



**Application Programming
Guide and Reference
for Java™**



**Application Programming
Guide and Reference
for Java™**

Note

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

Sixth edition (October 2009)

This edition applies to DB2 Version 9.1 for z/OS (DB2 V9.1 for z/OS), product number 5635-DB2, and to any subsequent releases until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Specific changes are indicated by a vertical bar to the left of a change. A vertical bar to the left of a figure caption indicates that the figure has changed. Editorial changes that have no technical significance are not noted.

© **Copyright International Business Machines Corporation 1998, 2009.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this information	ix
Who should read this information	ix
DB2 Utilities Suite	ix
Terminology and citations	ix
Accessibility features for DB2 Version 9.1 for z/OS	x
How to send your comments	xi
How to read syntax diagrams	xi
 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
DB2 for Linux, UNIX, and Windows and IBM Data Server Driver for JDBC and SQLJ levels.	5
 Chapter 3. JDBC application programming.	 7
Example of a simple JDBC application.	7
How JDBC applications connect to a data source	9
Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ.	11
Connecting to a data source using the DataSource interface	15
How to determine which type of IBM Data Server Driver for JDBC and SQLJ connectivity to use	17
JDBC connection objects	18
Creating and deploying DataSource objects.	19
Java packages for JDBC support	21
Learning about a data source using DatabaseMetaData methods.	21
DatabaseMetaData methods for identifying the type of data source.	23
Variables in JDBC applications	23
JDBC interfaces for executing SQL.	24
Creating and modifying database objects using the Statement.executeUpdate method	25
Updating data in tables using the PreparedStatement.executeUpdate method	26
JDBC executeUpdate methods against a DB2 for z/OS server.	28
Making batch updates in JDBC applications	28
Learning about parameters in a PreparedStatement using ParameterMetaData methods	31
Data retrieval in JDBC applications	32
Calling stored procedures in JDBC applications	46
LOBs in JDBC applications with the IBM Data Server Driver for JDBC and SQLJ	50
ROWIDs in JDBC with the IBM Data Server Driver for JDBC and SQLJ	56
Distinct types in JDBC applications	58
Savepoints in JDBC applications	59
Retrieving automatically generated keys in JDBC applications	60
Using named parameter markers in JDBC applications	62
Providing extended client information to the data source with IBM Data Server Driver for JDBC and SQLJ-only methods	65
Providing extended client information to the data source with client info properties	66
XML data in JDBC applications.	69
XML column updates in JDBC applications.	70
XML data retrieval in JDBC applications.	72
Java support for XML schema registration and removal.	75
Inserting data from file reference variables into tables in JDBC applications	78
Transaction control in JDBC applications.	80
IBM Data Server Driver for JDBC and SQLJ isolation levels	80
Committing or rolling back JDBC transactions.	81
Default JDBC autocommit modes	81
Exceptions and warnings under the IBM Data Server Driver for JDBC and SQLJ	82
Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ	84

Handling an SQLWarning under the IBM Data Server Driver for JDBC and SQLJ	88
Retrieving information from a BatchUpdateException	89
Memory use for IBM Data Server Driver for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.	91
Disconnecting from data sources in JDBC applications	92

Chapter 4. SQLJ application programming 93

Example of a simple SQLJ application	93
Connecting to a data source using SQLJ	95
SQLJ connection technique 1: JDBC DriverManager interface	96
SQLJ connection technique 2: JDBC DriverManager interface	97
SQLJ connection technique 3: JDBC DataSource interface	99
SQLJ connection technique 4: JDBC DataSource interface	100
SQLJ connection technique 5: Use a previously created connection context	101
SQLJ connection technique 6: Use the default connection	102
Java packages for SQLJ support	102
Variables in SQLJ applications	103
Comments in an SQLJ application	104
SQL statement execution in SQLJ applications	105
Creating and modifying DB2 objects in an SQLJ application	106
Performing positioned UPDATE and DELETE operations in an SQLJ application	106
Data retrieval in SQLJ applications	116
Calling stored procedures in SQLJ applications	128
LOBs in SQLJ applications with the IBM Data Server Driver for JDBC and SQLJ	130
SQLJ and JDBC in the same application	133
Controlling the execution of SQL statements in SQLJ	136
ROWIDs in SQLJ with the IBM Data Server Driver for JDBC and SQLJ	137
Distinct types in SQLJ applications	139
Savepoints in SQLJ applications	139
XML data in SQLJ applications	140
XML column updates in SQLJ applications	141
XML data retrieval in SQLJ applications	143
XMLCAST in SQLJ applications	145
SQLJ utilization of SDK for Java Version 5 function.	145
Transaction control in SQLJ applications	148
Setting the isolation level for an SQLJ transaction	148
Committing or rolling back SQLJ transactions	148
Handling SQL errors and warnings in SQLJ applications	149
Handling SQL errors in an SQLJ application	149
Handling SQL warnings in an SQLJ application	150
Closing the connection to a data source in an SQLJ application.	151

Chapter 5. Java stored procedures and user-defined functions 153

Setting up the environment for Java routines	153
Setting up the WLM application environment for Java routines.	154
Run-time environment for Java routines	156
Defining Java routines and JAR files to DB2	160
Definition of a Java routine to DB2	161
Definition of a JAR file for a Java routine to DB2	165
Java routine programming	174
Differences between Java routines and stand-alone Java programs.	175
Differences between Java routines and other routines	175
Static and non-final variables in a Java routine	176
Writing a Java stored procedure to return result sets	177
Techniques for testing a Java routine	178

Chapter 6. Preparing and running JDBC and SQLJ programs. 181

Program preparation for JDBC programs	181
Program preparation for SQLJ programs	181
Binding SQLJ applications to access multiple database servers	183

Program preparation for Java routines	185
Preparation of Java routines with no SQLJ clauses	185
Preparation of Java routines with SQLJ clauses	186
Creating JAR files for Java routines	189
Running JDBC and SQLJ programs	190
Chapter 7. JDBC and SQLJ reference information.	191
Data types that map to database data types in Java applications	191
Date, time, and timestamp values that can cause problems in JDBC and SQLJ applications	198
Properties for the IBM Data Server Driver for JDBC and SQLJ	201
Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products	202
Common IBM Data Server Driver for JDBC and SQLJ properties for DB2 servers.	219
Common IBM Data Server Driver for JDBC and SQLJ properties for DB2 for z/OS and IDS	227
Common IBM Data Server Driver for JDBC and SQLJ properties for IDS and DB2 Database for Linux, UNIX, and Windows	229
IBM Data Server Driver for JDBC and SQLJ properties for DB2 Database for Linux, UNIX, and Windows	230
IBM Data Server Driver for JDBC and SQLJ properties for DB2 for z/OS	233
IBM Data Server Driver for JDBC and SQLJ properties for IDS	237
IBM Data Server Driver for JDBC and SQLJ configuration properties.	243
Driver support for JDBC APIs	257
SQLJ statement reference information	285
SQLJ clause	285
SQLJ host-expression	285
SQLJ implements-clause.	286
SQLJ with-clause	287
SQLJ connection-declaration-clause	289
SQLJ iterator-declaration-clause	290
SQLJ executable-clause	291
SQLJ context-clause	292
SQLJ statement-clause	292
SQLJ SET-TRANSACTION-clause	295
SQLJ assignment-clause	296
SQLJ iterator-conversion-clause	297
Interfaces and classes in the sqlj.runtime package	297
sqlj.runtime.ConnectionContext interface	298
sqlj.runtime.ForUpdate interface	303
sqlj.runtime.NamedIterator interface.	303
sqlj.runtime.PositionedIterator interface.	304
sqlj.runtime.ResultSetIterator interface	304
sqlj.runtime.Scrollable interface	307
sqlj.runtime.AsciiStream class	309
sqlj.runtime.BinaryStream class	310
sqlj.runtime.CharacterStream class	311
sqlj.runtime.ExecutionContext class	312
sqlj.runtime.SQLNullException class.	320
sqlj.runtime.StreamWrapper class.	320
sqlj.runtime.UnicodeStream class	321
IBM Data Server Driver for JDBC and SQLJ extensions to JDBC	322
DBBatchUpdateException interface	324
DB2BaseDataSource class	325
DB2BlobFileReference class.	331
DB2ClientRerouteServerList class.	331
DB2ClobFileReference class.	332
DB2Connection interface	333
DB2ConnectionPoolDataSource class	350
DB2DatabaseMetaData interface	352
DB2Diagnosable interface	353
DB2ExceptionFormatter class	353
DB2FileReference class	354
DB2JCCPlugin class	355
DB2PooledConnection class	356

DB2PoolMonitor class	358
DB2PreparedStatement interface	361
DB2ResultSetMetaData interface	374
DB2RowID interface	374
DB2SimpleDataSource class	375
DB2Sqlca class	375
DB2Statement interface	376
DB2SystemMonitor interface	379
DB2TraceManager class	382
DB2TraceManagerMXBean interface	385
DB2Types class	389
DB2XADatasource class	389
DB2Xml interface	391
DB2XmlAsBlobFileReference class	393
DB2XmlAsClobFileReference class	394
JDBC differences between the current IBM Data Server Driver for JDBC and SQLJ and earlier DB2 JDBC drivers	394
JDBC differences between versions of the IBM Data Server Driver for JDBC and SQLJ	400
Examples of ResultSetMetaData.getColumnNames and ResultSetMetaData.getColumnLabels values	403
SQLJ differences between the IBM Data Server Driver for JDBC and SQLJ and other DB2 JDBC drivers	405
Error codes issued by the IBM Data Server Driver for JDBC and SQLJ	407
SQLSTATES issued by the IBM Data Server Driver for JDBC and SQLJ	413
How to find IBM Data Server Driver for JDBC and SQLJ version and environment information	415
Commands for SQLJ program preparation	416
sqlj - SQLJ translator	416
db2sqljcustomize - SQLJ profile customizer	420
db2sqljbind - SQLJ profile binder	431
db2sqljprint - SQLJ profile printer	437
Chapter 8. Installing the IBM Data Server Driver for JDBC and SQLJ	439
Installing the IBM Data Server Driver for JDBC and SQLJ as part of a DB2 installation	439
Jobs for loading the IBM Data Server Driver for JDBC and SQLJ libraries	440
Environment variables for the IBM Data Server Driver for JDBC and SQLJ	441
Customization of IBM Data Server Driver for JDBC and SQLJ configuration properties	442
Enabling the DB2-supplied stored procedures and defining the tables used by the IBM Data Server Driver for JDBC and SQLJ	444
DB2Binder utility	447
DB2T4XAIndoubtUtil for distributed transactions with DB2 UDB for OS/390 and z/OS Version 7 servers	454
DB2LobTableCreator utility	457
Verify the installation of the IBM Data Server Driver for JDBC and SQLJ	458
Upgrading the IBM Data Server Driver for JDBC and SQLJ to a new version	460
Installing the z/OS Application Connectivity to DB2 for z/OS feature	461
Jobs for loading the z/OS Application Connectivity to DB2 for z/OS libraries	463
Environment variables for the z/OS Application Connectivity to DB2 for z/OS feature	464
Chapter 9. Migrating from the JDBC/SQLJ Driver for OS/390 and z/OS to the IBM Data Server Driver for JDBC and SQLJ	465
Conversion of JDBC/SQLJ Driver for OS/390 and z/OS properties to IBM Data Server Driver for JDBC and SQLJ properties	468
Converting JDBC/SQLJ Driver for OS/390 and z/OS serialized profiles to IBM Data Server Driver for JDBC and SQLJ serialized profiles	471
db2sqljupgrade utility	472
Chapter 10. Security under the IBM Data Server Driver for JDBC and SQLJ	475
User ID and password security under the IBM Data Server Driver for JDBC and SQLJ	476
User ID-only security under the IBM Data Server Driver for JDBC and SQLJ	478
Encrypted password, user ID, or user ID and password security under the IBM Data Server Driver for JDBC and SQLJ	479
Kerberos security under the IBM Data Server Driver for JDBC and SQLJ	481
IBM Data Server Driver for JDBC and SQLJ trusted context support	484
IBM Data Server Driver for JDBC and SQLJ support for SSL	486

Configuring connections under the IBM Data Server Driver for JDBC and SQLJ	487
Configuring the Java Runtime Environment to use SSL	487
Security for preparing SQLJ applications with the IBM Data Server Driver for JDBC and SQLJ	490
Chapter 11. Java client support for high availability on IBM data servers.	493
Java client support for high availability for connections to DB2 Database for Linux, UNIX, and Windows servers	494
Configuration of DB2 Database for Linux, UNIX, and Windows high availability support for Java clients	494
Example of enabling DB2 Database for Linux, UNIX, and Windows high availability support in Java applications	497
Operation of automatic client reroute for connections to DB2 Database for Linux, UNIX, and Windows from Java clients	497
Java application programming requirements for high availability for connections to DB2 Database for Linux, UNIX, and Windows servers	501
Client affinities for DB2 Database for Linux, UNIX, and Windows Java clients.	501
Java client support for high availability for connections to IDS servers	504
Configuration of IDS high-availability support for Java clients	505
Example of enabling IDS high availability support in Java applications	508
Operation of automatic client reroute for connections to IDS from Java clients.	509
Operation of workload balancing for connections to IDS from Java clients	513
Application programming requirements for high availability for connections from Java clients to IDS servers	513
Client affinities for connections to IDS from Java clients	514
Java client support for high availability for connections to DB2 for z/OS servers	516
Configuration of Sysplex workload balancing at a Java client	519
Example of enabling DB2 for z/OS Sysplex workload balancing in Java applications	520
Operation of Sysplex workload balancing for connections from Java clients to DB2 for z/OS servers	522
Operation of automatic client reroute for connections from Java clients to DB2 for z/OS	523
Application programming requirements for high availability for connections from Java clients to DB2 for z/OS servers	524
Failover support with IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS	525
Chapter 12. JDBC and SQLJ connection pooling support	527
Chapter 13. IBM Data Server Driver for JDBC and SQLJ type 4 connectivity JDBC and SQLJ distributed transaction support	529
Example of a distributed transaction that uses JTA methods	530
Chapter 14. JDBC and SQLJ global transaction support	535
Chapter 15. Problem diagnosis with the IBM Data Server Driver for JDBC and SQLJ	537
Example of using configuration properties to start a JDBC trace	539
Example of a trace program under the IBM Data Server Driver for JDBC and SQLJ	540
Techniques for monitoring IBM Data Server Driver for JDBC and SQLJ Sysplex support	543
Chapter 16. Tracing IBM Data Server Driver for JDBC and SQLJ C/C++ native driver code	547
db2jcttrace - Format IBM Data Server Driver for JDBC and SQLJ trace data for C/C++ native driver code	547
Chapter 17. System monitoring for the IBM Data Server Driver for JDBC and SQLJ	549
Information resources for DB2 for z/OS and related products	553
How to obtain DB2 information.	559
How to use the DB2 library	563
Notices	567
Programming Interface Information	568
General-use Programming Interface and Associated Guidance Information	569

Trademarks	569
Glossary	571
Index	615

About this information

This information describes DB2® for z/OS® support for Java™. This support lets you access relational databases from Java application programs.

This information assumes that your DB2 subsystem is running in Version 9.1 new-function mode. Generally, new functions that are described, including changes to existing functions, statements, and limits, are available only in new-function mode. Two exceptions to this general statement are new and changed utilities and optimization enhancements, which are also available in conversion mode unless stated otherwise.

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

Important: In this version of DB2 for z/OS, the DB2 Utilities Suite 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.

The DB2 Utilities Suite is designed to work with the DFSORT™ program, which you are licensed to use 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 information

DB2 utilities packaging (Utility Guide)

Terminology and citations

In this information, DB2 Version 9.1 for z/OS is referred to as "DB2 for z/OS." In cases where the context makes the meaning clear, DB2 for z/OS is referred to as "DB2." When this information refers to titles of DB2 for z/OS books, a short title is used. (For example, "See *DB2 SQL Reference*" is a citation to IBM® *DB2 Version 9.1 for z/OS SQL Reference*.)

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.

OMEGAMON®

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 on 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 Version 9.1 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 Version 9.1 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 Information Management Software for z/OS Solutions Information Center (which includes information for DB2 Version 9.1 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

You can access DB2 Version 9.1 for z/OS ISPF panel functions by using a keyboard or keyboard shortcut keys.

For information about navigating the DB2 Version 9.1 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

Online documentation for DB2 Version 9.1 for z/OS is available in the Information Management Software for z/OS Solutions Information Center, which is available at

the following Web site: <http://publib.boulder.ibm.com/infocenter/dzichelp>

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. You can use the following methods to provide comments:

- Send your comments by e-mail 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).
- You can send comments from the Web. Visit the DB2 for z/OS - Technical Resources Web site at:

<http://www.ibm.com/support/docview.wss?&uid=swg27011656>

This Web site has an online reader comment form that you can use to send comments.

- You can also send comments by using the feedback link at the footer of each page in the Information Management Software for z/OS Solutions Information Center at <http://publib.boulder.ibm.com/infocenter/db2zhelp>.

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*—►◄

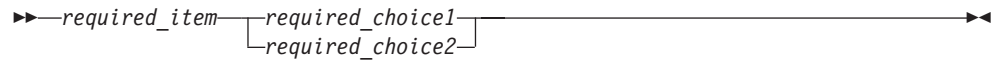
- Optional items appear below the main path.

►—*required_item*—
 └─*optional_item*—┘—►◄

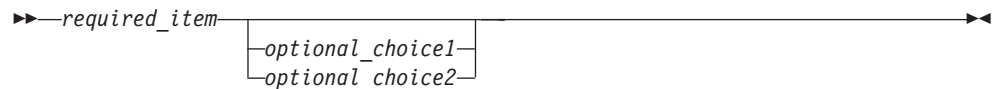
If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.



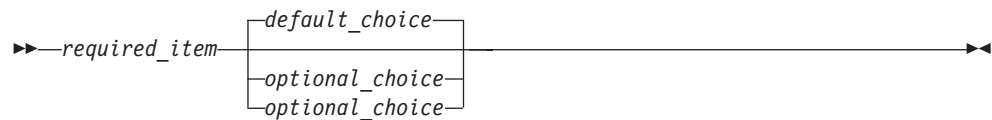
- If you can choose from two or more items, they appear vertically, in a stack.
If you *must* choose one of the items, one item of the stack appears on the main path.



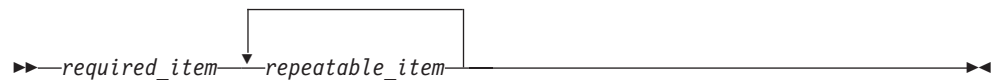
If choosing one of the items is optional, the entire stack appears below the main path.



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



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



If the repeat arrow contains a comma, you must separate repeated items with a comma.

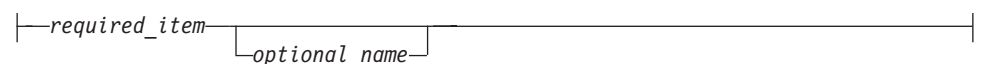


A repeat arrow above a stack indicates that you can repeat the items in the stack.

- 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.



fragment-name:



- 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.

Chapter 1. Java application development for IBM data servers

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

You can access data in DB2 and IDS 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 IDS 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 IDS, SQL statements in JDBC or SQLJ applications run dynamically.

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

pureQuery

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

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

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

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

Related concepts

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

Related reference

 [Integrated Data Management Information Center](#)

Chapter 2. Supported drivers for JDBC and SQLJ

The DB2 product includes support for two types of JDBC driver architecture.

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

Type 1

Drivers that implement the JDBC API as a mapping to another data access API, such as Open Database Connectivity (ODBC). Drivers of this type are generally dependent on a native library, which limits their portability. The DB2 database system does not provide a type 1 driver.

Type 2

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

Type 3

Drivers that use a pure Java client and communicate with a database using a database-independent protocol. The database then communicates the client's requests to the data source. The DB2 database system does not provide a type 3 driver.

Type 4

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

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 and later, and JDBC 3.0 and earlier support

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

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 3.5x is JDBC 3.0-compliant. IBM Data Server Driver for JDBC and SQLJ version 4.x is JDBC 4.0-compliant.

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

- Version 3.5x supports all of the methods that are described in the JDBC 3.0 specifications.
- Version 4.x supports all of the methods that are described in the JDBC 4.0 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 database within Java user-defined functions and stored procedures (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity only. Calling applications can use type 2 connectivity or type 4 connectivity.)
- 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 Database for Linux®, UNIX®, and Windows®).

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 zSeries® 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 Database for Linux, UNIX, and Windows database.

To use the IBM Data Server Driver for JDBC and SQLJ, you need Java 2 Technology Edition, SDK 1.4.2 or higher.

Related concepts

Chapter 1, “Java application development for IBM data servers,” on page 1
 “Environment variables for the z/OS Application Connectivity to DB2 for z/OS feature” on page 464
 “Environment variables for the IBM Data Server Driver for JDBC and SQLJ” on page 441
 “JDBC driver and database version compatibility”

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 3.57, which is shipped with DB2 Database for Linux, UNIX, and Windows Version 9.7, to a DB2 Database for Linux, UNIX, and Windows 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 DB2 for z/OS Version 9.1 database is supported, if the applications under which the driver runs contain no DB2 for z/OS Version 9.1 features.

IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to IBM Informix Dynamic Server is supported only for IDS 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 3.5x of the IBM Data
| Server Driver for JDBC and SQLJ is shipped with DB2 Database for Linux, UNIX,
| and Windows Version 9.5 and Version 9.7, and DB2 for z/OS Version 8 and Version
| 9.1.

However, for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to a local DB2 Database for Linux, UNIX, and Windows database, the database version can be one version earlier or one version later than the DB2 Database for Linux, UNIX, and Windows 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 DB2 for 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

DB2 for Linux, UNIX, and Windows and IBM Data Server Driver for JDBC and SQLJ levels

Each version of DB2 for Linux, UNIX, and Windows is shipped with a different version of the IBM Data Server Driver for JDBC and SQLJ.

The following table lists the DB2 for Linux, UNIX, and Windows versions and corresponding IBM Data Server Driver for JDBC and SQLJ versions. You can use this information to determine the level of DB2 for Linux, UNIX, and Windows 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 1. Versions of DB2 for Linux, UNIX, and Windows and IBM Data Server Driver for JDBC and SQLJ

DB2 for Linux, UNIX, and Windows version and fix pack level	IBM Data Server Driver for JDBC and SQLJ version
Version 8.1	1.0.581
Version 8.1 Fix Pack 1	1.1.67
Version 8.1 Fix Pack 2	1.2.117

Table 1. Versions of DB2 for Linux, UNIX, and Windows and IBM Data Server Driver for JDBC and SQLJ (continued)

DB2 for Linux, UNIX, and Windows version and fix pack level	IBM Data Server Driver for JDBC and SQLJ version
Version 8.1 Fix Pack 3	1.3.70
Version 8.1 Fix Pack 4a	1.5.54
Version 8.1 Fix Pack 5	1.9.23
Version 8.1 Fix Pack 6	2.2.49
Version 8.1 Fix Pack 7 (Version 8.2)	2.3.63
Version 8.1 Fix Pack 8 (Version 8.2 Fix Pack 1)	2.5.33
Version 8.1 Fix Pack 9 (Version 8.2 Fix Pack 2)	2.6.80
Version 8.1 Fix Pack 10 (Version 8.2 Fix Pack 3)	2.7.58
Version 8.1 Fix Pack 11 (Version 8.2 Fix Pack 4)	2.8.46
Version 8.1 Fix Pack 12 (Version 8.2 Fix Pack 5)	2.9.31
Version 8.1 Fix Pack 13 (Version 8.2 Fix Pack 6)	2.10.27
Version 8.1 Fix Pack 14 (Version 8.2 Fix Pack 7)	2.10.52
Version 9.1	3.1.xx ¹
Version 9.1 Fix Pack 1	3.2.xx ¹
Version 9.1 Fix Pack 2	3.3.xx ¹
Version 9.1 Fix Pack 3	3.4.xx ¹
Version 9.1 Fix Pack 4	3.6.xx ¹
Version 9.1 Fix Pack 5	3.7.xx ¹
Version 9.1 Fix Pack 6 and later	3.8.xx ¹
Version 9.5	3.5x.xx ¹ , 4.x.xx ¹
Version 9.7	3.5x.xx ¹ , 4.x.xx ¹

Note:

1. xx is different for each new version of the IBM Data Server Driver for JDBC and SQLJ that is introduced through an APAR.

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 DB2 tables.
- Connect to a data source.
- Execute SQL statements.
- Handle SQL errors and warnings.
- Disconnect from the data source.

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

Example of a simple JDBC application

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

Figure 1. Simple JDBC application

```
import java.sql.*; 1

public class EzJava
{
    public static void main(String[] args)
    {
        String urlPrefix = "jdbc:db2:";
        String url;
        String empNo; 2
        Connection con;
        Statement stmt;
        ResultSet rs;

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

        // Check the that first argument has the correct form for the portion
        // of the URL that follows jdbc: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==0)
        {
            System.err.println ("Invalid value. First argument appended to "+
                                "jdbc:db2: must specify a valid URL.");
            System.exit(1);
        }
        url = urlPrefix + args[0];
```

```

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);
    // 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
    }
}
} // End main
} // End EzJava

```


Notes to Figure 1 on page 7:

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

Related concepts

"Example of a simple JDBC application" on page 7

"Java packages for JDBC support" on page 21

"How JDBC applications connect to a data source"

"Variables in JDBC applications" on page 23

"JDBC interfaces for executing SQL" on page 24

Related tasks

"Handling an `SQLWarning` under the IBM Data Server Driver for JDBC and SQLJ" on page 88

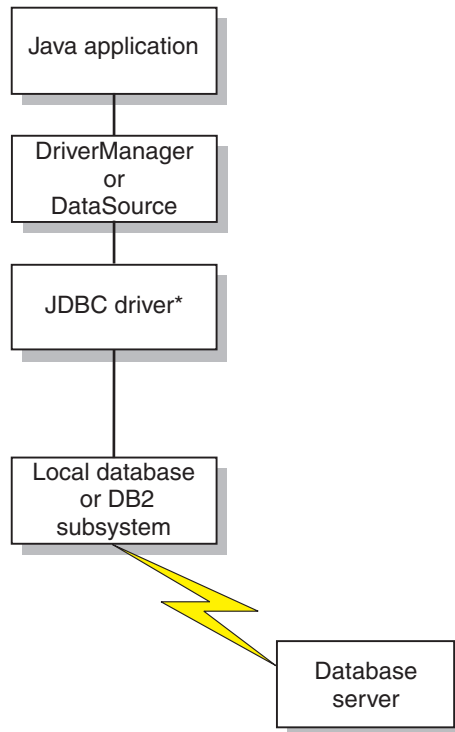
"Disconnecting from data sources in JDBC applications" on page 92

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 Dynamic Server (IDS) databases can use type 4 connectivity.

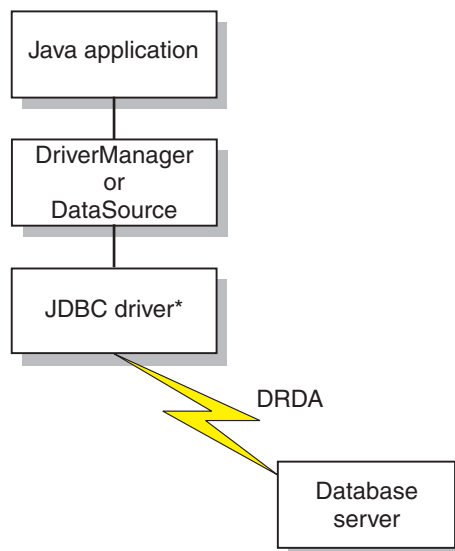
The following figure shows how a Java application connects to a data source using IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.



*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

Related concepts

“Example of a simple JDBC application” on page 7

Related tasks

“Connecting to a data source using SQLJ” on page 95

“Disconnecting from data sources in JDBC applications” on page 92

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

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

The steps for establishing a connection are:

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

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

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

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

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:

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

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

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

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

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

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

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

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

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

- Use the form of the `getConnection` method that specifies `url` with `property=value`; clauses, and include the user and password properties in the URL.
- Use the form of the `getConnection` method that specifies `user` and `password`.

- Use the form of the `getConnection` method that specifies *info*, after setting the user and password properties in a `java.util.Properties` object.

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

```
String url = "jdbc: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:

```
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:

```
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
```

Related concepts

“How to determine which type of IBM Data Server Driver for JDBC and SQLJ connectivity to use” on page 17

“User ID and password security under the IBM Data Server Driver for JDBC and SQLJ” on page 476

“User ID-only security under the IBM Data Server Driver for JDBC and SQLJ” on page 478

“Encrypted password, user ID, or user ID and password security under the IBM Data Server Driver for JDBC and SQLJ” on page 479

“Kerberos security under the IBM Data Server Driver for JDBC and SQLJ” on page 481

“IBM Data Server Driver for JDBC and SQLJ trusted context support” on page 484

“IBM Data Server Driver for JDBC and SQLJ support for SSL” on page 486

Related tasks

“Connecting to a data source using the `DriverManager` interface with the IBM Data Server Driver for JDBC and SQLJ” on page 11

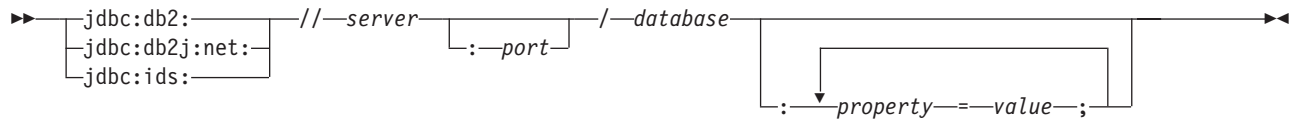
Related reference

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

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

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

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



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

The parts of the URL have the following meanings:

jdbc:db2: or jdbc:db2j:net:

The meanings of the initial portion of the URL are:

jdbc:db2:

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

jdbc:db2: can also be used for a connection to an IBM Informix Dynamic Server (IDS) database, for application portability.

jdbc:db2j:net:

Indicates that the connection is to a remote IBM Cloudscape® server.

jdbc:ids:

Indicates that the connection is to an IDS data source.

jdbc:informix-sqli: also indicates that the connection is to an IDS 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:

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

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

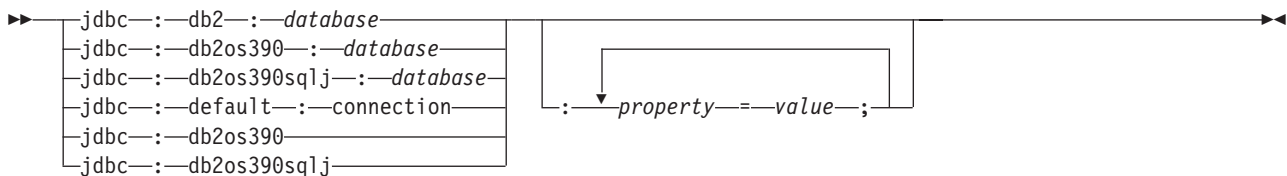
Some properties with an int data type have predefined constant field values. You must resolve constant field values to their integer values before you can use those values in the *url* parameter. For example, you cannot use `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL` in a *url* parameter. However, 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" +
    "user=dbadm;password=dbadm;" +
    "traceLevel=" +
    (com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL) + ";";
Connection con =
    java.sql.DriverManager.getConnection(url);
```

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



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 Database for Linux, UNIX, and Windows 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:

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

Connecting to a data source using the DataSource interface

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

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

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

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

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

data source. In addition, your system administrator can modify the data source attributes, and you do not need to change your application program.

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

<http://www.ibm.com/software/webservers/appserv/>

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

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

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

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

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

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

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

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

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

2
3
4

Figure 4. Obtaining a connection using a DataSource object

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

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

```
import java.sql.*;          // JDBC base  
import javax.sql.*;         // Additional methods for JDBC  
import com.ibm.db2.jcc.*;    // IBM Data Server Driver for JDBC and SQLJ  
                             // interfaces  
DB2SimpleDataSource dbds=new DB2SimpleDataSource();  
dbds.setDatabaseName("dbloc1");  
                             // Assign the location name
```

1
2
3


```

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

```

4

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 DB2's DataSource implementations.
- 3 The setDatabaseName, setDescription, setUser, and setPassword methods assign attributes to the DB2SimpleDataSource object. See "Properties for the IBM Data Server Driver for JDBC and SQLJ" for information about the attributes that you can set for a DB2SimpleDataSource object under the IBM Data Server Driver for JDBC and SQLJ.
- 4 Establishes a connection to the data source that DB2SimpleDataSource object dbds represents.

Related concepts

"User ID and password security under the IBM Data Server Driver for JDBC and SQLJ" on page 476

"User ID-only security under the IBM Data Server Driver for JDBC and SQLJ" on page 478

"Encrypted password, user ID, or user ID and password security under the IBM Data Server Driver for JDBC and SQLJ" on page 479

"Kerberos security under the IBM Data Server Driver for JDBC and SQLJ" on page 481

"IBM Data Server Driver for JDBC and SQLJ trusted context support" on page 484

"IBM Data Server Driver for JDBC and SQLJ support for SSL" on page 486

Related tasks

"Connecting to a data source using SQLJ" on page 95

"Creating and deploying DataSource objects" on page 19

Related reference

"Properties for the IBM Data Server Driver for JDBC and SQLJ" on page 201

"DB2SimpleDataSource class" on page 375

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 2. 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

Function	IBM Data Server Driver for JDBC and SQLJ type 2 connectivity support	IBM Data Server Driver for JDBC and SQLJ type 4 connectivity support
Performance	Better for accessing a local DB2 server	Better for accessing a remote DB2 server
Installation	Requires installation of native libraries in addition to Java classes	Requires installation of Java classes only
Stored procedures	Can be used to call or execute stored procedures	Can be used only to call stored procedures
Distributed transaction processing (XA)	Not supported	Supported
J2EE 1.4 compliance	Compliant	Compliant
CICS environment	Supported	Not supported
IMS environment	Supported	Not supported

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.

Related tasks

“Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ” on page 11

Related reference

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

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 24

Related tasks

"Disconnecting from data sources in JDBC applications" on page 92

"Committing or rolling back JDBC transactions" on page 81

"Learning about a data source using DatabaseMetaData methods" on page 21

"Setting the isolation level for an SQLJ transaction" on page 148

Related reference

"IBM Data Server Driver for JDBC and SQLJ isolation levels" on page 80

Creating and deploying DataSource objects

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

Using the DataSource interface involves two parts:

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

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

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

- `com.ibm.db2.jcc.DB2SimpleDataSource`, which does not support connection pooling. 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.

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

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

The following example shows how to perform these tasks.

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

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

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

Figure 6. Example of creating and deploying a DataSource object

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

Related concepts

“User ID and password security under the IBM Data Server Driver for JDBC and SQLJ” on page 476

“User ID-only security under the IBM Data Server Driver for JDBC and SQLJ” on page 478

“Encrypted password, user ID, or user ID and password security under the IBM Data Server Driver for JDBC and SQLJ” on page 479

“Kerberos security under the IBM Data Server Driver for JDBC and SQLJ” on page 481

“IBM Data Server Driver for JDBC and SQLJ trusted context support” on page 484

“IBM Data Server Driver for JDBC and SQLJ support for SSL” on page 486

Related tasks

“Connecting to a data source using the DataSource interface” on page 15

Related reference

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

“DB2SimpleDataSource class” on page 375

“DB2XADataSource class” on page 389

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

“Example of a simple JDBC application” on page 7

Learning about a data source using DatabaseMetaData methods

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

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

DatabaseMetaData methods provide the following types of information:

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

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

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

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

Figure 7. Using DatabaseMetaData methods to get information about a data source

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

Related concepts

“JDBC connection objects” on page 18

Related reference

“Driver support for JDBC APIs” on page 257

“JDBC differences between the current IBM Data Server Driver for JDBC and SQLJ and earlier DB2 JDBC drivers” on page 394

DatabaseMetaData methods for identifying the type of data source

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

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

database-productdatabase-product/operating-system

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

Table 3. Examples of DatabaseMetaData.getDatabaseProductName values

<code>getDatabaseProductName</code> value	Database product
DB2	DB2 for z/OS
DB2/LINUXx8664	DB2 Database for Linux, UNIX, and Windows on Linux on x86
IDS/UNIX64	IBM Informix Dynamic Server (IDS) on UNIX

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

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

Table 4. Examples of DatabaseMetaData.getDatabaseProductVersion values

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

Variables in JDBC applications

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

Some of those identifiers have the same function as host variables in other languages: they hold data that you pass to or retrieve from database tables.

Identifier `empNo` in the following code holds data that you retrieve from the `EMPNO` table column, which has the `CHAR` data type.

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

Your choice of Java data types can affect performance because DB2 picks better access paths when the data types of your Java variables map closely to the DB2 data types.

Related concepts

“Example of a simple JDBC application” on page 7

Related reference

“Data types that map to database data types in Java applications” on page 191

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 concepts

“Example of a simple JDBC application” on page 7

“JDBC connection objects” on page 18

Related tasks

“Retrieving data from tables using the PreparedStatement.executeQuery method” on page 33

“Updating data in tables using the PreparedStatement.executeUpdate method” on page 26

“Retrieving data from tables using the Statement.executeQuery method” on page 32

“Creating and modifying database objects using the Statement.executeUpdate method”

Related reference

“Driver support for JDBC APIs” on page 257

Creating and modifying database objects using the Statement.executeUpdate method

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

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

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

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

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

Suppose that you want to execute this SQL statement:

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

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

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

Figure 8. Using Statement.executeUpdate

Related concepts

"JDBC interfaces for executing SQL" on page 24

"JDBC executeUpdate methods against a DB2 for z/OS server" on page 28

Related tasks

"Retrieving automatically generated keys in JDBC applications" on page 60

Related reference

"Driver support for JDBC APIs" on page 257

"JDBC differences between the current IBM Data Server Driver for JDBC and SQLJ and earlier DB2 JDBC drivers" on page 394

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

The `Statement.executeUpdate` method works if you update DB2 tables with constant values. However, updates often need to involve passing values in variables to DB2 tables. To do that, you use the `PreparedStatement.executeUpdate` method.

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

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 Dynamic Server data sources, the `PreparedStatement` object can be a `CALL` statement or an `EXECUTE PROCEDURE` statement.

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

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

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

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

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

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

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

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

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

```

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

```

Figure 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. The following example demonstrates these steps.

```

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

```

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

Related concepts

"JDBC interfaces for executing SQL" on page 24

"JDBC executeUpdate methods against a DB2 for z/OS server"

Related tasks

"Retrieving automatically generated keys in JDBC applications" on page 60

Related reference

"Driver support for JDBC APIs" on page 257

"JDBC differences between the current IBM Data Server Driver for JDBC and SQLJ and earlier DB2 JDBC drivers" on page 394

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 25

"Updating data in tables using the PreparedStatement.executeUpdate method" on page 26

Making batch updates in JDBC applications

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

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

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

- addBatch
- executeBatch

- `clearBatch`

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

- `addBatch`

Restrictions on executing statements in a batch:

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

To make batch updates using several statements with no input parameters, follow these basic steps:

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

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

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

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

If the following conditions are true, the IBM Data Server Driver for JDBC and SQLJ returns `Statement.SUCCESS_NO_INFO` (-2), instead of the number of rows that were affected by each SQL statement:

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

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

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

- 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.

- 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.

- If errors occurred, process the BatchUpdateException.

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

```
try {
...
    PreparedStatement prepStmt = con.prepareStatement(
        "UPDATE DEPT SET MGRNO=? WHERE DEPTNO=?");
    prepStmt.setString(1,mgrnum1);
    prepStmt.setString(2,deptnum1);
    prepStmt.addBatch();

    prepStmt.setString(1,mgrnum2);
    prepStmt.setString(2,deptnum2);
    prepStmt.addBatch();
    int [] numUpdates=prepStmt.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");
    }
    con.commit();
} catch (BatchUpdateException b) {
    // process BatchUpdateException
}
```

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

```
try {
...
    PreparedStatement pStmt = con.prepareStatement(
        "INSERT INTO DEPT (DEPTNO, DEPTNAME, ADMRDEPT) " +
        "VALUES (?, ?, ?)",
        Statement.RETURN_GENERATED_KEYS);
    pStmt.setString(1,"X01");
    pStmt.setString(2,"Finance");
    pStmt.setString(3,"A00");
    pStmt.addBatch();
    pStmt.setString(1,"Y01");
    pStmt.setString(2,"Accounting");
    pStmt.setString(3,"A00");
```

```

pStmt.addBatch();

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

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

```

Related tasks

“Retrieving information from a BatchUpdateException” on page 89

“Making batch updates in SQLJ applications” on page 113

“Making batch queries in JDBC applications” on page 35

“Committing or rolling back JDBC transactions” on page 81

Related reference

“JDBC differences between the current IBM Data Server Driver for JDBC and SQLJ and earlier DB2 JDBC drivers” on page 394

Learning about parameters in a PreparedStatement using ParameterMetaData methods

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

ParameterMetaData methods provide the following types of information:

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

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

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

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

```
Connection con;
ParameterMetaData pmtadta;
int mtadtacnt;
String sqlType;
...
pstmt = con.prepareStatement(
    "UPDATE EMPLOYEE SET PHONENO=? WHERE EMPNO=?");
pmtadta = pstmt.getParameterMetaData();
mtadtacnt = pmtadta.getParameterCount();
System.out.println("Number of statement parameters: " + mtadtacnt);
for (int i = 1; i <= mtadtacnt; i++) {
    sqlType = pmtadta.getParameterTypeName(i);
    System.out.println("SQL type of parameter " + i + " is " + sqlType);
}
...
pstmt.close();
```

1
2
3
4

// Create a PreparedStatement object
// Create a ParameterMetaData object
// Determine the number of parameters
// Get SQL type for each parameter

// Close the PreparedStatement

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

Related reference

“Driver support for JDBC APIs” on page 257

Data retrieval in JDBC applications

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

Retrieving data from tables using the `Statement.executeQuery` method

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

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

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

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

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

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

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

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

Figure 12. Using `Statement.executeQuery`

Related concepts

“JDBC interfaces for executing SQL” on page 24

Related tasks

“Learning about a `ResultSet` using `ResultSetMetaData` methods” on page 36

“Specifying updatability, scrollability, and holdability for `ResultSet`s in JDBC applications” on page 38

“Retrieving multiple result sets from a stored procedure in a JDBC application” on page 48

Related reference

“Driver support for JDBC APIs” on page 257

Retrieving data from tables using the `PreparedStatement.executeQuery` method

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

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

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

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

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

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

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

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

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

Figure 13. Example of using `PreparedStatement.executeQuery`

Related concepts

“JDBC interfaces for executing SQL” on page 24

Related tasks

“Retrieving multiple result sets from a stored procedure in a JDBC application” on page 48

Related reference

“Driver support for JDBC APIs” on page 257

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.

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.executeDB2QueryBatch` 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.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();
```

1

2a

2b

```

// 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();
3
} catch (BatchUpdateException b) {
// process BatchUpdateException
}
ResultSet rs;
while(pstmt.getMoreResults()) {
4
rs = pstmt.getResultSet();
4a
while (rs.next()) {
4b
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 28

Related reference

“DB2PreparedStatement interface” on page 361

Learning about a ResultSet using ResultSetMetaData methods

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

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

ResultSetMetaData methods provide the following types of information:

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

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

1. Invoke the getMetaData method on the ResultSet object to create a ResultSetMetaData object.
2. Invoke the getColumnCount method to determine how many columns are in the ResultSet.
3. For each column in the ResultSet, execute ResultSetMetaData methods to determine column characteristics.

The results of ResultSetMetaData.getColumnName call reflects the column name information that is stored in the DB2 catalog for that data source.

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

```
String s;
Connection con;
Statement stmt;
ResultSet rs;
ResultSetMetaData rsmdta;
int colCount;
int mtadtaint;
int i;
String colName;
String colType;

...
stmt = con.createStatement();    // Create a Statement object
rs = stmt.executeQuery("SELECT * FROM EMPLOYEE");
rsmdta = rs.getMetaData();        // Get the ResultSet from the query
colCount = rsmdta.getColumnCount(); // Create a ResultSetMetaData object 1
// Find number of columns in EMP 2

for (i=1; i<= colCount; i++) {
    colName = rsmdta洗getColumn洗Name(); // Get column name
    colType = rsmdta洗getColumn洗TypeName(); // 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 multiple result sets from a stored procedure in a JDBC application” on page 48

“Calling stored procedures in JDBC applications” on page 46

“Retrieving data from tables using the *Statement.executeQuery* method” on page 32

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

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

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

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

The following terms describe characteristics of a *ResultSet*:

scrollability

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

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

updatability

Whether the cursor can be used to update or delete rows. This characteristic

does not apply to a `ResultSet` that is returned from a stored procedure, because a stored procedure `ResultSet` cannot be updated.

holdability

Whether the cursor stays open after a `COMMIT`.

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

Table 5. JDBC `ResultSet` characteristics and SQL cursor attributes

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

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 concepts

 Temporary table space storage requirements (DB2 Installation and Migration)

Related tasks

“Specifying updatability, scrollability, and holdability for `ResultSets` in JDBC applications”

Related reference

“Driver support for JDBC APIs” on page 257

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 6. Valid combinations of `resultSetType` and `resultSetConcurrency` for `ResultSet`s

<code>resultSetType</code> value	<code>resultSetConcurrency</code> value
<code>TYPE_FORWARD_ONLY</code>	<code>CONCUR_READ_ONLY</code>
<code>TYPE_FORWARD_ONLY</code>	<code>CONCUR_UPDATABLE</code>
<code>TYPE_SCROLL_INSENSITIVE</code>	<code>CONCUR_READ_ONLY</code>
<code>TYPE_SCROLL_SENSITIVE</code> ¹	<code>CONCUR_READ_ONLY</code>
<code>TYPE_SCROLL_SENSITIVE</code> ¹	<code>CONCUR_UPDATABLE</code>

Note:

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

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

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

- `LONG VARCHAR`
- `LONG VARGRAPHIC`
- `BLOB`
- `CLOB`

- XML
 - A distinct type that is based on any of the previous data types in this list
 - A structured type
2. If the SELECT statement has input parameters, invoke setXXX methods to pass values to the input parameters.
 3. Invoke the executeQuery method to obtain the result table from the SELECT statement in a ResultSet object.
 4. For each row that you want to access:
 - a. Position the cursor using one of the methods that are listed in the following table.

Table 7. ResultSet methods for positioning a scrollable cursor

Method	Positions the cursor
first ¹	On the first row of the ResultSet
last ¹	On the last row of the ResultSet
next ²	On the next row of the ResultSet
previous ^{1,3}	On the previous row of the ResultSet
absolute(int <i>n</i>) ^{1,4}	If <i>n</i> >0, on row <i>n</i> of the ResultSet. If <i>n</i> <0, and <i>m</i> is the number of rows in the ResultSet, on row <i>m</i> + <i>n</i> +1 of the ResultSet.
relative(int <i>n</i>) ^{1,5,6}	If <i>n</i> >0, on the row that is <i>n</i> rows after the current row. If <i>n</i> <0, on the row that is <i>n</i> rows before the current row. If <i>n</i> =0, on the current row.
afterLast ¹	After the last row in the ResultSet
beforeFirst ¹	Before the first row in the ResultSet

Notes:

1. This method does not apply to connections to IBM Informix Dynamic Server.
2. If the cursor is before the first row of the ResultSet, this method positions the cursor on the first row.
3. If the cursor is after the last row of the ResultSet, this method positions the cursor on the last row.
4. If the absolute value of *n* is greater than the number of rows in the result set, this method positions the cursor after the last row if *n* is positive, or before the first row if *n* is negative.
5. The cursor must be on a valid row of the ResultSet before you can use this method. If the cursor is before the first row or after the last row, the method throws an SQLException.
6. Suppose that *m* is the number of rows in the ResultSet and *x* is the current row number in the ResultSet. If *n*>0 and *x*+*n*>*m*, the driver positions the cursor after the last row. If *n*<0 and *x*+*n*<1, the driver positions the cursor before the first row.
 - b. If you need to know the current cursor position, use the getRow, isFirst, isLast, isBeforeFirst, or isAfterLast method to obtain this information.
 - c. If you specified a *resultSetType* value of TYPE_SCROLL_SENSITIVE in step 1 on page 39, and you need to see the latest values of the current row, invoke the refreshRow method.

Recommendation: Because refreshing the rows of a ResultSet can have a detrimental effect on the performance of your applications, you should invoke refreshRow *only* when you need to see the latest data.
 - d. Perform one or more of the following operations:

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

The `resultSetConcurrency` value for the `ResultSet` must be `CONCUR_UPDATABLE` for you to use these methods.

- To delete the current row from the underlying table, use the `deleteRow` method. Invoking `deleteRow` causes the driver to replace the current row of the `ResultSet` with a hole.

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

5. Invoke the `close` method to close the `ResultSet` object.

6. Invoke the `close` method to close the `Statement` or `PreparedStatement` object.

The following code demonstrates how to retrieve all 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);           1
                           // 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           3
rs.afterLast();                           // Position the cursor at the end of
                                           // the ResultSet           4a
while (rs.previous()) {                   // Position the cursor backward
    s = rs.getString("EMPNO");             // Retrieve the employee number           4d
                                           // (column 1 in the result
                                           // table)
    System.out.println("Employee number = " + s);
                                           // Print the column value
    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           5
stmt.close();                             // Close the Statement           6
```

Figure 15. Using a scrollable cursor

Related concepts

“Characteristics of a JDBC ResultSet under the IBM Data Server Driver for JDBC and SQLJ” on page 37

Related tasks

“Retrieving data from tables using the Statement.executeQuery method” on page 32

Multi-row SQL operations with the IBM Data Server Driver for JDBC and SQLJ:

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

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 in the following situations:

- For scrollable cursors in JDBC or SQLJ programs
- For forward-only cursors, in customized SQLJ programs

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

Multi-row FETCH

Multi-row FETCH can provide better performance than retrieving one row with each FETCH statement. For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, multi-row FETCH can be used for forward-only cursors and scrollable cursors. For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, multi-row FETCH can be used only for scrollable cursors.

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

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

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

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

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

- Second set of conditions:
 - The IBM Data Server Driver for JDBC and SQLJ version is 3.1.
 - The useRowsetCursor property value is com.ibm.db2.jcc.DB2BaseDataSource.YES (1).
 - The FETCH operation uses a scrollable cursor.

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

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

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

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

Multi-row positioned UPDATE or DELETE

The IBM Data Server Driver for JDBC and SQLJ supports a technique for performing positioned update or delete operations that follows the JDBC 1 standard. That technique involves using the ResultSet.setCursorName 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`.

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.

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.

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

```
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.TYPE_SCROLL_SENSITIVE);
                                // Can the database see delete holes?

rs.afterLast();
                                // Position the cursor at the end of
                                // the ResultSet

while (rs.previous()) {
    if (dbSeesDeletes) {
        if (!(rs.rowDeleted()))
            {
                s = rs.getString("EMPNO");
                System.out.println("Employee number = " + s);
                // Retrieve the employee number
                // Print the column value
            }
    }
}

rs.close();
stmt.close();
                                // Close the ResultSet
                                // 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`.

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

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

If an insert buffer already exists, all old values are cleared from the buffer.

- b. Call `ResultSet.updateXXX` methods to assign values to the row that you want to insert.

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

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

If you assign a null value to a column that is defined as NOT NULL, the JDBC driver throws 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.

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

1a

1b

1c

2

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

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

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

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

Calling stored procedures in JDBC applications

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

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

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

You can represent parameters with standard parameter markers (?) or named parameter markers. You cannot mix named parameter markers with standard parameter markers in the same `CALL` statement.

Use of named parameter markers requires the version of the IBM Data Server Driver for JDBC and SQLJ that is installed in `/usr/lpp/db2810/jcc3`. That driver supports IBM Data Server Driver for JDBC and SQLJ version 3.57 or later.

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. The following table lists the types of literals that are supported, and the JDBC types to which they map.

Table 8. Supported literal types in parameters in DB2 for z/OS stored procedure calls

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

2. Invoke the `CallableStatement.setXXX` methods to pass values to the input parameters (parameters that are defined as `IN` or `INOUT` in the `CREATE PROCEDURE` statement).

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

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

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

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

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

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

CallableStatement.executeUpdate

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

CallableStatement.executeQuery

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

CallableStatement.execute

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

Restriction: IBM Informix Dynamic Server (IDS) data sources do not support multiple result sets.

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

Restriction: IDS data sources do not support multiple result sets.

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

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

```
int ifcaret;  
int ifcareas;  
int xsbytes;  
String errbuff;  
Connection con;  
CallableStatement cstmt;  
ResultSet rs;  
...  
cstmt = con.prepareCall("CALL DSN8.DSN8ED2(?,?,?,?)");           1  
                                                                    // Create a CallableStatement object  
cstmt.setString (1, "DISPLAY THREAD(*)");                         2  
                                                                    // Set input parameter (DB2 command)  
cstmt.registerOutParameter (2, Types.INTEGER);                   3  
                                                                    // Register output parameters  
cstmt.registerOutParameter (3, Types.INTEGER);  
cstmt.registerOutParameter (4, Types.INTEGER);  
cstmt.registerOutParameter (5, Types.VARCHAR);  
cstmt.executeUpdate();                                           4  
                                                                    // Call the stored procedure  
ifcaret = cstmt.getInt(2);                                         6  
                                                                    // Get the output parameter values  
ifcareas = cstmt.getInt(3);  
xsbytes = cstmt.getInt(4);  
errbuff = cstmt.getString(5);  
cstmt.close();                                                    7
```


Related tasks

“Learning about a `ResultSet` using `ResultSetMetaData` methods” on page 36

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

Related reference

“Driver support for JDBC APIs” on page 257

“JDBC differences between the current IBM Data Server Driver for JDBC and SQLJ and earlier DB2 JDBC drivers” on page 394

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.

