td>
<td>The IP address of a Connection Manager. See &quot;Setting server and port properties for connecting to a Connection Manager&quot; on page 594.</td>
</tr>
</tbody>
</table>
Table 114. Basic settings to enable IBM Informix high availability support in Java applications (continued)

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection address: port</td>
<td>The SQL port number for the Connection Manager. See &quot;Setting server and port properties for connecting to a Connection Manager.&quot;</td>
</tr>
<tr>
<td>Connection address: database</td>
<td>The database name</td>
</tr>
</tbody>
</table>

If you want to enable the connection concentrator, but you do not want to enable workload balancing, you can use these properties.

Table 115. Settings to enable the IBM Informix connection concentrator without workload balancing in Java applications

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableSysplexWLB property</td>
<td>false</td>
</tr>
<tr>
<td>enableConnectionConcentrator property</td>
<td>true</td>
</tr>
</tbody>
</table>

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 116. Properties for fine-tuning IBM Informix high-availability support for connections from the IBM Data Server Driver for JDBC and SQLJ

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ configuration property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db2.jcc.maxTransportObjectIdleTime</td>
<td>Specifies the maximum elapsed time in number of seconds before an idle transport is dropped. The default is 10. The minimum supported value is 0.</td>
</tr>
<tr>
<td>db2.jcc.maxTransportObjectWaitTime</td>
<td>Specifies the number of seconds that the client will wait for a transport to become available. The default is 1. The minimum supported value is 0.</td>
</tr>
<tr>
<td>db2.jcc.minTransportObjects</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

Related tasks:
- "Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ” on page 13

Related reference:
- "IBM Data Server Driver for JDBC and SQLJ configuration properties” on page 295

- "Common IBM Data Server Driver for JDBC and SQLJ properties for DB2 for z/OS and IBM Informix” on page 276
- "Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products” on page 235

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. `[jdbc] 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>
<thead>
<tr>
<th>Property</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableSysplexWLB</td>
<td>true</td>
</tr>
<tr>
<td>maxTransportObjects</td>
<td>80</td>
</tr>
<tr>
<td>db2.jcc.maxRefreshInterval</td>
<td>10</td>
</tr>
</tbody>
</table>

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. 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=1000

c. Include the directory that contains DB2JccConfiguration.properties in the
   CLASSPATH concatenation.

Related concepts:

“Configuration of IBM Informix high-availability support for Java clients” on page 593

Related reference:

“IBM Data Server Driver for JDBC and SQLJ configuration properties” on page 295

“Common IBM Data Server Driver for JDBC and SQLJ properties for DB2 for z/OS
and IBM Informix” on page 276

“Common IBM Data Server Driver for JDBC and SQLJ properties for all supported
database products” on page 235

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
enabled:
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:
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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.NO (2)</td>
</tr>
<tr>
<td>serverName</td>
<td>host1</td>
</tr>
<tr>
<td>portNumber</td>
<td>port1</td>
</tr>
<tr>
<td>clientRerouteAlternateServerName</td>
<td>host2</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>port2</td>
</tr>
<tr>
<td>maxRetriesForClientReroute</td>
<td>3</td>
</tr>
<tr>
<td>retryIntervalForClientReroute</td>
<td>2</td>
</tr>
</tbody>
</table>

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.

Related reference:
- “Error codes issued by the IBM Data Server Driver for JDBC and SQLJ” on page 489
- “Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products” on page 235

**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.
   When the first SQL statement in a transaction runs, if the IBM Data Server
   Driver for JDBC and SQLJ receives a communication failure because the data
   server drops the connection or the blockingReadConnectionTimeout value was
   exceeded, the driver retries the SQL statement 10 times before it reports an
   error. On every retry, the driver closes the existing transport, obtains a new
   transport and then executes the transaction. During these retries, if the
   maxRetriesForClientReroute and retryIntervalForClientReroute properties are
   set, their values apply only to the process of obtaining a new transport during
   each retry.

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.

**Related reference:**

“Error codes issued by the IBM Data Server Driver for JDBC and SQLJ” on page 489

---

**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>
<thead>
<tr>
<th>Property Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.YES (1)</td>
</tr>
<tr>
<td>clientRerouteAlternateServerName</td>
<td>A comma-separated list of the primary server and alternate servers</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>A comma-separated list of the port numbers for the primary server and alternate servers</td>
</tr>
<tr>
<td>enableSeamlessFailover</td>
<td>DB2BaseDataSource.YES (1) for seamless failover; DB2BaseDataSource.NO (2) or enableSeamlessFailover not specified for no seamless failover</td>
</tr>
<tr>
<td>maxRetriesForClientReroute</td>
<td>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.</td>
</tr>
<tr>
<td>retryIntervalForClientReroute</td>
<td>The number of seconds to wait between retries. The default is no wait.</td>
</tr>
<tr>
<td>affinityFailbackInterval</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Related reference:**

“Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products” on page 235

**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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.YES (1)</td>
</tr>
<tr>
<td>clientRerouteAlternateServerName</td>
<td>host1,host2,host3</td>
</tr>
</tbody>
</table>
### Table

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>port1,port2,port3</td>
</tr>
<tr>
<td>maxRetriesForClientReroute</td>
<td>3</td>
</tr>
<tr>
<td>retryIntervalForClientReroute</td>
<td>2</td>
</tr>
</tbody>
</table>

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:

### Table

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.YES (1)</td>
</tr>
<tr>
<td>clientRerouteAlternateServerName</td>
<td>host1,host2,host3</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>port1,port2,port3</td>
</tr>
</tbody>
</table>
### 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>
<thead>
<tr>
<th>Data sharing access method</th>
<th>Description</th>
<th>Sysplex workload balancing possible?</th>
</tr>
</thead>
</table>
| Group access                | A requester uses the DB2 group IP address 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 |
Table 119. Data sharing access methods and Sysplex workload balancing  (continued)

<table>
<thead>
<tr>
<th>Data sharing access method</th>
<th>Description</th>
<th>Sysplex workload balancing possible?</th>
</tr>
</thead>
</table>
| 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 TCP/IP access methods (DB2 Data Sharing Planning and Administration).

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 data server through any available member of a Sysplex. Reconnection to another member is called failover. Automatic client reroute can be seamless when the application is rerouted, and the application does not receive an error after a network failure to a data sharing member. An example of a situation in which automatic reroute can be seamless is when a member is shut down for maintenance.

Sysplex workload balancing during migration of a data sharing group to DB2 for z/OS V10: 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 9.1 to Version 10 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 9.1 new-function mode to Version 10 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
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 and automatic client reroute for Java clients

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 and automatic client reroute, 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.

You should always configure Sysplex workload balancing and automatic client reroute together. When you configure a client to use Sysplex workload balancing, automatic client reroute is also enabled. Therefore, you need to change property settings that are related to automatic client reroute only to fine tune automatic client reroute operation.

The following table describes the basic property settings for Java applications.

<table>
<thead>
<tr>
<th>Data sharing access method</th>
<th>IBM Data Server Driver for JDBC and SQLJ setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group access</td>
<td>enableSysplexWLB property</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>Connection address:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>server</td>
<td>The group IP address or domain name of the data sharing group</td>
</tr>
<tr>
<td></td>
<td>port</td>
<td>The SQL port number for the DB2 location</td>
</tr>
<tr>
<td></td>
<td>database</td>
<td>The DB2 location name that is defined during installation</td>
</tr>
</tbody>
</table>
### Table 120. Basic settings to enable Sysplex high availability support in Java applications (continued)

<table>
<thead>
<tr>
<th>Data sharing access method</th>
<th>IBM Data Server Driver for JDBC and SQLJ setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member-specific access</td>
<td>enableSysplexWLB property</td>
<td>true</td>
</tr>
<tr>
<td>Connection address:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>server</td>
<td>The group IP address or domain name of the data sharing group</td>
<td></td>
</tr>
<tr>
<td>port</td>
<td>The port number for the DB2 location alias</td>
<td></td>
</tr>
<tr>
<td>database</td>
<td>The name of the DB2 location alias that represents a subset of the members of the data sharing group</td>
<td></td>
</tr>
<tr>
<td>Group access or member-specific access</td>
<td>commandTimeout</td>
<td>Specifies the maximum time in seconds that an application that runs under the IBM Data Server Driver for JDBC and SQLJ waits for any kind of request to the data server to complete before the driver throws an SQLException. The wait time includes time to obtain a transport, perform failover if needed, send the request, and wait for a response. The default is 0.</td>
</tr>
<tr>
<td></td>
<td>connectionTimeout</td>
<td>Specifies the maximum time in seconds that the IBM Data Server Driver for JDBC and SQLJ waits for a reply from a data sharing group when the driver attempts to establish a connection. If the driver does not receive a reply after the amount of time that is specified by connectionTimeout, it throws an SQLException with SQL error code -4499. The default is 0. If connectionTimeout is set to a positive value, that value overrides any other timeout values that are set on a connection, such as loginTimeout.</td>
</tr>
<tr>
<td></td>
<td>maxTransportObjects</td>
<td>Specifies the maximum number of connections that the requester can make to the data sharing group. The default is 1000. To determine the maxTransportObjects value, multiply the expected number of concurrent active connections to the DB2 for z/OS data sharing group by the number of members in the data sharing group.</td>
</tr>
</tbody>
</table>

Additional properties are available for fine tuning Sysplex workload balancing and automatic client reroute. You should initially set up Sysplex workload balancing using only the basic properties. In most cases, you should not need to set any of the additional properties.

The following IBM Data Server Driver for JDBC and SQLJ Connection or DataSource properties can be used to fine-tune Sysplex workload balancing and automatic client reroute:
- blockingReadConnectionTimeout
- enableSeamlessFailover
- loginTimeout
- maxRetriesForClientReroute
- memberConnectTimeout
retryIntervalForClientReroute

The following IBM Data Server Driver for JDBC and SQLJ configuration properties can be used to fine-tune Sysplex workload balancing and automatic client reroute:
- \texttt{db2.jcc.maxRefreshInterval}
- \texttt{db2.jcc.maxTransportObjectIdleTime}
- \texttt{db2.jcc.maxTransportObjectWaitTime}
- \texttt{db2.jcc.minTransportObjects}

Related reference:
- “IBM Data Server Driver for JDBC and SQLJ configuration properties” on page 295
- “Common IBM Data Server Driver for JDBC and SQLJ properties for DB2 for z/OS and IBM Informix” on page 276
- “Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products” on page 235

Example of enabling DB2 for z/OS Sysplex workload balancing and automatic client reroute in Java applications

Java client setup for Sysplex workload balancing and automatic client reroute 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 and automatic client reroute 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 Classification attributes (DB2 Performance).

- DB2 for z/OS, set up for data sharing

Example of setup with WebSphere Application Server for z/OS

This example assumes that you are using WebSphere Application Server for z/OS. 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 in UNIX System Services
      \texttt{java com.ibm.db2.jcc.DB2Jcc -version}
   b. Find a line in the output like this, and check that \textit{nnn} is 3.50 or later.

   \[
   \text{Driver: IBM Data Server Driver for JDBC and SQLJ Architecture nnn xxx}
   \]
2. In the WebSphere Application Server administrative console, set the IBM Data Server Driver for JDBC and SQLJ data source property enableSysplexWLB to true, to enable Sysplex workload balancing. Enabling Sysplex workload balancing enables automatic client reroute by default.

3. In the WebSphere Application Server administrative console, set other properties for which the defaults are unacceptable. Modify these WebSphere Application Server connection properties. The following settings are recommended when enableSysplexWLB is set to true:

<table>
<thead>
<tr>
<th>Connection property</th>
<th>Recommended setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reap Time</td>
<td>0</td>
<td>Specifies the interval, in seconds, between runs of the pool maintenance thread. The Reap Time interval affects performance. Because connections are not physical connections, disabling pool maintenance by setting this value to 0 is recommended.</td>
</tr>
<tr>
<td>Aged Timeout</td>
<td>0</td>
<td>Specifies the interval in seconds before a physical connection is discarded. Setting Aged Timeout to 0 allows connections to remain in the pool indefinitely.</td>
</tr>
<tr>
<td>Purge Policy</td>
<td></td>
<td>Specifies how to purge connections when a stale connection or fatal connection error is detected. Because Sysplex workload balancing isolates WebSphere Application Server from stale connections or fatal connections errors, FailingConnectionOnly is the recommended setting.</td>
</tr>
</tbody>
</table>

The Maximum Connections value does not need to be changed. Member Connections specifies the maximum number of physical connections that you create in your pool. It does not control the number of physical connections to a data sharing group. With Sysplex workload balancing, connections are logical, and use transports to associate a connection to a data sharing member. Physical connections are managed by transport pools in the driver. The maxTransportObjects property controls the maximum number of connections to the group.

4. Optional: 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. Beginning with versions 3.63 and 4.13 of the IBM Data Server Driver for JDBC and SQLJ, the default values should work for most environments.

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 configuration properties in the DB2JccConfiguration.properties file:
   
   
   property=value

   
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 and automatic client reroute by following these steps:
   a. Issue the following command in UNIX System Services
      
      ```
      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 and automatic client reroute 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. Enabling Sysplex workload balancing enables automatic client reroute by default. Set any other properties for which the defaults are unacceptable. For most users, the default values do not need to be changed.