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

“Learning about a `ResultSet` using `ResultSetMetaData` methods” on page 36

“Retrieving data from tables using the `PreparedStatement.executeQuery` method” on page 33

“Retrieving data from tables using the `Statement.executeQuery` method” on page 32

“Calling stored procedures in JDBC applications” on page 46

“Writing a Java stored procedure to return result sets” on page 177

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.

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.

```
CallableStatement cstmt;
ResultSet rs;
int i;
String s;
...
cstmt.execute(); // Call the stored procedure 1
rs = cstmt.getResultSet(); // Get the first result set 2
while (rs.next()) { // Position the cursor 3
    i = rs.getInt(1); // Retrieve current result set value
    System.out.println("Value from first result set = " + i);
    // Print the value
}
cstmt.getMoreResults(); // Point to the second result set 4a
// and close the first result set
rs = cstmt.getResultSet(); // Get the second result set 4b
while (rs.next()) { // Position the cursor 4c
    s = rs.getString(1); // Retrieve current result set value
    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.

To retrieve result sets when you do not know the number of result sets or their contents, you need to retrieve `ResultSet`s, until no more `ResultSet`s 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.

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 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.

```
CallableStatement cstmt;
ResultSet rs;
...
```

```

boolean resultsAvailable = cstmt.execute(); // Call the stored procedure
while (resultsAvailable) {                // Test for result sets 1
    ResultSet rs = cstmt.getResultSet();    // Get a result set 2a
    ...                                     // Process the ResultSet
                                           // as you would process
                                           // a ResultSet from a table
    resultsAvailable = cstmt.getMoreResults(); // Check for next result set 2c
                                           // (Also closes the
                                           // previous result set)
}

```

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`.

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

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

```
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 `ResultSet`s open until the final `ResultSet` has been retrieved, and then closes all `ResultSet`s.

```

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 concepts

“Java data types for retrieving or updating LOB column data in JDBC applications” on page 54

“Memory use for IBM Data Server Driver for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS” on page 91

Related reference

“Driver support for JDBC APIs” on page 257

“JDBC differences between the current IBM Data Server Driver for JDBC and SQLJ and earlier DB2 JDBC drivers” on page 394

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

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

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

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

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

Progressive streaming is the default behavior in the following environments:

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

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

For DB2 for z/OS Version 9.1 and later data sources, or DB2 Database for Linux, UNIX, and Windows 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.

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

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

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

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

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

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

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

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

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

- If the data source does not support progressive locators:
If the value of `fullyMaterializeLobData` is `true`, LOB data is fully materialized within the JDBC driver when a row is fetched. If the value is `false`, LOB data is streamed. The driver uses locators internally to retrieve LOB data in chunks on an as-needed basis. It is highly recommended that you set this value to `false` when you retrieve LOBs that contain large amounts of data. The default is `true`.
- If the data source supports progressive streaming, also known as dynamic data format:

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

`fullyMaterializeLobData` has no effect on stored procedure parameters.

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

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

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

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

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

For BLOB columns:

- `getBinaryStream`
- `getBlob`
- `getBytes`

For CLOB columns:

- `getAsciiStream`
- `getCharacterStream`
- `getClob`
- `getString`

- Call the following `ResultSet` methods to update a BLOB or CLOB column in an updatable `ResultSet`:

For BLOB columns:

- `updateBinaryStream`
- `updateBlob`

For CLOB columns:

- `updateAsciiStream`
- `updateCharacterStream`
- `updateClob`

If you specify -1 for the *length* parameter in any of the previously listed methods, the IBM Data Server Driver for JDBC and SQLJ reads the input data until it is exhausted.

- Use the following `PreparedStatement` methods to set the values for parameters that correspond to BLOB or CLOB columns:

For BLOB columns:

- `setBytes`
- `setBlob`
- `setBinaryStream`
- `setObject`, where the *Object* parameter value is an `InputStream`.

For CLOB columns:

- `setString`
- `setAsciiStream`

- `setClob`
- `setCharacterStream`
- `setObject`, where the *Object* parameter value is a `Reader`.

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

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

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

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

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


```

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

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


```

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

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

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

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

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:
`cstmt.setBlob(parmIndex, blobData);`
- Use a `CallableStatement.setObject` call that specifies that the target data type is BLOB:

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

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

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

Output parameters for BLOB columns

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

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

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

Input parameters for CLOB columns

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

- Use a `java.sql.Clob` input variable, which is an exact match for a CLOB column:
`cstmt.setClob(parmIndex, clobData);`
- Use a `CallableStatement.setObject` call that specifies that the target data type is CLOB:

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

- Use one of the following types of stream input parameters:

- A `java.io.StringReader` input parameter with a `cstmt.setCharacterStream` call:

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

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


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

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

- Use a String input parameter with a `cstmt.setString` call:
`cstmt.setString(parmIndex, charData);`

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

- Use a String input parameter with a `cstmt.setObject` call, and specify the target data type as VARCHAR or LONGVARCHAR:
`cstmt.setObject(parmIndex, charData, java.sql.Types.VARCHAR);`

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

Output parameters for CLOB columns

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

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

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

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

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

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

Related concepts

“LOBs in JDBC applications with the IBM Data Server Driver for JDBC and SQLJ” on page 50

Related reference

“Data types that map to database data types in Java applications” on page 191

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

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

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

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

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

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

- `updateRowId` (JDBC 4.0 and later)
`updateRowId` is valid only if the target database system supports updating of ROWID columns.

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

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

You can use the following `PreparedStatement` methods to set a value for a parameter that is associated with a ROWID column:

- `setRowId` (JDBC 4.0 and later)
- `setBytes`
- `setObject`

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

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

You can use the following `CallableStatement` methods to retrieve a ROWID column as an output parameter from a stored procedure call:

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

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

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

Table 9. DatabaseMetaData.getRowIdLifetime values for supported data sources

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

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

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

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

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

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

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

Related reference

“Data types that map to database data types in Java applications” on page 191

“DB2RowID interface” on page 374

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:

```

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 ('000010', 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 191

 [CREATE TYPE \(SQL Reference\)](#)

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. There are 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, DB2 executes the form of the `SAVEPOINT` statement that includes `ON ROLLBACK RETAIN CURSORS`.

Connection.releaseSavepoint(Savepoint savepoint)

Releases the specified savepoint, and all subsequently established savepoints.

Connection.rollback(Savepoint savepoint)

Rolls back work to the specified savepoint.

DatabaseMetaData.supportsSavepoints()

Indicates whether a data source supports savepoints.

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

default), multiple savepoints with the same name can be created within the same unit of recovery, but creation of a savepoint destroys a previously created savepoint with the same name.

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

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

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

Related tasks

“Committing or rolling back JDBC transactions” on page 81

Related reference

“Data types that map to database data types in Java applications” on page 191

Retrieving 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.

Automatically generated keys correspond to 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).

To enable retrieval of automatically generated keys from a table, you need to indicate when you insert rows that you will want to retrieve automatically generated key values. You do that by setting a flag in a Connection.prepareStatement, Statement.executeUpdate, or Statement.execute method call. The statement that is executed must be a single-row INSERT

statement or a multiple-row INSERT statement, such as an INSERT with a subselect clause. Otherwise, the JDBC driver ignores the parameter that sets the flag.

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

To retrieve automatically generated keys from a table, 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.

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

The following forms are valid only if the data source supports INSERT within SELECT 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.

```
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.

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

The following forms are valid only if the data source supports INSERT within SELECT 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.

```
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.

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.

```

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')",
    Statement.RETURN_GENERATED_KEYS); // Insert a row
    // 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
    // value
    System.out.println("automatically generated key value = " + idColVar);
}
rs.close();                           // Close ResultSet
stmt.close();                         // Close Statement

```

With any JDBC driver, you can retrieve the most recently assigned value of an identity column using 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()) {                   // Position the cursor
    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

```

Related tasks

“Updating data in tables using the `PreparedStatement.executeUpdate` method” on page 26

“Creating and modifying database objects using the `Statement.executeUpdate` method” on page 25

Using named parameter markers in JDBC applications

You can use named parameter markers instead of standard parameter markers in `PreparedStatement` and `CallableStatement` objects to assign values to the input

parameter markers. You can also use named parameter markers instead of standard parameter markers in `CallableStatement` objects to register named OUT parameters.

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, which are indicated by a question mark (?). When `enableNamedParameterMarkers` is `DB2BaseDataSource.YES`, you can use named parameter markers in your applications, regardless of whether the data server supports them.

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.

For best results, before you use named parameter markers in your applications, set the Connection or DataSource property `enableNamedParameterMarkers` to `DB2BaseDataSource.YES`.

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

1. Execute the `Connection.prepareStatement` method on an SQL statement string that contains named parameter markers. The named parameter markers must follow the rules for SQL host variable names.
You cannot mix named parameter markers and standard parameter markers in the same SQL statement string.
Named parameter markers are case-insensitive.
2. For each named parameter marker, use a `DB2PreparedStatement.setJccXXXAtName` method to assign a value to each named input parameter.
If you use the same named parameter marker more than once in the same SQL statement string, you need to call a `setJccXXXAtName` method for that parameter marker only once.

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

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

3. Execute the `PreparedStatement`.

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

```
Connection con;  
PreparedStatement pstmt;  
int numUpd;  
...  
pstmt = con.prepareStatement(  
    "UPDATE EMPLOYEE SET PHONENO=:phonenum WHERE EMPNO=:empnum");  
// Create a PreparedStatement object  
((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).setJccStringAtName
```

1


```

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

```

Using named parameter markers with CallableStatement objects

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

For best results, before you use named parameter markers in your applications, set the Connection or DataSource property enableNamedParameterMarkers to DB2BaseDataSource.YES.

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

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

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

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

Named parameter markers are case-insensitive.

2. 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.

3. 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.

4. Execute the CallableStatement.
5. Call CallableStatement.getXXX methods to retrieve output parameter values.

The following code illustrates calling a stored procedure that has one input VARCHAR parameter and one output INTEGER parameter, which are represented

by named parameter markers. The numbers to the right of selected statements correspond to the previously described steps.

```
...
CallableStatement cstmt =
    con.prepareCall("CALL MYSP(:inparm,:outparm)");
                                // Create a CallableStatement object 1
                                // Register OUT parameter data type 2
((com.ibm.db2.jcc.DB2CallableStatement)cstmt).
    registerOutParameterAtName("outparm", java.sql.Types.INTEGER);
((com.ibm.db2.jcc.DB2CallableStatement)cstmt).setJccStringAtName("inparm", "4567");
                                // Assign a value to inparm parameter 3

cstmt.executeUpdate();          // Call the stored procedure 4
int outssid = cstmt.getInt(2);  // Get the output parameter value 5
cstmt.close();
```

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.

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 10. Methods that provide client information to the DB2 server

Method	Information provided
<code>setDB2ClientAccountingInformation</code>	Accounting information
<code>setDB2ClientApplicationInformation</code>	Name of the application that is working with a connection
<code>setDB2ClientDebugInfo</code>	The CLIENT DEBUGINFO connection attribute for the Unified debugger
<code>setDB2ClientProgramId</code>	A caller-specified string that helps the caller identify which program is associated with a particular SQL statement. <code>setDB2ClientProgramId</code> does not apply to DB2 Database for Linux, UNIX, and Windows data servers.
<code>setDB2ClientUser</code>	User name for a connection
<code>setDB2ClientWorkstation</code>	Client workstation name for a connection

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 10.
4. Execute an SQL statement to cause the information to be sent to the DB2 server.

The following code performs the previous steps to pass a user name and a workstation name to the DB2 server. The numbers to the right of selected statements correspond to the previously-described steps.

```
public class ClientInfoTest {
    public static void main(String[] args) {
        String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose";
        try {
            Class.forName("com.ibm.db2.jcc.DB2Driver");
            String user = "db2adm";
            String password = "db2adm";
            Connection conn = DriverManager.getConnection(url,      1
                user, password);
            if (conn instanceof DB2Connection) {
                DB2Connection db2conn = (DB2Connection) conn;      2
                db2conn.setDB2ClientUser("Michael L Thompson");      3
                db2conn.setDB2ClientWorkstation("sjwkstn1");
                // Execute SQL to force extended client information to be sent
                // to the server
                conn.prepareStatement("SELECT * FROM SYSIBM.SYSDUMMY1"
                    + "WHERE 0 = 1").executeQuery();                4
            }
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
}
```

Figure 18. Example of passing extended client information to a DB2 server

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

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

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

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

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

To set client info properties, follow these steps:

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

The following code performs the previous steps to pass a client's user name and host name to the DB2 server. The numbers to the right of selected statements correspond to the previously-described steps.

```

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

```

Figure 19. Example of passing extended client information to aDB2 server

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

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

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

- NAME
- MAX_LEN
- DEFAULT_VALUE
- DESCRIPTION

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

Table 11. Client info property values for DB2 Database for Linux, UNIX, and Windows and for DB2 for i

NAME	MAX_LEN (bytes)	DEFAULT_VALUE	DESCRIPTION
ApplicationName	255	Empty string	The name of the application that is currently using the connection. This value is stored in DB2 special register CURRENT CLIENT_APPLNAME.
ClientAccountingInformation	255	Empty string	The value of the accounting string from the client information that is specified for the connection. This value is stored in DB2 special register CURRENT CLIENT_ACCTNG.

Table 11. Client info property values for DB2 Database for Linux, UNIX, and Windows and for DB2 for i (continued)

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

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

Table 12. Client info property values for type 4 connectivity to DB2 for z/OS

NAME	MAX_LEN (bytes)	DEFAULT_VALUE	DESCRIPTION
ApplicationName	32	clientProgramName property value, if set. "db2jcc_application" otherwise.	The name of the application that is currently using the connection. This value is stored in DB2 special register CURRENT CLIENT_APPLNAME.
ClientAccountingInformation	200	A string that is the concatenation of the following values: <ul style="list-style-type: none"> "JCCnnnnn", where nnnnn is the driver level, such as 04000. The value that is set by DB2Connection.setDB2ClientWorkstation. If the value is not set, the default is the host name of the local host. applicationName property value, if set. 20 blanks otherwise. clientUser property value, if set. Eight blanks otherwise. 	The value of the accounting string from the client information that is specified for the connection. This value is stored in DB2 special register CURRENT CLIENT_ACCTNG.
ClientHostname	18	The value that is set by DB2Connection.setDB2ClientWorkstation. If the value is not set, the default is the host name of the local host.	The host name of the computer on which the application that is using the connection is running. This value is stored in DB2 special register CURRENT CLIENT_WRKSTNNAME.
ClientUser	16	The value that is set by DB2Connection.setDB2ClientUser. If the value is not set, the default is the current user ID that is used to connect to the database.	The name of the user on whose behalf the application that is using the connection is running. This value is stored in DB2 special register CURRENT CLIENT_USERID.

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

Table 13. Client info property values for type 2 connectivity to DB2 for z/OS

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

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

Table 14. Client info property values for IBM Informix Dynamic Server

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

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.

In applications, XML data is in the serialized string format.

In JDBC applications that connect to DB2 Database for Linux, UNIX, and Windows, XML data is in textual XML format. In JDBC applications that connect to DB2 for z/OS, XML data can be in textual XML format or binary XML format.

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 column updates in JDBC applications”

“XML data retrieval in JDBC applications” on page 72

Related reference

“Data types that map to database data types in Java applications” on page 191

XML column updates in JDBC applications

When you update or insert data into XML columns of a database table, the input data in your JDBC applications must be in the serialized string format.

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

Table 15. Methods and data types for updating XML columns

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

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 Database for Linux, UNIX, and Windows, 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.

```
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());
    System.out.println("insertBinStream: SQL Error Code: " +
        sqle.getErrorCode());
}
}
}

```

Related concepts

“XML data in JDBC applications” on page 69

Related reference

“Data types that map to database data types in Java applications” on page 191

“DB2Xml interface” on page 391

XML data retrieval in JDBC applications

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

When you retrieve data from XML columns of a DB2 table, the output data is in the serialized string format. This is true whether you retrieve the entire contents of an XML column or a sequence from the column.

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.
- 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.

This technique uses the deprecated `DB2Xml` objects. Use of the previously described technique is preferable.

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

Table 16. *ResultSet* methods and data types for retrieving XML data

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

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 17. *SQLXML* and *DB2Xml* methods, data types, and added encoding specifications

Method	Output data type	Type of XML internal encoding declaration added
<code>SQLXML.getBinaryStream</code>	<code>InputStream</code>	None
<code>SQLXML.getCharacterStream</code>	<code>Reader</code>	None
<code>SQLXML.getSource</code>	<code>Source</code> ¹	None
<code>SQLXML.getString</code>	<code>String</code>	None
<code>DB2Xml.getDB2AsciiStream</code>	<code>InputStream</code>	None
<code>DB2Xml.getDB2BinaryStream</code>	<code>InputStream</code>	None
<code>DB2Xml.getDB2Bytes</code>	<code>byte[]</code>	None
<code>DB2Xml.getDB2CharacterStream</code>	<code>Reader</code>	None
<code>DB2Xml.getDB2String</code>	<code>String</code>	None
<code>DB2Xml.getDB2XmlAsciiStream</code>	<code>InputStream</code>	US-ASCII
<code>DB2Xml.getDB2XmlBinaryStream</code>	<code>InputStream</code>	Specified by <code>getDB2XmlBinaryStream</code> <i>targetEncoding</i> parameter
<code>DB2Xml.getDB2XmlBytes</code>	<code>byte[]</code>	Specified by <code>DB2Xml.getDB2XmlBytes</code> <i>targetEncoding</i> parameter
<code>DB2Xml.getDB2XmlCharacterStream</code>	<code>Reader</code>	ISO-10646-UCS-2
<code>DB2Xml.getDB2XmlString</code>	<code>String</code>	ISO-10646-UCS-2

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.

```

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 a String variable.

```

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.

```

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 concepts

“XML data in JDBC applications” on page 69

Related reference

“Data types that map to database data types in Java applications” on page 191

“DB2Xml interface” on page 391

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 DB2, 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 DB2.

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 Database for Linux, UNIX, and Windows.

Before you can invoke these methods, the stored procedures that support these methods must be installed on the DB2 database 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.

```
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 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`.

```

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 Database for Linux, UNIX, and Windows. 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);
}

```

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.

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

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

Input data type	Constructor
BLOB	<code>com.ibm.db2.jcc.DB2BlobFileReference</code>
CLOB	<code>com.ibm.db2.jcc.DB2ClobFileReference</code>
XML AS BLOB	<code>com.ibm.db2.jcc.DB2XmlAsBlobFileReference</code>
XML AS CLOB	<code>com.ibm.db2.jcc.DB2XmlAsClobFileReference</code>

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.

Input data type	DB2PreparedStatement.setXXX method
BLOB	<code>setDB2BlobFileReference</code>
CLOB	<code>setDB2ClobFileReference</code>
XML AS BLOB	<code>setDB2XmlAsBlobFileReference</code>
XML AS CLOB	<code>setDB2XmlAsClobFileReference</code>

If you use `DB2PreparedStatement` methods, you need to cast the `PreparedStatement` object that you created in step 1 to a `DB2PreparedStatement` object when you execute a `DB2PreparedStatement.setXXX` method.

You can set 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.execute` or `PreparedStatement.executeUpdate` method to update the table with the variable 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.executeBatch` 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 1
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 2
// 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. 3a
// Assign a null value to the XML parameter.
int numUpd = pstmt.executeUpdate();
// Perform the update 3b
pstmt.close(); // Close the PreparedStatement object 5

```

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.

```

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 18. Equivalent JDBC and DB2 isolation levels

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

The following table shows the values of *level* that you can specify in the `Connection.setTransactionIsolation` method and their IBM Informix Dynamic Server (IDS) equivalents.

Table 19. Equivalent JDBC and IDS isolation levels

JDBC value	IDS isolation level
<code>java.sql.Connection.TRANSACTION_SERIALIZABLE</code>	Repeatable read
<code>java.sql.Connection.TRANSACTION_REPEATABLE_READ</code>	Repeatable read

Table 19. Equivalent JDBC and IDS isolation levels (continued)

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

Related concepts

“JDBC connection objects” on page 18

Committing or rolling back JDBC transactions

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

For example:

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

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

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

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

While a connection is participating in a distributed 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

“JDBC connection objects” on page 18

“Savepoints in JDBC applications” on page 59

Related tasks

“Making batch updates in JDBC applications” on page 28

“Disconnecting from data sources in JDBC applications” on page 92

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 IDS data sources

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

Table 20. Default autocommit modes for IDS data sources

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

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

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

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

SQLException

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

- An `int` value that contains an error code. `SQLException.getErrorCode` retrieves this value.
- A `String` object that contains the `SQLSTATE`, or null. `SQLException.getSQLState` retrieves this value.
- A `String` object that contains a description of the error, or null. `SQLException.getMessage` retrieves this value.
- A pointer to the next `SQLException`, or null. `SQLException.getNextException` retrieves this value.

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

DB2Diagnosable

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

getSqlca

Returns an DB2Sqlca object with the following information:

- An SQL error code
- The SQLERRMC values
- The SQLERRP value
- The SQLERRD values
- The SQLWARN values
- The SQLSTATE

getThrowable

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

printTrace

Prints diagnostic information.

SQLException subclasses

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

- **SQLNonTransientException**

An SQLNonTransientException is thrown when an SQL operation that failed previously cannot succeed when the operation is retried, unless some corrective action is taken. The SQLNonTransientException class has these subclasses:

- SQLFeatureNotSupportedException
- SQLNonTransientConnectionException
- SQLDataException
- SQLIntegrityConstraintViolationException
- SQLInvalidAuthorizationSpecException
- SQLSyntaxException

- **SQLTransientException**

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

- SQLTransientConnectionException
- SQLTransientRollbackException
- SQLTimeoutException

- **SQLRecoverableException**

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

- **SQLClientInfoException**

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

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 DB2-specific information. The DB2-specific information for an SQLWarning object is the same as the DB2-specific information for an SQLException object.

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

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

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

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


```
import com.ibm.db2.jcc.DB2Diagnosable;
import com.ibm.db2.jcc.DB2Sqlca;
```
2. Optional: During a connection to a DB2 for z/OS or IBM Informix Dynamic Server (IDS) data source, 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 IDS data source, execute the `com.ibm.db2.jcc.DB2Statement.getIDSQLStatementOffset` method to determine which columns have syntax errors.
`DB2Statement.getIDSQLStatementOffset` returns the offset into the SQL statement of the first syntax error.
 - c. Optional: For an SQL statement that executes on an IDS 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.printStackTrace` method to write all `SQLException` information to a `java.io.PrintWriter` object.
 - 3) Invoke the `DB2Diagnosable.getThrowable` method to determine whether an underlying `java.lang.Throwable` caused the `SQLException`.
 - 4) Invoke the `DB2Diagnosable.getSqlca` method to retrieve the `DB2Sqlca` object.
 - 5) Invoke the `DB2Sqlca.getSqlCode` method to retrieve an SQL error code value.

- 6) Invoke the `DB2Sqlca.getSqlErrmc` method to retrieve a string that contains all `SQLERRMC` values, or invoke the `DB2Sqlca.getSqlErrmcTokens` method to retrieve the `SQLERRMC` values in an array.
 - 7) Invoke the `DB2Sqlca.getSqlErrp` method to retrieve the `SQLERRP` value.
 - 8) Invoke the `DB2Sqlca.getSqlErrd` method to retrieve the `SQLERRD` values in an array.
 - 9) Invoke the `DB2Sqlca.getSqlWarn` method to retrieve the `SQLWARN` values in an array.
 - 10) Invoke the `DB2Sqlca.getSqlState` method to retrieve the `SQLSTATE` value.
 - 11) Invoke the `DB2Sqlca.getMessage` method to retrieve error message text from the data source.
- e. Invoke the `SQLException.getNextException` method to retrieve the next `SQLException`.

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

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

```
import java.sql.*;           // Import JDBC API package
import com.ibm.db2.jcc.DB2Diagnosable; // Import packages for DB2
import com.ibm.db2.jcc.DB2Sqlca;   // SQLException support
import 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 {                           4
    // Code that could generate SQLExceptions
    ...
} catch(SQLException sqle) {
    while(sqle != null) {        // Check whether there are more
                                   // SQLExceptions to process
    //====> Optional IBM Data Server Driver for JDBC and SQLJ-only
    // error processing
        if (sqle instanceof DB2Diagnosable) {
            // 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 5d4
    if (sqlca != null) { // Check that DB2Sqlca is not null
        int sqlCode = sqlca.getSqlCode(); // Get the SQL error code 5d5
        String sqlErrmc = sqlca.getSqlErrmc(); // Get the entire SQLERRMC 5d6
        String[] sqlErrmcTokens = sqlca.getSqlErrmcTokens();
        // You can also retrieve the
        // individual SQLERRMC tokens
        String sqlErrp = sqlca.getSqlErrp(); // Get the SQLERRP 5d7
        int[] sqlErrd = sqlca.getSqlErrd(); // Get SQLERRD fields 5d8
        char[] sqlWarn = sqlca.getSqlWarn(); // Get SQLWARN fields 5d9
        String sqlState = sqlca.getSqlState(); // Get SQLSTATE 5d10
        String errMessage = sqlca.getMessage(); // Get error message 5d11

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

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

```


Related tasks

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

“Handling SQL errors in an SQLJ application” on page 149

“Handling SQL warnings in an SQLJ application” on page 150

Related reference

“Error codes issued by the IBM Data Server Driver for JDBC and SQLJ” on page 407

“DB2Diagnosable interface” on page 353

“DB2Sqlca class” on page 375

“SQLSTATEs issued by the IBM Data Server Driver for JDBC and SQLJ” on page 413

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

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

The basic steps for retrieving SQL warning information are:

1. Optional: During connection to the database server, set properties that affect SQLWarning objects.
If you want full message text from a DB2 for z/OS or IBM Informix Dynamic Server (IDS) data source 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 DB2-specific warning information, perform the same steps that you perform to get DB2-specific information for an SQLException.
 - f. Invoke the SQLWarning.getNextWarning method to retrieve the next SQLWarning.

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

steps.

```
String url = "jdbc: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());  
    System.out.println ("SQLSTATE: " + sqlwarn.getSQLState());  
    System.out.println ("Error code: " + sqlwarn.getErrorCode());  
    sqlwarn=sqlwarn.getNextWarning();    // Get next SQLWarning  
}
```

1

2

3a

3b

3c

3d

3f

Figure 21. Example of processing an SQLWarning

Related concepts

“Example of a simple JDBC application” on page 7

Related tasks

“Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ” on page 84

Retrieving information from a BatchUpdateException

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

To retrieve information from the `BatchUpdateException`, follow these steps:

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

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

n The number of rows that the statement updated.

Statement.SUCCESS_NO_INFO

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

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

Statement.EXECUTE_FAILED

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

2. If the batched statement can return automatically generated keys:
 - a. Cast the BatchUpdateException to a `com.ibm.db2.jcc.DBBatchUpdateException`.
 - b. Call the `DBBatchUpdateException.getDBGeneratedKeys` method to retrieve an array of `ResultSet` objects that contains the automatically generated keys for each execution of the batched SQL statement.
 - c. Test whether each `ResultSet` in the array is null.
Each `ResultSet` contains:
 - If the `ResultSet` is not null, it contains the automatically generated keys for an execution of the batched SQL statement.
 - If the `ResultSet` is null, execution of the batched statement failed.
3. Use `SQLException` methods `getMessage`, `getSQLState`, and `getErrorCode` to retrieve the description of the error, the `SQLSTATE`, and the error code for the first error.
4. Use the `BatchUpdateException.getNextException` method to get a chained `SQLException`.
5. In a loop, execute the `getMessage`, `getSQLState`, `getErrorCode`, and `getNextException` method calls to obtain information about an `SQLException` and get the next `SQLException`.

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

```
try {
    // Batch updates
} catch (BatchUpdateException buex) {
    System.err.println("Contents of BatchUpdateException:");
    System.err.println(" Update counts: ");
    int [] updateCounts = buex.getUpdateCounts();           1
    for (int i = 0; i < updateCounts.length; i++) {
        System.err.println(" Statement " + i + ":" + updateCounts[i]);
    }
    ResultSet[] resultList =
        ((DBBatchUpdateException)buex).getDBGeneratedKeys(); 2
    for (i = 0; i < resultList.length; i++)
    {
        if (resultList[i] == null)
            continue; // Skip the ResultSet for which there was a failure
        else {
            rs.next();
            java.math.BigDecimal idColVar = rs.getBigDecimal(1);
                                                    // Get automatically generated key
                                                    // value
            System.out.println("Automatically generated key value = " + idColVar);
        }
    }
    System.err.println(" Message: " + buex.getMessage());    3
    System.err.println(" SQLSTATE: " + buex.getSQLState());
    System.err.println(" Error code: " + buex.getErrorCode());
    SQLException ex = buex.getNextException();              4
    while (ex != null) {                                     5
        System.err.println("SQL exception:");
        System.err.println(" Message: " + ex.getMessage());
    }
}
```

```

        System.err.println(" SQLSTATE: " + ex.getSQLState());
        System.err.println(" Error code: " + ex.getErrorCode());
        ex = ex.getNextException();
    }
}

```

Related tasks

“Making batch updates in JDBC applications” on page 28

Memory use for IBM Data Server Driver for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS

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. 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. Therefore, applications that run under IBM Data Server Driver for JDBC and SQLJ type 2 connectivity and involve LOBs with large length attributes can fail with out-of-memory errors.

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 50

Related reference

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

Disconnecting from data sources in JDBC applications

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

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

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

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

For a connection to an IBM Informix Dynamic Server 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.

Related concepts

“Example of a simple JDBC application” on page 7

“JDBC connection objects” on page 18

“How JDBC applications connect to a data source” on page 9

Related tasks

“Committing or rolling back JDBC transactions” on page 81

Chapter 4. SQLJ application programming

Writing a 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 DB2 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.*; 1
import java.sql.*;

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

public class EzSqlj {
    public static void main(String args[])
        throws SQLException
    {
        EzSqljCtx ctx = null;
        String URLprefix = "jdbc:db2:";
        String url;
        url = new String(URLprefix + args[0]);

        String hvmgr="000010"; 2
        String hvdeptno="A00";
        try { 3b
            Class.forName("com.ibm.db2.jcc.DB2Driver");
        } catch (Exception e)
        {
            throw new SQLException("Error in EzSqlj: Could not load the driver");
        }
        try
        {
            System.out.println("About to connect using url: " + url);
            Connection con0 = DriverManager.getConnection(url); 3c
            // Create a JDBC Connection
            con0.setAutoCommit(false); 3d
            // set autocommit OFF
            ctx = new EzSqljCtx(con0);

            try
            {
```

```

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());
    // Retrieve rows from result table
    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 93:

Note	Description
1	These statements import the <code>java.sql</code> package, which contains the JDBC core API, and the <code>sqlj.runtime</code> 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 <code>hvmgr</code> and <code>hvdeptno</code> are <i>host identifiers</i> , which are equivalent to DB2 host variables. See "Variables in SQLJ applications" for more information.
3a, 3b, 3c, and 3d	These statements demonstrate how to connect to a data source using one of the three available techniques. See "Connecting to a data source using SQLJ" for more details.
	Step 3b (loading the JDBC driver) is not necessary if you use JDBC 4.0.
4a , 4b, 4c, and 4d	These statements demonstrate how to execute SQL statements in SQLJ. Statement 4a demonstrates the SQLJ equivalent of declaring an SQL cursor. Statements 4b and 4c show one way of doing the SQLJ equivalent of executing an SQL OPEN CURSOR and SQL FETCHes. Statement 4d shows how to do the SQLJ equivalent of performing an SQL UPDATE. For more information, see "SQL statements in an SQLJ application".
5	This try/catch block demonstrates the use of the <code>SQLException</code> 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".

Related concepts

"Java packages for SQLJ support" on page 102

"Variables in SQLJ applications" on page 103

"SQL statement execution in SQLJ applications" on page 105

"Comments in an SQLJ application" on page 104

Related tasks

"Connecting to a data source using SQLJ"

"Handling SQL errors in an SQLJ application" on page 149

"Handling SQL warnings in an SQLJ application" on page 150

"Closing the connection to a data source in an SQLJ application" on page 151

Connecting to a data source using SQLJ

In an SQLJ application, as in any other DB2 application, you must be connected to a data source before you can execute SQL statements.

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 9

"Example of a simple SQLJ application" on page 93

"SQLJ and JDBC in the same application" on page 133

Related tasks

"Connecting to a data source using the DataSource interface" on page 15

"Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 106

"Making batch updates in SQLJ applications" on page 113

"Committing or rolling back SQLJ transactions" on page 148

"Closing the connection to a data source in an SQLJ application" on page 151

Related reference

"SQLJ with-clause" on page 287

"SQLJ connection-declaration-clause" on page 289

"SQLJ context-clause" on page 292

SQLJ connection technique 1: JDBC DriverManager interface

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

To use SQLJ connection technique 1, follow these steps:

1. Execute an SQLJ *connection declaration clause*.

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

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

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

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

- Invoke `Class.forName` this way:

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

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

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

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

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

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

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

The meanings of the parameters are:

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

user and password

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

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.

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

```
#sql context Ctx;           // Create connection context class Ctx      1
String userid="dbadm";      // Declare variables for user ID and password
String password="dbadm";
String empname;            // Declare a host variable
...
try {                       // Load the JDBC driver
    Class.forName("com.ibm.db2.jcc.DB2Driver");                       2
}
catch (ClassNotFoundException e) {
    e.printStackTrace();
}
Ctx myConnCtx=              3
    new Ctx("jdbc: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

SQLJ connection technique 2: JDBC DriverManager interface

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

To use SQLJ connection technique 2, follow these steps:

1. Execute an SQLJ *connection declaration clause*.

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

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

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

2. Load a JDBC driver by invoking the `Class.forName` method.
 - Invoke `Class.forName` this way:

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

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

3. Invoke the JDBC `DriverManager.getConnection` method.

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

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

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

user and password

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

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 97

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

```
connection-context-class connection-context-object=  
new connection-context-class(Connection JDBC-connection-object);
```

The *JDBC-connection-object* parameter is the `Connection` object that you created in step 3.

The following code uses connection technique 2 to create a connection to location NEWYORK. The connection requires a user ID and password, and does not require autocommit. The numbers to the right of selected statements correspond to the previously-described steps.

```

#sql context Ctx;           // Create connection context class Ctx 1
String userid="dbadm";      // Declare variables for user ID and password
String password="dbadm";
String empname;             // Declare a host variable
...
try {                       // Load the JDBC driver
    Class.forName("com.ibm.db2.jcc.DB2Driver"); 2
}
catch (ClassNotFoundException e) {
    e.printStackTrace();
}
Connection jdbccon= 3
    DriverManager.getConnection("jdbc:db2://sysmvsl.stl.ibm.com:5021/NEWYORK",
        userid,password);

// Create JDBC connection object jdbccon
jdbccon.setAutoCommit(false); // Do not autocommit
Ctx myConnCtx=new Ctx(jdbccon); 4
// Create connection context object myConnCtx
// for the connection to NEWYORK
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
    WHERE EMPNO='000010'};
// Use myConnCtx for executing an SQL statement

```

Figure 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.

To use SQLJ connection technique 3, follow these steps:

1. Execute an SQLJ *connection declaration clause*.

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

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

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

2. If your system administrator created a DataSource object in a different program, follow these steps. Otherwise, create a DataSource object and assign properties to it.
 - a. Obtain the logical name of the data source to which you need to connect.
 - b. Create a context to use in the next step.
 - c. In your application program, use the Java Naming and Directory Interface (JNDI) to get the DataSource object that is associated with the logical data source name.
3. Invoke the JDBC DataSource.getConnection method.

Doing this creates a JDBC connection object for the connection to the data source. You can use one of the following forms of getConnection:

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

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

user and password

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

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:

```
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 99.

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

```
connection-context-class connection-context-object=
    new connection-context-class(Connection JDBC-connection-object);
```

The *JDBC-connection-object* parameter is the `Connection` object that you created in step 3 on page 99.

The following code uses connection technique 3 to create a connection to a location with logical name `jdbc/sampledb`. This example assumes that the system administrator created and deployed a `DataSource` object that is available through JNDI lookup. The numbers to the right of selected statements correspond to the previously-described steps.

```
import java.sql.*;
import javax.naming.*;
import javax.sql.*;
...
#sql context CtxSqlj;           // Create connection context class CtxSqlj 1
Context ctx=new InitialContext(); 2b
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb"); 2c
Connection con=ds.getConnection(); 3
String empname;                // Declare a host variable
...
con.setAutoCommit(false);      // Do not autocommit 4
CtxSqlj myConnCtx=new CtxSqlj(con); 5
                                // Create connection context object myConnCtx
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
    WHERE EMPNO='000010'};
                                // Use myConnCtx for executing an SQL statement
```

Figure 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.

To use SQLJ connection technique 4, follow these steps:

1. From your system administrator, obtain the logical name of the data source to which you need to connect.
2. Execute an SQLJ connection declaration clause.

For this type of connection, the connection declaration clause needs to be of this form:

```
#sql public static context context-class-name
with (dataSource="logical-name");
```

The connection context must be declared as public and static. *logical-name* is the data source name that you obtained in step 1 on page 100.

3. Invoke the constructor for the connection context class that you created in step 2 on page 100.

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

```
connection-context-class connection-context-object=
new connection-context-class();
```

```
connection-context-class connection-context-object=
new connection-context-class (String user,
String password);
```

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

user and password

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

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.

The following code uses connection technique 4 to create a connection to a location with logical name jdbc/sampledb. The connection requires a user ID and password.

```
#sql public static context Ctx
with (dataSource="jdbc/sampledb"); 2
// Create connection context class Ctx
String userid="dbadm"; // Declare variables for user ID and password
String password="dbadm";

String empname; // Declare a host variable
...
Ctx myConnCtx=new Ctx(userid, password); 3
// Create connection context object myConnCtx
// for the connection to jdbc/sampledb
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
WHERE EMPNO='000010'};
// Use myConnCtx for executing an SQL statement
```

Figure 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.

In general, one program declares a connection context class, creates connection contexts, and passes them as parameters to other programs. A program that uses the connection context invokes a constructor with the passed connection context object as its argument.

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

```
#sql context Ctx;
...
// Create connection context object oldCtx
```

Program test.sqlj receives oldCtx as a parameter and uses oldCtx as the argument of its connection context constructor:

```
void useContext(sqlj.runtime.ConnectionContext oldCtx)
    // oldCtx was created in CtxGen.sqlj
{
    Ctx myConnCtx=
        new Ctx(oldCtx);           // Create connection context object myConnCtx
    // from oldCtx
    #sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
        WHERE EMPNO='000010'};
    // Use myConnCtx for executing an SQL statement
    ...
}
```

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.

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 uses the implicit connection.

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

```
#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.

Related concepts

“Example of a simple SQLJ application” on page 93

Variables in SQLJ applications

In DB2 programs in other languages, you use host variables to pass data between the application program and DB2. In SQLJ programs, In SQLJ programs, you can use host variables or *host expressions*.

A host expression begins with a colon (:). The colon is followed by an optional parameter mode identifier (IN, OUT, or INOUT), which is followed by a parenthesized expression clause.

Host variables and host expressions are case sensitive.

A complex expression is an array element or Java expression that evaluates to a single value. A complex expression in an SQLJ clause must be surrounded by parentheses.

The following examples demonstrate how to use host expressions.

Example: Declaring a Java identifier and using it in a SELECT statement:

In this example, the statement that begins with #sql has the same function as a SELECT statement in other languages. This statement assigns the last name of the employee with employee number 000010 to Java identifier empname.

```
String empname;
...
#sql [ctxt]
    {SELECT LASTNAME INTO :empname FROM EMPLOYEE WHERE EMPNO='000010'};
```

Example: Declaring a Java identifier and using it in a stored procedure call:

In this example, the statement that begins with #sql has the same function as an SQL CALL statement in other languages. This statement uses Java identifier empno as an input parameter to stored procedure A. The keyword IN, which precedes empno, specifies that empno is an input parameter. For a parameter in a CALL statement, IN is the default. The explicit or default qualifier that indicates how the parameter is used (IN, OUT, or INOUT) must match the corresponding value in the parameter definition that you specified in the CREATE PROCEDURE statement for the stored procedure.

```
String empno = "0000010";
...
#sql [ctxt] {CALL A (:IN empno)};
```


Example: Using a complex expression as a host identifier:

This example uses complex expression `((int)yearsEmployed++/5)*500` as a host expression.

```
#sql [ctxt] {UPDATE EMPLOYEE  
    SET BONUS=((int)yearsEmployed++/5)*500) WHERE EMPNO=:empID};
```

SQLJ performs the following actions when it processes a complex host expression:

- Evaluates each of the host expressions in the statement, from left to right, before assigning their respective values to the database.
- Evaluates side effects, such as operations with postfix operators, according to normal Java rules. All host expressions are fully evaluated before any of their values are passed to DB2.
- Uses Java rules for rounding and truncation.

Therefore, if the value of `yearsEmployed` is 6 before the `UPDATE` statement is executed, the value that is assigned to column `BONUS` by the `UPDATE` statement is `((int)6/5)*500`, or 500. After 500 is assigned to `BONUS`, the value of `yearsEmployed` is incremented.

Restrictions on variable names: Two strings have special meanings in SQLJ programs. Observe the following restrictions when you use these strings in your SQLJ programs:

- The string `__sJT_` is a reserved prefix for variable names that are generated by SQLJ. Do not begin the following types of names with `__sJT_`:
 - Host expression names
 - Java variable names that are declared in blocks that include executable SQL statements
 - Names of parameters for methods that contain executable SQL statements
 - Names of fields in classes that contain executable SQL statements, or in classes with subclasses or enclosed classes that contain executable SQL statements
- The string `_SJ` is a reserved suffix for resource files and classes that are generated by SQLJ. Avoid using the string `_SJ` in class names and input source file names.

Related concepts

“Example of a simple SQLJ application” on page 93

Related reference

“Data types that map to database data types in Java applications” on page 191

“SQLJ host-expression” on page 285

Comments in an SQLJ application

To document your SQLJ program, you need to include comments. To do that, use Java comments. Java comments are denoted by `/* */` or `/**`.

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

- Within a host expression (`/* */` or `/**`).
- Within an SQL statement in an executable clause, if the data source supports a comment within the SQL statement (`/* */` or `--`).

`/*` and `*/` pairs in an SQL statement can be nested.

Related concepts

“Example of a simple SQLJ application” on page 93

“SQL statement execution in SQLJ applications”

SQL statement execution in SQLJ applications

You execute SQL statements in a traditional SQL program to create tables, update data in tables, retrieve data from the tables, call stored procedures, or commit or roll back transactions. In an SQLJ program, you also execute these statements, within SQLJ *executable clauses*.

An executable clause can have one of the following general forms:

```
#sql [connection-context] {sql-statement};  
#sql [connection-context,execution-context] {sql-statement};  
#sql [execution-context] {sql-statement};
```

execution-context specification

In an executable clause, you should **always** specify an explicit connection context, with one exception: you do not specify an explicit connection context for a FETCH statement. You include an execution context only for specific cases. See “Control the execution of SQL statements in SQLJ” for information about when you need an execution context.

connection-context specification

In an executable clause, if you do not explicitly specify a connection context, the executable clause uses the default connection context.

Related concepts

“Example of a simple SQLJ application” on page 93

“Comments in an SQLJ application” on page 104

“Data retrieval in SQLJ applications” on page 116

“Retrieving multiple result sets from a stored procedure in an SQLJ application” on page 129

“LOBs in SQLJ applications with the IBM Data Server Driver for JDBC and SQLJ” on page 130

“SQLJ and JDBC in the same application” on page 133

Related tasks

“Calling stored procedures in SQLJ applications” on page 128

“Committing or rolling back SQLJ transactions” on page 148

“Controlling the execution of SQL statements in SQLJ” on page 136

“Creating and modifying DB2 objects in an SQLJ application”

“Handling SQL errors in an SQLJ application” on page 149

“Handling SQL warnings in an SQLJ application” on page 150

“Making batch updates in SQLJ applications” on page 113

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

“Setting the isolation level for an SQLJ transaction” on page 148

“Using a named iterator in an SQLJ application” on page 117

“Using a positioned iterator in an SQLJ application” on page 119

“Using scrollable iterators in an SQLJ application” on page 124

Related reference

“SQLJ executable-clause” on page 291

Creating and modifying DB2 objects in an SQLJ application

Use SQLJ executable clauses to execute data definition statements (CREATE, ALTER, DROP, GRANT, REVOKE) or to execute INSERT, searched or positioned UPDATE, and searched or positioned DELETE statements.

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

```
#sql [myConnCtx] {INSERT INTO DEPARTMENT VALUES  
  ("X00", "Operations 2", "000030", "E01", NULL)};  
#sql [myConnCtx] {UPDATE DEPARTMENT  
  SET MGRNO="000090" WHERE MGRNO="000030"};  
#sql [myConnCtx] {DELETE FROM DEPARTMENT  
  WHERE DEPTNO="X00"};
```

Related concepts

“SQL statement execution in SQLJ applications” on page 105

Related tasks

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

Performing positioned UPDATE and DELETE operations in an SQLJ application

As in DB2 applications in other languages, performing positioned UPDATES and DELETES with SQLJ is an extension of retrieving rows from a result table.

The basic steps are:

1. Declare the iterator.

The iterator can be positioned or named. For positioned UPDATE or DELETE operations, declare the iterator as updatable, using one or both of the following clauses:

implements sqlj.runtime.ForUpdate

This clause causes the generated iterator class to include methods for using updatable iterators. This clause is required for programs with positioned UPDATE or DELETE operations.

with (updateColumns=*"column-list"*)

This clause specifies a comma-separated list of the columns of the result table that the iterator will update. This clause is optional.

You need to declare the iterator as `public`, so you need to follow the rules for declaring and using `public` iterators in the same file or different files.

If you declare the iterator in a file by itself, any SQLJ source file that has addressability to the iterator and imports the generated class can retrieve data and execute positioned UPDATE or DELETE statements using the iterator.

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.

The following code shows how to declare a positioned iterator and use it for positioned UPDATES. The numbers to the right of selected statements correspond to the previously described steps.

First, in one file, declare positioned iterator UpdByPos, specifying that you want to use the iterator to update column SALARY:

```
import java.math.*;    // Import this class for BigDecimal data type
#sql public iterator UpdByPos implements sqlj.runtime.ForUpdate 1
    with(updateColumns="SALARY") (String, BigDecimal);
```

Figure 27. Example of declaring a positioned iterator for a positioned UPDATE

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

```

import sqlj.runtime.*;      // Import files for SQLJ and JDBC APIs
import java.sql.*;
import java.math.*;         // Import this class for BigDecimal data type
import UpdByPos;           // Import the generated iterator class that
                           // was created by the iterator declaration clause
                           // for UpdByName in another file
#sql context HSCtx;        // Create a 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
    #sql [myConnCtx]
        upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE
                    WHERE WORKDEPT='D11'};
    // Assign result table to iterator object
    #sql {FETCH :upditer INTO :empnum,:sal};
    // Move cursor to next row
    while (!upditer.endFetch())
    {
        // Check if on a row
        #sql [myConnCtx] {UPDATE EMPLOYEE SET SALARY=SALARY*1.05
                        WHERE CURRENT OF :upditer};
        // Perform positioned update
        System.out.println("Updating row for " + empnum);
        #sql {FETCH :upditer INTO :empnum,:sal};
        // Move cursor to next row
    }
    upditer.close(); // Close the iterator
    #sql [myConnCtx] {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:

```

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:

```
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 2
    // do not destroy the cursor between updates
    HSCtx myConnCtx=new HSCtx(HSjdbccon);
    // Create a connection context object
    UpdByName upditer;      3
    // Declare iterator object of UpdByName class
    String empnum;          // Declare host variable to receive EmpNo
                           // column values
    #sql [myConnCtx]
        upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE
                    WHERE WORKDEPT='D11'};      4
    // Assign result table to iterator object
    while (upditer.next())  5a,5b
    {
        // 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};      5c
        // Perform positioned update
        System.out.println("Updating row for " + empnum);
    }
    upditer.close();        // Close the iterator      6
    #sql [myConnCtx] {COMMIT};
    // Commit the changes
    myConnCtx.close();      // Close the connection context
}
```

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

Related concepts

“SQL statement execution in SQLJ applications” on page 105

“Iterators as passed variables for positioned UPDATE or DELETE operations in an SQLJ application”

“Data retrieval in SQLJ applications” on page 116

 Authorization IDs and dynamic SQL (SQL Reference)

Related tasks

“Creating and modifying DB2 objects in an SQLJ application” on page 106

“Connecting to a data source using SQLJ” on page 95

“Using a named iterator in an SQLJ application” on page 117

“Using a positioned iterator in an SQLJ application” on page 119

Related reference

“SQLJ implements-clause” on page 286

“SQLJ with-clause” on page 287

“sqlj.runtime.ForUpdate interface” on page 303

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.

```

#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 DB2 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 on page 113 shows an example of how to do this.


```

#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 DB2 bind process can always associate the positioned UPDATE statement with a valid iterator.

Related tasks

“Performing positioned UPDATE and DELETE operations in an SQLJ application”
on page 106

Making batch updates in SQLJ applications

The IBM Data Server Driver for JDBC and SQLJ supports batch updates in SQLJ. With batch updates, instead of updating rows of a table one at a time, you can direct SQLJ to execute a group of updates at the same time.

You can include the following types of statements in a batch update:

- Searched INSERT, UPDATE, or DELETE, or MERGE statements
- CREATE, ALTER, DROP, GRANT, or REVOKE statements
- CALL statements with input parameters only

Unlike JDBC, SQLJ allows heterogeneous batches that contain statements with input parameters or host expressions. You can therefore combine any of the following items in an SQLJ batch:

- Instances of the same statement
- Different statements
- Statements with different numbers of input parameters or host expressions
- Statements with different data types for input parameters or host expressions
- Statements with no input parameters or host expressions

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.

To obtain information about warnings, use the `ExecutionContext.getWarnings` method on the `ExecutionContext` that you used to submit statements to be batched. You can then retrieve an error description, `SQLSTATE`, and error code for each `SQLWarning` object.

When a batch is executed implicitly because the program contains a statement that cannot be added to the batch, the batch is executed before the new statement is processed. If an error occurs during execution of the batch, the statement that caused the batch to execute does not execute.

The basic steps for creating, executing, and deleting a batch of statements are:

1. Disable `AutoCommit` for the connection.
Do this so that you can control whether to commit changes to already-executed statements when an error occurs during batch execution.
2. Acquire an execution context.
All statements that execute in a batch must use this execution context.
3. Invoke the `ExecutionContext.setBatching(true)` method to create a batch.
Subsequent batchable statements that are associated with the execution context that you created in step 2 are added to the batch for later execution.
If you want to batch sets of statements that are not batch compatible in parallel, you need to create an execution context for each set of batch compatible statements.
4. Include SQLJ executable clauses for SQL statements that you want to batch.
These clauses must include the execution context that you created in step 2.
If an SQLJ executable clause has input parameters or host expressions, you can include the statement in the batch multiple times with different values for the input parameters or host expressions.
To determine whether a statement was added to an existing batch, was the first statement in a new batch, or was executed inside or outside a batch, invoke the `ExecutionContext.getUpdateCount` method. This method returns one of the following values:
`ExecutionContext.ADD_BATCH_COUNT`
This is a constant that is returned if the statement was added to an existing batch.
`ExecutionContext.NEW_BATCH_COUNT`
This is a constant that is returned if the statement was the first statement in a new batch.
`ExecutionContext.EXEC_BATCH_COUNT`
This is a constant that is returned if the statement was part of a batch, and the batch was executed.
Other integer
This value is the number of rows that were updated by the statement. This value is returned if the statement was executed rather than added to a batch.
5. Execute the batch explicitly or implicitly.
 - Invoke the `ExecutionContext.executeBatch` method to execute the batch explicitly.

executeBatch returns an integer array that contains the number of rows that were updated by each statement in the batch. The order of the elements in the array corresponds to the order in which you added statements to the batch.

- Alternatively, a batch executes implicitly under the following circumstances:
 - You include a batchable statement in your program that is not compatible with statements that are already in the batch. In this case, SQLJ executes the statements that are already in the batch and creates a new batch that includes the incompatible statement.
 - You include a statement in your program that is not batchable. In this case, SQLJ executes the statements that are already in the batch. SQLJ also executes the statement that is not batchable.
 - After you invoke the `ExecutionContext.setBatchLimit(n)` method, you add a statement to the batch that brings the number of statements in the batch to *n* or greater. *n* can have one of the following values:

ExecutionContext.UNLIMITED_BATCH

This constant indicates that implicit execution occurs only when SQLJ encounters a statement that is batchable but incompatible, or not batchable. Setting this value is the same as not invoking `setBatchLimit`.

ExecutionContext.AUTO_BATCH

This constant indicates that implicit execution occurs when the number of statements in the batch reaches a number that is set by SQLJ.

Positive integer

When this number of statements have been added to the batch, SQLJ executes the batch implicitly. However, the batch might be executed before this many statements have been added if SQLJ encounters a statement that is batchable but incompatible, or not batchable.

To determine the number of rows that were updated by a batch that was executed implicitly, invoke the `ExecutionContext.getBatchUpdateCounts` method. `getBatchUpdateCounts` returns an integer array that contains the number of rows that were updated by each statement in the batch. The order of the elements in the array corresponds to the order in which you added statements to the batch. Each array element can be one of the following values:

- 2 This value indicates that the SQL statement executed successfully, but the number of rows that were updated could not be determined.
- 3 This value indicates that the SQL statement failed.

Other integer

This value is the number of rows that were updated by the statement.

6. Optionally, when all statements have been added to the batch, disable batching. Do this by invoking the `ExecutionContext.setBatching(false)` method. When you disable batching, you can still execute the batch implicitly or explicitly, but no more statements are added to the batch. Disabling batching is useful when a batch already exists, and you want to execute a batch compatible statement, rather than adding it to the batch.