   ```
   java.util.Properties properties = new java.util.Properties();
   properties.put("user", "xxxx");
   properties.put("password", "yyyy");
   properties.put("enableSysplexWLB", "true");
   java.sql.Connection con =
   java.sql.DriverManager.getConnection(url, properties);
   ```

3. Optional: 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. Beginning with versions 3.63 and 4.13 of the IBM Data Server Driver for JDBC and SQLJ, the default values should work for most environments. 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 configuration properties in the DB2JccConfiguration.properties file:
      ```
      property=value
      ```
   c. Include the directory that contains DB2JccConfiguration.properties in the CLASSPATH concatenation.

Related concepts:
- "Configuration of Sysplex workload balancing and automatic client reroute for Java clients" on page 609

Related reference:
- "IBM Data Server Driver for JDBC and SQLJ configuration properties" on page 295
- "Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products" on page 235

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. When the first SQL statement in a transaction runs, if the IBM Data Server Driver for JDBC and SQLJ receives a communication failure because the data server drops the connection or the blockingReadConnectionTimeout value was exceeded, the driver retries the SQL statement 10 times before it reports an error. On every retry, the driver closes the existing transport, obtains a new transport and then executes the transaction. During these retries, if the maxRetriesForClientReroute and retryIntervalForClientReroute properties are set, their values apply only to the process of obtaining a new transport during each retry.

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. Automatic client reroute should never be enabled when Sysplex workload balancing is disabled.

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. The IBM Data Server Driver for JDBC and SQLJ uses the distributed IP address as the group IP address to establish the initial connection to the data sharing group. If the connection to the group IP address fails, the connection is retried five times, with no wait between retries.

2. After a transaction has been established, if an SQL statement in the transaction fails, the application receives SQL error -30108 with reason code 2. The IBM Data Server Driver for JDBC and SQLJ does not acquire an active transport.

3. The application retries the transaction.

4. The driver attempts to acquire a transport to each member of the data sharing group in the order of their calculated priorities (WLM member weights) until a transport is acquired. The driver does not attempt to acquire a transport to the member to which the connection failed in step 2.

5. If the driver cannot acquire a transport, the driver attempts to contact the group IP address to check for any members that have become available.

6. If the driver still cannot acquire a transport, the driver continues to execute steps 4 and 5 until a transport is acquired, or the maxRetriesForClientReroute value is reached.

Related reference:
“Error codes issued by the IBM Data Server Driver for JDBC and SQLJ” on page 489

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 that were carried over from the failing data sharing member to the new (failover) data sharing member were the settings at the most recent commit point, or the settings at the point of failure. 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.

Seamless failover is attempted once. If a data sharing member on which seamless
failover is attempted goes down, failover to another data sharing member is
non-seamless.

---

**Failover support with IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS**

When you use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on
DB2 for z/OS, you can configure connections to a data sharing group so that when
a connection to a data sharing member fails, new connections switch automatically
to an alternative member of the DB2 data sharing group that is running on the
same LPAR.

To enable this function, you set the ssid Connection or DataSource property, or the
db2.jcc.ssid configuration property to the group attachment name or subgroup
attachment name that is associated with a data sharing group. The Connection or
DataSource property value overrides the configuration property value.

When a Java application uses IBM Data Server Driver for JDBC and SQLJ type 2
connectivity to connect to a data source that represents a data sharing group for
which a group attachment name or subgroup attachment name is defined, the IBM
Data Server Driver for JDBC and SQLJ connects to a member of the data sharing
group. While the data sharing member is active, all new type 2 connections to the
data sharing group also connect to that same data sharing member. If the data
sharing member terminates, existing connections to that member terminate.
However, when new type 2 connections to the data sharing group are requested,
DB2 for z/OS connects the application to a member of the data sharing group that
is active on the same LPAR.

**Related concepts:**

“Environment variables for the IBM Data Server Driver for JDBC and SQLJ” on page 520

**Related reference:**

“IBM Data Server Driver for JDBC and SQLJ properties for DB2 for z/OS” on page 282
Chapter 12. 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 data server threads. When JDBC reuses physical 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 physical connection to the data source. When the application establishes a connection to a data server, the data server creates a new physical connection to the data source. When the application calls the java.sql.Connection.close method, the data server terminates the physical connection to the data source.

In contrast, with connection pooling, a java.sql.Connection object is a temporary, logical representation of a physical data source connection. The physical data source connection can be serially reused by logical java.sql.Connection instances. The application can use the logical java.sql.Connection object in exactly the same manner as it uses a java.sql.Connection object when there is no connection pooling support.

With connection pooling, when a JDBC application invokes the DataSource.getConnection method, the data source determines whether an appropriate physical connection exists. If an appropriate physical connection exists, the data source returns a java.sql.Connection instance to the application. When the JDBC application invokes the java.sql.Connection.close method, JDBC does not close the physical data source connection. Instead, JDBC closes only JDBC resources, such as Statement or ResultSet objects. The data source returns the physical connection to the connection pool for reuse.

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 13. IBM Data Server Driver for JDBC and SQLJ
statement caching

The IBM Data Server Driver for JDBC and SQLJ can use an internal statement cache to improve the performance of Java applications by caching and pooling prepared statements.

Internal statement caching is available for connections that use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, or for connections that use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

You enable internal statement caching in any of the following ways:
- By setting one of the following properties to a positive value:
  - com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements, for objects that are created using the javax.sql.ConnectionPoolDataSource interface.
  - com.ibm.db2.jcc.DB2XADataSource.maxStatements, for objects that are created using the javax.sql.XADataSource interface.
  - com.ibm.db2.jcc.DB2SimpleDataSource.maxStatements, for objects that are created using the com.ibm.db2.jcc.DB2SimpleDataSource interfaces.
- By setting the maxStatements property in a URL, and passing the URL to the DriverManager.getConnection method.

When internal statement caching is enabled, the IBM Data Server Driver for JDBC and SQLJ can cache PreparedStatement objects, CallableStatement objects, and JDBC resources that are used by SQLJ statements when those objects or resources are logically closed. When you explicitly or implicitly invoke the close method on a statement, you logically close the statement.

Reuse of a previously cached statement is transparent to applications. The statement cache exists for the life of an open connection. When the connection is closed, the driver deletes the statement cache and closes all pooled statements.

A logically open statement becomes ineligible for caching under either of the following circumstances:
- An exception occurs on the statement.
- JDBC 4.0 method Statement.setPoolable(false) is called.

When the IBM Data Server Driver for JDBC and SQLJ attempts to cache a statement, and the internal statement cache is full, the driver purges the least recently used cached statement, and inserts the new statement.

The internal statement cache is purged under the following conditions:
- A SET statement is issued that affects target objects of the SQL statement.
- A SET statement is executed that the IBM Data Server Driver for JDBC and SQLJ does not recognize.
- The IBM Data Server Driver for JDBC and SQLJ detects that a property that modifies target objects of the SQL statement was modified during connection reuse. currentSchema is an example of a property that modifies target objects of an SQL statement.
In a Java program, you can test whether the internal statement cache is enabled by issuing the `DatabaseMetaData.supportsStatementPooling` method. The method returns `true` if the internal statement cache is enabled.

The IBM Data Server Driver for JDBC and SQLJ does not check whether the definitions of target objects of statements in the internal statement cache have changed. If you execute SQL data definition language statements in an application, you need to disable internal statement caching for that application.

The internal statement cache requires extra memory. If memory becomes constrained, you can increase the JVM size, or decrease the value of `maxStatements`.

**Related reference:**

“Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products” on page 235
Chapter 14. IBM Data Server Driver for JDBC and SQLJ type 4 connectivity JDBC and SQLJ distributed transaction support

The IBM Data Server Driver for JDBC and SQLJ in the z/OS environment supports distributed transaction management when you use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

This support implements the Java 2 Platform, Enterprise Edition (J2EE) Java Transaction Service (JTS) and Java Transaction API (JTA) specifications, and conforms to the X/Open standard for global transactions (Distributed Transaction Processing: The XA Specification, available from http://www.opengroup.org). IBM Data Server Driver for JDBC and SQLJ distributed transaction support lets Enterprise Java Beans (EJBs) and Java servlets that run under WebSphere Application Server Version 5.01 and above participate in a distributed transaction system.

JDBC and SQLJ distributed transaction support provides similar function to JDBC and SQLJ global transaction support. However, JDBC and SQLJ global transaction support is available with IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS only.

JDBC and SQLJ distributed transaction support is available for connections to DB2 for z/OS, DB2, and DB2 for IBM i servers.

A distributed transaction system consists of a resource manager, a transaction manager, and transactional applications. The following table lists the products and programs in the z/OS environment that provide those components.

<table>
<thead>
<tr>
<th>Distributed transaction system component</th>
<th>Component function provided by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource manager</td>
<td>DB2 for z/OS or DB2</td>
</tr>
<tr>
<td>Transaction manager</td>
<td>WebSphere Application Server or another application server</td>
</tr>
<tr>
<td>Transactional applications</td>
<td>JDBC or SQLJ applications</td>
</tr>
</tbody>
</table>

Your client application programs that run under the IBM Data Server Driver for JDBC and SQLJ can use distributed transaction support for connections to DB2 for z/OS or DB2 servers.

In JDBC or SQLJ applications, distributed transactions are supported for connections that are established using the DataSource interface. A connection is normally established by the application server.

Related concepts:
Chapter 15, “JDBC and SQLJ global transaction support,” on page 627

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 49 outlines code that executes local transactions.

```java
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 49. 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 50 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.

```java
javax.transaction.UserTransaction utx;
// Use the begin method on a UserTransaction object to indicate
// the beginning of a distributed transaction.
ux.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.
ux.commit();
...
```

Figure 50. Example of a distributed transaction under an application server

Figure 51 on page 623 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 51. 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;

global 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.XAResource.XA_RDONLY)
        {
            // The second connection is read-only, so just commit the
            // first connection.
            xares1.commit(xid1, false);
        }
    } else if(rc1 == javax.transaction.xa.XAResource.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.XAResource.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.
Chapter 15. JDBC and SQLJ global transaction support

JDBC and SQLJ global transaction support lets Enterprise Java Beans (EJB) and Java servlets access DB2 for z/OS relational data within global transactions.

WebSphere Application Server provides the environment to deploy EJBs and servlets, and RRS provides the transaction management.

JDBC and SQLJ global transaction support provides similar function to JDBC and SQLJ distributed transaction support. However, JDBC and SQLJ distributed transaction support is available with IBM Data Server Driver for JDBC and SQLJ type 4 connectivity on DB2 for z/OS or DB2.

You can use global transactions in JDBC or SQLJ applications. Global transactions are supported for connections that are established using the DriverManager or the DataSource interface.

The best way to demonstrate global transactions is to contrast them with local transactions. With local transactions, you call the commit or rollback methods of the Connection class to make the changes to the database permanent and indicate the end of each unit or work. Alternatively, you can use the setAutoCommit(true) method to perform a commit operation after every SQL statement. The following code shows an example of a local transaction.

```java
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
...
```

In contrast, applications cannot call the commit, rollback, or setAutoCommit(true) methods on the Connection object when the applications are in a global transaction. With global transactions, the commit or rollback methods on the Connection object do not indicate transaction boundaries. Instead, your applications let WebSphere manage transaction boundaries. Alternatively, you can use DB2-customized Java Transaction API (JTA) interfaces to indicate the boundaries of transactions. Although DB2 for z/OS does not implement the JTA specification, the methods for delimiting transaction boundaries are available with the JDBC driver. The following code demonstrates the use of the JTA interfaces to indicate global transaction boundaries.

```java
javax.transaction.UserTransaction utx;
// Use the begin method on a UserTransaction object to indicate
// the beginning of a global 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 global transaction.
utx.commit();
...
```
Related concepts:

Chapter 14, “IBM Data Server Driver for JDBC and SQLJ type 4 connectivity JDBC and SQLJ distributed transaction support,” on page 621
Chapter 16. 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 2 connectivity, the recommended method is to start the trace by setting the db2.jcc.override.traceFile property and the db2.jcc.t2zosTraceFile 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: 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 3: 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 “Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products” on page 235 for a list of traceLevel settings. See “Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 234 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 4:

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 “Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products” on page 235 for a list of traceLevel settings. See “Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 234 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 5: 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:db2://sysmvs1.stl.ibm.com:5021/san_jose" + 
   "traceFile=/u/db2p/jcctrace;" + 
   "traceLevel" + com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS + ";";

Procedure 6: 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 “Examples of using configuration properties to start a JDBC trace” on page 632.

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” on page 634.

Collecting SQLJ trace data during customization or bind

To collect trace data to diagnose problems during the SQLJ customization or bind process, specify the -tracelevel and -tracefile options when you run the db2sqljcustomize or db2sqljbind bind utility.

Formatting information about an SQLJ serialized profile

The profp utility formats information about each SQLJ clause in a serialized profile. The format of the profp utility is:

►►profp—serialized-profile-name-----------------------------------------------◄◄
Run the profp utility on the serialized profile for the connection in which the error occurs. If an exception is thrown, a Java stack trace is generated. You can determine which serialized profile was in use when the exception was thrown from the stack trace.

**Formatting information about an SQLJ customized serialized profile**

The db2sqljprint utility formats information about each SQLJ clause in a serialized profile that is customized for the IBM Data Server Driver for JDBC and SQLJ.

Run the db2sqljprint utility on the customized serialized profile for the connection in which the error occurs.

**Related reference:**

"db2sqljprint - SQLJ profile printer" on page 517

---

**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**