If you want to clear a batch without executing it, invoke the `ExecutionContext.cancel` method.

7. If batch execution was implicit, perform a final, explicit `executeBatch` to ensure that all statements have been executed.

In the following code fragment, raises are given to all managers by performing UPDATES in a batch. The numbers to the right of selected statements correspond to the previously-described steps.

```
#sql iterator GetMgr(String);          // Declare positioned iterator
...
{
    GetMgr deptiter;                    // Declare object of GetMgr class
    String mgrnum = null;                // Declare host variable for manager number
    int raise = 400;                     // Declare raise amount
    int currentSalary;                  // Declare current salary
    String url, username, password;     // Declare url, user ID, password
    ...
    TestContext c1 = new TestContext (url, username, password, false); 1
    ExecutionContext ec = new ExecutionContext(); 2
    ec.setBatching(true); 3

    #sql [c1] deptiter =
        {SELECT MGRNO FROM DEPARTMENT};
        // Assign the result table of the SELECT
        // to iterator object deptiter

    #sql {FETCH :deptiter INTO :mgrnum};
        // Retrieve the first manager number

    while (!deptiter.endFetch()) {      // Check whether the FETCH returned a row
        #sql [c1]
            {SELECT SALARY INTO :currentSalary FROM EMPLOYEE
                WHERE EMPNO=:mgrnum};
        #sql [c1, ec] 4
            {UPDATE EMPLOYEE SET SALARY=(currentSalary+raise)
                WHERE EMPNO=:mgrnum};
        #sql {FETCH :deptiter INTO :mgrnum };
            // Fetch the next row
    }
    ec.executeBatch(); 5
    ec.setBatching(false); 6
    #sql [c1] {COMMIT};
    deptiter.close();                // Close the iterator
    c1.close();                      // Close the connection
}
```

Figure 33. Example of performing a batch update

Related concepts

“SQL statement execution in SQLJ applications” on page 105

Related tasks

“Making batch updates in JDBC applications” on page 28

“Connecting to a data source using SQLJ” on page 95

“Controlling the execution of SQL statements in SQLJ” on page 136

Related reference

“sqlj.runtime.SQLNullException class” on page 320

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 DB2 applications in other languages, if you want to retrieve a single row from a table in an SQLJ application, you can write a SELECT INTO statement with a WHERE clause that defines a result table that contains only that row:

```
#sql [myConnCtx] {SELECT DEPTNO INTO :hvdeptno
    FROM DEPARTMENT WHERE DEPTNAME="OPERATIONS"};
```

However, most SELECT statements that you use create result tables that contain many rows. In DB2 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.

Related concepts

"SQL statement execution in SQLJ applications" on page 105

"Multiple open iterators for the same SQL statement in an SQLJ application" on page 122

"Multiple open instances of an iterator in an SQLJ application" on page 123

Related tasks

"Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 106

"Using a named iterator in an SQLJ application"

"Using a positioned iterator in an SQLJ application" on page 119

"Using scrollable iterators in an SQLJ application" on page 124

Related reference

"SQLJ iterator-declaration-clause" on page 290

Using a named iterator in an SQLJ application

Use a named iterator to refer to each of the columns in a result table by name.

The steps in using a named iterator are:

1. Declare the iterator.

You declare any result set iterator using an *iterator declaration clause*. This causes an iterator class to be created that has the same name as the iterator. For a named iterator, the iterator declaration clause specifies the following information:

- The name of the iterator
- A list of column names and Java data types

- Information for a Java class declaration, such as whether the iterator is `public` or `static`
- A set of attributes, such as whether the iterator is holdable, or whether its columns can be updated

When you declare a named iterator for a query, you specify names for each of the iterator columns. Those names must match the names of columns in the result table for the query. An iterator column name and a result table column name that differ only in case are considered to be matching names. The named iterator class that results from the iterator declaration clause contains *accessor methods*. There is one accessor method for each column of the iterator. Each accessor method name is the same as the corresponding iterator column name. You use the accessor methods to retrieve data from columns of the result table.

You need to specify Java data types in the iterators that closely match the corresponding DB2 column data types. See "Java, JDBC, and SQL data types" for a list of the best mappings between Java data types and DB2 data types.

You can declare an iterator in a number of ways. However, because a Java class underlies each iterator, you need to ensure that when you declare an iterator, the underlying class obeys Java rules. For example, iterators that contain a *with-clause* must be declared as `public`. Therefore, if an iterator needs to be `public`, it can be declared only where a `public` class is allowed. The following list describes some alternative methods of declaring an iterator:

- As `public`, in a source file by itself
This method lets you use the iterator declaration in other code modules, and provides an iterator that works for all SQLJ applications. In addition, there are no concerns about having other top-level classes or `public` classes in the same source file.
- As a top-level class in a source file that contains other top-level class definitions
Java allows only one `public`, top-level class in a code module. Therefore, if you need to declare the iterator as `public`, such as when the iterator includes a *with-clause*, no other classes in the code module can be declared as `public`.
- As a nested static class within another class
Using this alternative lets you combine the iterator declaration with other class declarations in the same source file, declare the iterator and other classes as `public`, and make the iterator class visible to other code modules or packages. However, when you reference the iterator from outside the nesting class, you must fully-qualify the iterator name with the name of the nesting class.
- As an inner class within another class
When you declare an iterator in this way, you can instantiate it only within an instance of the nesting class. However, you can declare the iterator and other classes in the file as `public`.
You cannot cast a `JDBC ResultSet` to an iterator if the iterator is declared as an inner class. This restriction does not apply to an iterator that is declared as a static nested class. See "Use SQLJ and JDBC in the same application" for more information on casting a `ResultSet` to a iterator.

2. Create an instance of the iterator class.

You declare an object of the named iterator class to retrieve rows from a result table.

3. Assign the result table of a SELECT to an instance of the iterator.

To assign the result table of a `SELECT` to an iterator, you use an SQLJ *assignment clause*. The format of the assignment clause for a named iterator is:

```
#sql context-clause iterator-object={select-statement};
```

See "SQLJ assignment-clause" and "SQLJ context-clause" for more information.

4. Retrieve rows.

Do this by invoking accessor methods in a loop. Accessor methods have the same names as the corresponding columns in the iterator, and have no parameters. An accessor method returns the value from the corresponding column of the current row in the result table. Use the `NamedIterator.next()` method to move the cursor forward through the result table.

To test whether you have retrieved all rows, check the value that is returned when you invoke the next method. `next` returns a boolean with a value of `false` if there is no next row.

5. Close the iterator.

Use the `NamedIterator.close` method to do this.

The following code demonstrates how to declare and use a named iterator. The numbers to the right of selected statements correspond to the previously-described steps.

```
#sql iterator ByName(String LastName, Date HireDate);           1
// Declare named iterator ByName
{
    ...
    ByName nameiter;           // Declare object of ByName class    2
    #sql [ctxt]
    nameiter={SELECT LASTNAME, HIREDATE FROM EMPLOYEE};          3
// Assign the result table of the SELECT
// to iterator object nameiter
    while (nameiter.next())    // Move the iterator through the result  4
// table and test whether all rows retrieved
    {
        System.out.println( nameiter.LastName() + " was hired on "
            + nameiter.HireDate()); // Use accessor methods LastName and
// HireDate to retrieve column values
    }
    nameiter.close();           // Close the iterator                5
}
```

Figure 34. Example of using a named iterator

Related concepts

"SQL statement execution in SQLJ applications" on page 105

"Data retrieval in SQLJ applications" on page 116

Related tasks

"Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 106

"Using a positioned iterator in an SQLJ application"

"Using scrollable iterators in an SQLJ application" on page 124

Related reference

"SQLJ iterator-declaration-clause" on page 290

"sqlj.runtime.NamedIterator interface" on page 303

Using a positioned iterator in an SQLJ application

Use a positioned iterator to refer to columns in a result table by their position in the result set.

The steps in using a positioned iterator are:

1. Declare the iterator.

You declare any result set iterator using an *iterator declaration clause*. This causes an iterator class to be created that has the same name and attributes as the iterator. For a positioned iterator, the iterator declaration clause specifies the following information:

- The name of the iterator
- A list of Java data types
- Information for a Java class declaration, such as whether the iterator is `public` or `static`
- A set of attributes, such as whether the iterator is holdable, or whether its columns can be updated

The data type declarations represent columns in the result table and are referred to as columns of the result set iterator. The columns of the result set iterator correspond to the columns of the result table, in left-to-right order. For example, if an iterator declaration clause has two data type declarations, the first data type declaration corresponds to the first column in the result table, and the second data type declaration corresponds to the second column in the result table.

You need to specify Java data types in the iterators that closely match the corresponding DB2 column data types. See "Java, JDBC, and SQL data types" for a list of the best mappings between Java data types and DB2 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 an iterator.

2. Create an instance of the iterator class.

You declare an object of the positioned iterator class to retrieve rows from a result table.

3. Assign the result table of a `SELECT` to an instance of the iterator.

To assign the result table of a `SELECT` to an iterator, you use an SQLJ *assignment clause*. The format of the assignment clause for a positioned iterator is:

```
#sql context-clause iterator-object={select-statement};
```

4. Retrieve rows.

Do this by executing `FETCH` statements in executable clauses in a loop. The `FETCH` statements look the same as `FETCH` statements in other languages.

To test whether you have retrieved all rows, invoke the `PositionedIterator.endFetch` method after each `FETCH`. `endFetch` returns a boolean with the value `true` if the `FETCH` failed because there are no rows to retrieve.

5. Close the iterator.

Use the `PositionedIterator.close` method to do this.

The following code demonstrates how to declare and use a positioned iterator. The numbers to the right of selected statements correspond to the previously-described steps.

```
#sql iterator ByPos(String,Date); // Declare positioned iterator ByPos 1
{
    ...
    ByPos positer;                // Declare object of ByPos class 2
    String name = null;           // Declare host variables
    Date hrdate;
    #sql [ctxt] positer =          3
        {SELECT LASTNAME, HIREDATE FROM EMPLOYEE};
        // Assign the result table of the SELECT
        // to iterator object positer
    #sql {FETCH :positer INTO :name, :hrdate }; 4
        // Retrieve the first row
    while (!positer.endFetch())    // Check whether the FETCH returned a row
    { System.out.println(name + " was hired in " +
        hrdate);
        #sql {FETCH :positer INTO :name, :hrdate };
        // Fetch the next row
    }
    positer.close();              // Close the iterator 5
}
```

Figure 35. Example of using a positioned iterator

Related concepts

“SQL statement execution in SQLJ applications” on page 105

“Data retrieval in SQLJ applications” on page 116

Related tasks

“Using a named iterator in an SQLJ application” on page 117

“Performing positioned UPDATE and DELETE operations in an SQLJ application” on page 106

“Using scrollable iterators in an SQLJ application” on page 124

Related reference

“SQLJ iterator-declaration-clause” on page 290

“sqlj.runtime.PositionedIterator interface” on page 304

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:

```
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 36 on page 123.

```

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 36. Example of concurrent table operations using iterators with different SQL statements

Figure 37 demonstrates how you can perform the same operations when you have the capability for multiple, concurrently open iterators for a single SQL statement.

```

...
MultiIter iter1 = openIter("000100"); // Invoke openIter to assign the result table
                                         // (for employee 100) to the first iterator
MultiIter iter2 = openIter("000200"); // Invoke openIter to assign the result
                                         // table to the second iterator
                                         // iter1 stays open when iter2 is opened

// Process with iter1
// Process with iter2

...
iter1.close();                          // Close the iterators
iter2.close();

...
public MultiIter openIter(String EmpNo)
                                   // Method to assign a result table
                                   // to an iterator instance
{
    MultiIter iter;
    #sql [ctx] iter =
        {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo};
    return iter;                      // Method returns an iterator instance
}

```

Figure 37. Example of concurrent table operations using iterators with the same SQL statement

Related concepts

“Data retrieval in SQLJ applications” on page 116

Multiple open instances of an iterator in an SQLJ application

Multiple instances of an iterator can be open concurrently in a single SQLJ application. One application for this ability is to open several instances of an iterator that uses host expressions. Each instance can use a different set of host expression values.

The following example shows an application with two concurrently open instances of an iterator.

```

...
ResultSet myFunc(String empid) // Method to open an iterator and get a resultSet
{
    MyIter iter;
    #sql iter = {SELECT * FROM EMPLOYEE WHERE EMPNO = :empid};
    return iter.getResultSet();
}

// An application can call this method to get a resultSet for each
// employee ID. The application can process each resultSet separately.
...
ResultSet rs1 = myFunc("000100"); // Get employee record for employee ID 000100
...
ResultSet rs2 = myFunc("000200"); // Get employee record for employee ID 000200

```

Figure 38. 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.

Related concepts

“Data retrieval in SQLJ applications” on page 116

Using scrollable iterators in an SQLJ application

In addition to moving forward, one row at a time, through a result table, you might want to move backward or go directly to a specific row. The IBM Data Server Driver for JDBC and SQLJ provides this capability.

An iterator in which you can move forward, backward, or to a specific row is called a *scrollable iterator*. A scrollable iterator in SQLJ is equivalent to the result table of a database cursor that is declared as SCROLL.

Like a scrollable cursor, a scrollable iterator 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 Database for Linux, UNIX, and Windows 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.

Important:

To create and use a scrollable iterator, you need to follow these steps:

1. Specify an iterator declaration clause that includes the following clauses:

- `implements sqlj.runtime.Scrollable`

This indicates that the iterator is scrollable.

- `with (sensitivity=INSENSITIVE|SENSITIVE|ASENSITIVE)` or `with (sensitivity=SENSITIVE, dynamic=true|false)`

`sensitivity=INSENSITIVE|SENSITIVE|ASENSITIVE` indicates whether update or delete operations on the underlying table can be visible to the iterator. The default sensitivity is `INSENSITIVE`.

`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=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=INSENSITIVE) (String EmpNo);
```

Restriction: You cannot use a scrollable iterator to select columns with the following data types from a table on a DB2 Database for Linux, UNIX, and Windows server:

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

2. Create an iterator object, which is an instance of your iterator class.
3. 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 21. `sqlj.runtime.Scrollable` methods for positioning a scrollable cursor

Method	Positions the cursor
<code>first</code> ¹	On the first row of the result table
<code>last</code> ¹	On the last row of the result table
<code>previous</code> ^{1,2}	On the previous row of the result table
<code>next</code>	On the next row of the result table
<code>absolute(int n)</code> ^{1,3}	If $n > 0$, on row n of the result table. If $n < 0$, and m is the number of rows in the result table, on row $m+n+1$ of the result table.
<code>relative(int n)</code> ^{1,4}	If $n > 0$, on the row that is n rows after the current row. If $n < 0$, on the row that is n rows before the current row. If $n = 0$, on the current row.
<code>afterLast</code> ¹	After the last row in the result table
<code>beforeFirst</code> ¹	Before the first row in the result table

Notes:

1. This method does not apply to connections to IBM Informix Dynamic Server.
2. If the cursor is after the last row of the result table, this method positions the cursor on the last row.
3. If the absolute value of n is greater than the number of rows in the result table, this method positions the cursor after the last row if n is positive, or before the first row if n is negative.
4. Suppose that m is the number of rows in the result table and x is the current row number in the result table. If $n > 0$ and $x+n > m$, the iterator is positioned after the last row. If $n < 0$ and $x+n < 1$, the iterator is positioned before the first row.

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

If you need to know the current fetch direction, invoke the `getFetchDirection` method.

- c. Use accessor methods to retrieve the current row of the result table.
- d. If update or delete operations by the iterator or by other means are visible in the result table, invoke the `getWarnings` method to check whether the current row is a hole.

For a positioned iterator, perform the following steps:

- a. Use a `FETCH` statement with a fetch orientation clause to position the iterator and retrieve the current row of the result table. Table 22 lists the clauses that you can use to position the cursor.

Table 22. `FETCH` clauses for positioning a scrollable cursor

Method	Positions the cursor
<code>FIRST</code> ¹	On the first row of the result table
<code>LAST</code> ¹	On the last row of the result table
<code>PRIOR</code> ^{1,2}	On the previous row of the result table
<code>NEXT</code>	On the next row of the result table
<code>ABSOLUTE(n)</code> ^{1,3}	If $n > 0$, on row n of the result table. If $n < 0$, and m is the number of rows in the result table, on row $m+n+1$ of the result table.

Table 22. *FETCH* clauses for positioning a scrollable cursor (continued)

Method	Positions the cursor
RELATIVE(<i>n</i>) ^{1,4}	If <i>n</i> >0, on the row that is <i>n</i> rows after the current row. If <i>n</i> <0, on the row that is <i>n</i> rows before the current row. If <i>n</i> =0, on the current row.
AFTER ^{1,5}	After the last row in the result table
BEFORE ^{1,5}	Before the first row in the result table

Notes:

1. This value is not supported for connections to IBM Informix Dynamic Server
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.

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://sysmvsl.stl.ibm.com:5021/NEWYORK",
            userid,password,false); // Create connection context object ctxt
                                   // for the connection to NEWYORK
    ScrollIter scliter;
    #sql [ctxt]
        scliter={SELECT EMPNO, LASTNAME FROM EMPLOYEE};
    scliter.afterLast();
    while (scliter.previous())
    {
        System.out.println(scliter.EmpNo() + " "
            + scliter.LastName());
    }
    scliter.close();
}
```

1

2

4a

4c

5

Related concepts

"SQL statement execution in SQLJ applications" on page 105

"Data retrieval in SQLJ applications" on page 116

 Temporary table space storage requirements (DB2 Installation and Migration)

Related tasks

"Using a named iterator in an SQLJ application" on page 117

"Using a positioned iterator in an SQLJ application" on page 119

Related reference

"SQLJ implements-clause" on page 286

"SQLJ with-clause" on page 287

"SQLJ iterator-declaration-clause" on page 290

"sqlj.runtime.Scrollable interface" on page 307

Calling stored procedures in SQLJ applications

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

You can execute the CALL statement with host identifier parameters. You can execute the CALL statement with literal parameters only if the DB2 server on which the CALL statement runs supports execution of the CALL statement dynamically.

The basic steps in calling a stored procedure are:

1. Assign values to input (IN or INOUT) parameters.
2. Call the stored procedure.
3. Process output (OUT or INOUT) parameters.
4. If the stored procedure returns multiple result sets, retrieve those result sets.

The following code illustrates calling a stored procedure that has three input parameters and three output parameters. The numbers to the right of selected statements correspond to the previously-described steps.

```
String FirstName="TOM";           // Input parameters 1
String LastName="NARISINST";
String Address="IBM";
int CustNo;                       // Output parameters
String Mark;
String MarkErrorText;
...
#sql [myConnCtx] {CALL ADD_CUSTOMER(:IN FirstName, 2
                                :IN LastName,
                                :IN Address,
                                :OUT CustNo,
                                :OUT Mark,
                                :OUT MarkErrorText));
                                // Call the stored procedure
System.out.println("Output parameters from ADD_CUSTOMER call: ");
System.out.println("Customer number for " + LastName + ": " + CustNo); 3
System.out.println(Mark);
If (MarkErrorText != null)
    System.out.println(" Error messages:" + MarkErrorText);
```

Figure 39. Example of calling a stored procedure in an SQLJ application

Related concepts

“SQL statement execution in SQLJ applications” on page 105

“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.

```
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);  
        }  
    }  
}
```

1
2

3a

3b

3c

Figure 40. Retrieving result sets from a stored procedure

Related concepts

“SQL statement execution in SQLJ applications” on page 105

Related tasks

“Calling stored procedures in SQLJ applications” on page 128

“Writing a Java stored procedure to return result sets” on page 177

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.

Related concepts

“SQL statement execution in SQLJ applications” on page 105

“Java data types for retrieving or updating LOB column data in SQLJ applications”

Related reference

“Data types that map to database data types in Java applications” on page 191

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

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.

Input parameters for BLOB columns

For input parameters for BLOB columns, you can use either of the following techniques:

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

```
java.sql.Blob blobData;  
#sql {CALL STORPROC(:IN blobData)};
```

Before you can use a `java.sql.Blob` input variable, you need to create a `java.sql.Blob` object, and then populate that object.

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:

```
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.io.ByteArrayInputStream byteStream =  
    new java.io.ByteArrayInputStream(byteData);  
int numBytes = byteData.length;  
sqlj.runtime.BinaryStream binStream =  
    new sqlj.runtime.BinaryStream(byteStream, numBytes);  
#sql {CALL STORPROC(:IN binStream)};
```

You cannot use this technique for INOUT parameters.

Output parameters for BLOB columns

For output or INOUT parameters for BLOB columns, you can use the following technique:

- Declare the output parameter or INOUT variable with a `java.sql.Blob` data type:

```
java.sql.Blob blobData = null;
#sql CALL STORPROC (:OUT blobData));

java.sql.Blob blobData = null;
#sql CALL STORPROC (:INOUT blobData));
```

Input parameters for CLOB columns

For input parameters for CLOB columns, you can use one of the following techniques:

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

```
#sql CALL STORPROC (:IN clobData));
```

Before you can use a `java.sql.Clob` input variable, you need to create a `java.sql.Clob` object, and then populate that object.

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:

```
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.lang.String charData;
java.io.StringReader reader = new java.io.StringReader(charData);
sqlj.runtime.CharacterStream charStream =
    new sqlj.runtime.CharacterStream (reader, charData.length);
#sql {CALL STORPROC (:IN charStream));
```

- A `sqlj.runtime.UnicodeStream` parameter, for Unicode UTF-16 data:

```
byte[] charDataBytes = charData.getBytes("UnicodeBigUnmarked");
java.io.ByteArrayInputStream byteStream =
    new java.io.ByteArrayInputStream(charDataBytes);
sqlj.runtime.UnicodeStream uniStream =
    new sqlj.runtime.UnicodeStream(byteStream, charDataBytes.length);
#sql {CALL STORPROC (:IN uniStream));
```

- A `sqlj.runtime.AsciiStream` parameter, for ASCII data:

```
byte[] charDataBytes = charData.getBytes("US-ASCII");
java.io.ByteArrayInputStream byteStream =
    new java.io.ByteArrayInputStream (charDataBytes);
sqlj.runtime.AsciiStream asciiStream =
    new sqlj.runtime.AsciiStream (byteStream, charDataBytes.length);
#sql {CALL STORPROC (:IN asciiStream));
```

For these calls, you need to specify the exact length of the input data. You cannot use this technique for INOUT parameters.

- Use a `java.lang.String` input parameter:

```
java.lang.String charData;
#sql {CALL STORPROC (:IN charData));
```

Output parameters for CLOB columns

For output or INOUT parameters for CLOB columns, you can use one of the following techniques:

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

```
java.sql.Clob clobData = null;  
#sql CALL STORPROC(:OUT clobData));
```
- Use a `java.lang.String` output variable:

```
java.lang.String charData = null;  
#sql CALL STORPROC(:OUT charData));
```

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

Output parameters for DBCLOB columns

DBCLOB output or INOUT parameters for stored procedures are not supported.

Related concepts

“LOBs in SQLJ applications with the IBM Data Server Driver for JDBC and SQLJ” on page 130

Related reference

“Data types that map to database data types in Java applications” on page 191

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 41 builds and executes a query using a JDBC call, executes an iterator conversion statement to convert the JDBC result set to an SQLJ iterator, and retrieves rows from the result table using the iterator.

```
#sql public iterator ByName(String LastName, Date HireDate); 1
public void HireDates(ConnectionContext connCtx, String whereClause)
{
    ByName nameiter;          // Declare object of ByName class
    Connection conn=connCtx.getConnection();
                                // Create JDBC connection
    Statement stmt = conn.createStatement(); 2
    String query = "SELECT LASTNAME, HIREDATE FROM EMPLOYEE";
    query+=whereClause; // Build the query
    ResultSet rs = stmt.executeQuery(query); 3
    #sql [connCtx] nameiter = {CAST :rs}; 4
    while (nameiter.next())
    {
        System.out.println( nameiter.LastName() + " was hired on "
            + nameiter.HireDate());
    }
    nameiter.close(); 5
    stmt.close();
}
```

Figure 41. Converting a JDBC result set to an SQLJ iterator

Notes to Figure 41:

Note	Description
1	This SQLJ clause creates the named iterator class ByName, which has accessor methods LastName() and HireDate() that return the data from result table columns LASTNAME and HIREDATE.
2	This statement and the following two statements build and prepare a query for dynamic execution using JDBC.
3	This JDBC statement executes the SELECT statement and assigns the result table to result set rs.

Note	Description
4	This iterator conversion clause converts the JDBC <code>ResultSet</code> <code>rs</code> to SQLJ iterator <code>nameiter</code> , and the following statements use <code>nameiter</code> to retrieve values from the result table.
5	The <code>nameiter.close()</code> method closes the SQLJ iterator and JDBC <code>ResultSet</code> <code>rs</code> .

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 42 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 42. Converting an SQLJ iterator to a JDBC `ResultSet`

Notes to Figure 42:

Note	Description
1	This SQLJ clause executes the SELECT statement, constructs an iterator object that contains the result table for the SELECT statement, and assigns the iterator object to variable <code>iter</code> .
2	The <code>getResultSet()</code> method accesses the <code>ResultSet</code> that underlies iterator <code>iter</code> .
3	The JDBC <code>getString()</code> and <code>getDate()</code> methods retrieve values from the <code>ResultSet</code> . The <code>next()</code> method moves the cursor to the next row in the <code>ResultSet</code> .
4	The <code>rs.close()</code> method closes the SQLJ iterator as well as the <code>ResultSet</code> .

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 concepts

“SQL statement execution in SQLJ applications” on page 105

Related tasks

“Connecting to a data source using SQLJ” on page 95

Related reference

“SQLJ with-clause” on page 287

“SQLJ assignment-clause” on page 296

“`sqlj.runtime.ConnectionContext` interface” on page 298

 `DESCRIBE FOR STATIC` field (`DESCSTAT` subsystem parameter) (DB2 Installation and Migration)

Controlling the execution of SQL statements in SQLJ

You can use selected methods of the SQLJ `ExecutionContext` class to control or monitor the execution of SQL statements.

To use `ExecutionContext` methods, follow these steps:

1. Acquire the default execution context from the connection context.

There are two ways to acquire an execution context:

- Acquire the default execution context from the connection context. For example:

```
ExecutionContext execCtx = connCtx.getExecutionContext();
```

- Create a new execution context by invoking the constructor for `ExecutionContext`. For example:

```
ExecutionContext execCtx=new ExecutionContext();
```

2. Associate the execution context with an SQL statement.

To do that, specify an execution context after the connection context in the execution clause that contains the SQL statement.

3. Invoke `ExecutionContext` methods.

Some `ExecutionContext` methods are applicable before the associated SQL statement is executed, and some are applicable only after their associated SQL statement is executed.

For example, you can use method `getUpdateCount` to count the number of rows that are deleted by a `DELETE` statement after you execute the `DELETE` statement.

The following code demonstrates how to acquire an execution context, and then use the `getUpdateCount` method on that execution context to determine the number

of rows that were deleted by a DELETE statement. The numbers to the right of selected statements correspond to the previously-described steps.

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

1
2
3

Related concepts

“SQL statement execution in SQLJ applications” on page 105

Related tasks

“Making batch updates in SQLJ applications” on page 113

“Handling SQL warnings in an SQLJ application” on page 150

Related reference

“SQLJ context-clause” on page 292

“sqlj.runtime.ExecutionContext class” on page 312

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

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

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

If you use columns with the ROWID data type in SQLJ programs, you need to customize those programs.

JDBC 4.0 includes interface `java.sql.RowId` that you can use in iterators and in CALL statement parameters. If you do not have JDBC 4.0, you can use the IBM Data Server Driver for JDBC and SQLJ-only class `com.ibm.db2.jcc.DB2RowID`. For an iterator, you can also use the `byte[]` object type to retrieve ROWID values.

The following code shows an example of an iterator that is used to select values from a ROWID column:

```

#sql iterator PosIter(int,String,java.sql.RowId);
                                // Declare positioned iterator
                                // for retrieving ITEM_ID (INTEGER),
                                // ITEM_FORMAT (VARCHAR), and ITEM_ROWID (ROWID)
                                // values from table ROWIDTAB
{
    PosIter positrowid;        // Declare object of PosIter class
    java.sql.RowId rowid = null;
    int id = 0;
    String i_fmt = null;

                                // Declare host expressions
    #sql [ctxt] positrowid =
        {SELECT ITEM_ID, ITEM_FORMAT, ITEM_ROWID FROM ROWIDTAB
         WHERE ITEM_ID=3};
                                // Assign the result table of the SELECT
                                // to iterator object positrowid
    #sql {FETCH :positrowid INTO :id, :i_fmt, :rowid};
                                // Retrieve the first row
    while (!positrowid.endFetch())
        // Check whether the FETCH returned a row
        {System.out.println("Item ID " + id + " Item format " +
            i_fmt + " Item ROWID ");
            MyUtilities.printBytes(rowid.getBytes());
                                // Use the getBytes method to
                                // convert the value to bytes for printing.
                                // Call a user-defined method called
                                // printBytes (not shown) to print
                                // the value.
            #sql {FETCH :positrowid INTO :id, :i_fmt, :rowid};
                                // Retrieve the next row
        }
    positrowid.close();        // Close the iterator
}

```

Figure 43. Example of using an iterator to retrieve ROWID values

The following code shows an example of calling a stored procedure that takes three ROWID parameters: an IN parameter, an OUT parameter, and an INOUT parameter.

```

java.sql.RowId in_rowid = rowid;
java.sql.RowId 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 44. Example of calling a stored procedure with a ROWID parameter

Related reference

“Data types that map to database data types in Java applications” on page 191
“DB2RowID interface” on page 374

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:

```
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 45. Defining and using a distinct type

Related reference

“Data types that map to database data types in Java applications” on page 191

 CREATE TYPE (SQL Reference)

Savepoints in SQLJ applications

Under the IBM Data Server Driver for JDBC and SQLJ, you can include any form of the SQL SAVEPOINT statement in your SQLJ program.

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

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

Figure 46. Setting, rolling back to, and releasing a savepoint in an SQLJ application

```

#sql context Ctx;           // Create connection context class Ctx
String empNumVar;
int shoeSizeVar;
...
try {                       // Load the JDBC driver
    Class.forName("com.ibm.db2.jcc.DB2Driver");
}
catch (ClassNotFoundException e) {
    e.printStackTrace();
}
Connection jdbccon=
    DriverManager.getConnection("jdbc:db2://sysmvsl.stl.ibm.com:5021/NEWYORK",
        userid,password);
// Create JDBC connection object jdbccon
jdbccon.setAutoCommit(false); // Do not autocommit
Ctx ctxt=new Ctx(jdbccon);
// Create connection context object myConnCtx
// for the connection to NEWYORK
...
// Perform some SQL
#sql [ctxt] {COMMIT};       // Commit the transaction
// Commit the create
#sql [ctxt]
    {INSERT INTO EMP_SHOE VALUES ('000010', 6)};
// Insert a row
#sql [ctxt]
    {SAVEPOINT SVPT1 ON ROLLBACK RETAIN CURSORS};
// Create a savepoint
...
#sql [ctxt]
    {INSERT INTO EMP_SHOE VALUES ('000020', 10)};
// Insert another row
#sql [ctxt] {ROLLBACK TO SAVEPOINT SVPT1};
// Roll back work to the point
// after the first insert
...
#sql [ctxt] {RELEASE SAVEPOINT SVPT1};
// Release the savepoint
ctx.close();               // Close the connection context

```

Related tasks

“Committing or rolling back SQLJ transactions” on page 148

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.

In applications, XML data is in the serialized string format.

In SQLJ applications that connect to DB2 Database for Linux, UNIX, and Windows, XML data is in textual XML format. In SQLJ applications that connect to DB2 for z/OS, XML data can be in textual XML format or binary XML format.

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.

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 column updates in SQLJ applications”

“XML data retrieval in SQLJ applications” on page 143

XML column updates in SQLJ applications

When you update or insert data into XML columns of a database table, the input data in your SQLJ applications must be in the serialized string format.

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 Database for Linux, UNIX, and Windows, 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 [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.

```
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.runtime.AsciiStream` host expression, and insert data from the `sqlj.runtime.AsciiStream` host expression into an XML column in a table on a data source.

```
byte[] b = xmlString.getBytes("US-ASCII");
java.io.ByteArrayInputStream xmlAsciiInputStream =
    new java.io.ByteArrayInputStream(b);
sqlj.runtime.AsciiStream sqljXmlAsciiStream =
    new sqlj.runtime.AsciiStream(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.io.StringReader xmlReader =
    new java.io.StringReader(xmlString);
sqlj.runtime.CharacterStream sqljXmlCharacterStream =
    new sqlj.runtime.CharacterStream(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.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.jcc.DB2Xml` host expression, and insert the data into an XML column in a table.

```
java.sql.ResultSet rs = s.executeQuery ("SELECT * FROM CUSTACC");
rs.next();
com.ibm.db2.jcc.DB2Xml xmlObject = (com.ibm.db2.jcc.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.

Related concepts

“XML data in SQLJ applications” on page 140

Related reference

“DB2Xml interface” on page 391

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 23. SQLXML and DB2Xml methods, data types, and added encoding specifications

Method	Output data type	Type of XML internal encoding declaration added
<code>SQLXML.getBinaryStream</code>	<code>InputStream</code>	None
<code>SQLXML.getCharacterStream</code>	<code>Reader</code>	None
<code>SQLXML.getSource</code>	<code>Source</code>	None
<code>SQLXML.getString</code>	<code>String</code>	None
<code>DB2Xml.getDB2AsciiStream</code>	<code>InputStream</code>	None
<code>DB2Xml.getDB2BinaryStream</code>	<code>InputStream</code>	None
<code>DB2Xml.getDB2Bytes</code>	<code>byte[]</code>	None
<code>DB2Xml.getDB2CharacterStream</code>	<code>Reader</code>	None
<code>DB2Xml.getDB2String</code>	<code>String</code>	None
<code>DB2Xml.getDB2XmlAsciiStream</code>	<code>InputStream</code>	US-ASCII
<code>DB2Xml.getDB2XmlBinaryStream</code>	<code>InputStream</code>	Specified by <code>getDB2XmlBinaryStream targetEncoding</code> parameter

Table 23. SQLXML and DB2Xml methods, data types, and added encoding specifications (continued)

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

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 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 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 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 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.

Related concepts

“XML data in SQLJ applications” on page 140

Related reference

“DB2Xml interface” on page 391

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.

```
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};
```

SQLJ utilization of SDK for Java Version 5 function

Your SQLJ applications can use a number of functions that were introduced with the SDK for Java Version 5.

Static import

The static import construct lets you access static members without qualifying those members with the name of the class to which they belong. For SQLJ applications, this means that you can use static members in host expressions without qualifying them.

Example: Suppose that you want to declare a host expression of this form:

```
double r = cos(PI * E);
```

cos, PI, and E are members of the java.lang.Math class. To declare r without explicitly qualifying cos, PI, and E, include the following static import statement in your program:

```
import static java.lang.Math.*;
```

Annotations

Java annotations are a means for adding metadata to Java programs that can also affect the way that those programs are treated by tools and libraries. Annotations are declared with annotation type declarations, which are similar to interface declarations. Java annotations can appear in the following types of classes or interfaces:

- Class declaration
- Interface declaration
- Nested class declaration
- Nested interface declaration

You cannot include Java annotations directly in SQLJ programs, but you can include annotations in Java source code, and then include that source code in your SQLJ programs.

Example: Suppose that you declare the following marker annotation in a program called `MyAnnot.java`:

```
public @interface MyAnot { }
```

You also declare the following marker annotation in a program called `MyAnnot2.java`:

```
public @interface MyAnot2 { }
```

You can then use those annotations in an SQLJ program:

```
// Class annotations
@MyAnot2 public @MyAnot class TestAnnotation
{
    // Field annotation
    @MyAnot
    private static final int field1 = 0;
    // Constructor annotation
    @MyAnot2 public @MyAnot TestAnnotation () { }
    // Method annotation
    @MyAnot
    public static void main (String a[])
    {
        TestAnnotation TestAnnotation_o = new TestAnnotation();
        TestAnnotation_o.runThis();
    }
    // Inner class annotation
    public static @MyAnot class TestAnotherInnerClass { }
    // Inner interface annotation
    public static @MyAnot interface TestAnotInnerInterface { }
}
```

Enumerated types

An enumerated type is a data type that consists of a set of ordered values. The SDK for Java version 5 introduces the `enum` type for enumerated types.

You can include enums in the following places:

- In Java source files (.java files) that you include in an SQLJ program
- In SQLJ class declarations

Example: The `TestEnum.sqlj` class declaration includes an `enum` type:

```
public class TestEnum2
{
    public enum Color {
        RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
    }
    Color color;
    ...
    // Get the value of color
    switch (color) {
    case RED:
        System.out.println("Red is at one end of the spectrum.");
        #sql[ctx] { INSERT INTO MYTABLE VALUES (:color) };
        break;
    case VIOLET:
        System.out.println("Violet is on the other end of the spectrum.");
        break;
    case ORANGE:
    case YELLOW:
    case GREEN:
    }
```

```

    case BLUE:
    case INDIGO:
        System.out.println("Everything else is in the middle.");
        break;
}

```

Generics

You can use generics in your Java programs to assign a type to a Java collection. The SQLJ translator tolerates Java generic syntax. Examples of generics that you can use in SQLJ programs are:

- A List of List objects:

```
List <List<String>> strList2 = new ArrayList<List<String>>();
```
- A HashMap in which the key/value pair has the String type:

```
Map <String,String> map = new HashMap<String,String>();
```
- A method that takes a List with elements of any type:

```
public void mthd(List <?> obj) {
    ...
}
```

Although you can use generics in SQLJ host variables, the value of doing so is limited because the SQLJ translator cannot determine the types of those host variables.

Enhanced for loop

The enhanced for lets you specify that a set of operations is performed on each member of a collection or array. You can use the iterator in the enhanced for loop in host expressions.

Example: INSERT each of the items in array names into table TAB.

```

String[] names = {"ABC","DEF","GHI"};
for (String n : names)
{
    #sql {INSERT INTO TAB (VARCHARCOL) VALUES(:n) };
}

```

Varargs

Varargs make it easier to pass an arbitrary number of values to a method. A Vararg in the last argument position of a method declaration indicates that the last arguments are an array or a sequence of arguments. An SQLJ program can use the passed arguments in host expressions.

Example: Pass an arbitrary number of parameters of type Object, to a method that inserts each parameter value into table TAB.

```

public void runThis(Object... objects) throws SQLException
{
    for (Object obj : objects)
    {
        #sql { INSERT INTO TAB (VARCHARCOL) VALUES(:obj) };
    }
}

```

Transaction control in SQLJ applications

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

Setting the isolation level for an SQLJ transaction

To set the isolation level for a unit of work within an SQLJ program, use the SET TRANSACTION ISOLATION LEVEL clause.

The following table shows the values that you can specify in the SET TRANSACTION ISOLATION LEVEL clause and their DB2 equivalents.

Table 24. Equivalent SQLJ and DB2 isolation levels

SET TRANSACTION value	DB2 isolation level
SERIALIZABLE	Repeatable read
REPEATABLE READ	Read stability
READ COMMITTED	Cursor stability
READ UNCOMMITTED	Uncommitted read

The isolation level affects the underlying JDBC connection as well as the SQLJ connection.

Related concepts

“SQL statement execution in SQLJ applications” on page 105

“JDBC connection objects” on page 18

Related reference

“SQLJ statement-clause” on page 292

“SQLJ SET-TRANSACTION-clause” on page 295

Committing or rolling back SQLJ transactions

If you disable autocommit for an SQLJ connection, you need to perform explicit commit or rollback operations.

You do this using execution clauses that contain the SQL COMMIT or ROLLBACK statements.

To commit a transaction in an SQLJ program, use a statement like this:

```
#sql [myConnCtx] {COMMIT};
```

To roll back a transaction in an SQLJ program, use a statement like this:

```
#sql [myConnCtx] {ROLLBACK};
```

Related concepts

“SQL statement execution in SQLJ applications” on page 105

“Savepoints in SQLJ applications” on page 139

Related tasks

“Committing or rolling back SQLJ transactions” on page 148

“Connecting to a data source using SQLJ” on page 95

Handling SQL errors and warnings in SQLJ applications

SQLJ clauses throw `SQLExceptions` when SQL errors occur, but not when most SQL warnings occur.

SQLJ generates an `SQLException` under the following circumstances:

- When any SQL statement returns a negative SQL error code
- When a `SELECT INTO` SQL statement returns a +100 SQL error code

You need to explicitly check for other SQL warnings.

- For SQL error handling, include try/catch blocks around SQLJ statements.
- For SQL warning handling, invoke the `getWarnings` method after every SQLJ statement.

Handling SQL errors in an SQLJ application

SQLJ clauses use the JDBC class `java.sql.SQLException` for error handling.

To handle SQL errors in SQLJ applications, following these steps:

1. Import the `java.sql.SQLException` class.
2. Use the Java error handling try/catch blocks to modify program flow when an SQL error occurs.
3. Obtain error information from the `SQLException`.

You can use the `getErrorCode` method to retrieve SQL error codes and the `getSQLState` method to retrieve `SQLSTATEs`.

If you are using the IBM Data Server Driver for JDBC and SQLJ, obtain additional information from the `SQLException` by casting it to a `DB2Diagnosable` object, in the same way that you obtain this information in a JDBC application.

The following code prints out the SQL error that occurred if a `SELECT` statement fails.

```
try {
    #sql [ctxt] {SELECT LASTNAME INTO :empname
                FROM EMPLOYEE WHERE EMPNO='000010'};
}
catch(SQLException e) {
    System.out.println("Error code returned: " + e.getErrorCode());
}
```

Related concepts

"Example of a simple SQLJ application" on page 93

"SQL statement execution in SQLJ applications" on page 105

Related tasks

"Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ" on page 84

"Handling SQL warnings in an SQLJ application"

Related reference

"Error codes issued by the IBM Data Server Driver for JDBC and SQLJ" on page 407

"SQLSTATEs issued by the IBM Data Server Driver for JDBC and SQLJ" on page 413

Handling SQL warnings in an SQLJ application

Other than a +100 SQL error code on a SELECT INTO statement, warnings from the data server do not throw SQLExceptions. To handle warnings from the data server, you need to give the program access to the `java.sql.SQLWarning` class.

If you want to retrieve data-server-specific information about a warning, you also need to give the program access to the `com.ibm.db2.jcc.DB2Diagnosable` interface and the `com.ibm.db2.jcc.DB2Sqlca` class. Then follow these steps:

1. Set up an execution context for that SQL clause. See "Control the execution of SQL statements in SQLJ" for information on how to set up an execution context.
2. To check for a warning from the data server, invoke the `getWarnings` method after you execute an SQLJ clause.
`getWarnings` returns the first `SQLWarning` object that an SQL statement generates. Subsequent `SQLWarning` objects are chained to the first one.
3. To retrieve data-server-specific information from the `SQLWarning` object with the IBM Data Server Driver for JDBC and SQLJ, follow the instructions in "Handle an SQLException under the IBM Data Server Driver for JDBC and SQLJ".

The following example demonstrates how to retrieve an `SQLWarning` object for an SQL clause with execution context `execCtx`. The numbers to the right of selected statements correspond to the previously-described steps.

```
ExecutionContext execCtx=myConnCtx.getExecutionContext(); 1
// Get default execution context from
// connection context

SQLWarning sqlWarn;
...
#sql [myConnCtx,execCtx] {SELECT LASTNAME INTO :empname
    FROM EMPLOYEE WHERE EMPNO='000010'};
if ((sqlWarn = execCtx.getWarnings()) != null) 2
    System.out.println("SQLWarning " + sqlWarn);
```

Related concepts

“Example of a simple SQLJ application” on page 93

“SQL statement execution in SQLJ applications” on page 105

Related tasks

“Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ” on page 84

“Controlling the execution of SQL statements in SQLJ” on page 136

“Handling SQL errors in an SQLJ application” on page 149

Related reference

“DB2Diagnosable interface” on page 353

“DB2Sqlca class” on page 375

Closing the connection to a data source in an SQLJ application

When you have finished with a connection to a data source, you need to close the connection to the data source. Doing so releases the connection context object's DB2 and SQLJ resources immediately.

To close the connection to the data source, use one of the `ConnectionContext.close` methods.

- If you execute `ConnectionContext.close()` or `ConnectionContext.close(ConnectionContext.CLOSE_CONNECTION)`, the connection context, as well as the connection to the data source, are closed.
- If you execute `ConnectionContext.close(ConnectionContext.KEEP_CONNECTION)` the connection context is closed, but the connection to the data source is not.

The following code closes the connection context, but does not close the connection to the data source.

```
...
ctx = new EzSqljctx(con0);           // Create a connection context object
                                     // from JDBC connection con0
...
                                     // Perform various SQL operations
EzSqljctx.close(ConnectionContext.KEEP_CONNECTION);
                                     // Close the connection context but keep
                                     // the connection to the data source open
```

Related concepts

“Example of a simple SQLJ application” on page 93

Related tasks

“Connecting to a data source using SQLJ” on page 95


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 \(DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond\)](#)

 [Java stored procedure returning a BLOB column \(DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond\)](#)

 [Java stored procedure returning a CLOB column \(DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond\)](#)

 [Debugging Java procedures on Linux, UNIX, and Windows \(DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond\)](#)

 [Java sample SQLJ stored procedure \(DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond\)](#)

Setting up the environment for Java routines

Before you can run Java routines, you need to set up a WLM environment and Java environment variables for them.

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 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.

- 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.

This topic discusses the setup tasks for preparing and running Java routines.

If you plan to use IBM Optim™ Development Studio to prepare and run your Java routines, see the information on developing database routines in the Integrated Data Management Information Center, at the following URL:

<http://publib.boulder.ibm.com/infocenter/idm/v2r1/index.jsp>

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 Java routines, which includes setting environment variables.

Setting up the WLM application environment for Java routines

To set up WLM application environments for stored procedures or user-defined functions, you need to define a JCL startup procedure for each WLM environment, and define the application environment to WLM. You need different WLM application environments for Java routines from the WLM application environments that you use for other routines.

To set up the WLM environment for Java routines, you need to perform these steps:

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.

```
//DSNWLM PROC RGN=0K,APPLENV=WLMIJAV,DB2SSN=DSN,NUMTCB=5 1
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
// PARM='&DB2SSN,&NUMTCB,&APPLENV'

//STEPLIB DD DISP=SHR,DSN=DSN910.RUNLIB.LOAD
// DD DISP=SHR,DSN=CEE.SCEERUN
// DD DISP=SHR,DSN=DSN910.SDSNEXIT
// DD DISP=SHR,DSN=DSN910.SDSNLOAD
// DD DISP=SHR,DSN=DSN910.SDSNLOAD2
//JAVAENV DD DISP=SHR,DSN=WLMIJAV.JSPENV 2
//JSPDEBUG DD SYSOUT=A 3
//NOWLMENC DD SYSOUT=A 4
//CEEDUMP DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
```

Figure 47. Startup procedure for a WLM address space in which a Java routine runs

Notes to Figure 47 on page 154:

- 1** In this line, 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. The maximum value of NUMTCB should be 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.
- 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 containing the Java SDK.
- 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”

“Run-time environment for Java routines” on page 156

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.

File Utilities Notes Options Help

Definition Menu WLM Appl

Command ==> _____

Definition data set . . : none

Definition name WLMENV

Description Environment for Java stored procedures

Select one of the

following options. . . 9

1. Policies

2. Workloads

3. Resource Groups

4. Service Classes

5. Classification Groups

6. Classification Rules

7. Report Classes

8. Service Coefficients/Options

9. Application Environments

10. Scheduling Environments

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).

Application-Environment
Notes
Options
Help

Create an Application Environment

Command ==> _____

Application Environment Name . . : WLMENV
Description Environment for Java stored procedures
Subsystem Type DB2
Procedure Name DSN8WLMF
Start Parameters DB2SSN=DB2T,NUMTCB=3,APPLENV=WLMENV

Limit on starting server address spaces for a subsystem instance:
1 1. No limit.
2. Single address space per system.
3. Single address spaces per sysplex.

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 154
“Run-time environment for Java routines”
“Environment variables for the IBM Data Server Driver for JDBC and SQLJ” on page 441

Run-time 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 run-time options for the routines that run in the stored procedure address space.

Create the data set for the run-time options with the characteristics that are listed in the following table.

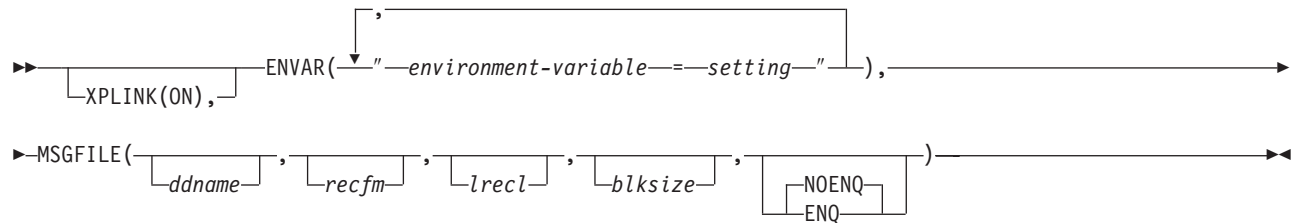
Table 25. Data set characteristics for the JAVAENV data set

Primary space allocation	1 block
Secondary space allocation	1 block
Record format	VB

Table 25. Data set characteristics for the JAVAENV data set (continued)

Record length	255
Block size	4096

After you create the data set, edit it to insert a Language Environment options string, which has this form:



The maximum length of the Language Environment run-time 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 run-time options string. The run-time 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:

XPLINK(ON)

Causes the initialization of the XPLINK environment. This parameter is **required**.

MSGFILE

Specifies the DD name of a data set in which Language Environment puts run-time 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.

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/db2/db2910_base
```

The default is /usr/lpp/db2910_base.

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/db2910_base
```

JCC_HOME must be set.

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/J1.4.2
```

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/jvmssp
```

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.

WORK_DIR

Modify WORK_DIR to change the default destination for STDOUT and STDERR output.

_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/db291/javasp/jspnolimit.txt for environment variable settings, specify:

_CEE_ENVFILE=/u/db291/javasp/jspnolimit.txt

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.

The following example shows the contents of a JAVAENV data set.

```
ENVAR("JCC_HOME=/usr/lpp/db2910",  
"JAVA_HOME=/usr/lpp/javas14/J1.4.2",  
"WORK_DIR=/u/db291/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 154

“WLM application environment values for Java routines” on page 155

“Techniques for testing a Java routine” on page 178

“Environment variables for the IBM Data Server Driver for JDBC and SQLJ” on page 441

Related tasks

“Preparing Java routines with no SQLJ clauses and no JAR file” on page 186

“Preparing Java routines with SQLJ clauses and no JAR file” on page 189

“Installing the z/OS Application Connectivity to DB2 for z/OS feature” on page 461

“Upgrading the IBM Data Server Driver for JDBC and SQLJ to a new version” on page 460

Defining Java routines and JAR files to DB2

Before you can use a Java routine, you need to define it to DB2.

If the routine is in a JAR file, it is recommended that you also define the JAR file to DB2. Alternatively, you can include the JAR file name in the CLASSPATH.

If you use IBM Optim Development Studio, IBM Optim Development Studio creates the definitions. If you do not use IBM Optim Development Studio, perform these steps:

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. If the routines are in JAR files, 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

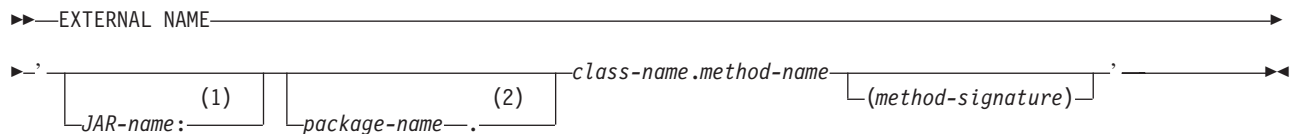
Specifies the application programming language in which the routine is written.

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:



Notes:

- 1 For compatibility with DB2 for Linux, UNIX, and Windows, 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.getSals
Type of SQL statements in the program	Statements that modify DB2 tables
WLM environment name	WLMIJAV
Maximum number of result sets returned	1

This CREATE PROCEDURE statement defines the stored procedure to DB2:

```
CREATE PROCEDURE SYSPROC.S1SAL
  (DECIMAL(10,2) INOUT)
  FENCED
  MODIFIES SQL DATA
  COLLID DSNJDBC
  LANGUAGE JAVA
  EXTERNAL NAME 's1.S1Sal.getSals'
  WLM ENVIRONMENT WLMIJAV
  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	MYSHEMA.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	WLMIJAV

This CREATE FUNCTION statement defines the user-defined function to DB2:


```
CREATE FUNCTION MYSCHEMA.S2SAL(INTEGER)
  RETURNS VARCHAR(20)
  FENCED
  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) (SQL Reference)

 ALTER PROCEDURE (external) (SQL Reference)

 CREATE FUNCTION (SQL Reference)

 CREATE PROCEDURE (external) (SQL Reference)

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 161

Related tasks

“Preparing Java routines with no SQLJ clauses to run from a JAR file” on page 185

“Preparing Java routines with SQLJ clauses to run from a JAR file” on page 186

SQLJ.INSTALL_JAR stored procedure

SQLJ.INSTALL_JAR creates a new definition of a JAR file in the local DB2 catalog.

SQLJ.INSTALL_JAR authorization

Privilege set: If the CALL statement is embedded in an application program, the privilege set consists of the privileges that are held by the authorization ID of the

owner of the plan or package. If the statement is dynamically prepared, the privilege set consists of the privileges that are held by the authorization IDs of the process.