```
java com.ibm.db2.jcc.DB2Jcc -version -db2ConnectVersion license-file-path
  -configuration -expirationDate
  -expirationDateWithLicenseType -help
  url-spec: -user user-ID -password password -sql-spec -tracing

url-spec:
  -url jdbc:db2://server/database:port

sql-spec:
  -sql 'SELECT * FROM SYSCAT.SYSDUMMY1'

```

**DB2Jcc parameters**
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://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 not contain any code to do 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 doing a circular trace with a fixed number of files and fixed file size**

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 set of trace files.
- The maximum number of trace files that are written for each connection is 4.
- When all trace files are full, the trace overwrites existing trace data, beginning with the first trace file that was written.
- The maximum size of each trace file is 4 MB.
- Each trace file name begins with jcc.log, and is written into a directory named /Trace.
- If trace files with the same names already exist, the trace data is overwritten.

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.

The configuration property settings look like this:

- db2.jcc.traceFile=jcc.log
- db2.jcc.traceOption=1
- db2.jcc.traceFileSize=4194304
- db2.jcc.traceFileCount=4
- db2.jcc.traceFileAppend=false

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
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. During execution of the program, the IBM Data Server Driver for JDBC and SQLJ writes 17 MB of data for the first connection, and 10 MB of data for the second connection.

When the program completes, the following files contain the trace data:

- /Trace/jcc.log_global_0.1
- /Trace/jcc.log_global_0.2
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 52. Example of tracing under the IBM Data Server Driver for JDBC and SQLJ

```java
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
        }
    }
    // Other methods for tracing...
}
```

/Trace/jcc.log_global_0.1 contains the last 1 MB of trace data that is written for the first connection, which overwrites the first 1 MB of trace data that was written for that connection.

Related reference:
"IBM Data Server Driver for JDBC and SQLJ configuration properties" on page 295
((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)庭");
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)庭");
s2.close();

c.close();

try{
classname
}
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] 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]");
}
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_DRA境内,floats and
    // com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS represent
    // int values. Those constants cannot be used directly in the

String databaseURL = "jdbc:db2://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


**Example:** The following set of configuration property settings cause error messages, dump pool error messages, and transport pool statistics 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:

```
group: DSNDB port: 446 hwmpo: 20 twte: 5 ngipr: 50 ttcpto: 2
  member: port: 446 DB1A hwmiut: 6 hwmt: 2 tmct: 1 trto: 0
  member: port: 446 DB1B hwmiut: 7 hwmt: 2 tmct: 1 trto: 0
  member: port: 446 DB1C hwmiut: 6 hwmt: 1 tmct: 0 trto: 1
```

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.

- **aoc**
  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.

group
The data sharing group for which transport pool statistics were gathered.

port
The port number of the data sharing group or member.

hwmpo
The maximum number of pool objects that were created since the pool was created.

twte
The number of times that the maxTransportWaitTime value was exceeded since the pool was created.

ngipr
The number of times that the group IP address was used since the pool was created.

ttcpto
The total number of times that members of the data sharing group had a connection timeout when they were establishing a new connection.

member
The member of the data sharing group for which transport statistics were gathered.

hwmiut
The maximum number of in-use transports for the data sharing member since the pool was created.

hwmt
The maximum number of transports that have been allocated to the data sharing member since the pool was created.

tmct
The number of times that the memberConnectTimeout value was reached for the data sharing member since the pool was created.

trto
The number of times that a read timeout occurred for the data sharing member since the pool was created.
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:

```java
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 17. Tracing IBM Data Server Driver for JDBC and SQLJ C/C++ native driver code

To debug applications that use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, you might need to trace the C/C++ native driver code.

Procedure

To collect, format, and print the trace data for the C/C++ native driver code, follow these steps:

1. Enable tracing of C/C++ native driver code by setting a value for the db2.jcc.t2zosTraceFile global configuration property.
   That value is the name of the file to which the IBM Data Server Driver for JDBC and SQLJ writes the trace data.
2. Run the db2jcctrace command from the z/OS UNIX System Services command line.
   By default, the trace data goes to stdout. You can pipe the data to another file.

Example

Suppose that db2.jcc.t2zosTraceFile has this setting:

```
db2.jcc.t2zosTraceFile=/SYSTEM/tmp/jdbctraceNative
```

Execute this command to format all available trace data for the C/C++ native driver code, and send the output to stdout:

```
db2jcctrace format flow /SYSTEM/tmp/jdbctraceNative
```

Related concepts:

“Customization of IBM Data Server Driver for JDBC and SQLJ configuration properties” on page 522

Related reference:

“db2jcctrace - Format IBM Data Server Driver for JDBC and SQLJ trace data for C/C++ native driver code”

---

**db2jcctrace - Format IBM Data Server Driver for JDBC and SQLJ trace data for C/C++ native driver code**

db2jcctrace writes formatted trace data for traces of C/C++ native driver code under IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

By default, the trace data is written to stdout. You can pipe the output to any file.

**db2jcctrace syntax**

```
  db2jcctrace format flow input-file-name
```

© Copyright IBM Corp. 1998, 2017
Notes:
1 You must specify one of these parameters.

**db2jcctrace parameters**

**format**
Specifies that the output trace file contains formatted trace data.
Abbreviation: fmt

**flow**
Specifies that the output trace file contains control flow information.
Abbreviation: flw

**information**
Specifies that the output trace file contains information about the trace, such as the version of the driver, the time at which the trace was taken, and whether the trace file wrapped or was truncated. This information is also included in the output trace file when you specify format or flow.
Abbreviation: inf or info

**input-file-name**
Specifies the name of the file from which db2jcctrace is to read the unformatted trace data. input-file-name is the same as the value of the db2jcc.t2zosTraceFile global configuration parameter.

Related concepts:
"Customization of IBM Data Server Driver for JDBC and SQLJ configuration properties" on page 522

Related tasks:
Chapter 17, "Tracing IBM Data Server Driver for JDBC and SQLJ C/C++ native driver code," on page 641
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.
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:

```java
```

The trace records with system monitor information look similar to this:

```java
import java.sql.*;
import com.ibm.db2.jcc.*;
public class TestSystemMonitor
{
    public static void main(String[] args)
    {
        String url = "jdbc:db2://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 = ((DB2Connection)conn).getDB2SystemMonitor();
            systemMonitor.enable(true);
            systemMonitor.start(DB2SystemMonitor.RESET_TIMES);
            Statement stmt = conn.createStatement();
            int numUpd = stmt.executeUpdate(
                "UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'");
            systemMonitor.stop();
            System.out.println("Server elapsed time (microseconds)=" + systemMonitor.getServerTimeMicros());
            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 53. Example of using DB2SystemMonitor methods to collect system monitoring data
[jcc][SystemMonitor:start]
...
[jcc][SystemMonitor:stop] core: 565.67ms | network: 211.695ms | server: 207.771ms
Information resources for DB2 10 for z/OS and related products

Information about DB2 10 for z/OS and products that you might use in conjunction with DB2 10 is available online in IBM Knowledge Center or on library websites.

Obtaining DB2 for z/OS publications

Current DB2 10 for z/OS publications are available from the following websites:


Links to IBM Knowledge Center and the PDF version of each publication are provided.

DB2 for z/OS publications are also available for download from the IBM Publications Center [http://www.ibm.com/shop/publications/order].

In addition, books for DB2 for z/OS are available on a CD-ROM that is included with your product shipment:

- DB2 10 for z/OS Licensed Library Collection, LK5T-7390, in English. The CD-ROM contains the collection of books for DB2 10 for z/OS in PDF format. Periodically, IBM refreshes the books on subsequent editions of this CD-ROM.

Installable information center

You can download or order an installable version of the Information Management Software for z/OS Solutions Information Center, which includes information about DB2 10 for z/OS, QMF™, IMS, and many DB2 Tools for z/OS products. You can install this information center on a local system or on an intranet server. For more information, see [http://www-01.ibm.com/support/knowledgecenter/ SSEPEK_11.0.0/com.ibm.db2z11.doc/src/alltoc/installabledzic.html]
Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

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 give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785 US

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

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 jurisdictions 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 Director of Licensing IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785 US

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.

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 actual people or business enterprises 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 shown below:

© (your company name) (year).
Portions of this code are derived from IBM Corp. Sample Programs.
© Copyright IBM Corp. (enter the year or years).

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

**Programming interface information**

This information is intended to help you write applications that use Java to access DB2 10 for z/OS servers. This book primarily documents General-use Programming Interface and Associated Guidance Information provided by DB2 10 for z/OS.
General-use Programming Interface and Associated Guidance Information

General-use Programming Interfaces allow the customer to write programs that obtain the services of DB2 10 for z/OS.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered marks 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.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo 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.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Privacy policy considerations

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at http://www.ibm.com/privacy and IBM’s Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled “Cookies, Web Beacons and Other Technologies” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at http://www.ibm.com/software/info/product-privacy.
Glossary

The glossary is available in IBM Knowledge Center.

See the Glossary topic for definitions of DB2 for z/OS terms.
Index

Special characters

- Infinity retrieving in Java applications 231

A

accessibility keyboard x shortcut keys x

ALLOW DEBUG MODE

CREATE PROCEDURE parameter 197

alternate groups

Db2 on Linux, UNIX, and Windows systems 582

application development

high availability

class connections to IBM Informix 602

direct connections to DB2 for z/OS servers 615

JDBC

application programming 9 SQLJ 125

application programming for high availability

class connections to Db2 on Linux, UNIX, and Windows systems 588

assignment-clause

SQLJ 357

generated keys retrieving for DELETE, JDBC application 79

retrieving for INSERT, JDBC application 77

retrieving for MERGE, JDBC application 79

retrieving for UPDATE, JDBC application 79

retrieving in JDBC application 77

auto-commit modes

default JDBC 113

automatic client reroute

client applications 569

DB2 for z/OS 614

IBM Informix servers 597

automatic client reroute support, client operation 578

automatically generated keys retrieving

DELETE statement, JDBC application 79

INSERT statement, JDBC application 77

JDBC applications 77

MERGE statement, JDBC application 79

UPDATE statement, JDBC application 79

B

batch queries

JDBC 41

batch updates

JDBC 33 SQLJ 146

BatchUpdateException exception

retrieving information 120

binary XML format

Java applications 103

binding SQLJ applications

accessing multiple servers 210

C

CallableStatement class 54

center

.client affinities .NET 588, 603

CLI 588, 603

IBM Data Server Driver for JDBC and SQLJ 588, 589, 603, 604

client affinities, example of enabling

Java clients 590, 604

client application

automatic client reroute 569

high availability 569

transaction-level load balancing 569

client authentication

IBM Data Server Driver for JDBC and SQLJ 565

client configuration, automatic client reroute support

Db2 on Linux, UNIX, and Windows systems 572

client configuration, high-availability support

IBM Informix 593

client configuration, Sysplex workload balancing

Db2 for z/OS 609

client configuration, workload balancing support

Db2 on Linux, UNIX, and Windows systems 575

client info properties

IBM Data Server Driver for JDBC and SQLJ 88, 89

class

alternate groups

connections to Db2 on Linux, UNIX, and Windows systems 582