For calling `SQLJ.INSTALL_JAR`, the privilege set must include at least one of the following items:

- EXECUTE privilege on `SQLJ.INSTALL_JAR`
- Ownership of `SQLJ.INSTALL_JAR`
- SYSADM authority

The privilege set must also include the authority to install a JAR, which consists of at least one of the following items:

- CREATEIN privilege on the schema of the JAR
The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.
- SYSADM or SYSCTRL authority

SQLJ.INSTALL_JAR syntax

►►—CALL—SQLJ.INSTALL_JAR—(—url,—JAR-name,—deploy—)—————►►

SQLJ.INSTALL_JAR parameters

url A VARCHAR(1024) input parameter that identifies the z/OS UNIX System Services full path name for the JAR file that is to be installed in the DB2 catalog. The format is `file://path-name` or `file:/path-name`.

JAR-name

A VARCHAR(257) input parameter that contains the DB2 name of the JAR, in the form `schema.JAR-id` or `JAR-id`. *JAR-name* is the name that you use when you refer to the JAR in SQL statements. If you omit *schema*, DB2 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.

deploy

An INTEGER input parameter that indicates whether additional actions are to be performed after the JAR file is installed. Additional actions are not supported, so this value is 0.

SQLJ.INSTALL_JAR example

Suppose that you want to install the JAR file that is in path `/u/db2inst3/apps/BUILDPLAN/BUILDPLAN.jar`. You want to refer to the JAR file as `DB2INST3.BUILDPLAN` in SQL statements. Use a CALL statement similar to this one.

```
CALL SQLJ.INSTALL_JAR('file:/u/db2inst3/apps/BUILDPLAN/BUILDPLAN.jar',  
  'DB2INST3.BUILDPLAN',0)
```

SQLJ.DB2_INSTALL_JAR stored procedure

`SQLJ.DB2_INSTALL_JAR` creates a new definition of a JAR file in the local DB2 catalog or in a remote DB2 catalog.

To install a JAR file at a remote location, you need to execute a `CONNECT` statement to connect to that location before you call `SQLJ.DB2_INSTALL_JAR`.

SQLJ.DB2_INSTALL_JAR authorization

Privilege set: If the CALL statement is embedded in an application program, the privilege set consists of the privileges that are held by the authorization ID of the owner of the plan or package. If the statement is dynamically prepared, the privilege set consists of the privileges that are held by the authorization IDs of the process.

For calling SQLJ.DB2_INSTALL_JAR, the privilege set must include at least one of the following items:

- EXECUTE privilege on SQLJ.DB2_INSTALL_JAR
- Ownership of SQLJ.DB2_INSTALL_JAR
- SYSADM authority

The privilege set must also include the authority to install a JAR, which consists of at least one of the following items:

- CREATEIN privilege on the schema of the JAR
The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.
- SYSADM or SYSCTRL authority

SQLJ.DB2_INSTALL_JAR syntax

```
►►—CALL—SQLJ.DB2_INSTALL_JAR—(—Jar-locator,—JAR-name,—deploy—)—————►►
```

SQLJ.DB2_INSTALL_JAR parameters

JAR-locator

A BLOB locator input parameter that points to the JAR file that is to be installed in the DB2 catalog.

JAR-name

A VARCHAR(257) input parameter that contains the DB2 name of the JAR, in the form *schema.JAR-id* or *JAR-id*. *JAR-name* is the name that you use when you refer to the JAR in SQL statements. If you omit *schema*, DB2 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.

deploy

An INTEGER input parameter that indicates whether additional actions are to be performed after the JAR file is installed. Additional actions are not supported, so this value is 0.

SQLJ.DB2_INSTALL_JAR example

Suppose that you want to install the JAR file that is in path /u/db2inst3/apps/BUILDPLAN/BUILDPLAN.jar. You want to refer to the JAR file as DB2INST3.BUILDPLAN in SQL statements. The following Java program installs that JAR file.

```
import java.sql.*; // JDBC classes
import java.io.IOException;
import java.io.File;
import java.io.FileInputStream;
class SimpleInstallJar
{
    public static void main (String argv[])
```

```

{
String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021";
String jarname = "DB2INST3.BUILDPLAN";
String jarfile =
"/u/db2inst3/apps/BUILDPLAN/BUILDPLAN.jar";
try
{
Class.forName ("com.ibm.db2.jcc.DB2Driver").newInstance ();
Connection con =
DriverManager.getConnection(url, "MYID", "MYPW");
File aFile = new File(jarfile);
FileInputStream inputStream = new FileInputStream(aFile);
CallableStatement stmt;
String sql = "Call SQLJ.DB2_INSTALL_JAR(?, ?, ?)";
stmt = con.prepareCall(sql);
stmt.setBinaryStream(1, inputStream, (int)aFile.length());
stmt.setString(2, jarname);
stmt.setInt(3, 0);
boolean isrs = stmt.execute();
stmt.close();
System.out.println("Installation of JAR succeeded");
con.commit();
con.close();
}
catch (Exception e)
{
System.out.println("Installation of JAR failed");
e.printStackTrace ();
}
}
}

```

SQLJ.REPLACE_JAR stored procedure

SQLJ.REPLACE_JAR replaces an existing JAR file in the local DB2 catalog.

SQLJ.REPLACE_JAR authorization

Privilege set: If the CALL statement is embedded in an application program, the privilege set consists of the privileges that are held by the authorization ID of the owner of the plan or package. If the statement is dynamically prepared, the privilege set consists of the privileges that are held by the authorization IDs of the process.

For calling SQLJ.REPLACE_JAR, the privilege set must include at least one of the following items:

- EXECUTE privilege on SQLJ.REPLACE_JAR
- Ownership of SQLJ.REPLACE_JAR
- SYSADM authority

The privilege set must also include the authority to replace a JAR, which consists of at least one of the following items:

- Ownership of the JAR
- ALTERIN privilege on the schema of the JAR

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.
- SYSADM or SYSCTRL authority

SQLJ.REPLACE_JAR syntax

►►—CALL—SQLJ.REPLACE_JAR—(—*url*,—*JAR-name*—)——►►

SQLJ.REPLACE_JAR parameters

url A VARCHAR(1024) input parameter that identifies the z/OS UNIX System Services full path name for the JAR file that replaces the existing JAR file in the DB2 catalog. The format is `file://path-name` or `file:/path-name`.

JAR-name

A VARCHAR(257) input parameter that contains the DB2 name of the JAR, in the form `schema.JAR-id` or `JAR-id`. *JAR-name* is the name that you use when you refer to the JAR in SQL statements. If you omit *schema*, DB2 uses the SQL authorization ID that is in the CURRENT SCHEMA special register.

SQLJ.REPLACE_JAR example

Suppose that you want to replace a previously installed JAR file that is named DB2INST3.BUILDPLAN with the JAR file that is in path `/u/db2inst3/apps/BUILDPLAN2/BUILDPLAN.jar`. Use a CALL statement similar to this one.

```
CALL SQLJ.REPLACE_JAR('file:/u/db2inst3/apps/BUILDPLAN2/BUILDPLAN.jar',  
  'DB2INST3.BUILDPLAN')
```

SQLJ.DB2_REPLACE_JAR stored procedure

SQLJ.DB2_REPLACE_JAR replaces an existing JAR file in the local DB2 catalog or in a remote DB2 catalog.

To replace a JAR file at a remote location, you need to execute a CONNECT statement to connect to that location before you call SQLJ.DB2_REPLACE_JAR.

SQLJ.DB2_REPLACE_JAR authorization

Privilege set: If the CALL statement is embedded in an application program, the privilege set consists of the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set consists of the privileges that are held by the authorization IDs of the process.

For calling SQLJ.DB2_REPLACE_JAR, the privilege set must include at least one of the following items:

- EXECUTE privilege on SQLJ.DB2_REPLACE_JAR
- Ownership of SQLJ.DB2_REPLACE_JAR
- SYSADM authority

The privilege set must also include the authority to replace a JAR, which consists of at least one of the following items:

- Ownership of the JAR
- ALTERIN privilege on the schema of the JAR
The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.
- SYSADM or SYSCTRL authority

SQLJ.DB2_REPLACE_JAR syntax

►►—CALL—SQLJ.DB2_REPLACE_JAR—(—*JAR-locator*,—*JAR-name*—)——►►

SQLJ.DB2_REPLACE_JAR parameters

JAR-locator

A BLOB locator input parameter that points to the JAR file that is to be replaced in the DB2 catalog.

JAR-name

A VARCHAR(257) input parameter that contains the DB2 name of the JAR, in the form *schema.JAR-id* or *JAR-id*. *JAR-name* is the name that you use when you refer to the JAR in SQL statements. If you omit *schema*, DB2 uses the SQL authorization ID that is in the CURRENT SCHEMA special register.

SQLJ.DB2_REPLACE_JAR example

Suppose that you want to replace a previously installed JAR file that is named DB2INST3.BUILDPLAN with the JAR file that is in path /u/db2inst3/apps/BUILDPLAN2/BUILDPLAN.jar. The following Java program replaces the JAR file.

```
import java.sql.*; // JDBC classes
import java.io.IOException;
import java.io.File;
import java.io.FileInputStream;
class SimpleInstallJar
{
    public static void main (String argv[])
    {
        String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021";
        String jarname = "DB2INST3.BUILDPLAN";
        String jarfile =
            "/u/db2inst3/apps/BUILDPLAN2/BUILDPLAN.jar";
        try
        {
            Class.forName ("com.ibm.db2.jcc.DB2Driver").newInstance ();
            Connection con =
                DriverManager.getConnection(url, "MYID", "MYPW");
            File aFile = new File(jarfile);
            FileInputStream inputStream = new FileInputStream(aFile);
            CallableStatement stmt;
            String sql = "Call SQLJ.DB2_REPLACE_JAR(?, ?)";
            stmt = con.prepareCall(sql);
            stmt.setBinaryStream(1, inputStream, (int)aFile.length());
            stmt.setString(2, jarname);
            boolean isrs = stmt.execute();
            stmt.close();
            System.out.println("Replacement of JAR succeeded");
            con.commit();
            con.close();
        }
        catch (Exception e)
        {
            System.out.println("Replacement of JAR failed");
            e.printStackTrace ();
        }
    }
}
```

SQLJ.REMOVE_JAR stored procedure

SQLJ.REMOVE_JAR deletes a JAR file from the local DB2 catalog or from a remote DB2 catalog.

To delete a JAR file at a remote location, you need to execute a CONNECT statement to connect to that location before you call SQLJ.REMOVE_JAR.

The JAR cannot be referenced in the EXTERNAL NAME clause of an existing routine, or in the path of an installed JAR.

SQLJ.REMOVE_JAR authorization

Privilege set: If the CALL statement is embedded in an application program, the privilege set consists of the privileges that are held by the authorization ID of the owner of the plan or package. If the statement is dynamically prepared, the privilege set consists of the privileges that are held by the authorization IDs of the process.

For calling SQLJ.REMOVE_JAR, the privilege set must include at least one of the following items:

- EXECUTE privilege on SQLJ.REMOVE_JAR
- Ownership of SQLJ.REMOVE_JAR
- SYSADM authority

The privilege set must also include the authority to remove a JAR, which consists of at least one of the following items:

- Ownership of the JAR
- DROPIN privilege on the schema of the JAR

The authorization ID that matches the schema name implicitly has the DROPIN privilege on the schema.

- SYSADM or SYSCTRL authority

SQLJ.REMOVE_JAR syntax

►►—CALL—SQLJ.REMOVE_JAR—(—JAR-name,—undeploy—)—————►►

SQLJ.REMOVE_JAR parameters

JAR-name

A VARCHAR(257) input parameter that contains the DB2 name of the JAR that is to be removed from the catalog, in the form *schema.JAR-id* or *JAR-id*.

JAR-name is the name that you use when you refer to the JAR in SQL statements. If you omit *schema*, DB2 uses the SQL authorization ID that is in the CURRENT SCHEMA special register.

undeploy

An INTEGER input parameter that indicates whether additional actions should be performed before the JAR file is removed. Additional actions are not supported, so this value is 0.

SQLJ.REMOVE_JAR example

Suppose that you want to remove a previously installed JAR file that is named DB2INST3.BUILDPLAN. Use a CALL statement similar to this one.

```
CALL SQLJ.REMOVE_JAR('DB2INST3.BUILDPLAN',0)
```

SQLJ.ALTER_JAVA_PATH stored procedure

SQLJ.ALTER_JAVA_PATH modifies the class resolution path of an installed JAR.

Privilege set: If the CALL statement is embedded in an application program, the privilege set consists of the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set consists of the privileges that are held by the authorization IDs of the process.

- EXECUTE privilege on SQLJ.ALTER_JAVA_PATH
- Ownership of SQLJ.ALTER_JAVA_PATH
- SYSADM authority

- Ownership of the JAR
- ALTERIN privilege on the schema of the JAR

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

- SYSADM or SYSCTRL authority

- Ownership of *jar2*
- USAGE privilege on *jar2*
- SYSADM authority

```

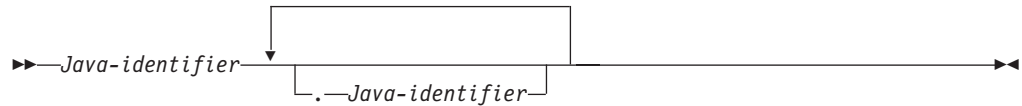
▶▶ CALL SQLJ.ALTER_JAVA_PATH (—JAR-name1, —'path' —)
                                     |
                                     | 'blanks'
                                     |
                                     | ''
                                     |
▶▶

```

Diagram illustrating a path element. A horizontal line with arrows at both ends represents a path. A vertical line segment connects the path to a box labeled *path-element*.

Diagram illustrating the structure of a JAR file name: `(-[*Java-package-name-.*class-name], -JAR-name2-)`. The diagram shows a sequence of components: a left arrow, an opening parenthesis, an asterisk, a box containing `Java-package-name-.*`, a box containing `class-name`, a closing parenthesis, a comma, another opening parenthesis, `JAR-name2`, and a right arrow. Brackets indicate that the `Java-package-name-.*` and `class-name` components are grouped together within the first set of parentheses.

172 Application Programming Guide and Reference for Java™



class-name:



SQLJ.ALTER_JAVA_PATH parameters

JAR-name1

A VARCHAR(257) input parameter that contains the DB2 name of the JAR whose path is to be altered, in the form *schema.JAR-id* or *JAR-id*. *JAR-name1* is the name that you use when you refer to the JAR in SQL statements. If you omit *schema*, DB2 uses the SQL authorization ID that is in the CURRENT SCHEMA special register.

path

A VARCHAR(2048) input parameter that specifies the class resolution path that the JVM uses when *JAR-name1* references a class that is neither contained in *JAR-name1*, found in the CLASSPATH, nor system-supplied.

During execution of the Java routine, when DB2 encounters an unresolved class reference, DB2 compares each path element in the path to the class reference. If a path element matches the class reference, DB2 searches for the class in the JAR that is specified by the path element.

- * Indicates that any class reference can be searched for in the JAR that is identified by *JAR-name2*. If an error prevents the class from being found, the search terminates, and a `java.lang.ClassNotFoundException` is thrown to report that error. If the class is not found in the JAR, the search continues with the next path element.

*Java-package-name.**

Indicates that class references for classes that are in the package named *Java-package-name* are searched for in the JAR that is identified by *JAR-name2*. If an error prevents a class from being found, the search terminates, and a `java.lang.ClassNotFoundException` is thrown to report that error. If a class is not found in the JAR, the search terminates, and a `java.lang.NoClassDefFoundError` is thrown.

If the class reference is to a class in a different package, the search continues with the next path element.

Java-package-name.class-name or class-name

Indicates that class references for classes whose fully qualified name matches *Java-package-name.class-name* or *class-name* are searched for in the JAR that is identified by *JAR-name2*. Class references for classes that are in packages within the package named *Java-package-name* are not searched for in the JAR that is identified by *JAR-name2*. If an error prevents a class from being found, the search terminates, and a `java.lang.ClassNotFoundException` is thrown to report that error. If a class is not found in the JAR, the search terminates and a `java.lang.NoClassDefFoundError` is thrown.

If the class reference is to a different class, the search continues with the next path element.

JAR-name2

Specifies the DB2 name of the JAR that is to be searched. The form of *JAR-name2* is *schema.JAR-id* or *JAR-id*. If *schema* is omitted, the JAR name is implicitly qualified with the schema name in the CURRENT SCHEMA special register. JAR *JAR-name2* must exist at the current server. *JAR-name2* must not be the same as *JAR-name1*.

SQLJ.ALTER_JAVA_PATH example

Suppose that the JAR file that is named DB2INST3.BUILDPLAN references classes that are in a previously installed JAR that is named DB2INST3.BUILDPLAN2. Those classes are in Java package buildPlan2. The following Java program calls SQLJ.ALTER_JAVA_PATH to add the classes in the buildPlan2 package to the resolution path for DB2INST3.BUILDPLAN.

```
import java.sql.*; // JDBC classes
import java.io.IOException;
import java.io.File;
import java.io.FileInputStream;
class SimpleInstallJar
{
    public static void main (String argv[])
    {
        String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021";
        String jarname = "DB2INST3.BUILDPLAN";
        String resolutionPath =
            "(buildPlan2.*,DB2INST3.BUILDPLAN2)";
        try
        {
            Class.forName ("com.ibm.db2.jcc.DB2Driver").newInstance ();
            Connection con =
                DriverManager.getConnection(url, "MYID", "MYPW");
            CallableStatement stmt;
            String sql = "Call SQLJ.ALTER_JAVA_PATH(?, ?)";
            stmt = con.prepareCall(sql);
            stmt.setString(1, jarname);
            stmt.setString(2, resolutionPath);
            boolean isrs = stmt.execute();
            stmt.close();
            System.out.println("Alteration of JAR resolution path succeeded");
            con.commit();
            con.close();
        }
        catch (Exception e)
        {
            System.out.println("Alteration of JAR resolution path failed");
            e.printStackTrace ();
        }
    }
}
```

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

There are a few basic difference between Java routines and 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 statements allowed in external functions and stored procedures (SQL Reference)

Differences between Java routines and other routines

There are a few basic difference between Java routines and routines 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 175

Related tasks

“Writing a Java stored procedure to return result sets” on page 177

 Creating an external stored procedure (Application Programming and SQL Guide)

 Writing an external user-defined function (Application Programming and SQL Guide)

Related reference

“Data types that map to database data types in Java applications” on page 191

Static and non-final variables in a Java routine

Using static and non-final variables 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.

Your 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.

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 `ResultSet`s that are closed before the stored procedure terminates.

The following code shows an example of a Java stored procedure that uses an SQLJ iterator to retrieve a result set.

```

package sl;

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 48. Java stored procedure that returns a result set

Notes to Figure 48:

- 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

“Differences between Java routines and other routines” on page 175

“Retrieving multiple result sets from a stored procedure in an SQLJ application” on page 129

Related tasks

“Retrieving multiple result sets from a stored procedure in a JDBC application” on page 48

Techniques for testing a Java routine

The most common techniques for testing a Java routine are testing the routine as a stand-alone program, using the DB2 Unified Debugger, enabling collection of DB2 debug information, and testing the routine as a stand-alone program, and writing your own debug information from your routine.

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 DB2 Unified Debugger (stored procedures only)

The DB2 Unified Debugger is available with DB2 Database for Linux, UNIX, and Windows. The DB2 Unified Debugger provides a GUI interface for debugging Java stored procedures. Information on the DB2 Unified Debugger is available in the DB2 Database for Linux, UNIX, and Windows information center, at <http://publib.boulder.ibm.com/infocenter/db2help>.

To set up a DB2 for z/OS subsystem to work with the DB2 Unified Debugger, when you set up your stored procedure environment, follow these additional steps:

1. Customize and run the DSNTIJS D program to define stored procedures that provide server support for the DB2 Unified Debugger.
DSNTIJS D 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 SYSOUT 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

“Run-time environment for Java routines” on page 156

Chapter 6. Preparing and running JDBC and SQLJ programs

DB2 for z/OS Java programs run in the z/OS UNIX System Services environment. The following topics contain information about preparing and running Java programs.

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/db2910_jdbc/` directory:

```
javac Sample01.java
```

Related tasks

“Preparing Java routines with no SQLJ clauses to run from a JAR file” on page 185

“Preparing Java routines with no SQLJ clauses and no JAR file” on page 186

Program preparation for SQLJ programs

Program preparation for SQLJ programs involves translating, compiling, customizing, and binding the program.

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.

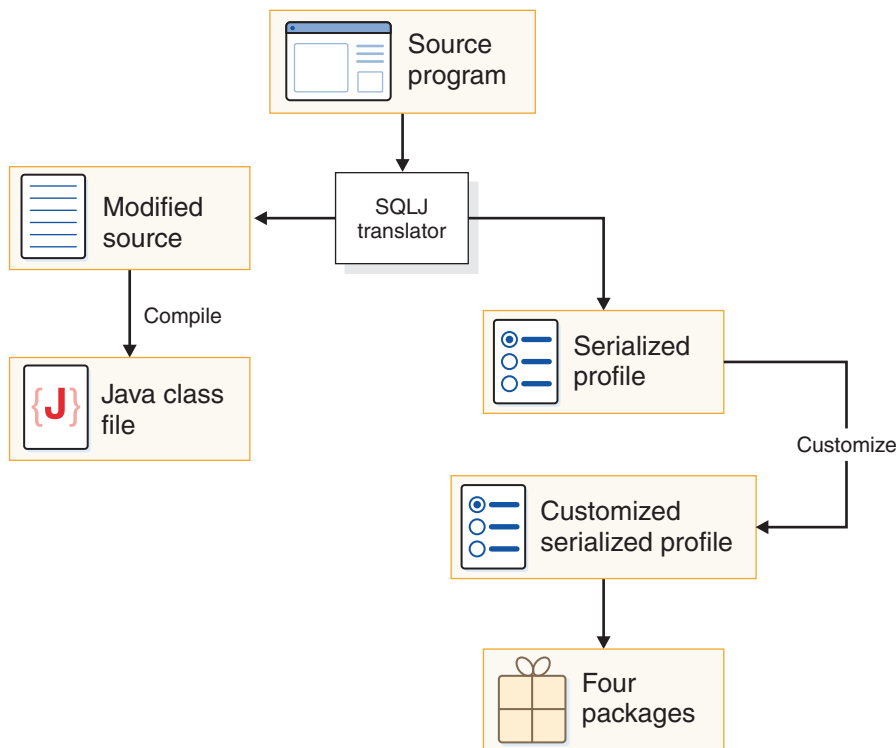


Figure 49. The SQLJ program preparation process

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 tasks

“Binding SQLJ applications to access multiple database servers”

“Preparing Java routines with SQLJ clauses to run from a JAR file” on page 186

“Preparing Java routines with SQLJ clauses and no JAR file” on page 189

Related reference

“sqlj - SQLJ translator” on page 416

“db2sqljcustomize - SQLJ profile customizer” on page 420

“db2sqljbind - SQLJ profile binder” on page 431

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.

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 db2sqljbind 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 db2sqljbind 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.

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)
BIND PACKAGE (ZOS2.MYCOL2) OWNER(USER01) QUALIFIER(WRK2) -
COPY(MYCOL.WRKSQLJ3) CURRENTDATA(NO)
BIND PACKAGE (ZOS2.MYCOL2) OWNER(USER01) QUALIFIER(WRK2) -
COPY(MYCOL.WRKSQLJ4) 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 181

Related reference

“db2sqljbind - SQLJ profile binder” on page 431

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. The program preparation process for Java routines that contain no SQLJ clauses and run from a JAR file includes compiling the program, creating the JAR file, defining the JAR file and the routine to DB2, and granting the appropriate privileges.

The steps in the process are:

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 181

“Definition of a JAR file for a Java routine to DB2” on page 165

Related tasks

“Creating JAR files for Java routines” on page 189

Preparing Java routines with no SQLJ clauses and no JAR file

The program preparation process for Java routines that contain no SQLJ clauses and do not run from a JAR file includes compiling the program, defining the routine to DB2, and granting the appropriate privileges.

The steps in the process are:

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 181

“Run-time environment for Java routines” on page 156

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 is to run them from a JAR file. The program preparation process for Java routines that contain SQLJ statements and run from a JAR file includes translating and compiling the program, customizing the serialized profiles, creating the JAR file, defining the JAR file and the routine to DB2, and granting the appropriate privileges.

The steps in the process are:

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.
8. Execute the `SQL GRANT` statement to grant the `EXECUTE` privilege on the routine to the appropriate users.

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:

```
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:

```
CREATE PROCEDURE SYSPROC.S1SAL  
  (DECIMAL(10,2) INOUT)  
  FENCED  
  MODIFIES SQL DATA  
  COLLID ADMCOLL  
  LANGUAGE JAVA  
  EXTERNAL NAME 'MYSCHEMA.S1SAL:s1.S1Sal.getSaIs'  
  WLM ENVIRONMENT WLMIJAV  
  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:

```
GRANT EXECUTE ON PROCEDURE SYSPROC.S1SAL TO PUBLIC;
```

Related concepts

“Definition of a JAR file for a Java routine to DB2” on page 165

Related tasks

“Program preparation for SQLJ programs” on page 181

“Creating JAR files for Java routines”

Preparing Java routines with SQLJ clauses and no JAR file

The program preparation process for Java routines that contain SQLJ clauses and do not run from a JAR file includes translating and compiling the program, customizing the serialized profiles, defining the routine to DB2, and granting the appropriate privileges.

The steps in the process are:

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

“Run-time environment for Java routines” on page 156

Related tasks

“Program preparation for SQLJ programs” on page 181

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.

The source code must be compiled. For Java routines with SQLJ clauses, the source code must be translated, compiled, and customized.

To create the JAR file, follow these steps:

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
```

Related tasks

“Preparing Java routines with no SQLJ clauses to run from a JAR file” on page 185

“Preparing Java routines with SQLJ clauses to run from a JAR file” on page 186

Running JDBC and SQLJ programs

You run a JDBC or SQLJ program using the java command. Before you run the program, you need to ensure that the JVM can find all of the files that it needs.

To run a JDBC or SQLJ program, follow these steps:

1. Ensure that the program files can be found.
 - For an SQLJ program, put the serialized profiles for the program in the same directory as the class files for the program.
 - Include directories for the class files that are used by the program in the CLASSPATH.
2. Run the java command on the z/OS UNIX System Services command line, with the top-level file name in the program as the argument.

To run a program that is in the EzJava class, add the directory that contains EzJava to the CLASSPATH. Then run this command:

```
java EzJava
```

Related concepts

“Environment variables for the IBM Data Server Driver for JDBC and SQLJ” on page 441

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 Database for Linux, UNIX, and Windows, DB2 for z/OS, or IBM Informix Dynamic Server (IDS) 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 26. Mappings of Java data types to database server data types for updating database tables

Java data type	Database data type
short	SMALLINT
boolean ¹ , byte ¹ , java.lang.Boolean	SMALLINT
int, java.lang.Integer	INTEGER
long, java.lang.Long	BIGINT ¹¹
float, java.lang.Float	REAL
double, java.lang.Double	DOUBLE
java.math.BigDecimal	DECIMAL(<i>p,s</i>) ²
java.math.BigDecimal	DECFLOAT(<i>n</i>) ^{3,4}
java.lang.String	CHAR(<i>n</i>) ⁵
java.lang.String	GRAPHIC(<i>m</i>) ⁶
java.lang.String	VARCHAR(<i>n</i>) ⁷
java.lang.String	VARGRAPHIC(<i>m</i>) ⁸
java.lang.String	CLOB ⁹
java.lang.String	XML ¹⁰
byte[]	CHAR(<i>n</i>) FOR BIT DATA ⁵
byte[]	VARCHAR(<i>n</i>) FOR BIT DATA ⁷
byte[]	BINARY(<i>n</i>) ^{5, 12}
byte[]	VARBINARY(<i>n</i>) ^{7, 12}
byte[]	BLOB ⁹
byte[]	ROWID
byte[]	XML ¹⁰

Table 26. Mappings of Java data types to database server data types for updating database tables (continued)

Java data type	Database data type
java.sql.Blob	BLOB
java.sql.Blob	XML ¹⁰
java.sql.Clob	CLOB
java.sql.Clob	DBCLOB ⁹
java.sql.Clob	XML ¹⁰
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.io.ByteArrayInputStream	BLOB
java.io.StringReader	CLOB
java.io.ByteArrayInputStream	CLOB
java.io.InputStream	XML ¹⁰
com.ibm.db2.jcc.DB2RowID (deprecated)	ROWID
java.sql.RowId	ROWID
com.ibm.db2.jcc.DB2Xml (deprecated)	XML ¹⁰
java.sql.SQLXML	XML ¹⁰

Notes:

1. The database server has no exact equivalent for the Java boolean or byte data types, but the best fit is SMALLINT.
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.
3. $n=16$ or $n=34$.
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 i V6R1, or later database servers. Use of DECFLOAT requires the SDK for Java Version 5 (1.5) or later.
5. $n \leq 255$.
6. $m \leq 127$.
7. $n \leq 32704$.
8. $m \leq 16352$.
9. This mapping is valid only if the database server can determine the data type of the column.
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.
11. 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 i database servers.
12. BINARY and VARBINARY are 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 DB2 for i5/OS® V5R3 and later database servers.

Data types for retrieval from table columns

The following table summarizes the mappings of DB2 or IDS 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 27. Mappings of database server data types to Java data types for retrieving data from database server tables

SQL data type	Recommended Java data type or Java object type	Other supported Java data types
SMALLINT	short	byte, int, long, float, double, java.math.BigDecimal, boolean, java.lang.String
INTEGER	int	short, byte, long, float, double, java.math.BigDecimal, boolean, java.lang.String
BIGINT ⁵	long	int, short, byte, float, double, java.math.BigDecimal, boolean, java.lang.String
DECIMAL(<i>p,s</i>) or NUMERIC(<i>p,s</i>)	java.math.BigDecimal	long, int, short, byte, float, double, boolean, java.lang.String
DECFLOAT(<i>n</i>) ^{1,2}	java.math.BigDecimal	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.lang.String
REAL	float	long, int, short, byte, double, java.math.BigDecimal, boolean, java.lang.String
DOUBLE	double	long, int, short, byte, float, java.math.BigDecimal, boolean, java.lang.String
CHAR(<i>n</i>)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
VARCHAR(<i>n</i>)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
CHAR(<i>n</i>) FOR BIT DATA	byte[]	java.lang.String, java.io.InputStream, java.io.Reader
VARCHAR(<i>n</i>) FOR BIT DATA	byte[]	java.lang.String, java.io.InputStream, java.io.Reader
BINARY(<i>n</i>) ⁶	byte[]	None
VARBINARY(<i>n</i>) ⁶	byte[]	None
GRAPHIC(<i>m</i>)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
VARGRAPHIC(<i>m</i>)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
CLOB(<i>n</i>)	java.sql.Clob	java.lang.String
BLOB(<i>n</i>)	java.sql.Blob	byte[] ³

Table 27. Mappings of database server data types to Java data types for retrieving data from database server tables (continued)

SQL data type	Recommended Java data type or Java object type	Other supported Java data types
DBCLOB(<i>m</i>)	No exact equivalent. Use <code>java.sql.Clob</code> .	
ROWID	<code>java.sql.RowId</code>	<code>byte[]</code> , <code>com.ibm.db2.jcc.DB2RowID</code> (deprecated)
XML ⁴	<code>java.sql.SQLXML</code>	<code>byte[]</code> , <code>java.lang.String</code> , <code>java.io.InputStream</code> , <code>java.io.Reader</code>
DATE	<code>java.sql.Date</code>	<code>java.sql.String</code> , <code>java.sql.Timestamp</code>
TIME	<code>java.sql.Time</code>	<code>java.sql.String</code> , <code>java.sql.Timestamp</code>
TIMESTAMP	<code>java.sql.Timestamp</code>	<code>java.sql.String</code> , <code>java.sql.Date</code> , <code>java.sql.Time</code> , <code>java.sql.Timestamp</code>

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 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 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 i database servers.
6. BINARY and VARBINARY are 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 DB2 for i5/OS V5R3 or later database servers.

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 IDS 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 28. Mappings of Java, JDBC, and SQL data types for calling stored procedures and user-defined functions

Java data type	JDBC data type	SQL data type ¹
<code>boolean</code> ²	BIT	SMALLINT
<code>byte</code> ²	TINYINT	SMALLINT
<code>short</code> , <code>java.lang.Short</code>	SMALLINT	SMALLINT
<code>int</code> , <code>java.lang.Integer</code>	INTEGER	INTEGER
<code>long</code>	BIGINT	BIGINT ⁶
<code>float</code> , <code>java.lang.Float</code>	REAL	REAL
<code>float</code> , <code>java.lang.Float</code>	FLOAT	REAL
<code>double</code> , <code>java.lang.Double</code>	DOUBLE	DOUBLE

Table 28. Mappings of Java, JDBC, and SQL data types for calling stored procedures and user-defined functions (continued)

Java data type	JDBC data type	SQL data type ¹
java.math.BigDecimal	DECIMAL	DECIMAL
java.math.BigDecimal	java.types.OTHER	DECFLOAT _n ³
java.math.BigDecimal	com.ibm.db2.jcc.DB2Types.DECFLOAT	DECFLOAT _n ³
java.lang.String	CHAR	CHAR
java.lang.String	CHAR	GRAPHIC
java.lang.String	VARCHAR	VARCHAR
java.lang.String	VARCHAR	VARGRAPHIC
java.lang.String	LONGVARCHAR	VARCHAR
java.lang.String	VARCHAR	CLOB
java.lang.String	LONGVARCHAR	CLOB
java.lang.String	CLOB	CLOB
byte[]	BINARY	CHAR FOR BIT DATA
byte[]	VARBINARY	VARCHAR FOR BIT DATA
byte[]	BINARY	BINARY ⁵
byte[]	VARBINARY	VARBINARY ⁵
byte[]	LONGVARBINARY	VARCHAR FOR BIT DATA
byte[]	VARBINARY	BLOB ⁴
byte[]	LONGVARBINARY	BLOB ⁴
java.sql.Date	DATE	DATE
java.sql.Time	TIME	TIME
java.sql.Timestamp	TIMESTAMP	TIMESTAMP
java.sql.Blob	BLOB	BLOB
java.sql.Clob	CLOB	CLOB
java.sql.Clob	CLOB	DBCLOB
java.io.ByteArrayInputStream	None	BLOB
java.io.StringReader	None	CLOB
java.io.ByteArrayInputStream	None	CLOB
com.ibm.db2.jcc.DB2RowID (deprecated)	com.ibm.db2.jcc.DB2Types.ROWID	ROWID
java.sql.RowId	java.sql.Types.ROWID	ROWID

Table 28. Mappings of Java, JDBC, and SQL data types for calling stored procedures and user-defined functions (continued)

Java data type	JDBC data type	SQL data type ¹
Notes:		
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 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. BINARY and VARBINARY are 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 DB2 for i5/OS V5R3 and later database servers.		
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 DB2 for i database servers.		

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 Database for Linux, UNIX, and Windows, 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 29. 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

SQL data type in CREATE PROCEDURE or CREATE FUNCTION ¹	Data type in Java stored procedure or user-defined function method ²
SMALLINT	short, java.lang.Integer
INTEGER	int, java.lang.Integer
BIGINT ⁴	long, java.lang.Long
REAL	float, java.lang.Float
DOUBLE	double, java.lang.Double
DECIMAL	java.math.BigDecimal
DECFLOAT ³	java.math.BigDecimal
CHAR	java.lang.String
VARCHAR	java.lang.String
CHAR FOR BIT DATA	byte[]
VARCHAR FOR BIT DATA	byte[]
BINARY ⁴	byte[]
VARBINARY ⁴	byte[]

Table 29. 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)

SQL data type in <i>CREATE PROCEDURE</i> or <i>CREATE FUNCTION</i> ¹	Data type in Java stored procedure or user-defined function method ²
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	java.sql.Blob
CLOB	java.sql.Clob
DBCLOB	java.sql.Clob
ROWID	java.sql.Types.ROWID

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 Database for Linux, UNIX, and Windows server, only the **first** data type is valid.
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 i. Use of DECFLOAT requires the SDK for Java Version 5 (1.5) or later.
4. BIGINT, BINARY, and VARBINARY are 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.

Related concepts

- “Variables in JDBC applications” on page 23
- “Java data types for retrieving or updating LOB column data in JDBC applications” on page 54
- “ROWIDs in JDBC with the IBM Data Server Driver for JDBC and SQLJ” on page 56
- “Distinct types in JDBC applications” on page 58
- “Savepoints in JDBC applications” on page 59
- “XML data in JDBC applications” on page 69
- “XML column updates in JDBC applications” on page 70
- “XML data retrieval in JDBC applications” on page 72
- “Variables in SQLJ applications” on page 103
- “LOBs in SQLJ applications with the IBM Data Server Driver for JDBC and SQLJ” on page 130
- “Java data types for retrieving or updating LOB column data in SQLJ applications” on page 131
- “ROWIDs in SQLJ with the IBM Data Server Driver for JDBC and SQLJ” on page 137
- “Distinct types in SQLJ applications” on page 139
- “Differences between Java routines and other routines” on page 175

Related reference

- “JDBC differences between the current IBM Data Server Driver for JDBC and SQLJ and earlier DB2 JDBC drivers” on page 394

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 30. Examples of updating DATE, TIME, or TIMESTAMP SQL values with Java date, time, or timestamp values that do not represent real dates or times

String input value	Target type in database	Value sent to table column, or exception
2008-13-35	DATE	2009-02-04
25:00:00	TIME	01:00:00
24:00:00	TIME	00:00:00
2008-15-36 28:63:74.0	TIMESTAMP	2009-04-06 05:04:14.0
9999-12-31 24:00:00.0	TIMESTAMP	Exception, because the adjusted value (10000-01-01 00:00:00.0) exceeds the maximum year of 9999.

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 31. Results of retrieving DATE, TIME, or TIMESTAMP SQL values that do not represent real dates or times into Java application variables

SELECT statement	Value in TIMESTAMP column TS_COL	Target type in application (getXXX method for retrieval)	Value retrieved from table column
SELECT TS_COL FROM TABLE1	2000-01-01 24:00:00.000000	java.sql.Timestamp (getTimestamp)	2000-01-02 00:00:00.000000
SELECT TS_COL FROM TABLE1	2000-01-01 24:00:00.000000	String (getString)	2000-01-02 00:00:00.000000
SELECT CHAR(TS_COL) FROM TABLE1	2000-01-01 24:00:00.000000	java.sql.Timestamp (getTimestamp)	2000-01-02 00:00:00.000000

Table 31. Results of retrieving DATE, TIME, or TIMESTAMP SQL values that do not represent real dates or times into Java application variables (continued)

SELECT statement	Value in TIMESTAMP column TS_COL	Target type in application (getXXX method for retrieval)	Value retrieved from table column
SELECT CHAR(TS_COL) FROM TABLE1	2000-01-01 24:00:00.000000	String (getString)	2000-01-01 24:00:00.000000 (no adjustment by Java)

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.

```
// 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.

```
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
```

Related reference

"JDBC differences between the current IBM Data Server Driver for JDBC and SQLJ and earlier DB2 JDBC drivers" on page 394

Properties for the IBM Data Server Driver for JDBC and SQLJ

IBM Data Server Driver for JDBC and SQLJ properties define how the connection to a particular data source should be made. Most properties can be set for a DataSource object or for a Connection object.

Methods for setting the properties

Properties can be set in one of the following ways:

- Using setXXX methods, where XXX is the unqualified property name, with the first character capitalized.

Properties are applicable to the following IBM Data Server Driver for JDBC and SQLJ-specific implementations that inherit from

com.ibm.db2.jcc.DB2BaseDataSource:

- com.ibm.db2.jcc.DB2SimpleDataSource
- com.ibm.db2.jcc.DB2ConnectionPoolDataSource
- com.ibm.db2.jcc.DB2XADataSource

- In a java.util.Properties value in the *info* parameter of a DriverManager.getConnection call.
- In a java.lang.String value in the *url* parameter of a DriverManager.getConnection call.

Some properties with an int data type have predefined constant field values. You must resolve constant field values to their integer values before you can use those values in the *url* parameter. For example, you cannot use

com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL in a *url* parameter. However, you can build a URL string that includes

com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL, and assign the URL string to a String variable. Then you can use the String variable in the *url* parameter:

```
String url =
    "jdbc:db2://sysmvs1.stl.ibm.com:5021" +
    "user=dbadm;password=dbadm;" +
    "traceLevel=" +
    (com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL) + ";";

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

Related concepts

“How to determine which type of IBM Data Server Driver for JDBC and SQLJ connectivity to use” on page 17

“LOBs in JDBC applications with the IBM Data Server Driver for JDBC and SQLJ” on page 50

“Memory use for IBM Data Server Driver for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS” on page 91

“LOBs in SQLJ applications with the IBM Data Server Driver for JDBC and SQLJ” on page 130

“IBM Data Server Driver for JDBC and SQLJ support for SSL” on page 486

Chapter 10, “Security under the IBM Data Server Driver for JDBC and SQLJ,” on page 475

“User ID and password security under the IBM Data Server Driver for JDBC and SQLJ” on page 476

“User ID-only security under the IBM Data Server Driver for JDBC and SQLJ” on page 478

“Encrypted password, user ID, or user ID and password security under the IBM Data Server Driver for JDBC and SQLJ” on page 479

“Kerberos security under the IBM Data Server Driver for JDBC and SQLJ” on page 481

“IBM Data Server Driver for JDBC and SQLJ trusted context support” on page 484

Related tasks

“Creating and deploying DataSource objects” on page 19

“Connecting to a data source using the DataSource interface” on page 15

“Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ” on page 11

 Keeping prepared statements after commit points (Application Programming and SQL Guide)

Related reference

“JDBC differences between the current IBM Data Server Driver for JDBC and SQLJ and earlier DB2 JDBC drivers” on page 394

“IBM Data Server Driver for JDBC and SQLJ extensions to JDBC” on page 322

“DB2BaseDataSource class” on page 325

“DB2ConnectionPoolDataSource class” on page 350

“DB2TraceManager class” on page 382

 SIGNON function for RRSF (Application Programming and SQL Guide)

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:

allowNextOnExhaustedResultSet

Specifies how the IBM Data Server Driver for JDBC and SQLJ handles a `ResultSet.next()` call for a forward-only cursor that is positioned after the last row of the `ResultSet`. The data type of this property is `int`.

Possible values are:

DB2BaseDataSource.YES (1)

For a `ResultSet` that is defined as `TYPE_FORWARD_ONLY`, `ResultSet.next()` returns `false` if the cursor was previously positioned after the last row of the `ResultSet`. `false` is returned, regardless of whether the cursor is open or closed.

DB2BaseDataSource.NO (2)

For a `ResultSet` that is defined as `TYPE_FORWARD_ONLY`, when `ResultSet.next()` is called, and the cursor was previously positioned after the last row of the `ResultSet`, the driver throws a `java.sql.SQLException` with error text "Invalid operation: result set is closed." This is the default.

atomicMultiRowInsert

Specifies whether batch operations that use `PreparedStatement` methods to modify a table are atomic or non-atomic. The data type of this property is `int`.

For connections to DB2 for z/OS, this property applies only to batch INSERT operations.

For connections to DB2 Database for Linux, UNIX, and Windows or IBM Informix Dynamic Server, this property applies to batch INSERT, MERGE, UPDATE or DELETE operations.

Possible values are:

DB2BaseDataSource.YES (1)

Batch operations are atomic. Insertion of all rows in the batch is considered to be a single operation. If insertion of a single row fails, the entire operation fails with a `BatchUpdateException`. Use of a batch statement that returns auto-generated keys fails with a `BatchUpdateException`.

If `atomicMultiRowInsert` is set to `DB2BaseDataSource.YES (1)`:

- Execution of statements in a heterogeneous batch is not allowed.
- If the target data source is DB2 for z/OS the following operations are not allowed:
 - Insertion of more than 32767 rows in a batch results in a `BatchUpdateException`.
 - Calling more than one of the following methods against the same parameter in different rows results in a `BatchUpdateException`:
 - `PreparedStatement.setAsciiStream`
 - `PreparedStatement.setCharacterStream`
 - `PreparedStatement.setUnicodeStream`

DB2BaseDataSource.NO (2)

Batch inserts are non-atomic. Insertion of each row is considered to be

a separate execution. Information on the success of each insert operation is provided by the `int[]` array that is returned by `Statement.executeBatch`.

DB2BaseDataSource.NOT_SET (0)

Batch inserts are non-atomic. Insertion of each row is considered to be a separate execution. Information on the success of each insert operation is provided by the `int[]` array that is returned by `Statement.executeBatch`. This is the default.

This property has no effect on SQLJ applications.

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.

clientRerouteAlternateServerName

Specifies one or more server names for client reroute. The data type of this property is String.

When `enableClientAffinitiesList=DB2BaseDataSource.YES (1)`, `clientRerouteAlternateServerName` must contain the name of the primary server as well as alternate server names. The server that is identified by `serverName` and `portNumber` is the primary server. That server name must appear at the beginning of the `clientRerouteAlternateServerName` list.

If more than one server name is specified, delimit the server names with commas (,) or spaces. The number of values that is specified for `clientRerouteAlternateServerName` must match the number of values that is specified for `clientRerouteAlternatePortNumber`.

`clientRerouteAlternateServerName` applies to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows and IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

clientRerouteAlternatePortNumber

Specifies one or more port numbers for client reroute. The data type of this property is String.

When `enableClientAffinitiesList=DB2BaseDataSource.YES (1)`, `clientRerouteAlternatePortNumber` must contain the port number for the primary server as well as port numbers for alternate servers. The server that is identified by `serverName` and `portNumber` is the primary server. That port number must appear at the beginning of the `clientRerouteAlternatePortNumber` list.

If more than one port number is specified, delimit the port numbers with commas (,) or spaces. The number of values that is specified for `clientRerouteAlternatePortNumber` must match the number of values that is specified for `clientRerouteAlternateServerName`.

`clientRerouteAlternatePortNumber` applies to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows and IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

clientRerouteServerListJNDIName

Identifies a JNDI reference to a `DB2ClientRerouteServerList` instance in a JNDI repository of reroute server information. `clientRerouteServerListJNDIName`

applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, and to connections that are established through the DataSource interface.

If the value of `clientRerouteServerListJNDIName` is not null, `clientRerouteServerListJNDIName` provides the following functions:

- Allows information about reroute servers to persist across JVMs
- Provides an alternate server location if the first connection to the data source fails

clientRerouteServerListJNDIContext

Specifies the JNDI context that is used for binding and lookup of the `DB2ClientRerouteServerList` instance. `clientRerouteServerListJNDIContext` applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, and to connections that are established through the DataSource interface.

If `clientRerouteServerListJNDIContext` is not set, the IBM Data Server Driver for JDBC and SQLJ creates an initial context using system properties or the `jndi.properties` file.

`clientRerouteServerListJNDIContext` can be set **only** by using the following method:

```
public void setClientRerouteServerListJNDIContext(javax.naming.Context registry)
```

databaseName

Specifies the name for the data source. This name is used as the *database* portion of the connection URL. The name depends on whether IBM Data Server Driver for JDBC and SQLJ type 4 connectivity or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity is used.

For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity:

- If the connection is to a DB2 for z/OS server, the `databaseName` value is the DB2 location name that is defined during installation. All characters in this value must be uppercase characters. You can determine the location name by executing the following SQL statement on the server:

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

- If the connection is to a DB2 Database for Linux, UNIX, and Windows server, the `databaseName` value is the database name that is defined during installation.
- If the connection is to an IDS 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 is the default.

DB2BaseDataSource.DECIMAL_SEPARATOR_PERIOD (1)

A period is used as the decimal separator.

DB2BaseDataSource.DECIMAL_SEPARATOR_COMMA (2)

A comma is used as the decimal separator.

When `DECIMAL_SEPARATOR_COMMA` is set, the result of `ResultSet.getString` on a decimal, floating point, or decimal floating-point value has a comma as a separator. However, if the `toString` method is executed on a value that is retrieved with a `ResultSet.getXXX` method that returns a decimal, floating point, or decimal floating-point value, the result has a decimal point as the decimal separator.

decimalStringFormat

Specifies the string format for data that is retrieved from a `DECIMAL` or `DECFLOAT` column when the SDK for Java is Version 1.5 or later. The data type of this property is `int`. Possible values are:

DB2BaseDataSource.DECIMAL_STRING_FORMAT_NOT_SET (0)

The IBM Data Server Driver for JDBC and SQLJ returns decimal values in the format that the `java.math.BigDecimal.toString` method returns them. This is the default.

For example, the value 0.0000000004 is returned as 4E-10.

DB2BaseDataSource.DECIMAL_STRING_FORMAT_TO_STRING (1)

The IBM Data Server Driver for JDBC and SQLJ returns decimal values in the format that the `java.math.BigDecimal.toString` method returns them.

For example, the value 0.0000000004 is returned as 4E-10.

DB2BaseDataSource.DECIMAL_STRING_FORMAT_TO_PLAIN_STRING (2)

The IBM Data Server Driver for JDBC and SQLJ returns decimal values in the format that the `java.math.BigDecimal.toPlainString` method returns them.

For example, the value 0.0000000004 is returned as 0.0000000004.

This property has no effect for earlier versions of the SDK for Java. For those versions, the IBM Data Server Driver for JDBC and SQLJ returns decimal values in the format that the `java.math.BigDecimal.toPlainString` 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 Dynamic Server (IDS) databases, the default depends on the type of data source. The following table shows the defaults.

Table 32. Default isolation levels for IDS databases

Type of data source	Default isolation level
ANSI-compliant database with logging	<code>java.sql.Connection.TRANSACTION_SERIALIZABLE</code>
Database without logging	<code>java.sql.Connection.TRANSACTION_READ_UNCOMMITTED</code>
Non-ANSI-compliant database with logging	<code>java.sql.Connection.TRANSACTION_READ_COMMITTED</code>

deferPrepares

Specifies whether invocation of the `Connection.prepareStatement` method results in immediate preparation of an SQL statement on the data source, or whether statement preparation is deferred until the `PreparedStatement.execute` method is executed. The data type of this property is boolean.

`deferPrepares` is supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows, and for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Possible values are:

- true** Statement preparation on the data source does not occur until the `PreparedStatement.execute` method is executed. This is the default.
- false** Statement preparation on the data source occurs when the `Connection.prepareStatement` method is executed.

Deferring prepare operations can reduce network delays. However, if you defer prepare operations, you need to ensure that input data types match table column types.

description

A description of the data source. The data type of this property is String.

downgradeHoldCursorsUnderXa

Specifies whether cursors that are defined WITH HOLD can be opened under XA connections.

`downgradeHoldCursorsUnderXa` applies to:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS servers.
- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows servers.

The default is `false`, which means that a cursor that is defined WITH HOLD cannot be opened under an XA connection. An exception is thrown when an attempt is made to open that cursor.

If `downgradeHoldCursorsUnderXa` is set to `true`, a cursor that is defined WITH HOLD can be opened under an XA connection. However, the cursor has the following restrictions:

- When the cursor is opened under an XA connection, the cursor does not have WITH HOLD behavior. The cursor is closed at XA End.

- A cursor that is open before XA Start on a local transaction is closed at XA Start.

driverType

For the DataSource interface, determines which driver to use for connections. The data type of this property is int. Valid values are 2 or 4. 2 is the default.

enableClientAffinitiesList

Specifies whether the IBM Data Server Driver for JDBC and SQLJ enables client affinities for cascaded failover support. The data type of this property is int. Possible values are:

DB2BaseDataSource.YES (1)

The IBM Data Server Driver for JDBC and SQLJ enables client affinities for cascaded failover support. This means that only servers that are specified in the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties are retried. The driver does not attempt to reconnect to any other servers.

For example, suppose that clientRerouteAlternateServerName contains the following string:

host1,host2,host3

Also suppose that clientRerouteAlternatePortNumber contains the following string:

port1,port2,port3

When client affinities are enabled, the retry order is:

1. host1:port1
2. host2:port2
3. host3:port3

DB2BaseDataSource.NO (2)

The IBM Data Server Driver for JDBC and SQLJ does not enable client affinities for cascaded failover support.

DB2BaseDataSource.NOT_SET (0)

The IBM Data Server Driver for JDBC and SQLJ does not enable client affinities for cascaded failover support. This is the default.

The effect of the maxRetriesForClientReroute and retryIntervalForClientReroute properties differs depending on whether enableClientAffinitiesList is enabled.

This property applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

enableNamedParameterMarkers

Specifies whether support for named parameter markers is enabled in the IBM Data Server Driver for JDBC and SQLJ. The data type of this property is int. Possible values are:

DB2BaseDataSource.YES (1)

Named parameter marker support is enabled in the IBM Data Server Driver for JDBC and SQLJ.

DB2BaseDataSource.NO (2)

Named parameter marker support is not enabled in the IBM Data Server Driver for JDBC and SQLJ.

The driver sends an SQL statement with named parameter markers to the target data source without modification. The success or failure of the statement depends on a number of factors, including the following ones:

- Whether the target data source supports named parameter markers
- Whether the deferPrepares property value is true 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 is the default.

enableSeamlessFailover

Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses seamless failover for client reroute. The data type of this property is `int`.

For connections to DB2 for z/OS, if `enableSysplexWLB` is set to `true`, `enableSeamlessFailover` has no effect. The IBM Data Server Driver for JDBC and SQLJ uses seamless failover regardless of the `enableSeamlessFailover` setting.

Possible values of `enableSeamlessFailover` are:

DB2BaseDataSource.YES (1)

The IBM Data Server Driver for JDBC and SQLJ uses seamless failover. This means that the driver does not throw an `SQLException` with error code -4498 after a failed connection has been successfully re-established if the following conditions are true:

- The connection was not being used for a transaction at the time the failure occurred.
- There are no outstanding global resources, such as global temporary tables or open, held cursors, or connection states that prevent a seamless failover to another server.

When seamless failover occurs, after the connection to a new data source has been established, the driver re-issues the SQL statement that was being processed when the original connection failed.

Recommendation: Set the `queryCloseImplicit` property to `DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_NO (2)` when you set `enableSeamlessFailover` to `DB2BaseDataSource.YES`, if the application uses held cursors.

DB2BaseDataSource.NO (2)

The IBM Data Server Driver for JDBC and SQLJ does not use seamless failover.

When this setting is in effect, if a server goes down, the driver tries to fail back or fail over to an alternate server. If failover or failback is successful, the driver throws an `SQLException` with error code -4498, which indicates that a connection failed but was successfully reestablished. An `SQLException` with error code -4498 informs the application that it should retry the transaction during which the connection failure occurred. If the driver cannot reestablish a connection, it throws an `SQLException` with error code -4499.

DB2BaseDataSource.NOT_SET (0)

The IBM Data Server Driver for JDBC and SQLJ does not use seamless failover. This is the default.

fetchSize

Specifies the default fetch size for `ResultSet` objects that are generated from `Statement` objects. The data type of this property is `int`.

The `fetchSize` default can be overridden by the `Statement.setFetchSize` method. The `fetchSize` property does not affect `Statement` objects that already exist when `fetchSize` is set. For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, and for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows data sources, `fetchSize` affects only scrollable cursors. For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS data sources, `fetchSize` affects scrollable cursors and forward-only cursors.

Possible values of `fetchSize` are:

0 or positive-integer

The default `fetchSize` value for newly created `Statement` objects. If the `fetchSize` property value is invalid, the IBM Data Server Driver for JDBC and SQLJ sets the default `fetchSize` value to 0.

DB2BaseDataSource.FETCHSIZE_NOT_SET (-1)

Indicates that the default `fetchSize` value for `Statement` objects is 0. This is the property default.

The `fetchSize` property differs from the `queryDataSize` property in the following ways:

- The `fetchSize` property applies to connections to all data sources that are supported by the IBM Data Server Driver for JDBC and SQLJ. The `queryDataSize` property applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS and DB2 Database for Linux, UNIX, and Windows, and to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows.
- For scrollable cursors, `fetchSize` controls the amount of data that is returned from the data source. A single response from the data source always contains up to `fetchSize` rows. The `queryDataSize` property has no impact on the total amount of data that is returned.
- For forward-only cursors, the amount of data that is returned from the data source in a single response is determined only by `queryDataSize`. The `fetchSize` property has no impact on forward-only cursors.

fullyMaterializeLobData

Indicates whether the driver retrieves LOB locators for `FETCH` operations. The data type of this property is `boolean`.

The effect of `fullyMaterializeLobData` depends on whether the data source supports progressive streaming, which is also known as dynamic data format:

- If the data source does not support progressive streaming:
If the value of `fullyMaterializeLobData` is `true`, LOB data is fully materialized within the JDBC driver when a row is fetched. If the value is `false`, LOB data is streamed. The driver uses locators internally to retrieve LOB data in chunks on an as-needed basis. It is highly recommended that you set this value to `false` when you retrieve LOBs that contain large amounts of data. The default is `true`.
- If the data source supports progressive streaming:

The JDBC driver ignores the value of `fullyMaterializeLobData` if the `progressiveStreaming` property is set to `DB2BaseDataSource.YES` or `DB2BaseDataSource.NOT_SET`.

This property has no effect on stored procedure parameters or on LOBs that are fetched using scrollable cursors. LOB stored procedure parameters are always fully materialized. LOBs that are fetched using scrollable cursors use LOB locators if progressive streaming is not in effect.

loginTimeout

The maximum time in seconds to wait for a connection to a data source. After the number of seconds that are specified by `loginTimeout` have elapsed, the driver closes the connection to the data source. The data type of this property is `int`. The default is 0. A value of 0 means that the timeout value is the default system timeout value. This property is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

logWriter

The character output stream to which all logging and trace messages for the `DataSource` object are printed. The data type of this property is `java.io.PrintWriter`. The default value is null, which means that no logging or tracing for the `DataSource` is output.

maxRetriesForClientReroute

During automatic client reroute, limit the number of retries if the primary connection to the data source fails.

The data type of this property is `int`.

The IBM Data Server Driver for JDBC and SQLJ uses the `maxRetriesForClientReroute` property only if the `retryIntervalForClientReroute` property is also set.

If the `enableClientAffinitiesList` is set to `DB2BaseDataSource.NO (2)`, an attempt to connect to the primary server and alternate servers counts as one retry. If `enableClientAffinitiesList` is set to `DB2BaseDataSource.YES (1)`, each server that is specified by the `clientRerouteAlternateServerName` and `clientRerouteAlternatePortNumber` values is retried the number of times that is specified by `maxRetriesForClientReroute`.

The default value for `maxRetriesForClientReroute` is 0 if `enableClientAffinitiesList` is `DB2BaseDataSource.NO (2)`, or 3 if `enableClientAffinitiesList` is `DB2BaseDataSource.YES (1)`.

If the value of `maxRetriesForClientReroute` is 0, client reroute processing does not occur.

password

The password to use for establishing connections. The data type of this property is `String`. When you use the `DataSource` interface to establish a connection, you can override this property value by invoking this form of the `DataSource.getConnection` method:

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

portNumber

The port number where the DRDA server is listening for requests. The data type of this property is `int`.

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 Database for Linux, UNIX, and Windows Version 9.5 and later, and IBM Informix Dynamic Server (IDS) Version 11.50 and later support progressive streaming for LOBs.

With progressive streaming, also known as dynamic data format, the data source dynamically determines the most efficient mode in which to return LOB or XML data, based on the size of the LOBs or XML objects. The value of the `streamBufferSize` parameter determines whether the data is materialized when it is returned.

The data type of `progressiveStreaming` is `int`. Valid values are `DB2BaseDataSource.YES` (1) and `DB2BaseDataSource.NO` (2). If the `progressiveStreaming` property is not specified, the `progressiveStreaming` value is `DB2BaseDataSource.NOT_SET` (0).

If the connection is to a data source that supports progressive streaming, and the value of `progressiveStreaming` is `DB2BaseDataSource.YES` or `DB2BaseDataSource.NOT_SET`, the JDBC driver uses progressive streaming to return LOBs and XML data.

If the value of `progressiveStreaming` is `DB2BaseDataSource.NO`, or the data source does not support progressive streaming, the way in which the JDBC driver returns LOB or XML data depends on the value of the `fullyMaterializeLobData` property.

queryCloseImplicit

Specifies whether cursors are closed immediately after all rows are fetched. `queryCloseImplicit` applies only to connections to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS Version 8 or later, and IBM Data Server Driver for JDBC and SQLJ type 4 connectivity or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity DB2 Database for Linux, UNIX, and Windows Version 9.7 or later. Possible values are:

DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_YES (1)

Close cursors immediately after all rows are fetched.

A value of `DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_YES` can provide better performance because this setting results in less network traffic.

DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_NO (2)

Do not close cursors immediately after all rows are fetched.

DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_COMMIT (3)

Perform these actions:

- Implicitly close the cursor after all rows are fetched.
- If the application is in autocommit mode, implicitly send a commit request to the data source for the current unit of work.

Important: When this value is set, there might be impacts on other resources, just as an explicit commit operation might impact other resources. For example, other non-held cursors are closed, LOB locators go out of scope, progressive references are reset, and scrollable cursors lose their position.

Restriction: The following restrictions apply to QUERY_CLOSE_IMPLICIT_COMMIT behavior:

- This behavior applies only to SELECT statements that are issued by the application. It does not apply to SELECT statements that are generated by the IBM Data Server Driver for JDBC and SQLJ.
- If QUERY_CLOSE_IMPLICIT_COMMIT is set, and the application is not in autocommit mode, the driver uses the default behavior (QUERY_CLOSE_IMPLICIT_NOT_SET behavior). If QUERY_CLOSE_IMPLICIT_COMMIT is the default behavior, the driver uses QUERY_CLOSE_IMPLICIT_YES behavior.
- If QUERY_CLOSE_IMPLICIT_COMMIT is set, and the data source does not support QUERY_CLOSE_IMPLICIT_COMMIT behavior, the driver uses QUERY_CLOSE_IMPLICIT_YES behavior.
- This behavior is not supported for batched statements.
- This behavior is supported on an XA Connection only when the connection is in a local transaction.

DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_NOT_SET (0)

This is the default. The following table describes the behavior for a connection to each type of data source.

Data source	Version	Data sharing environment	Behavior
DB2 for z/OS	Version 9 with APAR PK68746	Non-data sharing, or in a data sharing group but not in coexistence mode with Version 8 members	QUERY_CLOSE_IMPLICIT_COMMIT
DB2 for z/OS	Version 9 without APAR PK68746	Non-data sharing, or in a data sharing group but not in coexistence mode with Version 8 members	QUERY_CLOSE_IMPLICIT_YES
DB2 for z/OS	Version 9 with APAR PK68746	In a data sharing group in coexistence mode with Version 8 members	QUERY_CLOSE_IMPLICIT_COMMIT
DB2 for z/OS	Version 9 without APAR PK68746	In a data sharing group in coexistence mode with Version 8 members	QUERY_CLOSE_IMPLICIT_YES
DB2 for z/OS	Version 8 with or without APAR PK68746		QUERY_CLOSE_IMPLICIT_YES
DB2 Database for Linux, UNIX, and Windows	Version 9.7		QUERY_CLOSE_IMPLICIT_YES

DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_YES (1) and DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_NO (2). The default is DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_YES.

resultSetHoldability

Specifies whether cursors remain open after a commit operation. The data type of this property is int. Valid values are:

DB2BaseDataSource.HOLD_CURSORS_OVER_COMMIT (1)

Leave cursors open after a commit operation.

This setting is not valid for a connection that is part of a distributed (XA) transaction.

DB2BaseDataSource.CLOSE_CURSORS_AT_COMMIT (2)

Close cursors after a commit operation.

DB2BaseDataSource.NOT_SET (0)

This is the default value. The behavior is:

- For connections that are part of distributed (XA) transactions, cursors are closed after a commit operation.
- For connections that are not part of a distributed transaction:
 - For connections to all versions of DB2 for z/OS, DB2 Database for Linux, UNIX, and Windows, or DB2 for i servers, or to Cloudscape Version 8.1 or later servers, cursors remain open after a commit operation.
 - For connections to all versions of IBM Informix Dynamic Server, or to Cloudscape versions earlier than Version 8.1, cursors are closed after a commit operation.

retryIntervalForClientReroute

For automatic client reroute, specifies the amount of time in seconds between connection retries.

The data type of this property is int.

The IBM Data Server Driver for JDBC and SQLJ uses the `retryIntervalForClientReroute` property only if the `maxRetriesForClientReroute` property is also set.

If `maxRetriesForClientReroute` or `retryIntervalForClientReroute` is not set, the IBM Data Server Driver for JDBC and SQLJ performs retries for 10 minutes.

If the `enableClientAffinitiesList` is set to `DB2BaseDataSource.NO (2)`, an attempt to connect to the primary server and alternate servers counts as one retry. The driver waits the number of seconds that is specified by `retryIntervalForClientReroute` before retrying the connection. If `enableClientAffinitiesList` is set to `DB2BaseDataSource.YES (1)`, each server that is specified by the `clientRerouteAlternateServerName` and `clientRerouteAlternatePortNumber` values is retried after the number of seconds that is specified by `retryIntervalForClientReroute`.

The default value for `retryIntervalForClientReroute` is 0.

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 IDS.

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 IDS.

PLUGIN_SECURITY (15)

Plug-in security. This value applies to connections to DB2 Database for Linux, UNIX, and Windows only.

ENCRYPTED_USER_ONLY_SECURITY (16)

Encrypted user ID. This value does not apply to connections to IDS.

If this property is specified, the specified security mechanism is the only mechanism that is used. If the security mechanism is not supported by the connection, an exception is thrown.

The default value for securityMechanism is CLEAR_TEXT_PASSWORD_SECURITY. If the server does not support CLEAR_TEXT_PASSWORD_SECURITY but supports ENCRYPTED_USER_AND_PASSWORD_SECURITY, the IBM Data Server Driver for JDBC and SQLJ driver updates the security mechanism to ENCRYPTED_USER_AND_PASSWORD_SECURITY and attempts to connect to the server. Any other mismatch in security mechanism support between the requester and the server results in an error.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

sendDataAsIs

Specifies that the IBM Data Server Driver for JDBC and SQLJ does not convert input parameter values to the target column data types. The data type of this property is boolean. The default is false.

You should use this property only for applications that always ensure that the data types in the application match the data types in the corresponding database tables.

serverName

The host name or the TCP/IP address of the data source. The data type of this property is String.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

sslConnection

Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses an SSL socket to connect to the data source. If sslConnection is set to true, the connection uses an SSL socket. If sslConnection is set to false, the connection uses a plain socket.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

sslTrustStoreLocation

Specifies the name of the Java truststore on the client that contains the server certificate for an SSL connection.

The IBM Data Server Driver for JDBC and SQLJ uses this option only if the sslConnection property is set to true.

If `sslTrustStore` is set, and `sslConnection` is set to `true`, the IBM Data Server Driver for JDBC and SQLJ uses the `sslTrustStoreLocation` value instead of the value in the `javax.net.ssl.trustStore` Java property.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

sslTrustStorePassword

Specifies the password for the Java truststore on the client that contains the server certificate for an SSL connection.

The IBM Data Server Driver for JDBC and SQLJ uses this option only if the `sslConnection` property is set to `true`.

If `sslTrustStorePassword` is set, and `sslConnection` is set to `true`, the IBM Data Server Driver for JDBC and SQLJ uses the `sslTrustStorePassword` value instead of the value in the `javax.net.ssl.trustStorePassword` Java property.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

timestampFormat

Specifies the format in which the result of the `ResultSet.getString` or `CallableStatement.getString` method against a `TIMESTAMP` column is returned. The data type of `timestampFormat` is `int`.

Possible values of `timestampFormat` are:

Constant	Integer value	Format
<code>com.ibm.db2.jcc.DB2BaseDataSource.ISO</code>	1	<code>yyyy-mm-dd-hh:mm:ss.nnnnnn</code>
<code>com.ibm.db2.jcc.DB2BaseDataSource.JDBC</code>	5	<code>yyyy-mm-dd hh:mm:ss.nnnnnn</code>

Note:

The default is `com.ibm.db2.jcc.DB2BaseDataSource.JDBC`.

`timestampFormat` affects the format of output only.

timestampPrecisionReporting

Specifies whether trailing zeroes are truncated in the result of a `ResultSet.getString` call for a `TIMESTAMP` value. The data type of this property is `int`. Possible values are:

TIMESTAMP_JDBC_STANDARD (1)

Trailing zeroes are truncated in the result of a `ResultSet.getString` call for a `TIMESTAMP` value. This is the default.

For example:

- A `TIMESTAMP` value of `2009-07-19-10.12.00.000000` is truncated to `2009-07-19-10.12.00.0` after retrieval.
- A `TIMESTAMP` value of `2009-12-01-11.30.00.100000` is truncated to `2009-12-01-11.30.00.1` after retrieval.

TIMESTAMP_ZERO_PADDING (2)

Trailing zeroes are not truncated in the result of a `ResultSet.getString` call for a `TIMESTAMP` value.

traceDirectory

Specifies a directory into which trace information is written. The data type of

this property is String. When traceDirectory is specified, trace information for multiple connections on the same DataSource is written to multiple files.

When traceDirectory is specified, a connection is traced to a file named traceFile_origin_n.

n is the nth 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_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 (`~`) operator with a trace value to specify all except a certain trace. For example, to trace everything except DRDA flows, specify this value for `traceLevel`:
`~TRACE_DRDA_FLOWS`

user

The user ID to use for establishing connections. The data type of this property is `String`. When you use the `DataSource` interface to establish a connection, you can override this property value by invoking this form of the `DataSource.getConnection` method:

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

xaNetworkOptimization

Specifies whether XA network optimization is enabled for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity. You might need to disable XA network optimization in an environment in which an XA Start and XA End are issued from one Java process, and an XA Prepare and an XA Commit are issued from another Java process. With XA network optimization, the XA Prepare can reach the data source before the XA End, which results in an `XAER_PROTO` error. To prevent the `XAER_PROTO` error, disable XA network optimization.

The default is `true`, which means that XA network optimization is enabled. If `xaNetworkOptimization` is `false`, which means that XA network optimization is disabled, the driver closes any open cursors at XA End time.

`xaNetworkOptimization` can be set on a `DataSource` object, or in the `url` parameter in a `getConnection` call. The value of `xaNetworkOptimization` cannot be changed after a connection is obtained.

com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements

Controls an internal statement cache that is associated with a `PooledConnection`. The data type of this property is `int`. Possible values are:

positive integer

Enables the internal statement cache for a `PooledConnection`, and specifies the number of statements that the IBM Data Server Driver for JDBC and SQLJ keeps open in the cache.

0 or negative integer

Disables internal statement caching for the `PooledConnection`. 0 is the default.

`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 4 connectivity.

Common IBM Data Server Driver for JDBC and SQLJ properties for DB2 servers

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply to DB2 for z/OS and DB2 Database for Linux, UNIX, and Windows only.

Unless otherwise noted, all properties are in `com.ibm.db2.jcc.DB2BaseDataSource`.

Those properties are:

clientAccountingInformation

Specifies accounting information for the current client for the connection. This information is for client accounting purposes. This value can change during a connection. The data type of this property is String. The maximum length is 255 bytes. A Java empty string ("") is valid for this value, but a Java null value is not valid.

clientApplicationInformation

Specifies the application or transaction name of the end user's application. You can use this property to provide the identity of the client end user for accounting and monitoring purposes. This value can change during a connection. The data type of this property is String. For a DB2 for z/OS server, the maximum length is 32 bytes. For a DB2 Database for Linux, UNIX, and Windows server, the maximum length is 255 bytes. A Java empty string ("") is valid for this value, but a Java null value is not valid.

clientDebugInfo

Specifies a value for the CLIENT DEBUGINFO connection attribute, to notify the DB2 for z/OS 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.

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 `db2jccthread-name`.

This property applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

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.

currentFunctionPath

Specifies the SQL path that is used to resolve unqualified data type names and

function names in SQL statements that are in JDBC programs. The data type of this property is String. For a DB2 Database for Linux, UNIX, and Windows server, the maximum length is 254 bytes. For a DB2 for z/OS server, the maximum length is 2048 bytes. The value is a comma-separated list of schema names. Those names can be ordinary or delimited identifiers.

currentMaintainedTableTypesForOptimization

Specifies a value that identifies the types of objects that can be considered when the data source optimizes the processing of dynamic SQL queries. This register contains a keyword representing table types. The data type of this property is String.

Possible values of currentMaintainedTableTypesForOptimization are:

ALL

Indicates that all materialized query tables will be considered.

NONE

Indicates that no materialized query tables will be considered.

SYSTEM

Indicates that only system-maintained materialized query tables that are refresh deferred will be considered.

USER

Indicates that only user-maintained materialized query tables that are refresh deferred will be considered.

currentPackagePath

Specifies a comma-separated list of collections on the server. The database server searches these collections for JDBC and SQLJ packages.

The precedence rules for the currentPackagePath and currentPackageSet properties follow the precedence rules for the CURRENT PACKAGESET and CURRENT PACKAGE PATH special registers.

currentPackageSet

Specifies the collection ID to search for JDBC and SQLJ packages. The data type of this property is String. 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 i database servers, which do not support sensitive static cursors, `java.sql.ResultSet.TYPE_SCROLL_SENSITIVE` always maps to SENSITIVE DYNAMIC.

dateFormat

Specifies:

- The format in which the `String` argument of the `PreparedStatement.setString` method against a `DATE` column must be specified.
- The format in which the result of the `ResultSet.getString` or `CallableStatement.getString` method against a `DATE` column is returned.

The data type of `dateFormat` is `int`.

Possible values of `dateFormat` are:

Constant	Integer value	Format
<code>com.ibm.db2.jcc.DB2BaseDataSource.ISO</code>	1	<i>yyyy-mm-dd</i>
<code>com.ibm.db2.jcc.DB2BaseDataSource.USA</code>	2	<i>mm/dd/yyyy</i>
<code>com.ibm.db2.jcc.DB2BaseDataSource.EUR</code>	3	<i>dd.mm.yyyy</i>
<code>com.ibm.db2.jcc.DB2BaseDataSource.JIS</code>	4	<i>yyyy-mm-dd</i>

The default is `com.ibm.db2.jcc.DB2BaseDataSource.ISO`.

decimalRoundingMode

Specifies the rounding mode for decimal floating-point values on DB2 for z/OS Version 9 or later, or DB2 Database for Linux, UNIX, and Windows database 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 source.

The IBM Data Server Driver for JDBC and SQLJ uses the following values for its rounding mode:

- For DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows database servers, the rounding mode is ROUND_HALF_EVEN for decimal floating-point values.

If decimalRoundingMode is set, the decimalRoundingMode value is used to set the CURRENT DECFLOAT ROUNDING MODE special register on DB2 for z/OS database servers.

enableRowsetSupport

Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses multiple-row FETCH for forward-only cursors or scrollable cursors, if the data source supports multiple-row FETCH. The data type of this property is int.

When enableRowsetSupport is set, its value overrides the useRowsetCursor property value.

Possible values are:

DB2BaseDataSource.YES (1)

Specifies that:

- For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS, multiple-row FETCH is used for scrollable cursors and forward-only cursors, if the data source supports multiple-row FETCH.
- For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows, multiple-row fetch is used for scrollable cursors, if the data source supports multiple-row FETCH.

DB2BaseDataSource.NO (2)

Specifies that multiple-row fetch is not used.

DB2BaseDataSource.NOT_SET (0)

Specifies that if the enableRowsetSupport property is not set:

- For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS, multiple-row fetch is not used.
- For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows, the useRowsetCursor property determines whether multiple-row fetch is used for scrollable cursors.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS, multiple-row fetch is not compatible with progressive streaming. Therefore, if progressive streaming is used for a FETCH operation, multiple-row FETCH is not used.

encryptionAlgorithm

Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses 56-bit DES (weak) encryption or 256-bit AES (strong) encryption. The data type of this property is int. Possible values are:

- 1 The driver uses 56-bit DES encryption.
- 2 The driver uses 256-bit AES encryption, if the database server supports it. 256-bit AES encryption is available for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity only.

encryptionAlgorithm can be specified only if the securityMechanism value is ENCRYPTED_PASSWORD_SECURITY (7) or ENCRYPTED_USER_AND_PASSWORD_SECURITY (9).

fullyMaterializeInputStreams

Indicates whether streams are fully materialized before they are sent from the client to a data source. The data type of this property is boolean. The default is false.

If the value of fullyMaterializeInputStreams is true, the JDBC driver fully materialized the streams before sending them to the server.

gssCredential

For a data source that uses Kerberos security, specifies a delegated credential that is passed from another principal. The data type of this property is org.ietf.jgss.GSSCredential. Delegated credentials are used in multi-tier environments, such as when a client connects to WebSphere Application Server, which, in turn, connects to the data source. You obtain a value for this property from the client, by invoking the GSSContext.getDelegCred method. GSSContext is part of the IBM Java Generic Security Service (GSS) API. If you set this property, you also need to set the Mechanism and KerberosServerPrincipal properties.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

For more information on using Kerberos security with the IBM Data Server Driver for JDBC and SQLJ, see "Using Kerberos security under the IBM Data Server Driver for JDBC and SQLJ".

kerberosServerPrincipal

For a data source that uses Kerberos security, specifies the name that is used

for the data source when it is registered with the Kerberos Key Distribution Center (KDC). The data type of this property is String.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

pdqProperties

Specifies properties that control the interaction between the IBM Data Server Driver for JDBC and SQLJ and the client optimization feature of pureQuery.

The data type of this property is String.

Set the pdqProperties property **only** if you are using the client optimization feature of pureQuery. See the Integrated Data Management Information Center for information about valid values for pdqProperties.

readOnly

Specifies whether the connection is read-only. The data type of this property is boolean. The default is false.

resultSetHoldabilityForCatalogQueries

Specifies whether cursors for queries that are executed on behalf of DatabaseMetaData methods remain open after a commit operation. The data type of this property is int.

When an application executes DatabaseMetaData methods, the IBM Data Server Driver for JDBC and SQLJ executes queries against the catalog of the target data source. By default, the holdability of those cursors is the same as the holdability of application cursors. To use different holdability for catalog queries, use the resultSetHoldabilityForCatalogQueries property. Possible values are:

DB2BaseDataSource.HOLD_CURSORS_OVER_COMMIT (1)

Leave cursors for catalog queries open after a commit operation, regardless of the resultSetHoldability setting.

DB2BaseDataSource.CLOSE_CURSORS_AT_COMMIT (2)

Close cursors for catalog queries after a commit operation, regardless of the resultSetHoldability setting.

DB2BaseDataSource.NOT_SET (0)

Use the resultSetHoldability setting for catalog queries. This is the default value.

returnAlias

Specifies whether the JDBC driver returns rows for table aliases and synonyms for DatabaseMetaData methods that return table information, such as getTables. The data type of returnAlias is int. Possible values are:

- 0** Do not return rows for aliases or synonyms of tables in output from DatabaseMetaData methods that return table information.
- 1** For tables that have aliases or synonyms, return rows for aliases and synonyms of those tables, as well as rows for the tables, in output from DatabaseMetaData methods that return table information. This is the default.

streamBufferSize

Specifies the size, in bytes, of the JDBC driver buffers for chunking LOB or XML data. The JDBC driver uses the streamBufferSize value whether or not it uses progressive streaming. The data type of streamBufferSize is int. The default is 1048576.

If the JDBC driver uses progressive streaming, LOB or XML data is materialized if it fits in the buffers, and the driver does not use the `fullyMaterializeLobData` property.

DB2 for z/OS Version 9.1 and later supports progressive streaming for LOBs and XML objects. DB2 Database for Linux, UNIX, and Windows Version 9.5 and later, and IBM Informix Dynamic Server (IDS) Version 11.50 and later support progressive streaming for LOBs.

supportsAsynchronousXARollback

Specifies whether the IBM Data Server Driver for JDBC and SQLJ supports asynchronous XA rollback operations. The data type of this property is `int`. The default is `DB2BaseDataSource.NO (2)`. If the application runs against a BEA WebLogic Server application server, set `supportsAsynchronousXARollback` to `DB2BaseDataSource.YES (1)`.

sysSchema

Specifies the schema of the shadow catalog tables or views that are searched when an application invokes a `DatabaseMetaData` method. The `sysSchema` property was formerly called `cliSchema`.

timeFormat

Specifies:

- The format in which the `String` argument of the `PreparedStatement.setString` method against a `TIME` column must be specified.
- The format in which the result of the `ResultSet.getString` or `CallableStatement.getString` method against a `TIME` column is returned.

The data type of `timeFormat` is `int`.

Possible values of `timeFormat` are:

Constant	Integer value	Format
<code>com.ibm.db2.jcc.DB2BaseDataSource.ISO</code>	1	<i>hh:mm:ss</i>
<code>com.ibm.db2.jcc.DB2BaseDataSource.USA</code>	2	<i>hh:mm am</i> or <i>hh:mm pm</i>
<code>com.ibm.db2.jcc.DB2BaseDataSource.EUR</code>	3	<i>hh:mm:ss</i>
<code>com.ibm.db2.jcc.DB2BaseDataSource.JIS</code>	4	<i>hh:mm:ss</i>

The default is `com.ibm.db2.jcc.DB2BaseDataSource.ISO`.

useCachedCursor

Specifies whether the underlying cursor for `PreparedStatement` objects is cached and reused on subsequent executions of the `PreparedStatement`. The data type of `useCachedCursor` is `boolean`.

If `useCachedCursor` is set to `true`, the cursor for `PreparedStatement` objects is cached, which can improve performance. `true` is the default.

Set `useCachedCursor` to `false` if `PreparedStatement` objects access tables whose column types or lengths change between executions of those `PreparedStatement` objects.

useJDBC4ColumnNameAndLabelSemantics

Specifies how the IBM Data Server Driver for JDBC and SQLJ handles column labels in `ResultSetMetaData.getColumnName`, `ResultSetMetaData.getColumnLabel`, and `ResultSet.findColumn` method calls.

Possible values are:

DB2BaseDataSource.YES (1)

The IBM Data Server Driver for JDBC and SQLJ uses the following rules, which conform to the JDBC 4.0 specification, to determine the value that `ResultSetMetaData.getColumnNames`, `ResultSetMetaData.getColumnLabels`, and `ResultSet.findColumn` return:

- The column name that is returned by `ResultSetMetaData.getColumnNames` is its name from the database.
- The column label that is returned by `ResultSetMetaData.getColumnLabels` is the label that is specified with the SQL AS clause. If the SQL AS clause is not specified, the label is the name of the column.
- `ResultSet.findColumn` takes the label for the column, as specified with the SQL AS clause, as input. If the SQL AS clause was not specified, the label is the column name.
- The IBM Data Server Driver for JDBC and SQLJ does not use a column label that is assigned by the SQL LABEL ON statement.

These rules apply to IBM Data Server Driver for JDBC and SQLJ version 3.50 and later, for connections to the following database systems:

- DB2 for z/OS Version 8 or later
- DB2 Database for Linux, UNIX, and Windows Version 8.1 or later
- DB2 UDB for iSeries® V5R3 or later

For earlier versions of the driver or the database systems, the rules for a `useJDBC4ColumnNameAndLabelSemantics` value of `DB2BaseDataSource.NO` apply, even if `useJDBC4ColumnNameAndLabelSemantics` is set to `DB2BaseDataSource.YES`.

DB2BaseDataSource.NO (2)

The IBM Data Server Driver for JDBC and SQLJ uses the following rules to determine the values that `ResultSetMetaData.getColumnNames`, `ResultSetMetaData.getColumnLabels`, 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.getColumnNames` 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.getColumnLabels` 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.getColumnNames` 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.getColumnLabels` is the label that is specified in the LABEL ON statement.

- `ResultSet.findColumn` takes the column name as input.

These rules conform to the behavior of the IBM Data Server Driver for JDBC and SQLJ before Version 3.50.

DB2BaseDataSource.NOT_SET (0)

This is the default behavior.

For the IBM Data Server Driver for JDBC and SQLJ version 3.50 and earlier, the default behavior for `useJDBC4ColumnNameAndLabelSemantics` is the same as the behavior for `DB2BaseDataSource.NO`.

For the IBM Data Server Driver for JDBC and SQLJ version 4.0 and later:

- The default behavior for `useJDBC4ColumnNameAndLabelSemantics` is the same as the behavior for `DB2BaseDataSource.YES`, for connections to the following database systems:
 - DB2 for z/OS Version 8 or later
 - DB2 Database for Linux, UNIX, and Windows Version 8.1 or later
 - DB2 UDB for iSeries V5R3 or later
- For connections to earlier versions of these database systems, the default behavior for `useJDBC4ColumnNameAndLabelSemantics` is `DB2BaseDataSource.NO`.

com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements

Controls an internal statement cache that is associated with a `PooledConnection`. The data type of this property is `int`. Possible values are:

positive integer

Enables the internal statement cache for a `PooledConnection`, and specifies the number of statements that the IBM Data Server Driver for JDBC and SQLJ keeps open in the cache.

0 or negative integer

Disables internal statement caching for the `PooledConnection`. 0 is the default.

`maxStatements` controls the internal statement cache that is associated with a `PooledConnection` only when the `PooledConnection` object is created.

`maxStatements` has no effect on caching in an already existing `PooledConnection` object.

`maxStatements` applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, and to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Related reference

 Settings for properties for running JDBC applications in DYNAMIC mode

Common IBM Data Server Driver for JDBC and SQLJ properties for DB2 for z/OS and IDS

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply to IBM Informix Dynamic Server (IDS) and DB2 for z/OS database servers.

Properties that apply to IDS 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 connection concentrator function is available only for connections to DB2 for z/OS servers.

The data type of `enableConnectionConcentrator` is boolean. The default is `false`. However, if `enableSysplexWLB` is set to `true`, the default is `true`.

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`.

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 IDS data source, caching of dynamic statements for a connection is done if the property is not set.

`keepDynamic` is used with the `DB2Binder -keepdynamic` option. The `keepDynamic` property value that is specified must match the `-keepdynamic` value that was specified when `DB2Binder` was run.

For a DB2 for z/OS database server, dynamic statement caching can be done only if the EDM dynamic statement cache is enabled on the data source. The `CACHEDYN` subsystem parameter must be set to `DB2BaseDataSource.YES` to enable the dynamic statement cache.

maxTransportObjects

Specifies the maximum number of transport objects that can be used for all connections with the associated `DataSource` object. The IBM Data Server Driver for JDBC and SQLJ uses transport objects and a global transport objects pool to support the connection concentrator and Sysplex workload balancing. There is one transport object for each physical connection to the data source.

The data type of this property is `int`.

The `maxTransportObjects` value is ignored if the `enableConnectionConcentrator` or `enableSysplexWLB` properties are not set to enable the use of the connection concentrator or Sysplex workload balancing.

If the `maxTransportObjects` value has not been reached, and a transport object is not available in the global transport objects pool, the pool creates a new transport object. If the `maxTransportObjects` value has been reached, the application waits for the amount of time that is specified by the `db2.jcc.maxTransportObjectWaitTime` configuration property. After that amount of time has elapsed, if there is still no available transport object in the pool, the pool throws an `SQLException`.

`maxTransportObjects` does **not** override the `db2.jcc.maxTransportObjects` configuration property. `maxTransportObjects` has no effect on connections from other `DataSource` objects. If the `maxTransportObjects` value is larger than the `db2.jcc.maxTransportObjects` value, `maxTransportObjects` does not increase the `db2.jcc.maxTransportObjects` value.

The default value for `maxTransportObjects` is -1, which means that the number of transport objects for the `DataSource` is limited only by the `db2.jcc.maxTransportObjects` value for the driver.

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 `ADMFO01.NO_TABLE`:

```
ADMFO01.NO TABLE IS AN UNDEFINED NAME. SQLCODE=-204,  
SQLSTATE=42704, DRIVER=3.50.54
```

If `retrieveMessagesFromServerOnGetMessage` is set to false, a message similar to this one is returned:

```
DB2 SQL Error: SQLCODE=-204, SQLSTATE=42704, DRIVER=3.50.54
```

An alternative to setting this property to true is to use the IBM Data Server Driver for JDBC and SQLJ-only `DB2Sqlca.getMessage` method in applications. Both techniques result in a stored procedure call, which starts a unit of work.

Common IBM Data Server Driver for JDBC and SQLJ properties for IDS and DB2 Database for Linux, UNIX, and Windows

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply to IBM Informix Dynamic Server (IDS) and DB2 Database for Linux, UNIX, and Windows database servers.

Properties that apply to IDS and DB2 Database for Linux, UNIX, and Windows are:

currentLockTimeout

Specifies whether DB2 Database for Linux, UNIX, and Windows servers wait for a lock when the lock cannot be obtained immediately. The data type of this property is int. Possible values are:

integer Wait for integer seconds. *integer* is between -1 and 32767, inclusive.

LOCK_TIMEOUT_NO_WAIT

Do not wait for a lock. This is the default.

LOCK_TIMEOUT_WAIT_INDEFINITELY

Wait indefinitely for a lock.

LOCK_TIMEOUT_NOT_SET

Use the default for the data source.

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 32768 (32KB), two trips to the database server are required to retrieve the result set. However, if `queryDataSize` is set to 61440 (60 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 33. Minimum, maximum, and default values of queryDataSize

Data source	Minimum queryDataSize value	Maximum queryDataSize value	Default queryDataSize value
DB2 Database for Linux, UNIX, and Windows	4096	65535	32767
IDS	4096	10485760	32767
DB2 for i	4096	65535	32767
DB2 for z/OS	Not applicable	Not applicable	Query data size is always 32767.

IBM Data Server Driver for JDBC and SQLJ properties for DB2 Database for Linux, UNIX, and Windows

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply only to DB2 Database for Linux, UNIX, and Windows servers.

Those properties are:

connectNode

Specifies the target database partition server that an application connects to. The data type of this property is int. The value can be between 0 and 999. The default is database partition server that is defined with port 0. connectNode applies to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 Database for Linux, UNIX, and Windows servers only.

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.

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.

currentExplainSnapshot

Specifies the value for the CURRENT EXPLAIN SNAPSHOT special register. The CURRENT EXPLAIN SNAPSHOT special register enables and disables the Explain snapshot facility. The data type of this property is String. The maximum length is eight bytes. This property applies only to connections to data sources that support the CURRENT EXPLAIN SNAPSHOT special register, such as DB2 Database for Linux, UNIX, and Windows.

currentQueryOptimization

Specifies a value that controls the class of query optimization that is performed by the database manager when it binds dynamic SQL statements. The data type of this property is int. The possible values of currentQueryOptimization are:

- 0 Specifies that a minimal amount of optimization is performed to generate an access plan. This class is most suitable for simple dynamic SQL access to well-indexed tables.
- 1 Specifies that optimization roughly comparable to DB2 Database for Linux, UNIX, and Windows Version 1 is performed to generate an access plan.
- 2 Specifies a level of optimization higher than that of DB2 Database for Linux, UNIX, and Windows Version 1, but at significantly less optimization cost than levels 3 and above, especially for very complex queries.
- 3 Specifies that a moderate amount of optimization is performed to generate an access plan.
- 5 Specifies a significant amount of optimization is performed to generate an access plan. For complex dynamic SQL queries, heuristic rules are used to limit the amount of time spent selecting an access plan. Where possible, queries will use materialized query tables instead of the underlying base tables.
- 7 Specifies a significant amount of optimization is performed to generate an access plan. This value is similar to 5 but without the heuristic rules.
- 9 Specifies the maximum amount of optimization is performed to generate an access plan. This can greatly expand the number of possible access plans that are evaluated. This class should be used to determine if a better access plan can be generated for very complex and very long-running queries using large tables. Explain and performance measurements can be used to verify that a better plan has been generated.

optimizationProfile

Specifies an optimization profile that is used during SQL optimization. The data type of this property is String. The optimizationProfile value is used to set the OPTIMIZATION PROFILE special register. The default is null.

optimizationProfile applies to DB2 Database for Linux, UNIX, and Windows servers only.

optimizationProfileToFlush

Specifies the name of an optimization profile that is to be removed from the optimization profile cache. The data type of this property is String. The default is null.

plugin

The name of a client-side JDBC security plug-in. This property has the Object type and contains a new instance of the JDBC security plug-in method.

pluginName

The name of a server-side security plug-in module.

retryWithAlternativeSecurityMechanism

Specifies whether the IBM Data Server Driver for JDBC and SQLJ retries a connection with an alternative security mechanism if the security mechanism that is specified by property securityMechanism is not supported by the data source. The data type of this property is int. Possible values are:

com.ibm.db2.jcc.DB2BaseDataSource.YES (1)

Retry the connection using an alternative security mechanism. The IBM Data Server Driver for JDBC and SQLJ issues warning code +4222 and retries the connection with the most secure available security mechanism.

**com.ibm.db2.jcc.DB2BaseDataSource.NO (2) or
com.ibm.db2.jcc.DB2BaseDataSource.NOT_SET (0)**

Do not retry the connection using an alternative security mechanism.

retryWithAlternativeSecurityMechanism applies to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity connections to DB2 Database for Linux, UNIX, and Windows only.

statementConcentrator

Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses the data source's statement concentrator functionality. The statement concentrator is the ability to bypass preparation of a statement when it is the same as a statement in the dynamic statement cache, except for literal values. Statement concentrator functionality applies only to SQL statements that have literals but no parameter markers. Possible values are:

DB2BaseDataSource.STATEMENT_CONCENTRATOR_OFF (1)

The IBM Data Server Driver for JDBC and SQLJ does not use the data source's statement concentrator functionality.

DB2BaseDataSource.STATEMENT_CONCENTRATOR_WITH_LITERALS (2)

The IBM Data Server Driver for JDBC and SQLJ uses the data source's statement concentrator functionality.

DB2BaseDataSource.STATEMENT_CONCENTRATOR_NOT_SET (0)

The data source determines whether statement concentrator functionality is used. This is the default value.

For DB2 Database for Linux, UNIX, and Windows data sources that support statement concentrator functionality, the functionality is used if the STMT_CONC configuration parameter is set to ON. Otherwise, statement concentrator functionality is not used.

useTransactionRedirect

Specifies whether the DB2 system directs SQL statements to different database partitions for better performance. The data type of this property is boolean. The default is false.

This property is applicable only under the following conditions:

- The connection is to a DB2 Database for Linux, UNIX, and Windows server that uses the Database Partitioning Feature (DPF).
- The partitioning key remains constant throughout a transaction.

If `useTransactionRedirect` is true, the IBM Data Server Driver for JDBC and SQLJ sends connection requests to the DPF node that contains the target data of the first directable statement in the transaction. DB2 Database for Linux, UNIX, and Windows then directs the SQL statement to different partitions as needed.

IBM Data Server Driver for JDBC and SQLJ properties for DB2 for z/OS

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply only to DB2 for z/OS servers.

Those properties are:

accountingInterval

Specifies whether DB2 accounting records are produced at commit points or on termination of the physical connection to the data source. The data type of this property is String.

If the value of `accountingInterval` is "COMMIT", and there are no open, held cursors, DB2 writes an accounting record each time that the application commits work. If the value of `accountingInterval` is "COMMIT", and the application performs a commit operation while a held cursor is open, the accounting interval spans that commit point and ends at the next valid accounting interval end point. If the value of `accountingInterval` is not "COMMIT", accounting records are produced on termination of the physical connection to the data source.

The `accountingInterval` property sets the *accounting-interval* parameter for an underlying RRSF signon call. If the value of subsystem parameter ACCUMACC is not NO, the ACCUMACC value overrides the `accountingInterval` setting.

`accountingInterval` applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. `accountingInterval` is not applicable to connections under CICS or IMS, or for Java stored procedures.

The `accountingInterval` property overrides the `db2.jcc.accountingInterval` configuration property.

charOutputSize

Specifies the maximum number of bytes to use for INOUT or OUT stored procedure parameters that are registered as `Types.CHAR`. `charOutputSize` applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS database servers.

Because DESCRIBE information for stored procedure INOUT and OUT parameters is not available at run time, by default, the IBM Data Server Driver for JDBC and SQLJ sets the maximum length of each character INOUT or OUT parameter to 32767. For stored procedures with many `Types.CHAR` parameters, this maximum setting can result in allocation of much more storage than is necessary.

To use storage more efficiently, set `charOutputSize` to the largest expected length for any `Types.CHAR` INOUT or OUT parameter.

charOutputSize has no effect on INOUT or OUT parameters that are registered as Types.VARCHAR or Types.LONGVARCHAR. The driver uses the default length of 32767 for Types.VARCHAR and Types.LONGVARCHAR parameters.

The value that you choose for charOutputSize needs to take into account the possibility of expansion during character conversion. Because the IBM Data Server Driver for JDBC and SQLJ has no information about the server-side CCSID that is used for output parameter values, the driver requests the stored procedure output data in UTF-8 Unicode. The charOutputSize value needs to be the maximum number of bytes that are needed after the parameter value is converted to UTF-8 Unicode. UTF-8 Unicode characters can require up to three bytes. (The euro symbol is an example of a three-byte UTF-8 character.) To ensure that the value of charOutputSize is large enough, if you have no information about the output data, set charOutputSize to three times the defined length of the largest CHAR parameter.

clientUser

Specifies the current client user name for the connection. This information is for client accounting purposes. Unlike the JDBC connection user name, this value can change during a connection. For a DB2 for z/OS server, the maximum length is 16 bytes.

This property applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

clientWorkstation

Specifies the workstation name for the current client for the connection. This information is for client accounting purposes. This value can change during a connection. The data type of this property is String. For a DB2 for z/OS server, the maximum length is 18 bytes. A Java empty string ("") is valid for this value, but a Java null value is not valid.

This property applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

currentSQLID

Specifies:

- The authorization ID that is used for authorization checking on dynamically prepared CREATE, GRANT, and REVOKE SQL statements.
- The owner of a table space, database, storage group, or synonym that is created by a dynamically issued CREATE statement.
- The implicit qualifier of all table, view, alias, and index names specified in dynamic SQL statements.

currentSQLID sets the value in the CURRENT SQLID special register on a DB2 for z/OS server. If the currentSQLID property is not set, the default schema name is the value in the CURRENT SQLID special register.

jdbcCollection

Specifies the collection ID for the packages that are used by an instance of the IBM Data Server Driver for JDBC and SQLJ at run time. The data type of jdbcCollection is String. The default is NULLID.

This property is used with the DB2Binder -collection option. The DB2Binder utility must have previously bound IBM Data Server Driver for JDBC and SQLJ packages at the server using a -collection value that matches the jdbcCollection value.

The jdbcCollection setting does not determine the collection that is used for SQLJ applications. For SQLJ, the collection is determined by the -collection option of the SQLJ customizer.

jdbcCollection does not apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

maxRowsetSize

Specifies the maximum number of bytes that are used for rowset buffering for each statement, when the IBM Data Server Driver for JDBC and SQLJ uses multiple-row FETCH for cursors. The data type of this property is int. The default is 32767.

maxRowsetSize applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

pkList

Specifies a package list that is used for the underlying RRSF CREATE THREAD call when a JDBC or SQLJ connection to a data source is established. pkList applies only to 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 format of the package list is:



pkList overrides the value of the db2.jcc.pkList configuration property. If pkList, planName, and db2.jcc.pkList are not specified, the value of pkList is NULLID.*.

planName

Specifies a DB2 plan name that is used for the underlying RRSF 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**.

planName 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:

com.ibm.db2.jcc.DB2BaseDataSource.NO (2) or
com.ibm.db2.jcc.DB2BaseDataSource.NOT_SET (0)

Specifies that DatabaseMetaData methods that return information about a LONG VARCHAR or LONG VARGRAPHIC column return java.sql.Types.VARCHAR in the DATA_TYPE column and VARCHAR or VARGRAPHIC in the TYPE_NAME column of the result set. This is the default for DB2 for z/OS Version 9 or later.

com.ibm.db2.jcc.DB2BaseDataSource.YES (1)

Specifies that DatabaseMetaData methods that return information about a LONG VARCHAR or LONG VARGRAPHIC column return java.sql.Types.LONGVARCHAR in the DATA_TYPE column and LONG VARCHAR or LONG VARGRAPHIC in the TYPE_NAME column of the result set.

sendCharInputsUTF8

Specifies whether the IBM Data Server Driver for JDBC and SQLJ converts character input data to the CCSID of the DB2 for z/OS database server, or sends the data in UTF-8 encoding for conversion by the database server. sendCharInputsUTF8 applies to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS database servers only. The data type of this property is int. If this property is also set at the driver level (db2.jcc.sendCharInputsUTF8), this value overrides the driver-level value.

Possible values are:

com.ibm.db2.jcc.DB2BaseDataSource.NO (2)

Specifies that the IBM Data Server Driver for JDBC and SQLJ converts character input data to the target encoding before the data is sent to the DB2 for z/OS database server.

com.ibm.db2.jcc.DB2BaseDataSource.NO is the default.

com.ibm.db2.jcc.DB2BaseDataSource.YES (1)

Specifies that the IBM Data Server Driver for JDBC and SQLJ sends character input data to the DB2 for z/OS database server in UTF-8 encoding. The database server converts the data from UTF-8 encoding to the target CCSID.

Specify com.ibm.db2.jcc.DB2BaseDataSource.YES only if conversion to the target CCSID by the SDK for Java causes character conversion problems. The most common problem occurs when you use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to insert a Unicode line feed character (U+000A) into a table column that has CCSID 37, and then retrieve that data from a non-z/OS client. If the SDK for Java does the conversion during insertion of the character into the column, the line feed character is converted to the EBCDIC new line character X'15'. However, during retrieval, some SDKs for Java on operating systems other than z/OS convert the X'15' character to the Unicode next line character (U+0085) instead of the line feed character (U+000A). The next line character causes unexpected behavior for some XML parsers. If you set sendCharInputsUTF8 to com.ibm.db2.jcc.DB2BaseDataSource.YES, the DB2 for z/OS database server converts the U+000A character to the EBCDIC line feed character X'25' during insertion into the column, so the character is always retrieved as a line feed character.

Conversion of data to the target CCSID on the database server might cause the IBM Data Server Driver for JDBC and SQLJ to use more memory than conversion by the driver. The driver allocates memory for conversion of character data from the source encoding to the encoding of the data that it sends to the database server. The amount of space that the driver allocates for character data that is sent to a table column is based on the maximum possible length of the data. UTF-8 data can require up to three bytes for each character. Therefore, if the driver sends UTF-8 data to the database server, the driver needs to allocate three times the maximum number of characters in the input data. If the driver does the conversion, and the target CCSID is a

single-byte CCSID, the driver needs to allocate only the maximum number of characters in the input data.

sqljEnableClassLoaderSpecificProfiles

Specifies whether the IBM Data Server Driver for JDBC and SQLJ allows using and loading of SQLJ profiles with the same Java name in multiple J2EE application (.ear) files. The data type of this property is boolean. The default is `false`. `sqljEnableClassLoaderSpecificProfiles` is a `DataSource` property. This property is primarily intended for use with WebSphere Application Server.

ssid

Specifies the name of the local DB2 for z/OS subsystem to which a connection is established using IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. The data type of this property is `String`.

The `ssid` property overrides the `db2.jcc.ssid` configuration property.

`ssid` can be the subsystem name for a local subsystem or a group attachment name.

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 allows failover processing to occur if a data sharing group member fails. If the DB2 subsystem to which an application is connected fails, the connection terminates. However, when new connections use that group attachment name, DB2 for z/OS uses group attachment processing to find an active DB2 subsystem to which to connect.

`ssid` applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS.

useRowsetCursor

Specifies whether the IBM Data Server Driver for JDBC and SQLJ always uses multiple-row `FETCH` for scrollable cursors if the data source supports multiple-row fetch. The data type of this property is boolean.

This property applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, or to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS. If the `enableRowsetSupport` property is not set, the default for `useRowsetCursor` is `true`. If the `enableRowsetSupport` property is set, the `useRowsetCursor` property is not used.

Applications that use the JDBC 1 technique for performing positioned update or delete operations should set `useRowsetCursor` to `false`. Those applications do not operate properly if the IBM Data Server Driver for JDBC and SQLJ uses multiple-row `FETCH`.

Related concepts

“Failover support with IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS” on page 525

Related reference

➡ DDF/RRSAF ACCUM field (ACCUMACC subsystem parameter) (DB2 Installation and Migration)

IBM Data Server Driver for JDBC and SQLJ properties for IDS

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply only to IBM Informix Dynamic Server (IDS) databases. Those properties correspond to IDS environment variables.

Properties that are shown in uppercase characters in the following information must be specified in uppercase. For those properties, `getXXX` and `setXXX` methods are formed by prepending the uppercase property name with `get` or `set`. For example:

```
boolean dbDate = DB2BaseDataSource.getDBDATE();
```

The IDS-specific properties are:

DBANSIWARN

Specifies whether the IBM Data Server Driver for JDBC and SQLJ instructs the IDS 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 IDS 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 IDS 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` IDS property, but you cannot use the `DBANSIWARN` IBM Data Server Driver for JDBC and SQLJ property to reset the `DBANSIWARN` IDS property.

DBDATE

Specifies the end-user format of `DATE` values. The data type of this property is `String`. Possible values are in the description of the `DBDATE` environment variable in *IBM Informix Guide to SQL: Reference*.

The default value is `"Y4MD-"`.

DBPATH

Specifies a colon-separated list of values that identify the database servers that contain databases. The data type of this property is `String`. Each value can be:

- A full path name
- A relative path name
- The server name of an IDS database server
- A server name and full path name

The default is `"."`.

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 blob space of the client application. The data type of this property is String. A value of **"0"** indicates that the cache is not used.

If this property is not set, no value is sent to the server. The value for the `INFORMIXOPCACHE` environment variable is used.

INFORMIXSTACKSIZE

Specifies the stack size, in kilobytes, that the database server uses for the primary thread of a client session. The data type of this property is String.

If this property is not set, no value is sent to the server. The value for the `INFORMIXSTACKSIZE` environment variable is used.

NODEFDAC

Specifies whether the database server prevents default table privileges (`SELECT`, `INSERT`, `UPDATE`, and `DELETE`) from being granted to `PUBLIC` when a new table is created during the current session, in a database that is not ANSI compliant. The data type of this property is String. Possible values are:

- "yes"** The database server prevents default table privileges from being granted to `PUBLIC` when a new table is created during the current session, in a database that is not ANSI compliant.
- "no"** The database server does not prevent default table privileges from being granted to `PUBLIC` when a new table is created during the current session, in a database that is not ANSI compliant. This is the default.

OPTCOMPIND

Specifies the preferred method for performing a join operation on an ordered pair of tables. The data type of this property is String. Possible values are:

- "0"** The optimizer chooses a nested-loop join, where possible, over a sort-merge join or a hash join.
- "1"** When the isolation level is repeatable read, the optimizer chooses a nested-loop join, where possible, over a sort-merge join or a hash join. When the isolation level is not repeatable read, the optimizer chooses a join method based on costs.
- "2"** The optimizer chooses a join method based on costs, regardless of the transaction isolation mode.

If this property is not set, no value is sent to the server. The value for the `OPTCOMPIND` environment variable is used.

OPTOFC

Specifies whether to enable optimize-OPEN-FETCH-CLOSE functionality. The data type of this property is String. Possible values are:

- "0"** Disable optimize-OPEN-FETCH-CLOSE functionality for all threads of applications.

"1" Enable optimize-OPEN-FETCH-CLOSE functionality for all cursors in all threads of applications.

If this property is not set, no value is sent to the server. The value for the OPTOFCD environment variable is used.