automatic client reroute

connections to DB2 for z/OS 614

connections to IBM Informix 597

commands

db2sqlbind 513
db2sqlprint 517

sqlj 498

SQLJ 498

comments

JDBC applications 29 SQLJ applications 140

commits

SQLJ transactions 184

transactions

JDBC 112

configuration

JDBC 522

SQLJ 522

configuration properties

customizing 522

details 295

parameters 522

configuring

Java Runtime Environment 562

connection context

class 127

closing 186

default 127

object 127

connection declaration clause

SQLJ 350

© Copyright IBM Corp. 1998, 2017 655
connection pooling overview 617
connections closing importance 123, 186
data sources using SQLJ 127
DataSource interface 20 existing 133
context clause SQLJ 353, 354

D

data retrieving JDBC 39
data server connection testing with DB2Jcc 629
data sources connecting to DriverManager 13
JDBC 11
JDBC DataSource 20
data type mappings Java types to other types 219
DatabaseMetaData methods 26

databases compatibility IBM Data Server Driver for JDBC and SQLJ 5
DataSource interface SQL
connection technique 3 130
connection technique 4 132

DataSource objects creating 24
deploying 24
date value adjustment JDBC applications 226
SQLJ applications 226

DB2 for z/OS
binding packages 502
client configuration, Sysplex workload balancing 609
direct connections 613, 615
server access from Java programs 545
Sysplex support overview 606

DB2 for z/OS versions associated IBM Data Server Driver for JDBC and SQLJ versions 6

Db2 on Linux, UNIX, and Windows systems
client configuration, automatic client reroute support 572
client configuration, workload balancing support 575
high-availability support 570
workload balancing 586

Db2 on Linux, UNIX, and Windows systems high availability support, example of enabling IBM Data Server Driver for JDBC and SQLJ 574

Db2 on Linux, UNIX, and Windows systems versions associated IBM Data Server Driver for JDBC and SQLJ versions 7

Db2 on Linux, UNIX, and Windows systems workload balancing support, example of enabling IBM Data Server Driver for JDBC and SQLJ 576

Db2 on Linux, UNIX, and Windows systems, connections application programming for high availability 588

DB2BlobFileReference class 393
DB2CallableStatement interface 393
DB2ClientRerouteServerList class 400
DB2BlobFileReference class 401

DB2Connection interface 402
DB2ConnectionPoolDataSource class 420
DB2DatabaseMetaData interface 422
DB2DataSource class 424

DB2Diagnosable class retrieving the SQLCA 184, 185
DB2Diagnosable interface 423

DB2Driver class 425

DB2ExceptionFormatter class 426

DB2FileReference class 426

DB2Jcc utility details 631
testing a data server connection 629

DB2JCCPlugin interface 427
db2jctrace command 641

DB2JSONResultSet interface 428

DB2ParameterMetaData interface 429

DB2PooledConnection interface 430

DB2PoolMonitor class 432

DB2PreparedStatement interface 435

DB2ResultSet interface 450

DB2ResultSetMetaData interface 454

DB2RowID interface 455

DB2SimpleDataSource class definition 24
details 455

DB2sqlca class 456
db2sqljbind command 513
db2sqljcustomize command 502
db2sqljprint
formation JCC customized profile 631
db2sqljprint command
details 517
formatting information about SQLJ customized profile 629

DB2Statement interface 457

DB2SystemMonitor interface 459

DB2TraceManager class 463

DB2TraceManagerMXBean interface 466

DB2Types class 469

DB2Version class 469

DB2XADatasource class 471

DB2xml interface 473

DB2xmlAsBlobFileReference class 475

DB2xmlAsClobFileReference class 476

DBBatchUpdateException interface 385

DBINFO

CREATE FUNCTION parameter 197
CREATE PROCEDURE parameter 197

DBTimestamp class 476

DBTimestamp object IBM Data Server Driver for JDBC and SQLJ 66
default connection context 134
deregisterDB2XMLObject method 105
disability x

DISABLE DEBUG MODE CREATE PROCEDURE parameter 197

DISALLOW DEBUG MODE CREATE PROCEDURE parameter 197
distinct types JDBC applications 75
SQLJ applications 173
distributed transactions
   example 622
   IBM Data Server Driver for JDBC and SQLJ type 4 connectivity 621
DriverManager interface
   SQLJ
      SQLJ connection technique 1 127
      SQLJ connection technique 2 129
drivers
determining IBM Data Server Driver for JDBC and SQLJ version 5
dynamic data format 164

E
enabling
   IBM Data Server Driver for JDBC and SQLJ support routines 524
encryption
   IBM Data Server Driver for JDBC and SQLJ 552
environment
   Java stored procedures 189
   Java user-defined functions 189
environment variables
   IBM Data Server Driver for JDBC and SQLJ 520
   JDBC 522
   settings for Java routine 193
   SQLJ 522
   z/OS Application Connectivity to DB2 for z/OS feature 539
errors
   SQLJ 184, 185
escape syntax
   IBM Data Server Driver for JDBC and SQLJ 346
examples
deregisterDB2XMLObject 105
   registerDB2XMLSchema 105
exceptions
   IBM Data Server Driver for JDBC and SQLJ 113
executable clause 353
executeUpdate methods 33
extended client information 87
   DB2PreparedStatement constants 93, 94
   DB2PreparedStatement methods 93
   DB2ResultSet methods 94
extended parameter information
   IBM Data Server Driver for JDBC and SQLJ 91
EXTERNAL
   CREATE FUNCTION parameter 197
   CREATE PROCEDURE parameter 197

F
failover
   IBM Data Server Driver for JDBC and SQLJ type 2 connectivity 616
FINAL CALL
   CREATE FUNCTION parameter 197

G
general-use programming information, described 651
getCause method 113
getDatabaseProductName method 27
getDatabaseProductVersion method 27
global transactions
   JDBC and SQLJ 627
H
high availability
   client application 569
   IBM Informix 592
high-availability support
   Db2 on Linux, UNIX, and Windows systems 570
host expressions
   SQLJ 135, 136, 346
I
IBM data server clients
   alternate groups
      Db2 on Linux, UNIX, and Windows systems 582
   automatic client reroute
      DB2 for z/OS 614
      IBM Informix 597
IBM Data Server Driver for JDBC and SQLJ
   client info properties 88
   compatibility with databases 5
   connecting to data sources 13
   connection concentrator monitoring 637
   DB2 for z/OS 545
diagnostic utility 631
   errors 489
   example of enabling Db2 on Linux, UNIX, and Windows systems high availability support 574
   example of enabling Db2 on Linux, UNIX, and Windows systems workload balancing support 576
   example of enabling IBM Informix high availability support 596
   example of enabling Sysplex workload balancing 611
   exceptions 113
   extended client information 87
   installing 519, 541
   installing with DB2 519
   JDBC extensions 383
   Kerberos security 554
   LOB support
      JDBC 59, 61
      SQLJ 164
   properties 234
   security
      details 547
      encrypted password 552
      encrypted user ID 552
      user ID and password 548
      user ID-only 551
   SQL escape syntax 346
   SQLExceptions 116
   SQLSTATEs 496
   trace program example 634
   tracing with configuration parameters example 632
   trusted context support 558
   type 2 connectivity
      DB2 for z/OS failover support 616
      overview 22
   type 4 connectivity 22
   upgrading to a new version 536
   version determination 5