PDQPRIORITY

Specifies the degree of parallelism that the database server uses. The PDQPRIORITY value affects how the database server allocates resources, including memory, processors, and disk reads. The data type of this property is String. Possible values are:

"HIGH"

When the database server allocates resources among all users, it gives as many resources as possible to queries.

"LOW" or "1"

The database server fetches values from fragmented tables in parallel.

"OFF" or "0"

Parallel processing is disabled.

If this property is not set, no value is sent to the server. The value for the PDQPRIORITY environment variable is used.

PSORT_DBTEMP

Specifies the full path name of a directory in which the database server writes temporary files that are used for a sort operation. The data type of this property is String.

If this property is not set, no value is sent to the server. The value for the PSORT_DBTEMP environment variable is used.

PSORT_NPROCS

Specifies the maximum number of threads that the database server can use to sort a query. The data type of this property is String. The maximum value of PSORT_NPROCS is "10".

If this property is not set, no value is sent to the server. The value for the PSORT_NPROCS environment variable is used.

STMT_CACHE

Specifies whether the shared-statement cache is enabled. The data type of this property is String. Possible values are:

"0" The shared-statement cache is disabled.

"1" A 512 KB shared-statement cache is enabled.

If this property is not set, no value is sent to the server. The value for the STMT_CACHE environment variable is used.

dumpPool

Specifies the types of statistics on global transport pool events that are written, in addition to summary statistics. The global transport pool is used for the connection concentrator and Sysplex workload balancing.

The data type of dumpPool is int. dumpPoolStatisticsOnSchedule and dumpPoolStatisticsOnScheduleFile must also be set for writing statistics before any statistics are written.

You can specify one or more of the following types of statistics with the db2.jcc.dumpPool property:

- DUMP_REMOVE_OBJECT (hexadecimal: X'01', decimal: 1)

- DUMP_GET_OBJECT (hexadecimal: X'02', decimal: 2)
- DUMP_WAIT_OBJECT (hexadecimal: X'04', decimal: 4)
- DUMP_SET_AVAILABLE_OBJECT (hexadecimal: X'08', decimal: 8)
- DUMP_CREATE_OBJECT (hexadecimal: X'10', decimal: 16)
- DUMP_SYSPLEX_MSG (hexadecimal: X'20', decimal: 32)
- DUMP_POOL_ERROR (hexadecimal: X'80', decimal: 128)

To trace more than one type of event, add the values for the types of events that you want to trace. For example, suppose that you want to trace DUMP_GET_OBJECT and DUMP_CREATE_OBJECT events. The numeric equivalents of these values are 2 and 16, so you specify 18 for the dumpPool value.

The default is 0, which means that only summary statistics for the global transport pool are written.

This property does not have a setXXX or a getXXX method.

dumpPoolStatisticsOnSchedule

Specifies how often, in seconds, global transport pool statistics are written to the file that is specified by dumpPoolStatisticsOnScheduleFile. The global transport object pool is used for the connection concentrator and Sysplex workload balancing.

The default is -1. -1 means that global transport pool statistics are not written.

This property does not have a setXXX or a getXXX method.

dumpPoolStatisticsOnScheduleFile

Specifies the name of the file to which global transport pool statistics are written. The global transport pool is used for the connection concentrator and Sysplex workload balancing.

If dumpPoolStatisticsOnScheduleFile is not specified, global transport pool statistics are not written.

This property does not have a setXXX or a getXXX method.

maxTransportObjectIdleTime

Specifies the amount of time in seconds that an unused transport object stays in a global transport object pool before it can be deleted from the pool. Transport objects are used for the connection concentrator and Sysplex workload balancing.

The default value for maxTransportObjectIdleTime is 60. Setting maxTransportObjectIdleTime to a value less than 0 causes unused transport objects to be deleted from the pool immediately. Doing this is **not** recommended because it can cause severe performance degradation.

This property does not have a setXXX or a getXXX method.

maxTransportObjectWaitTime

Specifies the maximum amount of time in seconds that an application waits for a transport object if the maxTransportObjects value has been reached. Transport objects are used for the connection concentrator and Sysplex workload balancing. When an application waits for longer than the maxTransportObjectWaitTime value, the global transport object pool throws an SQLException.

The default value for maxTransportObjectWaitTime is -1. Any negative value means that applications wait forever.

This property does not have a setXXX or a getXXX method.

minTransportObjects

Specifies the lower limit for the number of transport objects in a global transport object pool for the connection concentrator and Sysplex workload balancing. When a JVM is created, there are no transport objects in the pool. Transport objects are added to the pool as they are needed. After the minTransportObjects value is reached, the number of transport objects in the global transport object pool never goes below the minTransportObjects value for the lifetime of that JVM.

The default value for minTransportObjects is 0. Any value that is less than or equal to 0 means that the global transport object pool can become empty.

This property does not have a setXXX or a getXXX method.

IBM Data Server Driver for JDBC and SQLJ configuration properties

The IBM Data Server Driver for JDBC and SQLJ configuration properties have driver-wide scope.

The following table summarizes the configuration properties and corresponding Connection or DataSource properties, if they exist.

Table 34. Summary of Configuration properties and corresponding Connection and DataSource properties

Configuration property name	Connection or DataSource property name: com.ibm.db2.jcc.DB2BaseDataSource. ...	Notes
db2.jcc.accountingInterval	accountingInterval	1 on page 244, 4 on page 244
db2.jcc.allowSqljDuplicateStaticQueries		4 on page 244
db2.jcc.charOutputSize	charOutputSize	1 on page 244, 4 on page 244
db2.jcc.currentSchema	currentSchema	1 on page 244, 4 on page 244, 6 on page 244
db2.jcc.override.currentSchema	currentSchema	2 on page 244, 4 on page 244, 6 on page 244
db2.jcc.currentSQLID	currentSQLID	1 on page 244, 4 on page 244
db2.jcc.override.currentSQLID	currentSQLID	2 on page 244, 4 on page 244
db2.jcc.decimalRoundingMode	decimalRoundingMode	1 on page 244, 4 on page 244, 6 on page 244
db2.jcc.override.decimalRoundingMode	decimalRoundingMode	2 on page 244, 4 on page 244, 6 on page 244
db2.jcc.defaultSQLState		4 on page 244
db2.jcc.disableSQLJProfileCaching		4 on page 244
db2.jcc.dumpPool	dumpPool	1 on page 244, 3 on page 244, 4 on page 244, 5 on page 244
db2.jcc.dumpPoolStatisticsOnSchedule	dumpPoolStatisticsOnSchedule	1 on page 244, 3 on page 244, 4 on page 244, 5 on page 244
db2.jcc.dumpPoolStatisticsOnScheduleFile	dumpPoolStatisticsOnScheduleFile	1 on page 244, 3 on page 244, 4 on page 244, 5 on page 244
db2.jcc.jmxEnabled		4 on page 244, 5 on page 244, 6 on page 244
db2.jcc.lobOutputSize		4 on page 244

Table 34. Summary of Configuration properties and corresponding Connection and DataSource properties (continued)

Configuration property name	Connection or DataSource property name: com.ibm.db2.jcc.DB2BaseDataSource. ...	Notes
db2.jcc.maxTransportObjectIdleTime	maxTransportObjectIdleTime	1, 4, 5
db2.jcc.maxTransportObjectWaitTime	maxTransportObjectWaitTime	1, 4, 5
db2.jcc.maxTransportObjects	maxTransportObjects	1, 4, 5
db2.jcc.minTransportObjects	minTransportObjects	1, 4, 5
db2.jcc.pkList	pkList	1, 4
db2.jcc.planName	planName	1, 4
db2.jcc.progressiveStreaming	progressiveStreaming	1, 4, 5, 6
db2.jcc.override.progressiveStreaming	progressiveStreaming	2, 4, 5, 6
db2.jcc.rollbackOnShutdown		4
db2.jcc.sendCharInputsUTF8	sendCharInputsUTF8	4
db2.jcc.sqljUncustomizedWarningOrException		4, 6
db2.jcc.ssid	ssid	1, 4
db2.jcc.traceDirectory	traceDirectory	1, 4, 5, 6
db2.jcc.override.traceDirectory	traceDirectory	2, 4, 5, 6
db2.jcc.traceFile	traceFile	1, 4, 5, 6
db2.jcc.override.traceFile	traceFile	2, 4, 5, 6
db2.jcc.traceFileAppend	traceFileAppend	1, 4, 5, 6
db2.jcc.override.traceFileAppend	traceFileAppend	2, 4, 5, 6
db2.jcc.traceLevel	traceLevel	1, 4, 5, 6
db2.jcc.override.traceLevel	traceLevel	2, 4, 5, 6
db2.jcc.tracePolling		4, 5, 6
db2.jcc.tracePollingInterval		4, 5, 6
db2.jcc.t2zosTraceFile		4
db2.jcc.t2zosTraceBufferSize		4
db2.jcc.t2zosTraceWrap		4
db2.jcc.useCcsid420ShapedConverter		4

Note:

1. The Connection or DataSource property setting overrides the configuration property setting. The configuration property provides a default value for the Connection or DataSource property.
2. The configuration property setting overrides the Connection or DataSource property.
3. The corresponding Connection or DataSource property is defined only for IBM Informix Dynamic Server.
4. The configuration property applies to DB2 for z/OS.
5. The configuration property applies to IBM Informix Dynamic Server.
6. The configuration property applies to DB2 Database for Linux, UNIX, and Windows.

The meanings of the configuration properties are:

db2.jcc.accountingInterval

Specifies whether DB2 accounting records are produced at commit points or on termination of the physical connection to the data source. If the value of db2.jcc.accountingInterval is COMMIT, DB2 accounting records are produced at commit points. For example:

```
db2.jcc.accountingInterval=COMMIT
```

Otherwise, accounting records are produced on termination of the physical connection to the data source.

db2.jcc.accountingInterval applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. db2.jcc.accountingInterval is not applicable to connections under CICS or IMS, or for Java stored procedures.

You can override db2.jcc.accountingInterval by setting the accountingInterval property for a Connection or DataSource object.

This configuration property applies only to DB2 for z/OS.

db2.jcc.allowSqljDuplicateStaticQueries

Specifies whether multiple open iterators on a single SELECT statement in an SQLJ application are allowed under IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

To enable this support, set db2.jcc.allowSqljDuplicateStaticQueries to YES or true.

db2.jcc.charOutputSize

Specifies the maximum number of bytes to use for INOUT or OUT stored procedure parameters that are registered as Types.CHAR.

Because DESCRIBE information for stored procedure INOUT and OUT parameters is not available at run time, by default, the IBM Data Server Driver for JDBC and SQLJ sets the maximum length of each character INOUT or OUT parameter to 32767. For stored procedures with many Types.CHAR parameters, this maximum setting can result in allocation of much more storage than is necessary.

To use storage more efficiently, set db2.jcc.charOutputSize to the largest expected length for any Types.CHAR INOUT or OUT parameter.

db2.jcc.charOutputSize has no effect on INOUT or OUT parameters that are registered as Types.VARCHAR or Types.LONGVARCHAR. The driver uses the default length of 32767 for Types.VARCHAR and Types.LONGVARCHAR parameters.

The value that you choose for db2.jcc.charOutputSize needs to take into account the possibility of expansion during character conversion. Because the IBM Data Server Driver for JDBC and SQLJ has no information about the server-side CCSID that is used for output parameter values, the driver requests the stored procedure output data in UTF-8 Unicode. The db2.jcc.charOutputSize value needs to be the maximum number of bytes that are needed after the parameter value is converted to UTF-8 Unicode. UTF-8 Unicode characters can require up to three bytes. (The euro symbol is an example of a three-byte UTF-8 character.) To ensure that the value of db2.jcc.charOutputSize is large enough, if you have no information about the output data, set db2.jcc.charOutputSize to three times the defined length of the largest CHAR parameter.

This configuration property applies only to DB2 for z/OS.

db2.jcc.currentSchema or db2.jcc.override.currentSchema

Specifies the default schema name that is used to qualify unqualified database objects in dynamically prepared SQL statements. This value of this property sets the value in the CURRENT SCHEMA special register on the database server. The schema name is case-sensitive, and must be specified in uppercase characters.

This configuration property applies only to DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows.

db2.jcc.currentSQLID or db2.jcc.override.currentSQLID

Specifies:

- The authorization ID that is used for authorization checking on dynamically prepared CREATE, GRANT, and REVOKE SQL statements.
- The owner of a table space, database, storage group, or synonym that is created by a dynamically issued CREATE statement.
- The implicit qualifier of all table, view, alias, and index names specified in dynamic SQL statements.

currentSQLID sets the value in the CURRENT SQLID special register on a DB2 for z/OS server. If the currentSQLID property is not set, the default schema name is the value in the CURRENT SQLID special register.

This configuration property applies only to DB2 for z/OS.

db2.jcc.decimalRoundingMode or db2.jcc.override.decimalRoundingMode

Specifies the rounding mode for decimal or decimal floating-point values on DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows database servers, and for decimal values on all other data sources that support the decimal data type.

Possible values are:

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_DOWN (1)

Rounds the value towards 0 (truncation). The discarded digits are ignored.

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_CEILING (2)

Rounds the value towards positive infinity. If all of the discarded digits are zero or if the sign is negative the result is unchanged other than the removal of the discarded digits. Otherwise, the result coefficient is incremented by 1.

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_HALF_EVEN (3)

Rounds the value to the nearest value; if the values are equidistant, rounds the value so that the final digit is even. If the discarded digits represents greater than half (0.5) of the value of one in the next left position then the result coefficient is incremented by 1. If they represent less than half, then the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise the result coefficient is unaltered if its rightmost digit is even, or is incremented by 1 if its rightmost digit is odd (to make an even digit).

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_HALF_UP (4)

Rounds the value to the nearest value; if the values are equidistant, rounds the value away from zero. If the discarded digits represent greater than or equal to half (0.5) of the value of one in the next left position then the result coefficient is incremented by 1. Otherwise the discarded digits are ignored.

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_FLOOR (6)

Rounds the value towards negative infinity. If all of the discarded digits are zero or if the sign is positive the result is unchanged other than the removal of discarded digits. Otherwise, the sign is negative and the result coefficient is incremented by 1.

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_UNSET (-2147483647)

No rounding mode was explicitly set. The IBM Data Server Driver for JDBC and SQLJ does not use the decimalRoundingMode to set the rounding mode on the data source.

The IBM Data Server Driver for JDBC and SQLJ uses the following values for its rounding mode:

- If the data source is DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows, the rounding mode is ROUND_HALF_EVEN for decimal or decimal floating-point values.
- For any other data source, the rounding mode is ROUND_DOWN for decimal values.

This configuration property applies only to DB2 for z/OS Version 9 or later, or DB2 Database for Linux, UNIX, and Windows Version 9.1 or later.

db2.jcc.defaultSQLState

Specifies the SQLSTATE value that the IBM Data Server Driver for JDBC and SQLJ returns to the client for SQLException or SQLWarning objects that have null SQLSTATE values. This configuration property can be specified in the following ways:

db2.jcc.defaultSQLState

If db2.jcc.defaultSQLState is specified with no value, the IBM Data Server Driver for JDBC and SQLJ returns 'FFFFF'.

db2.jcc.defaultSQLState=xxxxx

xxxxx is the value that the IBM Data Server Driver for JDBC and SQLJ returns when the SQLSTATE value is null. If xxxxx is longer than five bytes, the driver truncates the value to five bytes. If xxxxx is shorter than five bytes, the driver pads xxxxx on the right with blanks.

If db2.jcc.defaultSQLState is not specified, the IBM Data Server Driver for JDBC and SQLJ returns a null SQLSTATE value.

This configuration property applies only to DB2 for z/OS.

db2.jcc.disableSQLJProfileCaching

Specifies whether serialized profiles are cached when the JVM under which their application is running is reset. db2.jcc.disableSQLJProfileCaching applies only to applications that run in a resettable JVM (applications that run in the CICS, IMS, or Java stored procedure environment), and use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. Possible values are:

YES SQLJ serialized profiles are not cached every time the JVM is reset, so that new versions of the serialized profiles are loaded when the JVM is reset. Use this option when an application is under development, and new versions of the application and its serialized profiles are produced frequently.

NO SQLJ serialized profiles are cached when the JVM is reset. NO is the default.

This configuration property applies only to DB2 for z/OS.

db2.jcc.dumpPool

Specifies the types of statistics on global transport pool events that are written, in addition to summary statistics. The global transport pool is used for the connection concentrator and Sysplex workload balancing.

db2.jcc.dumpPoolStatisticsOnSchedule and db2.jcc.dumpPoolStatisticsOnScheduleFile must also be set for writing statistics before any statistics are written.

You can specify one or more of the following types of statistics with the db2.jcc.dumpPool property:

- DUMP_REMOVE_OBJECT (hexadecimal: X'01', decimal: 1)
- DUMP_GET_OBJECT (hexadecimal: X'02', decimal: 2)
- DUMP_WAIT_OBJECT (hexadecimal: X'04', decimal: 4)
- DUMP_SET_AVAILABLE_OBJECT (hexadecimal: X'08', decimal: 8)
- DUMP_CREATE_OBJECT (hexadecimal: X'10', decimal: 16)
- DUMP_SYSPLEX_MSG (hexadecimal: X'20', decimal: 32)
- DUMP_POOL_ERROR (hexadecimal: X'80', decimal: 128)

To trace more than one type of event, add the values for the types of events that you want to trace. For example, suppose that you want to trace DUMP_GET_OBJECT and DUMP_CREATE_OBJECT events. The numeric equivalents of these values are 2 and 16, so you specify 18 for the db2.jcc.dumpPool value.

The default is 0, which means that only summary statistics for the global transport pool are written.

This configuration property applies only to DB2 for z/OS or IBM Informix Dynamic Server.

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 Dynamic Server.

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 Dynamic Server.

db2.jcc.jmxEnabled

Specifies whether the Java Management Extensions (JMX) is enabled for the IBM Data Server Driver for JDBC and SQLJ instance. JMX must be enabled before applications can use the remote trace controller.

Possible values are:

true or yes

Indicates that JMX is enabled.

Any other value

Indicates that JMX is disabled. This is the default.

db2.jcc.lobOutputSize

Specifies the number of bytes of storage that the IBM Data Server Driver for JDBC and SQLJ needs to allocate for output LOB values when the driver cannot determine the size of those LOBs. This situation occurs for LOB stored procedure output parameters. db2.jcc.lobOutputSize applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

The default value for `db2.jcc.lobOutputSize` is 1048576. For systems with storage limitations and smaller LOBs, set the `db2.jcc.lobOutputSize` value to a lower number.

For example, if you know that the output LOB size is at most 64000, set `db2.jcc.lobOutputSize` to 64000.

This configuration property applies only to DB2 for z/OS.

db2.jcc.maxTransportObjectIdleTime

Specifies the amount of time in seconds that an unused transport object stays in a global transport object pool before it can be deleted from the pool. Transport objects are used for the connection concentrator and Sysplex workload balancing.

The default value for `db2.jcc.maxTransportObjectIdleTime` is 60. Setting `db2.jcc.maxTransportObjectIdleTime` to a value less than 0 causes unused transport objects to be deleted from the pool immediately. Doing this is **not** recommended because it can cause severe performance degradation.

This configuration property applies only to DB2 for z/OS or IBM Informix Dynamic Server.

db2.jcc.maxTransportObjects

Specifies the upper limit for the number of transport objects in a global transport object pool for the connection concentrator and Sysplex workload balancing. When the number of transport objects in the pool reaches the `db2.jcc.maxTransportObjects` value, transport objects that have not been used for longer than the `db2.jcc.maxTransportObjectIdleTime` value are deleted from the pool.

The default value for `db2.jcc.maxTransportObjects` is -1. Any value that is less than or equal to 0 means that there is no limit to the number of transport objects in the global transport object pool.

This configuration property applies only to DB2 for z/OS or IBM Informix Dynamic Server.

db2.jcc.maxTransportObjectWaitTime

Specifies the maximum amount of time in seconds that an application waits for a transport object if the `db2.jcc.maxTransportObjects` value has been reached. Transport objects are used for the connection concentrator and Sysplex workload balancing. When an application waits for longer than the `db2.jcc.maxTransportObjectWaitTime` value, the global transport object pool throws an `SQLException`.

The default value for `db2.jcc.maxTransportObjectWaitTime` is -1. Any negative value means that applications wait forever.

This configuration property applies only to DB2 for z/OS or IBM Informix Dynamic Server.

db2.jcc.minTransportObjects

Specifies the lower limit for the number of transport objects in a global transport object pool for the connection concentrator and Sysplex workload balancing. When a JVM is created, there are no transport objects in the pool. Transport objects are added to the pool as they are needed. After the `db2.jcc.minTransportObjects` value is reached, the number of transport objects in the global transport object pool never goes below the `db2.jcc.minTransportObjects` value for the lifetime of that JVM.

The default value for `db2.jcc.minTransportObjects` is 0. Any value that is less than or equal to 0 means that the global transport object pool can become empty.

This configuration property applies only to DB2 for z/OS or IBM Informix Dynamic Server.

db2.jcc.pkList

Specifies a package list that is used for the underlying RRSF CREATE THREAD call when a JDBC or SQLJ connection to a data source is established. Specify this property if you do not bind plans for your SQLJ programs or for the JDBC driver. If you specify this property, **do not specify `db2.jcc.planName`**.

`db2.jcc.pkList` applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. `db2.jcc.pkList` does not apply to applications that run under CICS or IMS, or to Java stored procedures. The JDBC driver ignores the `db2.jcc.pkList` setting in those cases.

Recommendation: Use `db2.jcc.pkList` instead of `db2.jcc.planName`.

The format of the package list is:



The default value of `db2.jcc.pkList` is `NULLID.*`.

If you specify the `-collection` parameter when you run `com.ibm.db2.jcc.DB2Binder`, the collection ID that you specify for IBM Data Server Driver for JDBC and SQLJ packages when you run `com.ibm.db2.jcc.DB2Binder` must also be in the package list for the `db2.jcc.pkList` property.

You can override `db2.jcc.pkList` by setting the `pkList` property for a `Connection` or `DataSource` object.

The following example specifies a package list for a IBM Data Server Driver for JDBC and SQLJ instance whose packages are in collection `JDBCCID`. SQLJ applications that are prepared under this driver instance are bound into collections `SQLJCID1`, `SQLJCID2`, or `SQLJCID3`.

```
db2.jcc.pkList=JDBCCID.*,SQLJCID1.*,SQLJCID2.*,SQLJCID3.*
```

This configuration property applies only to DB2 for z/OS.

db2.jcc.planName

Specifies a DB2 for z/OS plan name that is used for the underlying RRSF CREATE THREAD call when a JDBC or SQLJ connection to a data source is established. Specify this property if you bind plans for your SQLJ programs and for the JDBC driver packages. If you specify this property, **do not specify `db2.jcc.pkList`**.

`db2.jcc.planName` applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. `db2.jcc.planName` does not apply to applications that run under CICS or IMS, or to Java stored procedures. The JDBC driver ignores the `db2.jcc.planName` setting in those cases.

If you do not specify this property or the `db2.jcc.pkList` property, the IBM Data Server Driver for JDBC and SQLJ uses the `db2.jcc.pkList` default value of `NULLID.*`.

If you specify `db2.jcc.planName`, you need to bind the packages that you produce when you run `com.ibm.db2.jcc.DB2Binder` into a plan whose name is the value of this property. You also need to bind all SQLJ packages into a plan whose name is the value of this property.

You can override `db2.jcc.planName` by setting the `planName` property for a `Connection` or `DataSource` object.

The following example specifies a plan name of MYPLAN for the IBM Data Server Driver for JDBC and SQLJ JDBC packages and SQLJ packages.

```
db2.jcc.planName=MYPLAN
```

This configuration property applies only to DB2 for z/OS.

db2.jcc.progressiveStreaming or db2.jcc.override.progressiveStreaming

Specifies whether the JDBC driver uses progressive streaming when progressive streaming is supported on the data source.

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

Valid values are:

- 1 Use progressive streaming, if the data source supports it.
- 2 Do not use progressive streaming.

db2.jcc.rollbackOnShutdown

Specifies whether DB2 for z/OS forces a rollback operation and disables further operations on JDBC connections that are in a unit of work during processing of JVM shutdown hooks.

`db2.jcc.rollbackOnShutdown` applies to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity only.

`db2.jcc.rollbackOnShutdown` does not apply to the CICS, IMS, stored procedure, or WebSphere Application Server environments.

Possible values are:

yes or true

The IBM Data Server Driver for JDBC and SQLJ directs DB2 for z/OS to force a rollback operation and disables further operations on JDBC connections that are in a unit of work during processing of JVM shutdown hooks.

Any other value

The IBM Data Server Driver for JDBC and SQLJ takes no action with respect to rollback processing during processing of JVM shutdown hooks. This is the default.

This configuration property applies only to DB2 for z/OS.

db2.jcc.sendCharInputsUTF8

Specifies whether the IBM Data Server Driver for JDBC and SQLJ converts character input data to the CCSID of the DB2 for z/OS database server, or sends the data in UTF-8 encoding for conversion by the database server.

`db2.jcc.sendCharInputsUTF8` applies to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS database servers only. If this property is also set at the connection level, the connection-level setting overrides this value.

Possible values are:

no, false, or 2

Specifies that the IBM Data Server Driver for JDBC and SQLJ converts character input data to the target encoding before the data is sent to the DB2 for z/OS database server. This is the default.

yes, true, or 1

Specifies that the IBM Data Server Driver for JDBC and SQLJ sends character input data to the DB2 for z/OS database server in UTF-8 encoding. The data source converts the data from UTF-8 encoding to the target CCSID.

Specify yes, true, or 1 only if conversion to the target CCSID by the SDK for Java causes character conversion problems. The most common problem occurs when you use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to insert a Unicode line feed character (U+000A) into a table column that has CCSID 37, and then retrieve that data from a non-z/OS client. If the SDK for Java does the conversion during insertion of the character into the column, the line feed character is converted to the EBCDIC new line character X'15'. However, during retrieval, some SDKs for Java on operating systems other than z/OS convert the X'15' character to the Unicode next line character (U+0085) instead of the line feed character (U+000A). The next line character causes unexpected behavior for some XML parsers. If you set `db2.jcc.sendCharInputsUTF8` to yes, the DB2 for z/OS database server converts the U+000A character to the EBCDIC line feed character X'25' during insertion into the column, so the character is always retrieved as a line feed character.

Conversion of data to the target CCSID on the data source might cause the IBM Data Server Driver for JDBC and SQLJ to use more memory than conversion by the driver. The driver allocates memory for conversion of character data from the source encoding to the encoding of the data that it sends to the data source. The amount of space that the driver allocates for character data that is sent to a table column is based on the maximum possible length of the data. UTF-8 data can require up to three bytes for each character. Therefore, if the driver sends UTF-8 data to the data source, the driver needs to allocate three times the maximum number of characters in the input data. If the driver does the conversion, and the target CCSID is a single-byte CCSID, the driver needs to allocate only the maximum number of characters in the input data.

For example, any of the following settings for `db2.jcc.sendCharInputsUTF8` causes the IBM Data Server Driver for JDBC and SQLJ to convert input character strings to UTF-8, rather than the target encoding, before sending the data to the data source:

```
db2.jcc.sendCharInputsUTF8=yes
db2.jcc.sendCharInputsUTF8=true
db2.jcc.sendCharInputsUTF8=1
```

This configuration property applies only to DB2 for z/OS.

db2.jcc.sqljUncustomizedWarningOrException

Specifies the action that the IBM Data Server Driver for JDBC and SQLJ takes when an uncustomized SQLJ application runs.

`db2.jcc.sqljUncustomizedWarningOrException` can have the following values:

- 0** The IBM Data Server Driver for JDBC and SQLJ does not throw a Warning or Exception when an uncustomized SQLJ application is run. This is the default.

- 1 The IBM Data Server Driver for JDBC and SQLJ throws a Warning when an uncustomized SQLJ application is run.
- 2 The IBM Data Server Driver for JDBC and SQLJ throws an Exception when an uncustomized SQLJ application is run.

This configuration property applies only to DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows.

db2.jcc.traceDirectory or db2.jcc.override.traceDirectory

Enables the IBM Data Server Driver for JDBC and SQLJ trace for Java driver code, and specifies a directory into which trace information is written. 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.

xads The log writer for a `DB2XADataSource` object.

The `db2.jcc.override.traceDirectory` property overrides the `traceDirectory` property for a `Connection` or `DataSource` object.

For example, specifying the following setting for `db2.jcc.override.traceDirectory` enables tracing of the IBM Data Server Driver for JDBC and SQLJ Java code to files in a directory named `/SYSTEM/tmp`:

```
db2.jcc.override.traceDirectory=/SYSTEM/tmp
```

You should set the trace properties under the direction of IBM Software Support.

db2.jcc.traceLevel or db2.jcc.override.traceLevel

Specifies what to trace.

The `db2.jcc.override.traceLevel` property overrides the `traceLevel` property for a `Connection` or `DataSource` object.

You specify one or more trace levels by specifying a decimal value. The trace levels are the same as the trace levels that are defined for the `traceLevel` property on a `Connection` or `DataSource` object.

To specify more than one trace level, do an OR (|) operation on the values, and specify the result in decimal in the `db2.jcc.traceLevel` or `db2.jcc.override.traceLevel` specification.

For example, suppose that you want to specify `TRACE_DRDA_FLOWS` and `TRACE_CONNECTIONS` for `db2.jcc.override.traceLevel`.

TRACE_DRDA_FLOWS has a hexadecimal value of X'40'.
TRACE_CONNECTION_CALLS has a hexadecimal value of X'01'. To specify both traces, do a bitwise OR operation on the two values, which results in X'41'. The decimal equivalent is 65, so you specify:

```
db2.jcc.override.traceLevel=65
```

db2.jcc.ssid

Specifies the DB2 for z/OS subsystem to which applications make connections with IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

The db2.jcc.ssid value can be the name of the local DB2 subsystem or a group attachment name.

For example:

```
db2.jcc.ssid=DB2A
```

The ssid Connection and DataSource property overrides db2.jcc.ssid.

If you specify a group attachment name, and the DB2 subsystem to which an application is connected fails, the connection terminates. However, when new connections use that group attachment name, DB2 for z/OS uses group attachment processing to find an active DB2 subsystem to which to connect.

If you do not specify the db2.jcc.ssid property, the IBM Data Server Driver for JDBC and SQLJ uses the SSID value from the DSNHDECP data-only load module. When you install DB2 for z/OS, a DSNHDECP module is created in the *prefix*.SDSNEXIT data set and the *prefix*.SDSNLOAD data set. Other DSNHDECP load modules might be created in other data sets for selected applications.

The IBM Data Server Driver for JDBC and SQLJ must load a DSNHDECP module before it can read the SSID value. z/OS searches data sets in the following places, and in the following order, for the DSNHDECP 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 DSNHDECP module, z/OS finds the data set that contains the correct copy for the IBM Data Server Driver for JDBC and SQLJ first.

This configuration property applies only to DB2 for z/OS.

db2.jcc.traceFile or db2.jcc.override.traceFile

Enables the IBM Data Server Driver for JDBC and SQLJ trace for Java driver code, and specifies the name on which the trace file names are based. 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 should set the trace properties under the direction of IBM Software Support.

db2.jcc.traceFileAppend or db2.jcc.override.traceFileAppend

Specifies whether to append to or overwrite the file that is specified by the db2.jcc.override.traceFile property. 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 should set the trace properties under the direction of IBM Software Support.

db2.jcc.tracePolling

Indicates whether the IBM Data Server Driver for JDBC and SQLJ polls the global configuration file for changes in trace directives and modifies the trace behavior to match the new trace directives. The driver modifies the trace behavior at the beginning of the next polling interval after the configuration properties file is changed. Possible values are true or false. False is the default. For trace polling to be enabled, the db2.jcc.tracePolling property must be enabled *before* the driver is loaded and initialized.

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 kilobytes, of a trace buffer in virtual storage that is used for tracing the processing that is done by the C/C++ native driver code. This value is also the maximum amount of C/C++ native driver trace information that can be collected.

Specify an integer between 64 (64 KB) and 4096 (4096 KB). The default is 256 (256 KB).

The JDBC driver determines the trace buffer size as shown in the following table:

Specified value (<i>n</i>)	Trace buffer size (KB)
<64	64
64≤ <i>n</i> <128	64
128≤ <i>n</i> <256	128
256≤ <i>n</i> <512	256
512≤ <i>n</i> <1024	512
1024≤ <i>n</i> <2048	1024
2048≤ <i>n</i> <4096	2048
<i>n</i> ≥4096	4096

db2.jcc.t2zosTraceBufferSize is used only if the db2.jcc.t2zosTraceFile property is set.

Recommendation: To avoid a performance impact, specify a value of 1024 or less.

For example, to set a trace buffer size of 1024 KB, use this setting:

```
db2.jcc.t2zosTraceBufferSize=1024
```

You should set the trace properties under the direction of IBM Software Support.

This configuration property applies only to DB2 for z/OS.

db2.jcc.t2zosTraceWrap

Enables or disables wrapping of the SQLJ trace. db2.jcc.t2zosTraceWrap can have one of the following values:

- 1** Wrap the trace
- 0** Do not wrap the trace

The default is 1. This parameter is optional. For example:

```
DB2SQLJ_TRACE_WRAP=0
```

You should set db2.jcc.t2zosTraceWrap only under the direction of IBM Software Support.

This configuration property applies only to DB2 for z/OS.

db2.jcc.useCcsid420ShapedConverter

Specifies whether Arabic character data that is in EBCDIC CCSID 420 maps to Cp420S encoding.

db2.jcc.useCcsid420ShapedConverter applies only to connections to DB2 for z/OS database servers.

If the value of db2.jcc.useCcsid420ShapedConverter is `true`, CCSID 420 maps to Cp420S encoding. If the value of db2.jcc.useCcsid420ShapedConverter is `false`, CCSID 420 maps to Cp420 encoding. `false` is the default.

This configuration property applies only to DB2 for z/OS.

Related concepts

“Customization of IBM Data Server Driver for JDBC and SQLJ configuration properties” on page 442

Related tasks

“Installing the z/OS Application Connectivity to DB2 for z/OS feature” on page 461

Chapter 9, “Migrating from the JDBC/SQLJ Driver for OS/390 and z/OS to the IBM Data Server Driver for JDBC and SQLJ,” on page 465

“Upgrading the IBM Data Server Driver for JDBC and SQLJ to a new version” on page 460

Driver support for JDBC APIs

The JDBC drivers that are supported by DB2 and IBM Informix Dynamic Server (IDS) database systems have different levels of support for JDBC methods.

The following tables list the JDBC interfaces and indicate which drivers supports them. The drivers and their supported platforms are:

Table 35. JDBC drivers for DB2 and IDS database systems

JDBC driver name	Associated data source
IBM Data Server Driver for JDBC and SQLJ	DB2 Database for Linux, UNIX, and Windows, DB2 for z/OS, or IBM Informix Dynamic Server (IDS)
DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (deprecated)	DB2 Database for Linux, UNIX, and Windows
IBM Informix JDBC Driver (IDS JDBC Driver)	IDS

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 36. Support for Array methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ1 on page 258 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
free ²	Yes	No	No
getArray	Yes	No	Yes
getBaseType	Yes	No	Yes
getBaseTypeName	Yes	No	Yes

Table 36. Support for Array methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
getResultSet	Yes	No	Yes

Notes:

1. Under the IBM Data Server Driver for JDBC and SQLJ, Array methods are supported for connections to DB2 Database for Linux, UNIX, and Windows data sources only.
2. This is a JDBC 4.0 method.

Table 37. Support for BatchUpdateException methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from java.lang.Exception	Yes	Yes	Yes
getUpdateCounts	Yes	Yes	Yes

Table 38. Support for Blob methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
free ¹	Yes	No	No
getBinaryStream	Yes ²	Yes	Yes
getBytes	Yes	Yes	Yes
length	Yes	Yes	Yes
position	Yes	Yes	Yes
setBinaryStream ³	Yes	No	No
setBytes ³	Yes	No	No
truncate ³	Yes	No	No

Notes:

1. This is a JDBC 4.0 method.
2. Supported forms of this method include the following JDBC 4.0 form:
getBinaryStream(long pos, long length)
3. For versions of the IBM Data Server Driver for JDBC and SQLJ before version 3.50, these methods cannot be used if a Blob is passed to a stored procedure as an IN or INOUT parameter, and the methods are used on the Blob in the stored procedure.

Table 39. Support for CallableStatement methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from java.sql.Statement	Yes	Yes	Yes
Methods inherited from java.sql.PreparedStatement	Yes ¹	Yes	Yes
getArray	No	No	No

Table 39. Support for CallableStatement methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
getBigDecimal	Yes ³	Yes	Yes
getBlob	Yes ³	Yes	Yes
getBoolean	Yes ³	Yes	Yes
getByte	Yes ³	Yes	Yes
getBytes	Yes ³	Yes	Yes
getClob	Yes ³	Yes	Yes
getDate	Yes ^{3,4}	Yes ⁴	Yes
getDouble	Yes ³	Yes	Yes
getFloat	Yes ³	Yes	Yes
getInt	Yes ³	Yes	Yes
getLong	Yes ³	Yes	Yes
getObject	Yes ^{3,5}	Yes ⁵	Yes
getRef	No	No	No
getRowId ²	Yes	No	No
getShort	Yes ³	Yes	Yes
getString	Yes ³	Yes	Yes
getTime	Yes ^{3,4}	Yes ⁴	Yes
getTimestamp	Yes ^{3,4}	Yes ⁴	Yes
getURL	Yes	No	No
registerOutParameter	Yes ⁶	Yes ⁶	Yes ⁶
setAsciiStream	Yes ⁷	No	Yes
setBigDecimal	Yes ⁷	No	Yes
setBinaryStream	Yes ⁷	No	Yes
setBoolean	Yes ⁷	No	Yes
setByte	Yes ⁷	No	Yes
setBytes	Yes ⁷	No	Yes
setCharacterStream	Yes ⁷	No	Yes
setDate	Yes ⁷	No	Yes
setDouble	Yes ⁷	No	Yes
setFloat	Yes ⁷	No	Yes
setInt	Yes ⁷	No	Yes
setLong	Yes ⁷	No	Yes
setNull	Yes ^{7,8}	No	Yes
setObject	Yes ⁷	No	Yes
setShort	Yes ⁷	No	Yes
setString	Yes ⁷	No	Yes
setTime	Yes ⁷	No	Yes
setTimestamp	Yes ⁷	No	Yes

Table 39. Support for CallableStatement methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
setURL	Yes	No	No
wasNull	Yes	Yes	Yes

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 database server does no timezone adjustment for datetime values. The JDBC driver adjusts a value for the local timezone after retrieving the value from the server if you specify a form of the `getDate`, `getTime`, or `getTimestamp` method that includes a `java.util.Calendar` parameter.
5. The following form of the `getObject` method is not supported:
`getObject(int parameterIndex, java.util.Map map)`
6. The following form of the `registerOutParameter` method is not supported:
`registerOutParameter(int parameterIndex, int jdbcType, String typeName)`
7. The following forms of `CallableStatement.setXXX` methods are not supported if the data source is DB2 for z/OS:
`setXXX(String parameterName,...)`
8. The following form of `setNull` is not supported:
`setNull(int parameterIndex, int jdbcType, String typeName)`

Table 40. Support for Clob methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
free ¹	Yes	No	No
getAsciiStream	Yes	Yes	Yes
getCharacterStream	Yes ²	Yes	Yes
getSubString	Yes	Yes	Yes
length	Yes	Yes	Yes
position	Yes	Yes	Yes
setAsciiStream ³	Yes	No	Yes
setCharacterStream ³	Yes	No	Yes
setString ³	Yes	No	Yes
truncate ³	Yes	No	Yes

Notes:

1. This is a JDBC 4.0 method.
2. Supported forms of this method include the following JDBC 4.0 form:
`getCharacterStream(long pos, long length)`
3. For versions of the IBM Data Server Driver for JDBC and SQLJ before version 3.50, these methods cannot be used if a `Clob` is passed to a stored procedure as an `IN` or `INOUT` parameter, and the methods are used on the `Clob` in the stored procedure.

Table 41. Support for Connection methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
clearWarnings	Yes	Yes	Yes
close	Yes	Yes	Yes
commit	Yes	Yes	Yes
createStatement	Yes	Yes ²	Yes
createBlob ¹	Yes	No	No
createClob ¹	Yes	No	No
getAutoCommit	Yes	Yes	Yes
getCatalog	Yes	Yes	Yes
getClientInfo ³	Yes	No	No
getHoldability	Yes	No	No
getMetaData	Yes	Yes	Yes
getTransactionIsolation	Yes	Yes	Yes
getTypeMap	No	No	Yes
getWarnings	Yes	Yes	Yes
isClosed	Yes	Yes	Yes
isReadOnly	Yes	Yes	Yes
isValid ^{3,4}	Yes	No	No
nativeSQL	Yes	Yes	Yes
prepareCall	Yes ⁵	Yes	Yes
prepareStatement	Yes	Yes ²	Yes
releaseSavepoint	Yes	No	No
rollback	Yes	Yes ²	Yes
setAutoCommit	Yes	Yes	Yes
setCatalog	Yes	Yes	No
setClientInfo ³	Yes	No	No
setReadOnly	Yes ⁶	Yes	No
setSavepoint	Yes	No	No
setTransactionIsolation	Yes	Yes	Yes
setTypeMap	No	No	Yes

Notes:

1. This is a JDBC 4.0 method.
2. The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows does not support the JDBC 3.0 forms of this method.
3. This is a JDBC 4.0 method.
4. Under IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, an `SQLException` is thrown if the *timeout* parameter value is less than 0. Under IBM Data Server Driver for JDBC and SQLJ type 2 connectivity, an `SQLException` is thrown if the if the *timeout* parameter value is not 0.
5. If the stored procedure in the CALL statement is on DB2 for z/OS, the parameters of the CALL statement cannot be expressions.
6. 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 42. Support for ConnectionEvent methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from java.util.EventObject	Yes	Yes	Yes
getSQLException	Yes	Yes	Yes

Table 43. Support for ConnectionEventListener methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
connectionClosed	Yes	Yes	Yes
connectionErrorOccurred	Yes	Yes	Yes

Table 44. Support for ConnectionPoolDataSource methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
getLoginTimeout	Yes	Yes	Yes
getLogWriter	Yes	Yes	Yes
getPooledConnection	Yes	Yes	Yes
setLoginTimeout	Yes ¹	Yes	Yes
setLogWriter	Yes	Yes	Yes

Note:

1. This method is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

Table 45. Support for DatabaseMetaData methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
allProceduresAreCallable	Yes	Yes	Yes
allTablesAreSelectable	Yes ¹	Yes	Yes ¹
dataDefinitionCausesTransactionCommit	Yes	Yes	Yes
dataDefinitionIgnoredInTransactions	Yes	Yes	Yes
deletesAreDetected	Yes	Yes	Yes
doesMaxRowSizeIncludeBlobs	Yes	Yes	Yes
getAttributes	Yes ²	No	No
getBestRowIdentifier	Yes	Yes	Yes
getCatalogs	Yes	Yes	Yes
getCatalogSeparator	Yes	Yes	Yes
getCatalogTerm	Yes	Yes	Yes
getClientInfoProperties ⁶	Yes	No	No

Table 45. Support for DatabaseMetaData methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
getColumnPrivileges	Yes	Yes	Yes
getColumns	Yes ⁷	Yes ¹⁰	Yes ¹⁰
getConnection	Yes	Yes	Yes
getCrossReference	Yes	Yes	Yes
getDatabaseMajorVersion	Yes	No	No
getDatabaseMinorVersion	Yes	No	No
getDatabaseProductName	Yes	Yes	Yes
getDatabaseProductVersion	Yes	Yes	Yes
getDefaultTransactionIsolation	Yes	Yes	Yes
getDriverMajorVersion	Yes	Yes	Yes
getDriverMinorVersion	Yes	Yes	Yes
getDriverName	Yes ⁸	Yes	Yes
getDriverVersion	Yes	Yes	Yes
getExportedKeys	Yes	Yes	Yes
getFunctionColumns ⁶	Yes	No	No
getFunctions ⁶	Yes	No	No
getExtraNameCharacters	Yes	Yes	Yes
getIdentifierQuoteString	Yes	Yes	Yes
getImportedKeys	Yes	Yes	Yes
getIndexInfo	Yes	Yes	Yes
getJDBCMinorVersion	Yes	No	No
getJDBCMajorVersion	Yes	No	No
getMaxBinaryLiteralLength	Yes	Yes	Yes
getMaxCatalogNameLength	Yes	Yes	Yes
getMaxCharLiteralLength	Yes	Yes	Yes
getMaxColumnNameLength	Yes	Yes	Yes
getMaxColumnsInGroupBy	Yes	Yes	Yes
getMaxColumnsInIndex	Yes	Yes	Yes
getMaxColumnsInOrderBy	Yes	Yes	Yes
getMaxColumnsInSelect	Yes	Yes	Yes
getMaxColumnsInTable	Yes	Yes	Yes
getMaxConnections	Yes	Yes	Yes
getMaxCursorNameLength	Yes	Yes	Yes
getMaxIndexLength	Yes	Yes	Yes
getMaxProcedureNameLength	Yes	Yes	Yes
getMaxRowSize	Yes	Yes	Yes

Table 45. Support for DatabaseMetaData methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
getMaxSchemaNameLength	Yes	Yes	Yes
getMaxStatementLength	Yes	Yes	Yes
getMaxStatements	Yes	Yes	Yes
getMaxTableNameLength	Yes	Yes	Yes
getMaxTablesInSelect	Yes	Yes	Yes
getMaxUserNameLength	Yes	Yes	Yes
getNumericFunctions	Yes	Yes	Yes
getPrimaryKeys	Yes	Yes	Yes
getProcedureColumns	Yes ⁷ on page 267	Yes	Yes
getProcedures	Yes ⁷ on page 267	Yes	Yes
getProcedureTerm	Yes	Yes	Yes
getResultSetHoldability	Yes	No	No
getRowIdLifetime ⁶	Yes	No	No
getSchemas	Yes ⁹ on page 267	Yes ¹⁰	Yes ¹⁰
getSchemaTerm	Yes	Yes	Yes
getSearchStringEscape	Yes	Yes	Yes
getSQLKeywords	Yes	Yes	Yes
getSQLStateType	Yes	No	No
getStringFunctions	Yes	Yes	Yes
getSuperTables	Yes ²	No	No
getSuperTypes	Yes ²	No	No
getSystemFunctions	Yes	Yes	Yes
getTablePrivileges	Yes	Yes	Yes
getTables	Yes	Yes ¹⁰	Yes ¹⁰
getTableTypes	Yes	Yes	Yes
getTimeDateFunctions	Yes	Yes	Yes
getTypeInfo	Yes	Yes	Yes
getUDTs	No	Yes ¹¹	Yes ¹¹
getURL	Yes	Yes	Yes
getUserName	Yes	Yes	Yes
getVersionColumns	Yes	Yes	Yes
insertsAreDetected	Yes	Yes	Yes
isCatalogAtStart	Yes	Yes	Yes
isReadOnly	Yes	Yes	Yes

Table 45. Support for DatabaseMetaData methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
locatorsUpdateCopy	Yes ³	Yes	Yes ³
nullPlusNonNullsNull	Yes	Yes	Yes
nullsAreSortedAtEnd	Yes ⁴	Yes	Yes ⁴
nullsAreSortedAtStart	Yes	Yes	Yes
nullsAreSortedHigh	Yes ⁵	Yes	Yes ⁵
nullsAreSortedLow	Yes ¹	Yes	Yes ¹
othersDeletesAreVisible	Yes	Yes	Yes
othersInsertsAreVisible	Yes	Yes	Yes
othersUpdatesAreVisible	Yes	Yes	Yes
ownDeletesAreVisible	Yes	Yes	Yes
ownInsertsAreVisible	Yes	Yes	Yes
ownUpdatesAreVisible	Yes	Yes	Yes
storesLowerCaseIdentifiers	Yes ¹	Yes	Yes ¹
storesLowerCaseQuotedIdentifiers	Yes ⁴	Yes	Yes ⁴
storesMixedCaseIdentifiers	Yes	Yes	Yes
storesMixedCaseQuotedIdentifiers	Yes	Yes	Yes
storesUpperCaseIdentifiers	Yes ⁵	Yes	Yes ⁵
storesUpperCaseQuotedIdentifiers	Yes	Yes	Yes
supportsAlterTableWithAddColumn	Yes	Yes	Yes
supportsAlterTableWithDropColumn	Yes ¹	Yes	Yes ¹
supportsANSI92EntryLevelSQL	Yes	Yes	Yes
supportsANSI92FullSQL	Yes	Yes	Yes
supportsANSI92IntermediateSQL	Yes	Yes	Yes
supportsBatchUpdates	Yes	Yes	Yes
supportsCatalogsInDataManipulation	Yes ¹	Yes	Yes ¹
supportsCatalogsInIndexDefinitions	Yes	Yes	Yes
supportsCatalogsInPrivilegeDefinitions	Yes	Yes	Yes
supportsCatalogsInProcedureCalls	Yes ¹	Yes	Yes ¹
supportsCatalogsInTableDefinitions	Yes	Yes	Yes
SupportsColumnAliasing	Yes	Yes	Yes
supportsConvert	Yes	Yes	Yes
supportsCoreSQLGrammar	Yes	Yes	Yes
supportsCorrelatedSubqueries	Yes	Yes	Yes
supportsDataDefinitionAndDataManipulationTransactions	Yes	Yes	Yes
supportsDataManipulationTransactionsOnly	Yes	Yes	Yes
supportsDifferentTableCorrelationNames	Yes ⁴	Yes	Yes ⁴

Table 45. Support for DatabaseMetaData methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
supportsExpressionsInOrderBy	Yes	Yes	Yes
supportsExtendedSQLGrammar	Yes	Yes	Yes
supportsFullOuterJoins	Yes ³	Yes	Yes ³
supportsGetGeneratedKeys	Yes	No	No
supportsGroupBy	Yes	Yes	Yes
supportsGroupByBeyondSelect	Yes	Yes	Yes
supportsGroupByUnrelated	Yes	Yes	Yes
supportsIntegrityEnhancementFacility	Yes	Yes	Yes
supportsLikeEscapeClause	Yes	Yes	Yes
supportsLimitedOuterJoins	Yes	Yes	Yes
supportsMinimumSQLGrammar	Yes	Yes	Yes
supportsMixedCaseIdentifiers	Yes	Yes	Yes
supportsMixedCaseQuotedIdentifiers	Yes ³	Yes	Yes ³
supportsMultipleOpenResults	Yes ⁵	No	Yes ⁵
supportsMultipleResultSets	Yes ⁵	Yes	Yes ⁵
supportsMultipleTransactions	Yes	Yes	Yes
supportsNamedParameters	Yes	No	No
supportsNonNullableColumns	Yes	Yes	Yes
supportsOpenCursorsAcrossCommit	Yes ³	Yes	Yes ³
supportsOpenCursorsAcrossRollback	Yes	Yes	Yes
supportsOpenStatementsAcrossCommit	Yes ³	Yes	Yes ³
supportsOpenStatementsAcrossRollback	Yes ³	Yes	Yes ³
supportsOrderByUnrelated	Yes	Yes	Yes
supportsOuterJoins	Yes	Yes	Yes
supportsPositionedDelete	Yes	Yes	Yes
supportsPositionedUpdate	Yes	Yes	Yes
supportsResultSetConcurrency	Yes	Yes	Yes
supportsResultSetHoldability	Yes	No	No
supportsResultSetType	Yes	Yes	Yes
supportsSavepoints	Yes	No	Yes
supportsSchemasInDataManipulation	Yes	Yes	Yes
supportsSchemasInIndexDefinitions	Yes	Yes	Yes
supportsSchemasInPrivilegeDefinitions	Yes	Yes	Yes
supportsSchemasInProcedureCalls	Yes	Yes	Yes
supportsSchemasInTableDefinitions	Yes	Yes	Yes
supportsSelectForUpdate	Yes	Yes	Yes

Table 45. Support for DatabaseMetaData methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
supportsStoredProcedures	Yes	Yes	Yes
supportsSubqueriesInComparisons	Yes	Yes	Yes
supportsSubqueriesInExists	Yes	Yes	Yes
supportsSubqueriesInIns	Yes	Yes	Yes
supportsSubqueriesInQuantifieds	Yes	Yes	Yes
supportsSuperTables	Yes	No	No
supportsSuperTypes	Yes	No	No
supportsTableCorrelationNames	Yes	Yes	Yes
supportsTransactionIsolationLevel	Yes	Yes	Yes
supportsTransactions	Yes	Yes	Yes
supportsUnion	Yes	Yes	Yes
supportsUnionAll	Yes	Yes	Yes
updatesAreDetected	Yes	Yes	Yes
usesLocalFilePerTable	Yes	Yes	Yes
usesLocalFiles	Yes	Yes	Yes

Notes:

1. DB2 data sources return false for this method. IDS data sources return true.
2. This method is supported for connections to DB2 Database for Linux, UNIX, and Windows and IDS only.
3. Under the IBM Data Server Driver for JDBC and SQLJ, DB2 data sources and IDS data sources return true for this method. Under the IDS JDBC Driver, IDS data sources return false.
4. Under the IBM Data Server Driver for JDBC and SQLJ, DB2 data sources and IDS data sources return false for this method. Under the IDS JDBC Driver, IDS data sources return true.
5. DB2 data sources return true for this method. IDS data sources return false.
6. This is a JDBC 4.0 method.
7. This method returns the additional column that is described by the JDBC 4.0 specification.
8. JDBC 3.0 and earlier implementations of the IBM Data Server Driver for JDBC and SQLJ return "IBM DB2 JDBC Universal Driver Architecture."
The JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ returns "IBM Data Server Driver for JDBC and SQLJ."
9. The JDBC 4.0 form and previous forms of this method are supported.
10. The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows does not support the JDBC 3.0 form of this method.
11. The method can be executed, but it returns an empty ResultSet.

Table 46. Support for DataSource methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
getConnection	Yes	Yes	Yes
getLoginTimeout	Yes	Yes ¹	Yes

Table 46. Support for DataSource methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
getLogWriter	Yes	Yes	Yes
setLoginTimeout	Yes ²	Yes ¹	Yes
setLogWriter	Yes	Yes	Yes

Notes:

1. The DB2 JDBC Type 2 Driver does not use this setting.
2. This method is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

Table 47. Support for DataTruncation methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from java.lang.Throwable	Yes	Yes	Yes
Methods inherited from java.sql.SQLException	Yes	Yes	Yes
Methods inherited from java.sql.SQLWarning	Yes	Yes	Yes
getDataSize	Yes	Yes	Yes
getIndex	Yes	Yes	Yes
getParameter	Yes	Yes	Yes
getRead	Yes	Yes	Yes
getTransferSize	Yes	Yes	Yes

Table 48. Support for Driver methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
acceptsURL	Yes	Yes	Yes
connect	Yes	Yes	Yes
getMajorVersion	Yes	Yes	Yes
getMinorVersion	Yes	Yes	Yes
getPropertyInfo	Yes	Yes	Yes
jdbcCompliant	Yes	Yes	Yes

Table 49. Support for DriverManager methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
deregisterDriver	Yes	Yes	Yes
getConnection	Yes	Yes	Yes
getDriver	Yes	Yes	Yes

Table 49. Support for DriverManager methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
getDrivers	Yes	Yes	Yes
getLoginTimeout	Yes	Yes ¹	Yes ¹
getLogStream	Yes	Yes	Yes
getLogWriter	Yes	Yes	Yes
println	Yes	Yes	Yes
registerDriver	Yes	Yes	Yes
setLoginTimeout	Yes ²	Yes ¹	Yes ¹
setLogStream	Yes	Yes	Yes
setLogWriter	Yes	Yes	Yes

Notes:

1. The DB2 JDBC Type 2 Driver does not use this setting.
2. This method is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

Table 50. Support for ParameterMetaData methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
getParameterClassName	No	No	No
getParameterCount	Yes	No	No
getParameterMode	Yes	No	No
getParameterType	Yes	No	No
getParameterTypeName	Yes	No	No
getPrecision	Yes	No	No
getScale	Yes	No	No
isNullable	Yes	No	No
isSigned	Yes	No	No

Table 51. Support for PooledConnection methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
addConnectionEventListener	Yes	Yes	Yes
addStatementEventListener ¹	Yes	No	No
close	Yes	Yes	Yes
getConnection	Yes	Yes	Yes
removeConnectionEventListener	Yes	Yes	Yes
removeStatementEventListener ¹	Yes	No	No

Notes:

1. This is a JDBC 4.0 method.

Table 52. Support for PreparedStatement methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from java.sql.Statement	Yes	Yes	Yes
addBatch	Yes	Yes	Yes
clearParameters	Yes	Yes	Yes
execute	Yes	Yes	Yes
executeQuery	Yes	Yes	Yes
executeUpdate	Yes	Yes	Yes
getMetaData	Yes	Yes	Yes
getParameterMetaData	Yes	Yes	Yes
setArray	No	No	No
setAsciiStream	Yes ^{1,2}	Yes	Yes
setBigDecimal	Yes	Yes	Yes
setBinaryStream	Yes ^{1,3}	Yes	Yes
setBlob	Yes ⁴	Yes	Yes
setBoolean	Yes	Yes	Yes
setByte	Yes	Yes	Yes
setBytes	Yes	Yes	Yes
setCharacterStream	Yes ^{1,5}	Yes	Yes
setClob	Yes ⁶	Yes	Yes
setDate	Yes ⁸	Yes ⁸	Yes ⁸
setDouble	Yes	Yes	Yes
setFloat	Yes	Yes	Yes
setInt	Yes	Yes	Yes
setLong	Yes	Yes	Yes
setNull	Yes ⁹	Yes ⁹	Yes ⁹
setObject	Yes	Yes	Yes
setRef	No	No	No
setRowId ⁷	Yes	No	No
setShort	Yes	Yes	Yes
setString	Yes ¹⁰	Yes ¹⁰	Yes ¹⁰
setTime	Yes ⁸	Yes ⁸	Yes ⁸
setTimestamp	Yes ⁸	Yes ⁸	Yes ⁸
setUnicodeStream	Yes	Yes	Yes
setURL	Yes	Yes	Yes

Table 52. Support for PreparedStatement methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Notes:			
1. If the value of the <i>length</i> 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 <i>parameterIndex</i> , InputStream <i>x</i> , long <i>length</i>) setAsciiStream(int <i>parameterIndex</i> , InputStream <i>x</i>)			
3. Supported forms of this method include the following JDBC 4.0 forms: setBinaryStream(int <i>parameterIndex</i> , InputStream <i>x</i> , long <i>length</i>) setBinaryStream(int <i>parameterIndex</i> , InputStream <i>x</i>)			
4. Supported forms of this method include the following JDBC 4.0 form: setBlob(int <i>parameterIndex</i> , InputStream <i>inputStream</i> , long <i>length</i>)			
5. Supported forms of this method include the following JDBC 4.0 forms: setCharacterStream(int <i>parameterIndex</i> , Reader <i>reader</i> , long <i>length</i>) setCharacterStream(int <i>parameterIndex</i> , Reader <i>reader</i>)			
6. Supported forms of this method include the following JDBC 4.0 form: setClob(int <i>parameterIndex</i> , Reader <i>reader</i> , long <i>length</i>)			
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 <i>parameterIndex</i> , int <i>jdbcType</i> , String <i>typeName</i>)			
10. setString is not supported if the column has the FOR BIT DATA attribute or the data type is BLOB.			

Table 53. Support for Ref methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
get BaseTypeName	No	No	No

Table 54. Support for ResultSet methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
absolute	Yes	Yes	Yes
afterLast	Yes	Yes	Yes
beforeFirst	Yes	Yes	Yes
cancelRowUpdates	Yes	No	No
clearWarnings	Yes	Yes	Yes
close	Yes	Yes	Yes
deleteRow	Yes	No	No
findColumn	Yes	Yes	Yes
first	Yes	Yes	Yes
getArray	No	No	No

Table 54. Support for ResultSet methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
getAsciiStream	Yes	Yes	Yes
getBigDecimal	Yes	Yes	Yes
getBinaryStream	Yes ¹	Yes	Yes
getBlob	Yes	Yes	Yes
getBoolean	Yes	Yes	Yes
getByte	Yes	Yes	Yes
getBytes	Yes	Yes	Yes
getCharacterStream	Yes	Yes	Yes
getClob	Yes	Yes	Yes
getConcurrency	Yes	Yes	Yes
getCursorName	Yes	Yes	Yes
getDate	Yes ³	Yes ³	Yes ³
getDouble	Yes	Yes	Yes
getFetchDirection	Yes	Yes	Yes
getFetchSize	Yes	Yes	Yes
getFloat	Yes	Yes	Yes
getInt	Yes	Yes	Yes
getLong	Yes	Yes	Yes
getMetaData	Yes	Yes	Yes
getObject	Yes ⁴	Yes ⁴	Yes ⁴
getRef	No	No	No
getRow	Yes	Yes	Yes
getRowId ¹⁰	Yes	No	No
getShort	Yes	Yes	Yes
getStatement	Yes	Yes	Yes
getString	Yes	Yes	Yes
getTime	Yes ³	Yes ³	Yes ³
getTimestamp	Yes ³	Yes ³	Yes ³
getType	Yes	Yes	Yes
getUnicodeStream	Yes	Yes	Yes
getURL	Yes	Yes	Yes
getWarnings	Yes	Yes	Yes
insertRow	Yes	No	No
isAfterLast	Yes	Yes	Yes
isBeforeFirst	Yes	Yes	Yes
isFirst	Yes	Yes	Yes
isLast	Yes	Yes	Yes
last	Yes	Yes	Yes

Table 54. Support for ResultSet methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
moveToCurrentRow	Yes	No	No
moveToInsertRow	Yes	No	No
next	Yes	Yes	Yes
previous	Yes	Yes	Yes
refreshRow	Yes	No	No
relative	Yes	Yes	Yes
rowDeleted	Yes	No	No
rowInserted	Yes	No	No
rowUpdated	Yes	No	No
setFetchDirection	Yes	Yes	Yes
setFetchSize	Yes	Yes	Yes
updateArray	No	No	No
updateAsciiStream	Yes ⁵	No	No
updateBigDecimal	Yes	No	No
updateBinaryStream	Yes ⁶	No	No
updateBlob	Yes ⁷	No	No
updateBoolean	Yes	No	No
updateByte	Yes	No	No
updateBytes	Yes	No	No
updateCharacterStream	Yes ⁸	No	No
updateClob	Yes ⁹	No	No
updateDate	Yes	No	No
updateDouble	Yes	No	No
updateFloat	Yes	No	No
updateInt	Yes	No	No
updateLong	Yes	No	No
updateNull	Yes	No	No
updateObject	Yes	No	No
updateRef	No	No	No
updateRow	Yes	No	No
updateRowId ¹⁰	Yes	No	No
updateShort	Yes	No	No
updateString	Yes	No	No
updateTime	Yes	No	No
updateTimestamp	Yes	No	No
wasNull	Yes	Yes	Yes

Table 54. Support for ResultSet methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Notes:			
1. getBinaryStream is not supported for CLOB columns.			
2. getMetaData pads the schema name, if the returned schema name is less than 8 characters, to fill 8 characters.			
3. The database server does no timezone adjustment for datetime values. The JDBC driver adjusts a value for the local timezone after retrieving the value from the server if you specify a form of the getDate, getTime, or getTimestamp method that includes a java.util.Calendar parameter.			
4. The following form of the getObject method is not supported: getObject(int <i>parameterIndex</i> , java.util.Map <i>map</i>)			
5. Supported forms of this method include the following JDBC 4.0 forms: updateAsciiStream(int <i>columnIndex</i> , InputStream <i>x</i>) updateAsciiStream(String <i>columnLabel</i> , InputStream <i>x</i>) updateAsciiStream(int <i>columnIndex</i> , InputStream <i>x</i> , long <i>length</i>) updateAsciiStream(String <i>columnLabel</i> , InputStream <i>x</i> , long <i>length</i>)			
6. Supported forms of this method include the following JDBC 4.0 forms: updateBinaryStream(int <i>columnIndex</i> , InputStream <i>x</i>) updateBinaryStream(String <i>columnLabel</i> , InputStream <i>x</i>) updateBinaryStream(int <i>columnIndex</i> , InputStream <i>x</i> , long <i>length</i>) updateBinaryStream(String <i>columnLabel</i> , InputStream <i>x</i> , long <i>length</i>)			
7. Supported forms of this method include the following JDBC 4.0 forms: updateBlob(int <i>columnIndex</i> , InputStream <i>x</i>) updateBlob(String <i>columnLabel</i> , InputStream <i>x</i>) updateBlob(int <i>columnIndex</i> , InputStream <i>x</i> , long <i>length</i>) updateBlob(String <i>columnLabel</i> , InputStream <i>x</i> , long <i>length</i>)			
8. Supported forms of this method include the following JDBC 4.0 forms: updateCharacterStream(int <i>columnIndex</i> , Reader <i>reader</i>) updateCharacterStream(String <i>columnLabel</i> , Reader <i>reader</i>) updateCharacterStream(int <i>columnIndex</i> , Reader <i>reader</i> , long <i>length</i>) updateCharacterStream(String <i>columnLabel</i> , Reader <i>reader</i> , long <i>length</i>)			
9. Supported forms of this method include the following JDBC 4.0 forms: updateClob(int <i>columnIndex</i> , Reader <i>reader</i>) updateClob(String <i>columnLabel</i> , Reader <i>reader</i>) updateClob(int <i>columnIndex</i> , Reader <i>reader</i> , long <i>length</i>) updateClob(String <i>columnLabel</i> , Reader <i>reader</i> , long <i>length</i>)			
10. This is a JDBC 4.0 method.			

Table 55. Support for ResultSetMetaData methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
getCatalogName	Yes	Yes	Yes
getColumnClassName	No	Yes	Yes
getColumnCount	Yes	Yes	Yes
getColumnDisplaySize	Yes	Yes	Yes
getColumnLabel	Yes	Yes	Yes
getColumnName	Yes	Yes	Yes
getColumnType	Yes	Yes	Yes
getColumnTypeName	Yes	Yes	Yes

Table 55. Support for *ResultSetMetaData* methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
<code>getPrecision</code>	Yes	Yes	Yes
<code>getScale</code>	Yes	Yes	Yes
<code>getSchemaName</code>	Yes	Yes	Yes
<code>getTableName</code>	Yes ¹	Yes	Yes
<code>isAutoIncrement</code>	Yes	Yes	Yes
<code>isCaseSensitive</code>	Yes	Yes	Yes
<code>isCurrency</code>	Yes	Yes	Yes
<code>isDefinitelyWritable</code>	Yes	Yes	Yes
<code>isNullable</code>	Yes	Yes	Yes
<code>isReadOnly</code>	Yes	Yes	Yes
<code>isSearchable</code>	Yes	Yes	Yes
<code>isSigned</code>	Yes	Yes	Yes
<code>isWritable</code>	Yes	Yes	Yes

Notes:

1. For IDS 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 56. Support for *RowId* methods¹

JDBC method	IBM Data Server Driver for JDBC and SQLJ support ²	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
<code>equals</code>	Yes	No	No
<code>getBytes</code>	Yes	No	No
<code>hashCode</code>	No	No	No
<code>toString</code>	Yes	No	No

Notes:

1. These methods are JDBC 4.0 methods.
2. These methods are supported for connections to DB2 for z/OS, DB2 for i, and IDS data sources.

Table 57. Support for *SQLException* methods¹

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from <code>java.lang.Exception</code>	Yes	No	No
Methods inherited from <code>java.lang.Throwable</code>	Yes	No	No
Methods inherited from <code>java.lang.Object</code>	Yes	No	No
<code>getFailedProperties</code>	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 58. Support for *SQLData* methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
getSQLTypeName	No	No	No
readSQL	No	No	No
writeSQL	No	No	No

Table 59. Support for *SQLDataException* methods¹

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 60. Support for *SQLException* methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from java.lang.Exception	Yes	Yes	Yes
getSQLState	Yes	Yes	Yes
getErrorCode	Yes	Yes	Yes
getNextException	Yes	Yes	Yes
setNextException	Yes	Yes	Yes

Table 61. Support for *SQLFeatureNotSupported* methods¹

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 62. Support for *SQLInput* methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
readArray	No	No	No
readAsciiStream	No	No	No
readBigDecimal	No	No	No
readBinaryStream	No	No	No
readBlob	No	No	No
readBoolean	No	No	No
readByte	No	No	No
readBytes	No	No	No
readCharacterStream	No	No	No
readClob	No	No	No
readDate	No	No	No
readDouble	No	No	No
readFloat	No	No	No
readInt	No	No	No
readLong	No	No	No
readObject	No	No	No
readRef	No	No	No
readShort	No	No	No
readString	No	No	No
readTime	No	No	No
readTimestamp	No	No	No
wasNull	No	No	No

Table 63. Support for *SQLIntegrityConstraintViolationException* methods¹

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from <code>java.lang.Exception</code>	Yes	No	No
Methods inherited from <code>java.lang.Throwable</code>	Yes	No	No
Methods inherited from <code>java.lang.Object</code>	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 64. Support for *SQLInvalidAuthorizationSpecException* methods¹

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from <code>java.lang.Exception</code>	Yes	No	No
Methods inherited from <code>java.lang.Throwable</code>	Yes	No	No
Methods inherited from <code>java.lang.Object</code>	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 65. Support for *SQLNonTransientConnectionException* methods¹

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from <code>java.lang.Exception</code>	Yes	No	No
Methods inherited from <code>java.lang.Throwable</code>	Yes	No	No
Methods inherited from <code>java.lang.Object</code>	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 66. Support for *SQLNonTransientException* methods¹

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from <code>java.lang.Exception</code>	Yes	No	No
Methods inherited from <code>java.lang.Throwable</code>	Yes	No	No
Methods inherited from <code>java.lang.Object</code>	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 67. Support for *SQLOutput* methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
<code>writeArray</code>	No	No	No
<code>writeAsciiStream</code>	No	No	No
<code>writeBigDecimal</code>	No	No	No
<code>writeBinaryStream</code>	No	No	No
<code>writeBlob</code>	No	No	No

Table 67. Support for *SQLOutput* methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
<code>writeBoolean</code>	No	No	No
<code>writeByte</code>	No	No	No
<code>writeBytes</code>	No	No	No
<code>writeCharacterStream</code>	No	No	No
<code>writeClob</code>	No	No	No
<code>writeDate</code>	No	No	No
<code>writeDouble</code>	No	No	No
<code>writeFloat</code>	No	No	No
<code>writeInt</code>	No	No	No
<code>writeLong</code>	No	No	No
<code>writeObject</code>	No	No	No
<code>writeRef</code>	No	No	No
<code>writeShort</code>	No	No	No
<code>writeString</code>	No	No	No
<code>writeStruct</code>	No	No	No
<code>writeTime</code>	No	No	No
<code>writeTimestamp</code>	No	No	No

Table 68. Support for *SQLRecoverableException* methods¹

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from <code>java.lang.Exception</code>	Yes	No	No
Methods inherited from <code>java.lang.Throwable</code>	Yes	No	No
Methods inherited from <code>java.lang.Object</code>	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 69. Support for *SQLSyntaxErrorException* methods¹

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from <code>java.lang.Exception</code>	Yes	No	No
Methods inherited from <code>java.lang.Throwable</code>	Yes	No	No
Methods inherited from <code>java.lang.Object</code>	Yes	No	No

Table 69. Support for *SQLException* methods¹ (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
-------------	--	--	----------------------------

Note:

1. This is a JDBC 4.0 class.

Table 70. Support for *TimeoutException* methods¹

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
-------------	--	--	----------------------------

Methods inherited from <code>java.lang.Exception</code>	Yes	No	No
Methods inherited from <code>java.lang.Throwable</code>	Yes	No	No
Methods inherited from <code>java.lang.Object</code>	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 71. Support for *TransientConnectionException* methods¹

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
-------------	--	--	----------------------------

Methods inherited from <code>java.lang.Exception</code>	Yes	No	No
Methods inherited from <code>java.lang.Throwable</code>	Yes	No	No
Methods inherited from <code>java.lang.Object</code>	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 72. Support for *TransientException* methods¹

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
-------------	--	--	----------------------------

Methods inherited from <code>java.lang.Exception</code>	Yes	No	No
Methods inherited from <code>java.lang.Throwable</code>	Yes	No	No
Methods inherited from <code>java.lang.Object</code>	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 73. Support for *SQLTransientRollbackException* methods¹

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from java.lang.Exception	Yes	No	No
Methods inherited from java.lang.Throwable	Yes	No	No
Methods inherited from java.lang.Object	Yes	No	No

Note:

1. This is a JDBC 4.0 class.

Table 74. Support for *SQLXML* methods¹

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
free	Yes	No	No
getBinaryStream	Yes	No	No
getCharacterStream	Yes	No	No
getSource	Yes	No	No
getString	Yes	No	No
setBinaryStream	Yes	No	No
setCharacterStream	Yes	No	No
setResult	Yes	No	No
setString	Yes	No	No

Notes:

1. These are JDBC 4.0 methods. These methods are not supported for connections to IBM Informix Dynamic Server servers.

Table 75. Support for *Statement* methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
addBatch	Yes	Yes	Yes
cancel	Yes ¹	Yes ²	Yes
clearBatch	Yes	Yes	Yes
clearWarnings	Yes	Yes	Yes
close	Yes	Yes	Yes
execute	Yes	Yes ³	Yes ³
executeBatch	Yes	Yes	Yes
executeQuery	Yes	Yes	Yes
executeUpdate	Yes	Yes ³	Yes ³
getConnection	Yes	Yes	Yes
getFetchDirection	Yes	Yes	Yes
getFetchSize	Yes	Yes	Yes

Table 75. Support for Statement methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
getGeneratedKeys	Yes	No	No
getMaxFieldSize	Yes	Yes	Yes
getMaxRows	Yes	Yes	Yes
getMoreResults	Yes	Yes ³	Yes ³
getQueryTimeout	Yes ²	Yes	Yes
getResultSet	Yes	Yes	Yes
getResultSetConcurrency	Yes	Yes	Yes
getResultSetHoldability	Yes	No	No
getResultSetType	Yes	Yes	Yes
getUpdateCount ⁴	Yes	Yes	Yes
getWarnings	Yes	Yes	Yes
isClosed ⁷	Yes	No	No
isPoolable ⁷	Yes	No	No
setCursorName	Yes	Yes	Yes
setEscapeProcessing	Yes	Yes	Yes
setFetchDirection	Yes	Yes	Yes
setFetchSize	Yes	Yes	Yes
setMaxFieldSize	Yes	Yes	Yes
setMaxRows	Yes	Yes	Yes
setPoolable ⁷	Yes	No	No
setQueryTimeout	Yes ^{5,6}	Yes	Yes

Table 75. Support for Statement methods (continued)

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Notes:			
1. For the IBM Data Server Driver for JDBC and SQLJ, <code>Statement.cancel()</code> is supported only in the following environments:			
<ul style="list-style-type: none"> • Type 2 and type 4 connectivity from a Linux, UNIX, or Windows client to a DB2 Database for Linux, UNIX, and Windows server, Version 8 or later • Type 2 and type 4 connectivity from a Linux, UNIX, or Windows client to a DB2 for z/OS server, Version 9 or later • Type 4 connectivity from a z/OS client to a DB2 Database for Linux, UNIX, and Windows server, Version 8 or later • Type 4 connectivity from a z/OS client to a DB2 for z/OS server, Version 8 or later 			
2. For the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows, <code>Statement.cancel()</code> is supported only in the following environments:			
<ul style="list-style-type: none"> • Connections to a DB2 Database for Linux, UNIX, and Windows server, Version 8 or later • Connections to a DB2 for z/OS server, Version 9 or later 			
3. The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows does not support the JDBC 3.0 form of this method.			
4. Not supported for stored procedure <code>ResultSets</code> .			
5. For DB2 for i and for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, this method is supported only for a <i>seconds</i> value of 0.			
6. For the IBM Data Server Driver for JDBC and SQLJ Version 4.0 and later, <code>Statement.setQueryTimeout</code> is supported for the following methods:			
<ul style="list-style-type: none"> • <code>Statement.execute</code> • <code>Statement.executeUpdate</code> • <code>Statement.executeQuery</code> 			
<code>Statement.setQueryTimeout</code> is not supported for the <code>Statement.executeBatch</code> method.			
7. This is a JDBC 4.0 method.			

Table 76. Support for Struct methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
<code>getSQLTypeName</code>	No	No	No
<code>getAttributes</code>	No	No	No

Table 77. Support for Wrapper methods

JDBC method ¹	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
<code>isWrapperFor</code>	Yes	No	No
<code>unwrap</code>	Yes	No	No

Notes:

1. These are JDBC 4.0 methods.

Table 78. Support for `javax.sql.XAConnection` methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support ¹	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
Methods inherited from <code>javax.sql.PooledConnection</code>	Yes	Yes	Yes
<code>getXAResource</code>	Yes	Yes	Yes

Notes:

1. These methods are supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to a DB2 Database for Linux, UNIX, and Windows server or IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Table 79. Support for `XADataSource` methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
<code>getLoginTimeout</code>	Yes	Yes	Yes
<code>getLogWriter</code>	Yes	Yes	Yes
<code>getXAConnection</code>	Yes	Yes	Yes
<code>setLoginTimeout</code>	Yes	Yes	Yes
<code>setLogWriter</code>	Yes	Yes	Yes

Table 80. Support for `javax.transaction.xa.XAResource` methods

JDBC method	IBM Data Server Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support	IDS JDBC Driver support
<code>commit</code>	Yes ¹	Yes	Yes
<code>end</code>	Yes ¹	Yes	Yes
<code>forget</code>	Yes ¹	Yes	Yes
<code>getTransactionTimeout</code>	Yes ²	Yes	Yes
<code>isSameRM</code>	Yes ¹	Yes	Yes
<code>prepare</code>	Yes ¹	Yes	Yes
<code>recover</code>	Yes ¹	Yes	Yes
<code>rollback</code>	Yes ¹	Yes	Yes
<code>setTransactionTimeout</code>	Yes ²	Yes	Yes
<code>start</code>	Yes ¹	Yes	Yes

Notes:

1. This method is supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to a DB2 Database for Linux, UNIX, and Windows server or IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.
2. This method is supported for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 Database for Linux, UNIX, and Windows Version 9.1 or later.

Related concepts

"JDBC interfaces for executing SQL" on page 24

"Characteristics of a JDBC ResultSet under the IBM Data Server Driver for JDBC and SQLJ" on page 37

"LOBs in JDBC applications with the IBM Data Server Driver for JDBC and SQLJ" on page 50

Related tasks

"Learning about a data source using DatabaseMetaData methods" on page 21

"Creating and modifying database objects using the Statement.executeUpdate method" on page 25

"Updating data in tables using the PreparedStatement.executeUpdate method" on page 26

"Learning about parameters in a PreparedStatement using ParameterMetaData methods" on page 31

"Retrieving data from tables using the Statement.executeQuery method" on page 32

"Retrieving data from tables using the PreparedStatement.executeQuery method" on page 33

"Calling stored procedures in JDBC applications" on page 46

Related reference

"JDBC differences between the current IBM Data Server Driver for JDBC and SQLJ and earlier DB2 JDBC drivers" on page 394

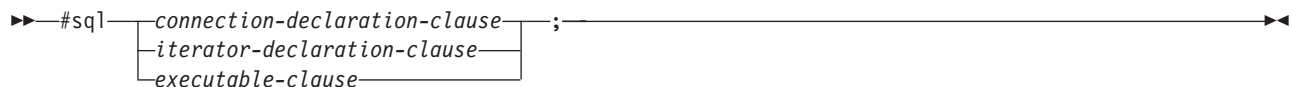
SQLJ statement reference information

SQLJ statements are used for transaction control and SQL statement execution.

SQLJ clause

The SQL statements in an SQLJ program are in SQLJ clauses.

Syntax



Usage notes

Keywords in an SQLJ clause are case sensitive, unless those keywords are part of an SQL statement in an executable clause.

Related reference

"SQLJ connection-declaration-clause" on page 289

"SQLJ executable-clause" on page 291

"SQLJ iterator-declaration-clause" on page 290

SQLJ host-expression

A host expression is a Java variable or expression that is referenced by SQLJ clauses in an SQLJ application program.

Syntax



Description

: Indicates that the variable or expression that follows is a host expression. The colon must immediately precede the variable or expression.

IN|OUT|INOUT

For a host expression that is used as a parameter in a stored procedure call, identifies whether the parameter provides data to the stored procedure (IN), retrieves data from the stored procedure (OUT), or does both (INOUT). The default is IN.

simple-variable

Specifies a Java unqualified identifier.

complex-expression

Specifies a Java expression that results in a single value.

Usage notes

- A complex expression must be enclosed in parentheses.
- ANSI/ISO rules govern where a host expression can appear in a static SQL statement.

Related concepts

“Variables in SQLJ applications” on page 103

SQLJ implements-clause

The implements clause derives one or more classes from a Java interface.

Syntax



interface-element:



Description

interface-element

Specifies a user-defined Java interface, the SQLJ interface `sqlj.runtime.ForUpdate` or the SQLJ interface `sqlj.runtime.Scrollable`.

You need to implement `sqlj.runtime.ForUpdate` when you declare an iterator for a positioned UPDATE or positioned DELETE operation. See “Perform

positioned UPDATE and DELETE operations in an SQLJ application” for information on performing a positioned UPDATE or positioned DELETE operation in SQLJ.

You need to implement `sqlj.runtime.Scrollable` when you declare a scrollable iterator. See “Use scrollable iterators in an SQLJ application” for information on scrollable iterators.

Related tasks

“Performing positioned UPDATE and DELETE operations in an SQLJ application” on page 106

“Using scrollable iterators in an SQLJ application” on page 124

Related reference

“SQLJ connection-declaration-clause” on page 289

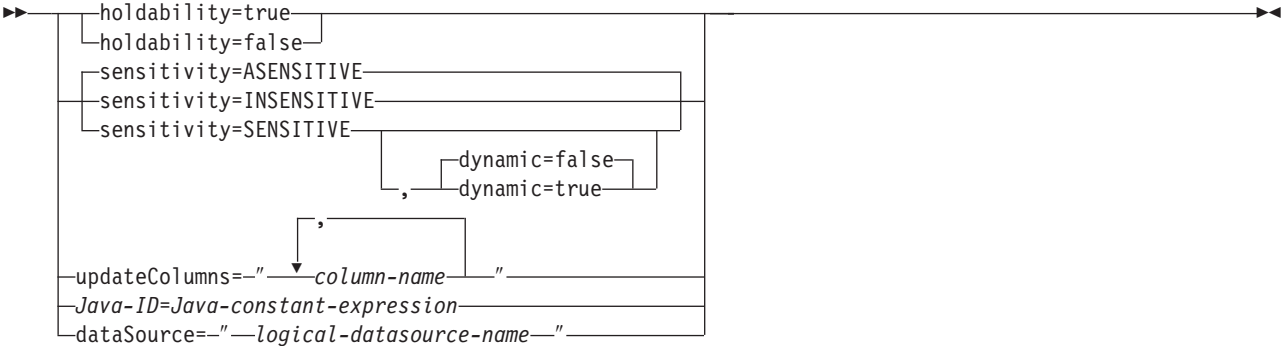
SQLJ with-clause

The with clause specifies a set of one or more attributes for an iterator or a connection context.

Syntax



with-element:



Description

holdability

For an iterator, specifies whether an iterator keeps its position in a table after a COMMIT is executed. The value for holdability must be true or false.

sensitivity

For an iterator, specifies whether changes that are made to the underlying table can be visible to the iterator after it is opened. The value must be INSENSITIVE, SENSITIVE, or ASENSITIVE. The default is ASENSITIVE.

For connections to IBM Informix Dynamic Server (IDS), only INSENSITIVE is supported.

dynamic

For an iterator that is defined with `sensitivity=SENSITIVE`, specifies whether the following cases are true:

- When the application executes positioned UPDATE and DELETE statements with the iterator, those changes are visible to the iterator.
- When the application executes INSERT, UPDATE, and DELETE statements within the application but outside the iterator, those changes are visible to the iterator.

The value for dynamic must be true or false. The default is false.

DB2 Database for Linux, UNIX, and Windows servers do not support dynamic scrollable cursors. Specify true only if your application accesses data on DB2 for z/OS servers, at Version 9 or later.

For connections to IDS, only false is supported. IDS 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 133

Related tasks

"Connecting to a data source using SQLJ" on page 95

"Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 106

"Using scrollable iterators in an SQLJ application" on page 124

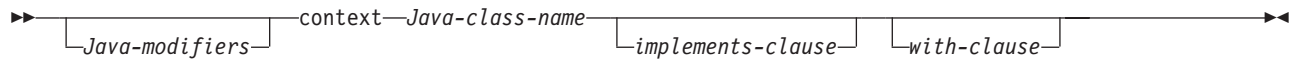
Related reference

"SQLJ connection-declaration-clause"

SQLJ connection-declaration-clause

The connection declaration clause declares a connection to a data source in an SQLJ application program.

Syntax



Description

Java-modifiers

Specifies modifiers that are valid for Java class declarations, such as static, public, private, or protected.

Java-class-name

Specifies a valid Java identifier. During the program preparation process, SQLJ generates a connection context class whose name is this identifier.

implements-clause

See "SQLJ implements-clause" for a description of this clause. In a connection declaration clause, the interface class to which the implements clause refers must be a user-defined interface class.

with-clause

See "SQLJ with-clause" for a description of this clause.

Usage notes

- SQLJ generates a connection class declaration for each connection declaration clause you specify. SQLJ data source connections are objects of those generated connection classes.
- You can specify a connection declaration clause anywhere that a Java class definition can appear in a Java program.

Related tasks

“Connecting to a data source using SQLJ” on page 95

Related reference

“SQLJ clause” on page 285

“SQLJ implements-clause” on page 286

“SQLJ with-clause” on page 287

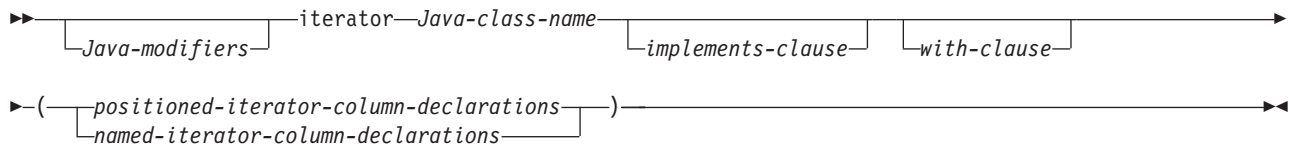
SQLJ iterator-declaration-clause

An iterator declaration clause declares a positioned iterator class or a named iterator class in an SQLJ application program.

An iterator contains the result table from a query. SQLJ generates an iterator class for each iterator declaration clause you specify. An iterator is an object of an iterator class.

An iterator declaration clause has a form for a positioned iterator and a form for a named iterator. The two kinds of iterators are distinct and incompatible Java types that are implemented with different interfaces.

Syntax



positioned-iterator-column declarations:



named-iterator-column-declarations:



Description

Java-modifiers

Any modifiers that are valid for Java class declarations, such as static, public, private, or protected.

Java-class-name

Any valid Java identifier. During the program preparation process, SQLJ generates an iterator class whose name is this identifier.

implements-clause

See “SQLJ implements-clause” for a description of this clause. For an iterator declaration clause that declares an iterator for a positioned UPDATE or positioned DELETE operation, the implements clause must specify interface

`sqlj.runtime.ForUpdate`. For an iterator declaration clause that declares a scrollable iterator, the implements clause must specify interface `sqlj.runtime.Scrollable`.

with-clause

See "SQLJ with-clause" for a description of this clause.

positioned-iterator-column-declarations

Specifies a list of Java data types, which are the data types of the columns in the positioned iterator. The data types in the list must be separated by commas. The order of the data types in the positioned iterator declaration is the same as the order of the columns in the result table. For online checking during serialized profile customization to succeed, the data types of the columns in the iterator must be compatible with the data types of the columns in the result table. See "Java, JDBC, and SQL data types" for a list of compatible data types.

named-iterator-column-declarations

Specifies a list of Java data types and Java identifiers, which are the data types and names of the columns in the named iterator. Pairs of data types and names must be separated by commas. The name of a column in the iterator must match, except for case, the name of a column in the result table. For online checking during serialized profile customization to succeed, the data types of the columns in the iterator must be compatible with the data types of the columns in the result table. See "Java, JDBC, and SQL data types" for a list of compatible data types.

Usage notes

- An iterator declaration clause can appear anywhere in a Java program that a Java class declaration can appear.
- When a named iterator declaration contains more than one pair of Java data types and Java IDs, all Java IDs within the list must be unique. Two Java IDs are not unique if they differ only in case.

Related concepts

"Data retrieval in SQLJ applications" on page 116

Related tasks

"Using a named iterator in an SQLJ application" on page 117

"Using a positioned iterator in an SQLJ application" on page 119

"Using scrollable iterators in an SQLJ application" on page 124

Related reference

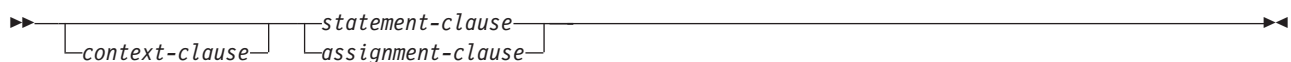
"SQLJ clause" on page 285

SQLJ executable-clause

An executable clause contains an SQL statement or an assignment statement. An assignment statement assigns the result of an SQL operation to a Java variable.

This topic describes the general form of an executable clause.

Syntax



Usage notes

- An executable clause can appear anywhere in a Java program that a Java statement can appear.
- SQLJ reports negative SQL codes from executable clauses through class `java.sql.SQLException`.
If SQLJ raises a run-time exception during the execution of an executable clause, the value of any host expression of type OUT or INOUT is undefined.

Related concepts

“SQL statement execution in SQLJ applications” on page 105

Related reference

“SQLJ clause” on page 285

“SQLJ assignment-clause” on page 296

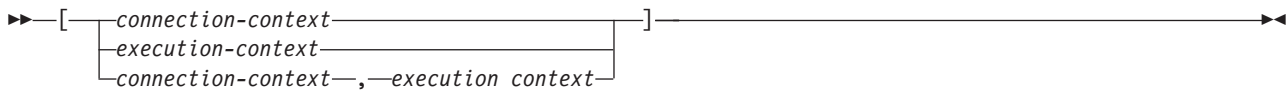
“SQLJ context-clause”

“SQLJ statement-clause”

SQLJ context-clause

A context clause specifies a connection context, an execution context, or both. You use a connection context to connect to a data source. You use an execution context to monitor and modify SQL statement execution.

Syntax



Description

connection-context

Specifies a valid Java identifier that is declared earlier in the SQLJ program. That identifier must be declared as an instance of the connection context class that SQLJ generates for a connection declaration clause.

execution-context

Specifies a valid Java identifier that is declared earlier in the SQLJ program. That identifier must be declared as an instance of class `sqlj.runtime.ExecutionContext`.

Usage notes

- If you do not specify a connection context in an executable clause, SQLJ uses the default connection context.
- If you do not specify an execution context, SQLJ obtains the execution context from the connection context of the statement.

Related tasks

“Connecting to a data source using SQLJ” on page 95

“Controlling the execution of SQL statements in SQLJ” on page 136

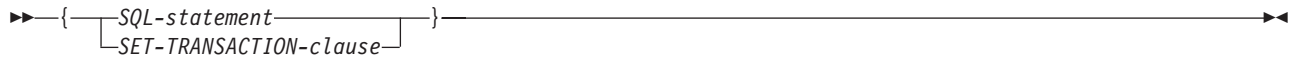
Related reference

“SQLJ executable-clause” on page 291

SQLJ statement-clause

A statement clause contains an SQL statement or a SET TRANSACTION clause.

Syntax



Description

SQL-statement

You can include SQL statements in Table 81 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 81. Valid SQL statements in an SQLJ statement clause

Statement	Applicable data sources
ALTER DATABASE	1 on page 295, 2 on page 295
ALTER FUNCTION	1 on page 295, 2 on page 295, 3 on page 295
ALTER INDEX	1 on page 295, 2 on page 295, 3 on page 295
ALTER PROCEDURE	1 on page 295, 2 on page 295, 3 on page 295
ALTER STOGROUP	1 on page 295, 2 on page 295
ALTER TABLE	1 on page 295, 2 on page 295, 3 on page 295
ALTER TABLESPACE	1 on page 295, 2 on page 295
CALL	1 on page 295, 2 on page 295, 3 on page 295
COMMENT ON	1 on page 295, 2 on page 295
COMMIT	1 on page 295, 2 on page 295, 3 on page 295
Compound SQL (BEGIN ATOMIC...END)	2 on page 295
CREATE ALIAS	1 on page 295, 2 on page 295
CREATE DATABASE	1 on page 295, 2 on page 295, 3a on page 295
CREATE DISTINCT TYPE	1 on page 295, 2 on page 295, 3 on page 295
CREATE FUNCTION	1 on page 295, 2 on page 295, 3 on page 295
CREATE GLOBAL TEMPORARY TABLE	1 on page 295, 2 on page 295
CREATE TEMP TABLE	3 on page 295
CREATE INDEX	1 on page 295, 2 on page 295, 3 on page 295
CREATE PROCEDURE	1 on page 295, 2 on page 295, 3 on page 295
CREATE STOGROUP	1 on page 295, 2 on page 295
CREATE SYNONYM	1 on page 295, 2 on page 295, 3 on page 295
CREATE TABLE	1 on page 295, 2 on page 295, 3 on page 295
CREATE TABLESPACE	1 on page 295, 2 on page 295
CREATE TYPE (cursor)	2 on page 295
CREATE TRIGGER	1 on page 295, 2 on page 295, 3 on page 295
CREATE VIEW	1 on page 295, 2 on page 295, 3 on page 295
DECLARE GLOBAL TEMPORARY TABLE	1 on page 295, 2 on page 295
DELETE	1 on page 295, 2 on page 295, 3 on page 295

Table 81. Valid SQL statements in an SQLJ statement clause (continued)

Statement	Applicable data sources
DROP ALIAS	1 on page 295, 2 on page 295
DROP DATABASE	1 on page 295, 2 on page 295, 3a on page 295
DROP DISTINCT TYPE	1 on page 295, 2 on page 295
DROP TYPE	3 on page 295
DROP FUNCTION	1 on page 295, 2 on page 295, 3 on page 295
DROP INDEX	1 on page 295, 2 on page 295, 3 on page 295
DROP PACKAGE	1 on page 295, 2 on page 295
DROP PROCEDURE	1 on page 295, 2 on page 295, 3 on page 295
DROP STOGROUP	1 on page 295, 2 on page 295
DROP SYNONYM	1 on page 295, 2 on page 295, 3 on page 295
DROP TABLE	1 on page 295, 2 on page 295, 3 on page 295
DROP TABLESPACE	1 on page 295, 2 on page 295
DROP TRIGGER	1 on page 295, 2 on page 295, 3 on page 295
DROP VIEW	1 on page 295, 2 on page 295, 3 on page 295
FETCH	1 on page 295, 2 on page 295, 3 on page 295
GRANT	1 on page 295, 2 on page 295, 3 on page 295
INSERT	1 on page 295, 2 on page 295, 3 on page 295
LOCK TABLE	1 on page 295, 2 on page 295, 3 on page 295
MERGE	1 on page 295, 2 on page 295
REVOKE	1 on page 295, 2 on page 295, 3 on page 295
ROLLBACK	1 on page 295, 2 on page 295, 3 on page 295
SAVEPOINT	1 on page 295, 2 on page 295, 3 on page 295
SELECT INTO	1 on page 295, 2 on page 295, 3 on page 295
SET CURRENT APPLICATION ENCODING SCHEME	1 on page 295
SET CURRENT DEBUG MODE	1 on page 295
SET CURRENT DEFAULT TRANSFORM GROUP	2 on page 295
SET CURRENT DEGREE	1 on page 295, 2 on page 295
SET CURRENT EXPLAIN MODE	2 on page 295
SET CURRENT EXPLAIN SNAPSHOT	2 on page 295
SET CURRENT ISOLATION	1 on page 295, 2 on page 295
SET CURRENT LOCALE LC_CTYPE	1 on page 295
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	1 on page 295, 2 on page 295
SET CURRENT OPTIMIZATION HINT	1 on page 295, 2 on page 295
SET CURRENT PACKAGE PATH	1 on page 295
SET CURRENT PACKAGESET (USER is not supported)	1 on page 295, 2 on page 295
SET CURRENT PRECISION	1 on page 295, 2 on page 295
SET CURRENT QUERY OPTIMIZATION	2 on page 295
SET CURRENT REFRESH AGE	1 on page 295, 2 on page 295
SET CURRENT ROUTINE VERSION	1 on page 295

Table 81. Valid SQL statements in an SQLJ statement clause (continued)

Statement	Applicable data sources
SET CURRENT RULES	1
SET CURRENT SCHEMA	2
SET CURRENT SQLID	1
SET PATH	1, 2
TRUNCATE	1
UPDATE	1, 2, 3

Note: The SQL statement applies to connections to the following data sources:

1. DB2 for z/OS
2. DB2 Database for Linux, UNIX, and Windows
3. IBM Informix Dynamic Server
 - a. IBM Informix Dynamic Server, 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 148

Related reference

“SQLJ executable-clause” on page 291

“SQLJ SET-TRANSACTION-clause”

 Statements (SQL Reference)

SQLJ SET-TRANSACTION-clause

The SET TRANSACTION clause sets the isolation level for the current unit of work.

Syntax



Description

ISOLATION LEVEL

Specifies one of the following isolation levels:

READ COMMITTED

Specifies that the current DB2 isolation level is cursor stability.

READ UNCOMMITTED

Specifies that the current DB2 isolation level is uncommitted read.

REPEATABLE READ

Specifies that the current DB2 isolation level is read stability.

SERIALIZABLE

Specifies that the current DB2 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 148

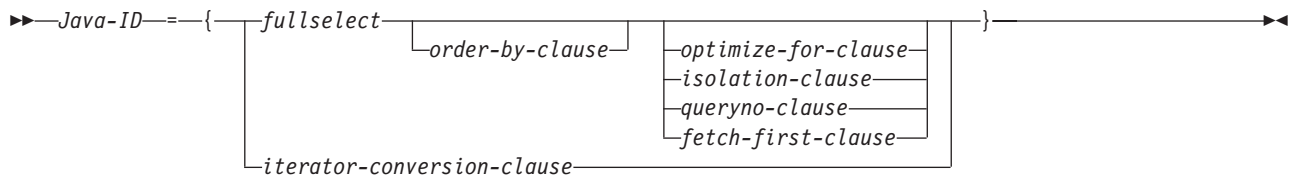
Related reference

"SQLJ statement-clause" on page 292

SQLJ assignment-clause

The assignment clause assigns the result of an SQL operation to a Java variable.

Syntax



Description

Java-ID

Identifies an iterator that was declared previously as an instance of an iterator class.

fullselect

Generates a result table.

iterator-conversion-clause

See "SQLJ iterator-conversion-clause" for a description of this clause.

Usage notes

- If the object that is identified by *Java-ID* is a positioned iterator, the number of columns in the result set must match the number of columns in the iterator. In addition, the data type of each column in the result set must be compatible with the data type of the corresponding column in the iterator. See "Java, JDBC, and SQL data types" for a list of compatible Java and SQL data types.
- If the object that is identified by *Java-ID* is a named iterator, the name of each accessor method must match, except for case, the name of a column in the result set. In addition, the data type of the object that an accessor method returns must be compatible with the data type of the corresponding column in the result set.
- You can put an assignment clause anywhere in a Java program that a Java assignment statement can appear. However, you cannot put an assignment clause where a Java assignment expression can appear. For example, you cannot specify an assignment clause in the control list of a for statement.

Related concepts

“SQLJ and JDBC in the same application” on page 133

Related reference

“SQLJ executable-clause” on page 291

“SQLJ iterator-conversion-clause”

SQLJ iterator-conversion-clause

The iterator conversion clause converts a JDBC `ResultSet` to an iterator.

Syntax

►►—`CAST—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 reference

“SQLJ assignment-clause” on page 296

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 82. Summary of `sqlj.runtime` interfaces

Interface name	Purpose
<code>ConnectionContext</code>	Manages the SQL operations that are performed during a connection to a data source.
<code>ForUpdate</code>	Implemented by iterators that are used in a positioned UPDATE or DELETE statement.

Table 82. Summary of `sqlj.runtime` interfaces (continued)

Interface name	Purpose
<code>NamedIterator</code>	Implemented by iterators that are declared as named iterators.
<code>PositionedIterator</code>	Implemented by iterators that are declared as positioned iterators.
<code>ResultSetIterator</code>	Implemented by all iterators to allow query results to be processed using a JDBC <code>ResultSet</code> .
<code>Scrollable</code>	Provides a set of methods for manipulating scrollable iterators.

sqlj.runtime classes

The following table summarizes the classes in `sqlj.runtime`.

Table 83. Summary of `sqlj.runtime` classes

Class name	Purpose
<code>AsciiStream</code>	A class for handling an input stream whose bytes should be interpreted as ASCII.
<code>BinaryStream</code>	A class for handling an input stream whose bytes should be interpreted as binary.
<code>CharacterStream</code>	A class for handling an input stream whose bytes should be interpreted as Character.
<code>DefaultRuntime</code>	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.
<code>ExecutionContext</code>	Implemented when an SQLJ execution context is declared, to control the execution of SQL operations.
<code>RuntimeContext</code>	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.
<code>SQLException</code>	Derived from the <code>java.sql.SQLException</code> class. An <code>sqlj.runtime.SQLException</code> is thrown when an SQL NULL value is fetched into a host identifier with a Java primitive type.
<code>StreamWrapper</code>	Wraps a <code>java.io.InputStream</code> instance.
<code>UnicodeStream</code>	A class for handling an input stream whose bytes should be interpreted as Unicode.

sqlj.runtime.ConnectionContext interface

The `sqlj.runtime.ConnectionContext` interface provides a set of methods that manage SQL operations that are performed during a session with a specific data source.

Translation of an SQLJ connection declaration clause causes SQLJ to create a connection context class. A connection context object maintains a JDBC Connection object on which dynamic SQL operations can be performed. A connection context object also maintains a default `ExecutionContext` object.

Variables

CLOSE_CONNECTION

Format:

```
public static final boolean CLOSE_CONNECTION=true;
```

A constant that can be passed to the close method. It indicates that the underlying JDBC Connection object should be closed.

KEEP_CONNECTION

Format:

```
public static final boolean KEEP_CONNECTION=false;
```

A constant that can be passed to the close method. It indicates that the underlying JDBC Connection object should not be closed.

Methods

close()

Format:

```
public abstract void close() throws SQLException
```

Performs the following functions:

- Releases all resources that are used by the given connection context object
- Closes any open ConnectedProfile objects
- Closes the underlying JDBC Connection object

close() is equivalent to close(CLOSE_CONNECTION).

close(boolean)

Format:

```
public abstract void close (boolean close-connection)  
throws SQLException
```

Performs the following functions:

- Releases all resources that are used by the given connection context object
- Closes any open ConnectedProfile objects
- Closes the underlying JDBC Connection object, depending on the value of the *close-connection* parameter

Parameters:

close-connection

Specifies whether the underlying JDBC Connection object is closed when a connection context object is closed:

CLOSE_CONNECTION

Closes the underlying JDBC Connection object.