   warnings 113
   XML support 175

Index 657
IBM Data Server Driver for JDBC and SQLJ versions
associated DB2 for z/OS versions 6
associated Db2 on Linux, UNIX, and Windows systems
versions 7
IBM Data Server Driver for JDBC and SQLJ-only fields
DB2Types class 469
IBM Data Server Driver for JDBC and SQLJ-only methods
DB2BaseDataSource class 386
DB2Binder class 392
DB2BlobFileReference class 393
DB2CallableStatement interface 393
DB2ClientRerouteServerList class 400
DB2ClobFileReference class 401
DB2Connection interface 402
DB2ConnectionPoolDataSource class 420
DB2DataSource interface 422
DB2DataSource class 424
DB2Diagnosable interface 423
DB2Driver class 425
DB2ExceptionFormatter class 426
DB2FileReference interface 427
DB2ParameterMetadata interface 429
DB2PooledConnection interface 430
DB2PoolMonitor class 432
DB2PreparedStatement interface 435
DB2ResultSet interface 450
DB2ResultSetMetaData interface 455
DB2SimpleDataSource class 456
DB2sqlca class 476
JSON interface 428
IBM Data Server Driver for JDBC and SQLJ-only properties
DB2BaseDataSource class 386
DB2ClientRerouteServerList class 400
DB2ConnectionPoolDataSource class 420
DB2SimpleDataSource class 455
IBM data server drivers
alternate groups
DB2 on Linux, UNIX, and Windows systems 582
automatic client reroute
DB2 for z/OS 614
IBM Informix 597
IBM Informix
client configuration, high-availability support 593
high availability
application programming 602
cluster support 592
workload balancing 601
IBM Informix high availability support, example of enabling
IBM Data Server Driver for JDBC and SQLJ 596
implements clause
SQLJ 348
Infinity
retrieving in Java applications 231

installation
IBM Data Server Driver for JDBC and SQLJ 541
installing
IBM Data Server Driver for JDBC and SQLJ 519
installing IBM Data Server Driver for JDBC and SQLJ
with DB2 519
internal statement cache
IBM Data Server Driver for JDBC and SQLJ 619
isolation levels
JDBC 111
SQLJ 184
iterator conversion clause
SQLJ 358
iterator declaration clause
SQLJ 351

J

JAR files
creating for JDBC routines 217
defining to DB2 196, 201
Java
applications
accessing z/OS servers 545
overview 1
environment
customization 522
Java programs
preparing and running 209
Java routines
environment variable settings 193
preparing 212
testing 206
WLM environment 190
Java routines with no SQLJ
preparing 212, 214
program preparation 213
Java routines with SQLJ
program preparation 214, 216
Java stored procedures
defining to DB2 196
differences from Java programs 202
differences from other stored procedures 202
parameters specific to 197
WLM environment definitions 524
writing 202
Java user-defined functions
defining to DB2 196
differences from Java program 202
differences from other user-defined functions 202
parameters specific to 197
writing 202
JAVAVENV data set
characteristics 193
JDBC
4.0
columnLabel change 487
columnName change 487
accessing packages 25
APIs 320
applications
24 as hour value 226
data retrieval 39
PROGRAM TYPE
CREATE FUNCTION parameter 197
CREATE PROCEDURE parameter 197
programming interface information, described 651
progressive streaming
IBM Data Server Driver for JDBC and SQLJ 59, 61
JDBC 164
properties
IBM Data Server Driver for JDBC and SQLJ
customizing 522
for all database products 235
for DB2 on Linux, UNIX, and Windows systems 276, 278, 279
for DB2 servers 263
for IBM Informix 276, 278, 290
overview 234

R
reference information
Java 219
registerDB2XMLSchema method 105
resources
releasing
closing connections 123, 186
restrictions
SQLJ variable names 135, 136
result set iterator
details 151
generating JDBC ResultSets from SQLJ iterators 166
named 151
positioned 153
retrieving data from JDBC result sets using SQLJ iterators 166
ResultSet
holdability 44
inserting row 52
testing for delete hole 51
testing for inserted row 53
ResultSet holdability
JDBC 45
ResultSet objects
converting to JSON documents 84
ResultSetMetaData methods
ResultSetMetaData.getColumnLabel change in value 487
ResultSetMetaData.getColumnType change in value 487
retrieving result set information 43
retrieving data
JDBC
data source information 26
PreparedStatement.executeQuery method 40
result set information 43
tables 39
SQLJ 151, 156, 157
retrieving parameter information
JDBC 38
retrieving SQLCA
DB2Diagnosable class 184, 185
return codes
IBM Data Server Driver for JDBC and SQLJ errors 489
rollbacks
JDBC transactions 112
SQLJ transactions 184

S
sample program
JDBC 534
savepoints
JDBC applications 75
SQLJ applications 174
SCRATCHPAD
CREATE FUNCTION parameter 197
scrollable iterators
SQLJ 158
scrollable ResultSet
JDBC 45
scrollable ResultSets
JDBC 44
SDKs
version 1.5 181
security
IBM Data Server Driver for JDBC and SQLJ
encrypted security-sensitive data 552
encrypted user ID or encrypted password 552
Kerberos 554
security mechanisms 547
user ID and password 548
user ID only 551
SQL program preparation 566
SECURITY
CREATE FUNCTION parameter 197
CREATE PROCEDURE parameter 197
SET TRANSACTION clause 356
shortcut keys
keyboard x
SQL statements
error handling
SQLJ applications 184, 185
executing
JDBC interfaces 29
SQLJ applications 140, 170
SQLException
IBM Data Server Driver for JDBC and SQLJ 116
SQLJ
accessing packages for 134
applications
24 as hour value 226
examples 125
invalid Gregorian date 226
programming 125
transaction control 184
assignment clause 357
batch updates 146
binding applications to access multiple servers 210
calling stored procedures 162
clauses 346
collecting trace data 629
comments 140
connecting to data source 127
connecting using default context 134
updatable ResultSet
- inserting row 52
  JDBC 44, 45
- testing for delete hole 51
- testing for inserted row 53
updates
  PreparedStatement.executeUpdate method 31
URL format
  DB2BaseDataSource class 14, 18
user ID and password security
  IBM Data Server Driver for JDBC and SQLJ 548
user ID-only security
  IBM Data Server Driver for JDBC and SQLJ 551
user-defined functions
  access to z/OS UNIX System Services 197
  Java 189

warnings
  IBM Data Server Driver for JDBC and SQLJ 113
with clause
  SQLJ 348
WLM ENVIRONMENT
  CREATE FUNCTION parameter 197
  CREATE PROCEDURE parameter 197
WLM environments
  defining for Java stored procedures 524
  values for Java routines 191
WLM setup
  Java stored procedures 190
  Java user-defined functions 190
WLM startup procedures
  Java routines 190
workload balancing
  connections to Db2 on Linux, UNIX, and Windows systems 586
  IBM Informix
    operation 601

XML
  IBM Data Server Driver for JDBC and SQLJ 175
- parameters
  invoking routines from Java programs 102
XML data
  Java applications 95
  updating
    tables in Java applications 96, 176
XML data retrieval
  Java applications 98, 178
XML schemas
  registering 105
  removing 105
XMLCAST
  SQLJ applications 180
  xmlFormat property
    binary XML format 103

z/OS Application Connectivity to DB2 for z/OS
  SMP/E jobs for loading 538
z/OS Application Connectivity to DB2 for z/OS feature
  environment variables 539
z/OS Application Connectivity to z/OS
  installing 537
z/OS UNIX System Services
  accessing by Java routines 197