KEEP_CONNECTION

Does not close the underlying JDBC Connection object.

getConnectedProfile

Format:

```
public abstract ConnectedProfile getConnectedProfile(Object profileKey)  
throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

getConnection

Format:

```
public abstract Connection getConnection()
```

Returns the underlying JDBC Connection object for the given connection context object.

getExecutionContext

Format:

```
public abstract ExecutionContext 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 Ctx setDefaultContext(Ctx default-context)
```

Sets the default connection context object for the Ctx class.

Recommendation: Do not use this method for multithreaded applications. Instead, use explicit contexts.

Related concepts

“SQLJ and JDBC in the same application” on page 133

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.

Related tasks

“Performing positioned UPDATE and DELETE operations in an SQLJ application” on page 106

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.

Related tasks

“Using a named iterator in an SQLJ application” on page 117

Related reference

“sqlj.runtime.ResultSetIterator interface”

sqlj.runtime.PositionedIterator interface

The sqlj.runtime.PositionedIterator interface is implemented when an SQLJ application executes an iterator declaration clause for a positioned iterator.

The order of columns in a positioned iterator must be the same as the order of columns in the result table, and a positioned iterator does not include result table column names.

Methods

sqlj.runtime.PositionedIterator inherits all **ResultSetIterator** methods, and includes the following additional method:

endFetch

Format:

```
public abstract boolean endFetch() throws SQLException
```

Returns a value of true if the iterator is not positioned on a row. Returns a value of false if the iterator is positioned on a row.

Related tasks

“Using a positioned iterator in an SQLJ application” on page 119

Related reference

“sqlj.runtime.ResultSetIterator interface”

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 Dynamic Server.

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 Dynamic Server.

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 Dynamic Server.

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 Dynamic Server.

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.

Related reference

“sqlj.runtime.NamedIterator interface” on page 303

“sqlj.runtime.PositionedIterator interface” on page 304

“sqlj.runtime.ExecutionContext class” on page 312

sqlj.runtime.Scrollable interface

sqlj.runtime.Scrollable provides methods to move around in the result table and to check the position in the result table.

sqlj.runtime.Scrollable is implemented when a scrollable iterator is declared.

Methods

absolute(int)

Format:

`public abstract boolean absolute (int n) throws SQLException`

Moves the iterator to a specified row.

If $n > 0$, positions the iterator on row n of the result table. If $n < 0$, and m is the number of rows in the result table, positions the iterator on row $m+n+1$ of the result table.

If the absolute value of n is greater than the number of rows in the result table, positions the cursor after the last row if n is positive, or before the first row if n is negative.

absolute(0) is the same as beforeFirst(). absolute(1) is the same as first(). absolute(-1) is the same as last().

Returns true if the iterator is on a row. Otherwise, returns false.

afterLast()

Format:

```
public abstract void afterLast() throws SQLException
```

Moves the iterator after the last row of the result table.

beforeFirst()

Format:

```
public abstract void beforeFirst() throws SQLException
```

Moves the iterator before the first row of the result table.

first()

Format:

```
public abstract boolean first() throws SQLException
```

Moves the iterator to the first row of the result table.

Returns true if the iterator is on a row. Otherwise, returns false.

getFetchDirection()

Format:

```
public abstract int getFetchDirection() throws SQLException
```

Returns the fetch direction of the iterator. Possible values are:

sqlj.runtime.ResultSetIterator.FETCH_FORWARD

Rows are processed in a forward direction, from first to last.

sqlj.runtime.ResultSetIterator.FETCH_REVERSE

Rows are processed in a backward direction, from last to first.

sqlj.runtime.ResultSetIterator.FETCH_UNKNOWN

The order of processing is not known.

isAfterLast()

Format:

```
public abstract boolean isAfterLast() throws SQLException
```

Returns true if the iterator is positioned after the last row of the result table. Otherwise, returns false.

isBeforeFirst()

Format:

```
public abstract boolean isBeforeFirst() throws SQLException
```

Returns true if the iterator is positioned before the first row of the result table. Otherwise, returns false.

isFirst()

Format:

```
public abstract boolean isFirst() throws SQLException
```

Returns true if the iterator is positioned on the first row of the result table. Otherwise, returns false.

isLast()

Format:

```
public abstract boolean isLast() throws SQLException
```

Returns true if the iterator is positioned on the last row of the result table. Otherwise, returns false.

last()

Format:

```
public abstract boolean last() throws SQLException
```

Moves the iterator to the last row of the result table.

Returns true if the iterator is on a row. Otherwise, returns false.

previous()

Format:

```
public abstract boolean previous() throws SQLException
```

Moves the iterator to the previous row of the result table.

Returns true if the iterator is on a row. Otherwise, returns false.

relative(int)

Format:

```
public abstract boolean relative(int n) throws SQLException
```

If $n > 0$, positions the iterator on the row that is n rows after the current row. If $n < 0$, positions the iterator on the row that is n rows before the current row. If $n = 0$, positions the iterator on the current row.

The cursor must be on a valid row of the result table before you can use this method. If the cursor is before the first row or after the last throw, the method throws an `SQLException`.

Suppose that m is the number of rows in the result table and x is the current row number in the result table. If $n > 0$ and $x + n > m$, the iterator is positioned after the last row. If $n < 0$ and $x + n < 1$, the iterator is positioned before the first row.

Returns true if the iterator is on a row. Otherwise, returns false.

setFetchDirection(int)

Format:

```
public abstract void setFetchDirection (int) throws SQLException
```

Gives the SQLJ runtime environment a hint as to the direction in which rows of this iterator object are processed. Possible values are:

sqlj.runtime.ResultSetIterator.FETCH_FORWARD

Rows are processed in a forward direction, from first to last.

sqlj.runtime.ResultSetIterator.FETCH_REVERSE

Rows are processed in a backward direction, from last to first.

sqlj.runtime.ResultSetIterator.FETCH_UNKNOWN

The order of processing is not known.

Related tasks

“Using scrollable iterators in an SQLJ application” on page 124

sqlj.runtime.AsciiStream class

The `sqlj.runtime.AsciiStream` class is for an input stream of ASCII data with a specified length.

The `sqlj.runtime.AsciiStream` class is derived from the `java.io.InputStream` class, and extends the `sqlj.runtime.StreamWrapper` class. SQLJ interprets the bytes in an `sqlj.runtime.AsciiStream` object as ASCII characters. An `InputStream` object with ASCII characters needs to be passed as a `sqlj.runtime.AsciiStream` object.

Constructors

AsciiStream(InputStream)

Format:

```
public AsciiStream(java.io.InputStream input-stream)
```

Creates an ASCII `java.io.InputStream` object with an unspecified length.

Parameters:

input-stream

The `InputStream` object that SQLJ interprets as an `AsciiStream` object.

AsciiStream(InputStream, int)

Format:

```
public AsciiStream(java.io.InputStream input-stream, int length)
```

Creates an ASCII `java.io.InputStream` object with a specified length.

Parameters:

input-stream

The `InputStream` object that SQLJ interprets as an `AsciiStream` object.

length

The length of the `InputStream` object that SQLJ interprets as an `AsciiStream` object.

Related reference

“`sqlj.runtime.BinaryStream` class”

“`sqlj.runtime.CharacterStream` class” on page 311

“`sqlj.runtime.StreamWrapper` class” on page 320

“`sqlj.runtime.UnicodeStream` class” on page 321

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.

Related reference

“sqlj.runtime.AsciiStream class” on page 309

“sqlj.runtime.CharacterStream class”

“sqlj.runtime.StreamWrapper class” on page 320

“sqlj.runtime.UnicodeStream class” on page 321

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.

Related reference

“`sqlj.runtime.AsciiStream` class” on page 309

“`sqlj.runtime.BinaryStream` class” on page 310

“`sqlj.runtime.StreamWrapper` class” on page 320

“`sqlj.runtime.UnicodeStream` class” on page 321

sqlj.runtime.ExecutionContext class

The `sqlj.runtime.ExecutionContext` class is defined for execution contexts. An execution context is used to control the execution of SQL statements.

Variables

ADD_BATCH_COUNT

Format:

```
public static final int ADD_BATCH_COUNT
```

A constant that can be returned by the `getUpdateCount` method. It indicates that the previous statement was not executed but was added to the existing statement batch.

AUTO_BATCH

Format:

```
public static final int AUTO_BATCH
```

A constant that can be passed to the `setBatchLimit` method. It indicates that implicit batch execution should be performed, and that SQLJ should determine the batch size.

EXEC_BATCH_COUNT

Format:

```
public static final int EXEC_BATCH_COUNT
```

A constant that can be returned from the `getUpdateCount` method. It indicates that a statement batch was just executed.

EXCEPTION_COUNT

Format:

```
public static final int EXCEPTION_COUNT
```

A constant that can be returned from the `getUpdateCount` method. It indicates that an exception was thrown before the previous execution completed, or that no operation has been performed on the execution context object.

NEW_BATCH_COUNT

Format:

```
public static final int NEW_BATCH_COUNT
```

A constant that can be returned from the `getUpdateCount` method. It indicates that the previous statement was not executed, but was added to a new statement batch.

QUERY_COUNT

Format:

```
public static final int QUERY_COUNT
```

A constant that can be passed to the `setBatchLimit` method. It indicates that the previous execution produced a result set.

UNLIMITED_BATCH

Format:

```
public static final int UNLIMITED_BATCH
```

A constant that can be returned from the `getUpdateCount` method. It indicates that statements should continue to be added to a statement batch, regardless of the batch size.

Constructors:

ExecutionContext

Format:

```
public ExecutionContext()
```

Creates an `ExecutionContext` instance.

Methods

cancel

Format:

```
public void cancel() throws SQLException
```

Cancels an SQL operation that is currently being executed by a thread that uses the execution context object. If there is a pending statement batch on the execution context object, the statement batch is canceled and cleared.

The cancel method throws an SQLException if the statement cannot be canceled.

execute

Format:

```
public boolean execute ( ) throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

executeBatch

Format:

```
public synchronized int[] executeBatch() throws SQLException
```

Executes the pending statement batch and returns an array of update counts. If no pending statement batch exists, null is returned. When this method is called, the statement batch is cleared, even if the call results in an exception.

Each element in the returned array can be one of the following values:

- 2 This value indicates that the SQL statement executed successfully, but the number of rows that were updated could not be determined.
- 3 This value indicates that the SQL statement failed.

Other integer

This value is the number of rows that were updated by the statement.

The executeBatch method throws an SQLException if a database error occurs while the statement batch executes.

executeQuery

Format:

```
public ResultSet executeQuery ( ) throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

executeUpdate

Format:

```
public int executeUpdate() throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

getBatchLimit

Format:

```
synchronized public int getBatchLimit()
```

Returns the number of statements that are added to a batch before the batch is implicitly executed.

The returned value is one of the following values:

UNLIMITED_BATCH

This value indicates that the batch size is unlimited.

AUTO_BATCH

This value indicates that the batch size is finite but unknown.

Other integer

The current batch limit.

getBatchUpdateCounts

Format:

```
public synchronized int[] getBatchUpdateCounts()
```

Returns an array that contains the number of rows that were updated by each statement that successfully executed in a batch. The order of elements in the array corresponds to the order in which statements were inserted into the batch. Returns null if no statements in the batch completed successfully.

Each element in the returned array can be one of the following values:

- 2 This value indicates that the SQL statement executed successfully, but the number of rows that were updated could not be determined.
- 3 This value indicates that the SQL statement failed.

Other integer

This value is the number of rows that were updated by the statement.

getFetchDirection

Format:

```
synchronized public int getFetchDirection() throws SQLException
```

Returns the current fetch direction for scrollable iterator objects that were generated from the given execution context. If a fetch direction was not set for the execution context, `sqlj.runtime.ResultSetIterator.FETCH_FORWARD` is returned.

getFetchSize

Format:

```
synchronized public int getFetchSize() throws SQLException
```

Returns the number of rows that should be fetched by SQLJ when more rows are needed. This value applies only to iterator objects that were generated from the given execution context. The returned value is the value that was set by the `setFetchSize` method, or 0 if no value was set by `setFetchSize`.

getMaxFieldSize

Format:

```
public synchronized int getMaxFieldSize()
```

Returns the maximum number of bytes that are returned for any string (character, graphic, or varying-length binary) column in queries that use the given execution context. If this limit is exceeded, SQLJ discards the remaining bytes. A value of 0 means that the maximum number of bytes is unlimited.

getMaxRows

Format:

```
public synchronized int getMaxRows()
```

Returns the maximum number of rows that are returned for any query that uses the given execution context. If this limit is exceeded, SQLJ discards the remaining rows. A value of 0 means that the maximum number of rows is unlimited.

getNextResultSet()

Format:

```
public ResultSet getNextResultSet() throws SQLException
```

After a stored procedure call, returns a result set from the stored procedure.

A null value is returned if any of the following conditions are true:

- There are no more result sets to be returned.
- The stored procedure call did not produce any result sets.
- A stored procedure call has not been executed under the execution context.

When you invoke `getNextResultSet()`, SQLJ closes the currently-open result set and advances to the next result set.

If an error occurs during a call to `getNextResultSet`, resources for the current JDBC `ResultSet` object are released, and an `SQLException` is thrown.

Subsequent calls to `getNextResultSet` return null.

getNextResultSet(int)

Formats:

```
public ResultSet getNextResultSet(int current)
```

After a stored procedure call, returns a result set from the stored procedure.

A null value is returned if any of the following conditions are true:

- There are no more result sets to be returned.
- The stored procedure call did not produce any result sets.
- A stored procedure call has not been executed under the execution context.

If an error occurs during a call to `getNextResultSet`, resources for the current JDBC `ResultSet` object are released, and an `SQLException` is thrown.

Subsequent calls to `getNextResultSet` return null.

Parameters:

current

Indicates what SQLJ does with the currently open result set before it advances to the next result set:

java.sql.Statement.CLOSE_CURRENT_RESULT

Specifies that the current `ResultSet` object is closed when the next `ResultSet` object is returned.

java.sql.Statement.KEEP_CURRENT_RESULT

Specifies that the current `ResultSet` object stays open when the next `ResultSet` object is returned.

java.sql.Statement.CLOSE_ALL_RESULTS

Specifies that all open `ResultSet` objects are closed when the next `ResultSet` object is returned.

getQueryTimeout

Format:

```
public synchronized int getQueryTimeout()
```

Returns the maximum number of seconds that SQL operations that use the given execution context object can execute. If an SQL operation exceeds the limit, an `SQLException` is thrown. The returned value is the value that was set by the `setQueryTimeout` method, or 0 if no value was set by `setQueryTimeout`. 0 means that execution time is unlimited.

getUpdateCount

Format:

```
public abstract int getUpdateCount() throws SQLException
```

Returns:

ExecutionContext.ADD_BATCH_COUNT

If the statement was added to an existing batch.

ExecutionContext.NEW_BATCH_COUNT

If the statement was the first statement in a new batch.

ExecutionContext.EXCEPTION_COUNT

If the previous statement generated an SQLException, or no previous statement was executed.

ExecutionContext.EXEC_BATCH_COUNT

If the statement was part of a batch, and the batch was executed.

ExecutionContext.QUERY_COUNT

If the previous statement created an iterator object or JDBC ResultSet.

Other integer

If the statement was executed rather than added to a batch. This value is the number of rows that were updated by the statement.

getWarnings

Format:

```
public synchronized SQLWarning getWarnings()
```

Returns the first warning that was reported by the last SQL operation that was executed using the given execution context. Subsequent warnings are chained to the first warning. If no warnings occurred, null is returned.

getWarnings is used to retrieve positive SQLCODEs.

isBatching

Format:

```
public synchronized boolean isBatching()
```

Returns true if batching is enabled for the execution context. Returns false if batching is disabled.

registerStatement

Format:

```
public RTStatement registerStatement(ConnectionContext connCtx,  
    Object profileKey, int stmtNdx)  
    throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

releaseStatement

Format:

```
public void releaseStatement() throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

setBatching

Format:


```
public synchronized void setBatching(boolean batching)
```

Parameters:

batching

Indicates whether batchable statements that are registered with the given execution context can be added to a statement batch:

true

Statements can be added to a statement batch.

false

Statements are executed individually.

setBatching affects only statements that occur in the program after setBatching is called. It does not affect previous statements or an existing statement batch.

setBatchLimit

Format:

```
public synchronized void setBatchLimit(int batch-size)
```

Sets the maximum number of statements that are added to a batch before the batch is implicitly executed.

Parameters:

batch-size

One of the following values:

ExecutionContext.UNLIMITED_BATCH

Indicates that implicit execution occurs only when SQLJ encounters a statement that is batchable but incompatible, or not batchable. Setting this value is the same as not invoking setBatchLimit.

ExecutionContext.AUTO_BATCH

Indicates that implicit execution occurs when the number of statements in the batch reaches a number that is set by SQLJ.

Positive integer

The number of statements that are added to the batch before SQLJ executes the batch implicitly. The batch might be executed before this many statements have been added if SQLJ encounters a statement that is batchable but incompatible, or not batchable.

setBatchLimit affects only statements that occur in the program after setBatchLimit is called. It does not affect an existing statement batch.

setFetchDirection

Format:

```
public synchronized void setFetchDirection(int direction) throws SQLException
```

Gives SQLJ a hint as to the current fetch direction for scrollable iterator objects that were generated from the given execution context.

Parameters:

direction

One of the following values:

sqlj.runtime.ResultSetIterator.FETCH_FORWARD

Rows are fetched in a forward direction. This is the default.

sqlj.runtime.ResultSetIterator.FETCH_REVERSE

Rows are fetched in a backward direction.

sqlj.runtime.ResultSetIterator.FETCH_UNKNOWN

The order of fetching is unknown.

Any other input value results in an SQLException.

setFetchSize

Format:

synchronized public void setFetchSize(int *number-of-rows*) throws SQLException

Gives SQLJ a hint as to the number of rows that should be fetched when more rows are needed.

Parameters:

number-of-rows

The expected number of rows that SQLJ should fetch for the iterator that is associated with the given execution context.

If *number-of-rows* is less than 0 or greater than the maximum number of rows that can be fetched, an SQLException is thrown.

setMaxFieldSize

Format:

public void setMaxFieldSize(int *max-bytes*)

Specifies the maximum number of bytes that are returned for any string (character, graphic, or varying-length binary) column in queries that use the given execution context. If this limit is exceeded, SQLJ discards the remaining bytes.

Parameters:

max-bytes

The maximum number of bytes that SQLJ should return from a BINARY, VARBINARY, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC column. A value of 0 means that the number of bytes is unlimited. 0 is the default.

setMaxRows

Format:

public synchronized void setMaxRows(int *max-rows*)

Specifies the maximum number of rows that are returned for any query that uses the given execution context. If this limit is exceeded, SQLJ discards the remaining rows.

Parameters:

max-rows

The maximum number of rows that SQLJ should return for a query that uses the given execution context. A value of 0 means that the number of rows is unlimited. 0 is the default.

setQueryTimeout

Format:

public synchronized void setQueryTimeout(int *timeout-value*)

Specifies the maximum number of seconds that SQL operations that use the given execution context object can execute. If an SQL operation exceeds the limit, an `SQLException` is thrown.

Parameters:

timeout-value

The maximum number of seconds that SQL operations that use the given execution context object can execute. 0 means that execution time is unlimited. 0 is the default. For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS database servers, 0 is the only valid value.

Related tasks

“Controlling the execution of SQL statements in SQLJ” on page 136

Related reference

“`sqlj.runtime.ResultSetIterator` interface” on page 304

`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'.

Related tasks

“Making batch updates in SQLJ applications” on page 113

`sqlj.runtime.StreamWrapper` class

The `sqlj.runtime.StreamWrapper` class wraps a `java.io.InputStream` instance and extends the `java.io.InputStream` class.

The `sqlj.runtime.AsciiStream`, `sqlj.runtime.BinaryStream`, and `sqlj.runtime.UnicodeStream` classes extend `sqlj.runtime.StreamWrapper`. `sqlj.runtime.StreamWrapper` supports methods for specifying the length of `sqlj.runtime.AsciiStream`, `sqlj.runtime.BinaryStream`, and `sqlj.runtime.UnicodeStream` objects.

Constructors

`StreamWrapper(InputStream)`

Format:

```
protected StreamWrapper(InputStream input-stream)
```

Creates an `sqlj.runtime.StreamWrapper` object with an unspecified length.

Parameters:

input-stream

The `InputStream` object that the `sqlj.runtime.StreamWrapper` object wraps.

`StreamWrapper(InputStream, int)`

Format:

```
protected StreamWrapper(java.io.InputStream input-stream, int length)
```

Creates an `sqlj.runtime.StreamWrapper` object with a specified length.

Parameters:

input-stream

The `InputStream` object that the `sqlj.runtime.StreamWrapper` object wraps.

length

The length of the `InputStream` object in bytes.

Methods

getInputStream

Format:

```
public InputStream getInputStream()
```

Returns the underlying `InputStream` object that is wrapped by the `StreamWrapper` object.

getLength

Format:

```
public void getLength()
```

Returns the length in bytes of the wrapped `InputStream` object, as specified by the constructor or in the last call to `setLength`.

setLength

Format:

```
public void setLength (int length)
```

Sets the number of bytes that are read from the wrapped `InputStream` object when the object is passed as an input argument to an SQL operation.

Parameters:

length

The number of bytes that are read from the wrapped `InputStream` object.

Related reference

“`sqlj.runtime.AsciiStream` class” on page 309

“`sqlj.runtime.BinaryStream` class” on page 310

“`sqlj.runtime.CharacterStream` class” on page 311

“`sqlj.runtime.UnicodeStream` class”

sqlj.runtime.UnicodeStream class

The `sqlj.runtime.UnicodeStream` class is for an input stream of Unicode data with a specified length.

The `sqlj.runtime.UnicodeStream` class is derived from the `java.io.InputStream` class, and extends the `sqlj.runtime.StreamWrapper` class. SQLJ interprets the bytes in an `sqlj.runtime.UnicodeStream` object as Unicode characters. An `InputStream` object with Unicode characters needs to be passed as a `sqlj.runtime.UnicodeStream` object.

Constructors

UnicodeStream(InputStream)

Format:

```
public UnicodeStream(java.io.InputStream input-stream)
```

Creates a Unicode `java.io.InputStream` object with an unspecified length.

Parameters:

input-stream

The InputStream object that SQLJ interprets as an UnicodeStream object.

UnicodeStream(InputStream, int)

Format:

```
public UnicodeStream(java.io.InputStream input-stream, int length)
```

Creates a Unicode java.io.InputStream object with a specified length.

Parameters:

input-stream

The InputStream object that SQLJ interprets as an UnicodeStream object.

length

The length of the InputStream object that SQLJ interprets as an UnicodeStream object.

Related reference

“sqlj.runtime.AsciiStream class” on page 309

“sqlj.runtime.BinaryStream class” on page 310

“sqlj.runtime.CharacterStream class” on page 311

“sqlj.runtime.StreamWrapper class” on page 320

IBM Data Server Driver for JDBC and SQLJ extensions to JDBC

The IBM Data Server Driver for JDBC and SQLJ provides a set of extensions to the support that is provided by the JDBC specification.

To use IBM Data Server Driver for JDBC and SQLJ-only methods in classes that have corresponding, standard classes, cast an instance of the related, standard JDBC class to an instance of the IBM Data Server Driver for JDBC and SQLJ-only class. For example:

```
javax.sql.DataSource ds =  
    new com.ibm.db2.jcc.DB2SimpleDataSource();  
((com.ibm.db2.jcc.DB2BaseDataSource) ds).setServerName("sysmvs1.stl.ibm.com");
```

Table 84 summarizes the IBM Data Server Driver for JDBC and SQLJ-only interfaces.

Table 84. Summary of IBM Data Server Driver for JDBC and SQLJ-only interfaces provided by the IBM Data Server Driver for JDBC and SQLJ

Interface name	Applicable data sources	Purpose
DB2CallableStatement	2 on page 323	Extends the java.sql.CallableStatement and the com.ibm.db2.jcc.DB2PreparedStatement interfaces.
DB2Connection	1 on page 323, 2 on page 323, 3 on page 323	Extends the java.sql.Connection interface.
DB2DatabaseMetaData	1 on page 323, 2 on page 323, 3 on page 323	Extends the java.sql.DatabaseMetaData interface.
DB2Diagnosable	1 on page 323, 2 on page 323, 3 on page 323	Provides a mechanism for getting DB2 diagnostics from a DB2 SQLException.
DB2PreparedStatement	1 on page 323, 2 on page 323, 3 on page 323	Extends the com.ibm.db2.jcc.DB2Statement and java.sql.PreparedStatement interfaces.
DB2RowID	1 on page 323, 2 on page 323	Used for declaring Java objects for use with the ROWID data type.

Table 84. Summary of IBM Data Server Driver for JDBC and SQLJ-only interfaces provided by the IBM Data Server Driver for JDBC and SQLJ (continued)

Interface name	Applicable data sources	Purpose
DB2Statement	1, 2, 3	Extends the <code>java.sql.Statement</code> interface.
DB2SystemMonitor	1, 2, 3	Used for collecting system monitoring data for a connection.
DB2TraceManagerMXBean	1, 2, 3	Provides the MBean interface for the remote trace controller.
DB2Xml	1, 2	Used for updating data in XML columns and retrieving data from XML columns.
DBBatchUpdateException	1, 2, 3	Used for retrieving error information about batch execution of statements that return automatically generated keys.

Note: The interface applies to connections to the following data sources:

1. DB2 for z/OS
2. DB2 Database for Linux, UNIX, and Windows
3. IBM Informix Dynamic Server

Table 85 summarizes the IBM Data Server Driver for JDBC and SQLJ-only classes.

Table 85. Summary of IBM Data Server Driver for JDBC and SQLJ-only classes provided by the IBM Data Server Driver for JDBC and SQLJ

Class name	Applicable data sources	Purpose
DB2Administrator (DB2 Database for Linux, UNIX, and Windows only)	2 on page 324	Instances of the DB2Administrator class are used to retrieve DB2CataloguedDatabase objects.
DB2BaseDataSource	1 on page 324, 2 on page 324, 3 on page 324	The abstract data source parent class for all IBM Data Server Driver for JDBC and SQLJ-specific implementations of <code>javax.sql.DataSource</code> , <code>javax.sql.ConnectionPoolDataSource</code> , and <code>javax.sql.XADataSource</code> .
DB2BlobFileReference	1 on page 324	A subclass of DB2FileReference for creating BLOB file reference variable objects.
DB2CataloguedDatabase	2 on page 324	Contains methods that retrieve information about a local DB2 Database for Linux, UNIX, and Windows database.
DB2ClientRerouteServerList	1 on page 324, 2 on page 324	Implements the <code>java.io.Serializable</code> and <code>javax.naming.Referenceable</code> interfaces.
DB2ClobFileReference	1 on page 324	A subclass of DB2FileReference for creating CLOB file reference variable objects.
DB2ConnectionPoolDataSource	1 on page 324, 2 on page 324, 3 on page 324	A factory for PooledConnection objects.
DB2ExceptionFormatter	1 on page 324, 2 on page 324, 3 on page 324	Contains methods for printing diagnostic information to a stream.
DB2FileReference	1 on page 324	Provides methods for inserting data into tables from file reference variables.
DB2JCCPlugin	2 on page 324	The abstract class for implementation of JDBC security plug-ins.

Table 85. Summary of IBM Data Server Driver for JDBC and SQLJ-only classes provided by the IBM Data Server Driver for JDBC and SQLJ (continued)

Class name	Applicable data sources	Purpose
DB2PooledConnection	1, 2, 3	Provides methods that an application server can use to switch users on a preexisting trusted connection.
DB2PoolMonitor	1, 2	Provides methods for monitoring the global transport objects pool for the connection concentrator and Sysplex workload balancing.
DB2SimpleDataSource	1, 2, 3	Extends the DataBaseDataSource class. Does not support connection pooling or distributed transactions.
DB2Sqlca	1, 2, 3	An encapsulation of the DB2 SQLCA.
DB2TraceManager	1, 2, 3	Controls the global log writer.
DB2Types	1 on page 323	Defines data type constants.
DB2XADataSource	1, 2, 3	A factory for XADataSource objects. An object that implements this interface is registered with a naming service that is based on the Java Naming and Directory Interface (JNDI).
DB2XmlAsBlobFileReference	1	A subclass of DB2FileReference for creating XML AS BLOB file reference variable objects.
DB2XmlAsClobFileReference	1	A subclass of DB2FileReference for creating XML AS CLOB file reference variable objects.

Note: The class applies to connections to the following data sources:

1. DB2 for z/OS
2. DB2 Database for Linux, UNIX, and Windows
3. IBM Informix Dynamic Server

Related reference

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

DBBatchUpdateException interface

The `com.ibm.db2.jcc.DBBatchUpdateException` interface is used for retrieving error information about batch execution of statements that return automatically generated keys.

DBBatchUpdateException methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

`getDBGeneratedKeys`

Format:

```
public java.sql.ResultSet[] getDBGeneratedKeys()
    throws java.sql.SQLException
```

Retrieves automatically generated keys that were created when INSERT statements were executed in a batch. Each `ResultSet` object that is returned contains the automatically generated keys for a single statement in the batch. `ResultSet` objects that are null correspond to failed statements.

DB2BaseDataSource class

The `com.ibm.db2.jcc.DB2BaseDataSource` class is the abstract data source parent class for all IBM Data Server Driver for JDBC and SQLJ-specific implementations of `javax.sql.DataSource`, `javax.sql.ConnectionPoolDataSource`, and `javax.sql.XADataSource`.

`DB2BaseDataSource` implements the `java.sql.Wrapper` interface.

DB2BaseDataSource properties

The following properties are defined only for the IBM Data Server Driver for JDBC and SQLJ.

You can set all properties on a `DataSource` or in the `url` parameter in a `DriverManager.getConnection` call.

All properties **except** the following properties have a `setXXX` method to set the value of the property and a `getXXX` method to retrieve the value:

- `minTransportObjects`
- `maxTransportObjectIdleTime`
- `maxTransportObjectWaitTime`
- `dumpPool`
- `dumpPoolStatisticsOnSchedule`
- `dumpPoolStatisticsOnScheduleFile`

A `setXXX` method has this form:

```
void setProperty-name(data-type property-value)
```

A `getXXX` method has this form:

```
data-type getProperty-name()
```

Property-name is the unqualified property name. For properties that are not specific to IBM Informix Dynamic Server (IDS), the first character of the property name is capitalized. For properties that are used only by IDS, 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 86. *DB2BaseDataSource* properties and their data types

Property name	Applicable data sources	Data type
<code>com.ibm.db2.jcc.DB2BaseDataSource.accountingInterval</code>	1 on page 330	String
<code>com.ibm.db2.jcc.DB2BaseDataSource.allowNextOnExhaustedResultSet</code>	1 on page 330, 2 on page 330, 3 on page 330	int
<code>com.ibm.db2.jcc.DB2BaseDataSource.atomicMultiRowInsert</code>	1 on page 330, 2 on page 330, 3 on page 330	int
<code>com.ibm.db2.jcc.DB2BaseDataSource.blockingReadConnectionTimeout</code>	1 on page 330, 2 on page 330, 3 on page 330	int
<code>com.ibm.db2.jcc.DB2BaseDataSource.charOutputSize</code>	1 on page 330	short
<code>com.ibm.db2.jcc.DB2BaseDataSource.clientAccountingInformation</code>	1 on page 330, 2 on page 330	String

Table 86. DB2BaseDataSource properties and their data types (continued)

Property name	Applicable data sources	Data type
com.ibm.db2.jcc.DB2BaseDataSource.clientApplicationInformation	1 on page 330, 2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.clientDebugInfo (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	1 on page 330, 2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.clientProgramId	1 on page 330, 2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.clientProgramName (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	1 on page 330, 2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteAlternateServerName	1 on page 330, 2 on page 330, 3 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteAlternatePortNumber	1 on page 330, 2 on page 330, 3 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteServerListJNDIContext	1 on page 330, 2 on page 330, 3 on page 330	javax.naming.Context
com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteServerListJNDIName	1 on page 330, 2 on page 330, 3 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.clientUser (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS only)	1 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.clientWorkstation (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS only)	1 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.connectNode	2 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.currentDegree	1 on page 330, 2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.currentExplainMode	2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.currentExplainSnapshot	2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.currentFunctionPath	1 on page 330, 2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.currentLockTimeout	2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.currentMaintainedTableTypesForOptimization	1 on page 330, 2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.currentPackagePath	1 on page 330, 2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.currentPackageSet	1 on page 330, 2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.currentQueryOptimization	2 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.currentRefreshAge	1 on page 330, 2 on page 330	long
com.ibm.db2.jcc.DB2BaseDataSource.currentSchema	1 on page 330, 2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.cursorSensitivity	1 on page 330, 2 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.currentSQLID	1 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.databaseName	1 on page 330, 2 on page 330, 3 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.dateFormat	1 on page 330, 2 on page 330	int

Table 86. DB2BaseDataSource properties and their data types (continued)

Property name	Applicable data sources	Data type
com.ibm.db2.jcc.DB2BaseDataSource.decimalRoundingMode	1 on page 330, 2 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.decimalSeparator	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.decimalStringFormat	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.defaultIsolationLevel	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.deferPrepares	1 on page 330, 2 on page 330, 3 on page 330	boolean
com.ibm.db2.jcc.DB2BaseDataSource.description	1 on page 330, 2 on page 330, 3 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.downgradeHoldCursorsUnderXa	1 on page 330, 2 on page 330, 3 on page 330	boolean
com.ibm.db2.jcc.DB2BaseDataSource.driverType	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.dumpPool	3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.dumpPoolStatisticsOnSchedule	3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.dumpPoolStatisticsOnScheduleFile	3 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.enableClientAffinitiesList	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.enableNamedParameterMarkers	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.enableConnectionConcentrator	1 on page 330, 3 on page 330	boolean
com.ibm.db2.jcc.DB2BaseDataSource.enableRowsetSupport	1 on page 330, 2 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.enableSeamlessFailover	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.enableSysplexWLB	1 on page 330, 3 on page 330	boolean
com.ibm.db2.jcc.DB2BaseDataSource.encryptionAlgorithm	1 on page 330, 2 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.fetchSize	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.fullyMaterializeInputStreams	1 on page 330, 2 on page 330	boolean
com.ibm.db2.jcc.DB2BaseDataSource.fullyMaterializeLobData	1 on page 330, 2 on page 330, 3 on page 330	boolean
com.ibm.db2.jcc.DB2BaseDataSource.gssCredential	1 on page 330, 2 on page 330	Object
com.ibm.db2.jcc.DB2BaseDataSource.jdbcCollection	1 on page 330	String

Table 86. DB2BaseDataSource properties and their data types (continued)

Property name	Applicable data sources	Data type
com.ibm.db2.jcc.DB2BaseDataSource.keepDynamic	1 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.kerberosServerPrincipal	1 on page 330, 2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.loginTimeout (not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS)	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.logWriter	1 on page 330, 2 on page 330, 3 on page 330	PrintWriter
com.ibm.db2.jcc.DB2BaseDataSource.maxRetriesForClientReroute	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.maxRowsetSize (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS only)	1 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.maxTransportObjectIdleTime	3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.maxTransportObjectWaitTime	3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.maxTransportObjects	1 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.minTransportObjects	3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.optimizationProfile	2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.optimizationProfileToFlush	2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.password	1 on page 330, 2 on page 330, 3 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.pdqProperties	1 on page 330, 2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.pkList (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity)	1 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.planName (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity only)	1 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.plugin	2 on page 330	Object
com.ibm.db2.jcc.DB2BaseDataSource.pluginName	2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.portNumber	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.progressiveStreaming	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.queryDataSize	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.queryCloseImplicit	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.readOnly	1 on page 330, 2 on page 330	boolean
com.ibm.db2.jcc.DB2BaseDataSource.reportLongTypes	1 on page 330	short
com.ibm.db2.jcc.DB2BaseDataSource.resultSetHoldability	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.resultSetHoldabilityForCatalogQueries	1 on page 330, 2 on page 330	int

Table 86. DB2BaseDataSource properties and their data types (continued)

Property name	Applicable data sources	Data type
com.ibm.db2.jcc.DB2BaseDataSource.retrieveMessagesFromServerOnGetMessage	1 on page 330, 3 on page 330	boolean
com.ibm.db2.jcc.DB2BaseDataSource.retryIntervalForClientReroute	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.retryWithAlternativeSecurityMechanism (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	2 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.returnAlias	1 on page 330, 2 on page 330	short
com.ibm.db2.jcc.DB2BaseDataSource.securityMechanism	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.sendCharInputsUTF8	1 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.sendDataAsIs	1 on page 330, 2 on page 330, 3 on page 330	boolean
com.ibm.db2.jcc.DB2BaseDataSource.serverName	1 on page 330, 2 on page 330, 3 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.sqljEnableClassLoaderSpecificProfiles	1 on page 330	boolean
com.ibm.db2.jcc.DB2BaseDataSource.ssid (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS only)	1 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.sslConnection (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	1 on page 330, 2 on page 330, 3 on page 330	boolean
com.ibm.db2.jcc.DB2BaseDataSource.sslTrustStoreLocation (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	1 on page 330, 2 on page 330, 3 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.sslTrustStorePassword (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)	1 on page 330, 2 on page 330, 3 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.statementConcentrator	2 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.streamBufferSize	1 on page 330, 2 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.supportsAsynchronousXARollback	1 on page 330, 2 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.sysSchema	1 on page 330, 2 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.timeFormat	1 on page 330, 2 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.timestampFormat	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.timestampPrecisionReporting	1 on page 330, 2 on page 330, 3 on page 330	int
com.ibm.db2.jcc.DB2BaseDataSource.traceDirectory	1 on page 330, 2 on page 330, 3 on page 330	String
com.ibm.db2.jcc.DB2BaseDataSource.traceFile	1 on page 330, 2 on page 330, 3 on page 330	String

Table 86. DB2BaseDataSource properties and their data types (continued)

Property name	Applicable data sources	Data type
com.ibm.db2.jcc.DB2BaseDataSource.traceFileAppend	1, 2, 3	boolean
com.ibm.db2.jcc.DB2BaseDataSource.traceLevel	1, 2, 3	int
com.ibm.db2.jcc.DB2BaseDataSource.useCachedCursor	1, 2	boolean
com.ibm.db2.jcc.DB2BaseDataSource.useJDBC4ColumnNameAndLabelSemantics	1, 2	int
com.ibm.db2.jcc.DB2BaseDataSource.user	1, 2, 3	String
com.ibm.db2.jcc.DB2BaseDataSource.useRowsetCursor	1	boolean
com.ibm.db2.jcc.DB2BaseDataSource.useTransactionRedirect	2	boolean
com.ibm.db2.jcc.DB2BaseDataSource.xaNetworkOptimization	1, 2, 3	boolean
com.ibm.db2.jcc.DB2BaseDataSource.DBANSIWARN	3	boolean
com.ibm.db2.jcc.DB2BaseDataSource.DBDATE	3	String
com.ibm.db2.jcc.DB2BaseDataSource.DBPATH	3	String
com.ibm.db2.jcc.DB2BaseDataSource.DBSPACETEMP	3	String
com.ibm.db2.jcc.DB2BaseDataSource.DBTEMP	3	String
com.ibm.db2.jcc.DB2BaseDataSource.DBUPSPACE	3	String
com.ibm.db2.jcc.DB2BaseDataSource.DELIMIDENT	3	boolean
com.ibm.db2.jcc.DB2BaseDataSource.IFX_DIRECTIVES	3	String
com.ibm.db2.jcc.DB2BaseDataSource.IFX_EXTDIRECTIVES	3	String
com.ibm.db2.jcc.DB2BaseDataSource.IFX_UPDESC	3	String
com.ibm.db2.jcc.DB2BaseDataSource.IFX_XASTDCOMPLIANCE_XAEND	3	String
com.ibm.db2.jcc.DB2BaseDataSource.INFORMIXOPCACHE	3	String
com.ibm.db2.jcc.DB2BaseDataSource.INFORMIXSTACKSIZE	3	String
com.ibm.db2.jcc.DB2BaseDataSource.NODEFDAC	3	String
com.ibm.db2.jcc.DB2BaseDataSource.OPTCOMPIND	3	String
com.ibm.db2.jcc.DB2BaseDataSource.OPTOFC	3	String
com.ibm.db2.jcc.DB2BaseDataSource.PDQPRIORITY	3	String
com.ibm.db2.jcc.DB2BaseDataSource.PSORT_DBTEMP	3	String
com.ibm.db2.jcc.DB2BaseDataSource.PSORT_NPROCS	3	String
com.ibm.db2.jcc.DB2BaseDataSource.STMT_CACHE	3	String

Note: The property applies to connections to the following data sources:

1. DB2 for z/OS
2. DB2 Database for Linux, UNIX, and Windows
3. IBM Informix Dynamic Server

DB2BaseDataSource methods

In addition to the `getXXX` and `setXXX` methods for the `DB2BaseDataSource` properties, the following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getReference

Format:

```
public javax.naming.Reference getReference()  
    throws javax.naming.NamingException
```

Retrieves the Reference of a DataSource object. For an explanation of a Reference, see the description of javax.naming.Referenceable in the JNDI documentation at:

<http://java.sun.com/products/jndi/docs.html>

Related reference

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

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.

DB2BlobFileReference

Format:

```
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 an existing HFS file.

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.

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 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.

getDB2ClientProgramId

Format:

```
public String getDB2ClientProgramId()  
    throws java.sql.SQLException
```

Returns the user-defined program identifier for the client. The program identifier can be used to identify the application at the data source.

getDB2ClientProgramId does not apply to DB2 Database for Linux, UNIX, and Windows data servers.

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.

getDB2ClientUser

Format:

```
public String getDB2ClientUser()  
    throws java.sql.SQLException
```

Returns the current client user name for the connection. This name is not the user value for the JDBC connection.

Important: getDB2ClientUser is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use `java.sql.Connection.getClientInfo` instead.

getDB2ClientWorkstation

Format:

```
public String getDB2ClientWorkstation()  
    throws java.sql.SQLException
```

Returns current client workstation name for the current client.

Important: getDB2ClientWorkstation is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use `java.sql.Connection.getClientInfo` instead.

getDB2Correlator

Format:

```
String getDB2Correlator()  
    throws java.sql.SQLException
```

Returns the value of the `crrtkn` (correlation token) instance variable that DRDA sends with the `ACCRDB` command. The correlation token uniquely identifies a logical connection to a server.

getDB2CurrentPackagePath

Format:

```
public String getDB2CurrentPackagePath()  
    throws java.sql.SQLException
```

Returns the list of DB2 package collections that are searched for JDBC and SQLJ packages.

The `getDB2CurrentPackagePath` method applies only to connections to DB2 database systems.

getDB2CurrentPackageSet

Format:

```
public String getDB2CurrentPackageSet()  
    throws java.sql.SQLException
```

Returns the collection ID for the connection.

The `getDB2CurrentPackageSet` method applies only to connections to DB2 database systems.

getDB2ProgressiveStreaming

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).

getDB2SecurityMechanism

Format:

```
public int getDB2SecurityMechanism()  
    throws java.sql.SQLException
```

Returns the security mechanism that is in effect for the connection:

- 3 Clear text password security
- 4 User ID-only security
- 7 Encrypted password security
- 9 Encrypted user ID and password security
- 11 Kerberos security
- 12 Encrypted user ID and data security
- 13 Encrypted user ID, password, and data security
- 15 Plugin security
- 16 Encrypted user ID-only security

getDB2SystemMonitor

Format:

```
public abstract DB2SystemMonitor getDB2SystemMonitor()  
    throws java.sql.SQLException
```

Returns the system monitor object for the connection. Each IBM Data Server Driver for JDBC and SQLJ connection can have a single system monitor.

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 Dynamic Server data sources.

getSavePointUniqueOption

Format:

```
public boolean getSavePointUniqueOption()
    throws java.sql.SQLException
```

Returns true if a unique savepoint was previously set. 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 Dynamic Server data sources.

isDB2Alive

Format:

```
public boolean DB2Connection.isDB2Alive()  
    throws java.sql.SQLException
```

Returns true if the socket for a connection to the data source is still active.

Important: `isDB2Alive` is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use `Connection.isValid` instead.

isValid

Format:

```
public boolean DB2Connection.isValid(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 `isValid` throws an `SQLException` if the connection is not valid. Possible values are:

true `isValid` throws an `SQLException` if the connection is not valid.

false `isValid` throws an `SQLException` only if the value of *timeout* is less than 0.

timeout

The time in seconds to wait for a database operation that the driver submits to complete. The driver submits that database operation to the data source to validate the connection. If the timeout period expires before the database operation completes, `isValid` returns false. A value of 0 indicates that there is no timeout period for the database operation.

This method does not apply to connections to IBM Informix Dynamic Server data sources.

reconfigureDB2Connection

Format:

```
public void reconfigureDB2Connection(java.util.Properties properties)  
    throws SQLException
```

Reconfigures a connection with new settings. The connection does not need to be returned to a connection pool before it is reconfigured. This method can be called while a transaction is in progress, and can be used for trusted or untrusted connections.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 Database for Linux, UNIX, and Windows Version 9.5 or later, and DB2 for z/OS Version 9.1 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to 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:

```
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
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 is deprecated for connections to DB2 for z/OS data sources. A value of true might not be valid in future releases.

This method does not apply to connections to IBM Informix Dynamic Server data sources.

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.

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 Dynamic Server data sources.

removeDB2JavaStoredProcedure

Format:

```
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 Dynamic Server 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 Dynamic Server 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 Database for Linux, UNIX, and Windows Version 9.5 or later, and DB2 for z/OS Version 9.1 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS Version 9.1 or later

The second of these forms of reuseDB2Connection does not apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

These forms of reuseDB2Connection are used by a trusted application server to reuse a preexisting trusted connection on behalf of a new user. Properties that can be reset are passed, including the new user ID. The database server resets the associated physical connection. If reuseDB2Connection executes successfully, the connection becomes available for immediate use, with different properties, by the new user.

Parameter descriptions:

cookie

A unique cookie that the JDBC driver generates for the Connection instance. The cookie is known only to the application server and the underlying JDBC driver that established the initial trusted connection. The application server passes the cookie that was created by the driver when the pooled connection instance was created. The JDBC driver checks that the supplied cookie matches the cookie of the underlying trusted physical connection to ensure that the request originated from the application server that established the trusted physical connection. If the cookies match, the connection becomes available for immediate use, with different properties, by the new user .

user

The client ID that the database system uses to establish the database authorization ID. If the user was not authenticated by the application server, the application server needs to pass a client ID that represents an unauthenticated user.

password

The password for *user*.

gssCredential

If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

userNameRegistry

A name that identifies a mapping service that maps a workstation user ID to a z/OS RACF ID. An example of a mapping service is the Integrated Security Services Enterprise Identity Mapping (EIM). The mapping service

is defined by a plugin. Valid values for *userNameRegistry* are defined by the plugin providers. If *userNameRegistry* is null, no mapping of *user* is done.

userSecToken

The client's security tokens. This value is traced as part of DB2 for z/OS accounting data. The content of *userSecToken* is described by the application server and is referred to by the database system as an application server security token.

originalUser

The original user ID that was used by the application server.

properties

Properties for the reused connection.

reuseDB2Connection (untrusted reuse with reauthentication)

Formats:

```
public void reuseDB2Connection(String user,
    String password,
    java.util.Properties properties)
    throws java.sql.SQLException
public void reuseDB2Connection(
    org.ietf.jgss.GSSCredential gssCredential,
    java.util.Properties properties)
    throws java.sql.SQLException
```

The first of these forms of `reuseDB2Connection` is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

The second of these forms of `reuseDB2Connection` does not apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

In a heterogeneous pooling environment, these forms of `reuseDB2Connection` reuse an existing Connection instance after reauthentication.

Parameter description:

user

The authorization ID that is used to establish the connection.

password

The password for the authorization ID that is used to establish the connection.

gssCredential

If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

properties

Properties for the reused connection. These properties override any properties that are already defined on the `DB2Connection` instance.

reuseDB2Connection (untrusted or trusted reuse without reauthentication)

Formats:

```
public void reuseDB2Connection(java.util.Properties properties)
    throws java.sql.SQLException
```

Reuses an existing Connection instance without reauthentication. This method is intended for reuse of a Connection instance when the properties do not change.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 Database for Linux, UNIX, and Windows Version 9.5 or later, and DB2 for z/OS Version 9.1 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS Version 9.1 or later

This method is for *dirty reuse* of a connection. This means that the connection state is not reset when the object is reused from the pool. Special register settings and property settings remain in effect unless they are overridden by passed properties. Global temporary tables are not deleted. Properties that are not specified are not re-initialized. All JDBC standard transient properties, such as the isolation level, autocommit mode, and read-only mode are reset to their JDBC defaults. Certain properties, such as user, password, databaseName, serverName, portNumber, planName, and pkList remain unchanged.

Parameter description:

properties

Properties for the reused connection. These properties override any properties that are already defined on the DB2Connection instance.

setDB2ClientAccountingInformation

Format:

```
public void setDB2ClientAccountingInformation(String info)
    throws java.sql.SQLException
```

Specifies accounting information for the connection. This information is for client accounting purposes. This value can change during a connection.

setDB2ClientAccountingInformation sets the value in the CLIENT ACCTNG special register.

Parameter description:

info

User-specified accounting information. The maximum length depends on the server. For a DB2 Database for Linux, UNIX, and Windows server, the maximum length is 255 bytes. For a DB2 for z/OS server, the maximum length is 22 bytes. A Java empty string ("") is valid for this parameter value, but a Java null value is not valid.

Important: setDB2ClientAccountingInformation is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use java.sql.Connection.setClientInfo instead.

setDB2ClientApplicationInformation

Format:

```
public String setDB2ClientApplicationInformation(String info)
    throws java.sql.SQLException
```

Specifies application information for the current client.

Important: setDB2ClientApplicationInformation is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use java.sql.Connection.setClientInfo instead.

Parameter description:

info

User-specified application information. The maximum length depends on the server. For a DB2 Database for Linux, UNIX, and Windows server, the maximum length is 255 bytes. For a DB2 for z/OS server, the maximum

length is 32 bytes. A Java empty string ("") is valid for this parameter value, but a Java null value is not valid.

setDB2ClientDebugInfo

Formats:

```
public void setDB2ClientDebugInformation(String debugInfo)
    throws java.sql.SQLException
public void setDB2ClientDebugInformation(String mgrInfo,
    String traceInfo)
    throws java.sql.SQLException
```

Sets a value for the CLIENT DEBUGINFO connection attribute, to notify the database system that stored procedures and user-defined functions that are using the connection are running in debug mode. CLIENT DEBUGINFO is used by the DB2 Unified Debugger. Use the first form to set the entire CLIENT DEBUGINFO string. Use the second form to modify only the session manager and trace information in the CLIENT DEBUGINFO string.

The setDB2ClientDebugInfo method applies only to connections to DB2 for z/OS database systems.

Setting the CLIENT DEBUGINFO attribute to a string of length greater than zero requires one of the following privileges:

- The DEBUGSESSION privilege
- SYSADM authority

Parameter description:

debugInfo

A string of up to 254 bytes, in the following form:

Mip:port,Iip,Ppid,Ttid,Cid,Llvl

The parts of the string are:

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.

This method does not apply to connections to IBM Informix Dynamic Server data sources.

setDB2ClientProgramId

Format:

```
public abstract void setDB2ClientProgramId(String program-ID)
    throws java.sql.SQLException
```

Sets a user-defined program identifier for the connection, on DB2 for z/OS servers. That program identifier is an 80-byte string that is used to identify the caller.

setDB2ClientProgramId does not apply to DB2 Database for Linux, UNIX, and Windows 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. For a DB2 Database for Linux, UNIX, and Windows server, the maximum length is 255 bytes. For a DB2 for z/OS server, the maximum length is 16 bytes. A Java empty string ("") is valid for this parameter value, but a Java null value is not valid.

Important: 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.

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 server. For a DB2 Database for Linux, UNIX, and Windows server, the maximum length is 255 bytes. For a DB2 for z/OS server, the maximum length is 18 bytes. A Java empty string ("") is valid for this parameter value, but a Java null value is not valid.

Important: getDB2ClientWorkstation is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use java.sql.Connection.getClientInfo instead.

setDB2CurrentPackagePath

Format:

```
public void setDB2CurrentPackagePath(String packagePath)
    throws java.sql.SQLException
```

Specifies a list of collection IDs that the database system searches for JDBC and SQLJ packages.

The setDB2CurrentPackagePath method applies only to connections to DB2 database systems.

Parameter description:

packagePath

A comma-separated list of collection IDs.

setDB2CurrentPackageSet

Format:

```
public void setDB2CurrentPackageSet(String packageSet)
    throws java.sql.SQLException
```

Specifies the collection ID for the connection. When you set this value, you also set the collection ID of the IBM Data Server Driver for JDBC and SQLJ instance that is used for the connection.

The setDB2CurrentPackageSet method applies only to connections to DB2 database systems.

Parameter description:

packageSet

The collection ID for the connection. The maximum length for the *packageSet* value is 18 bytes. You can invoke this method as an alternative to executing the SQL SET CURRENT PACKAGESET statement in your program.

setDB2ProgressiveStreaming

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.

setJccLogWriter

Formats:

```
public void setJccLogWriter(PrintWriter logWriter)
    throws java.sql.SQLException
```

```
public void setJccLogWriter(PrintWriter logWriter, int traceLevel)
    throws java.sql.SQLException
```

Enables or disables the IBM Data Server Driver for JDBC and SQLJ trace, or changes the trace destination during an active connection.

Parameter descriptions:

logWriter

An object of type `java.io.PrintWriter` to which the IBM Data Server Driver for JDBC and SQLJ writes trace output. To turn off the trace, set the value of *logWriter* to `null`.

traceLevel

Specifies the types of traces to collect. See the description of the *traceLevel* property in "Properties for the IBM Data Server Driver for JDBC and SQLJ" for valid values.

setSavePointUniqueOption

Format:

```
public void setSavePointUniqueOption(boolean flag)
    throws java.sql.SQLException
```

Specifies whether an application can reuse a savepoint name within a unit of recovery. Possible values are:

true A `Connection.setSavepoint(savepoint-name)` method cannot specify the same value for *savepoint-name* more than once within the same unit of recovery.

false A `Connection.setSavepoint(savepoint-name)` method can specify the same value for *savepoint-name* more than once within the same unit of recovery.

When `false` is specified, if the `Connection.setSavepoint(savepoint-name)` method is executed, and a savepoint with the name *savepoint-name* already exists within the unit of recovery, the database manager destroys the existing savepoint, and creates a new savepoint with the name *savepoint-name*.

Reuse of a savepoint is not the same as executing `Connection.releaseSavepoint(savepoint-name)`.

Connection.releaseSavepoint(savepoint-name) releases *savepoint-name*, and any savepoints that were subsequently set.

Related concepts

Chapter 15, “Problem diagnosis with the IBM Data Server Driver for JDBC and SQLJ,” on page 537

DB2ConnectionPoolDataSource class

DB2ConnectionPoolDataSource is a factory for PooledConnection objects. An object that implements this interface is registered with a naming service that is based on the Java Naming and Directory Interface (JNDI).

The com.ibm.db2.jcc.DB2ConnectionPoolDataSource class extends the com.ibm.db2.jcc.DB2BaseDataSource class, and implements the javax.sql.ConnectionPoolDataSource, java.io.Serializable, and javax.naming.Referenceable interfaces.

DB2ConnectionPoolDataSource properties

These properties are defined only for the IBM Data Server Driver for JDBC and SQLJ. “Properties for the IBM Data Server Driver for JDBC and SQLJ” for explanations of these properties.

These properties have a setXXX method to set the value of the property and a getXXX method to retrieve the value. A setXXX method has this form:

`void setProperty-name(data-type property-value)`

A getXXX method has this form:

`data-type getProperty-name()`

Property-name is the unqualified property name, with the first character capitalized.

The following table lists the IBM Data Server Driver for JDBC and SQLJ properties and their data types.

Table 87. DB2ConnectionPoolDataSource properties and their data types

Property name	Data type
com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements	int

DB2ConnectionPoolDataSource methods

getDB2PooledConnection

Formats:

```
public DB2PooledConnection getDB2PooledConnection(String user,
String password,
java.util.Properties properties)
throws java.sql.SQLException
public DB2PooledConnection getDB2PooledConnection(
org.ietf.jgss.GSSCredential gssCredential,
java.util.Properties properties)
throws java.sql.SQLException
```

Establishes the initial untrusted connection in a heterogeneous pooling environment.

The first form `getDB2PooledConnection` provides a user ID and password. The second form of `getDB2PooledConnection` is for connections that use Kerberos security.

Parameter descriptions:

user

The authorization ID that is used to establish the connection.

password

The password for the authorization ID that is used to establish the connection.

gssCredential

If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

properties

Properties for the connection.

getDB2TrustedPooledConnection

Formats:

```
public Object[] getDB2TrustedPooledConnection(String user,
String password,
java.util.Properties properties)
throws java.sql.SQLException
public Object[] getDB2TrustedPooledConnection(
java.util.Properties properties)
throws java.sql.SQLException
public Object[] getDB2TrustedPooledConnection(
org.ietf.jgss.GSSCredential gssCredential,
java.util.Properties properties)
throws java.sql.SQLException
```

An application server using a system authorization ID uses this method to establish a trusted connection.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 Database for Linux, UNIX, and Windows Version 9.5 or later, and DB2 for z/OS Version 9.1 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS Version 9.1 or later

The following elements are returned in `Object[]`:

- The first element is a trusted `DB2PooledConnection` instance.
- The second element is a unique cookie for the generated pooled connection instance.

The first form `getDB2TrustedPooledConnection` provides a user ID and password, while the second form of `getDB2TrustedPooledConnection` uses the user ID and password of the `DB2ConnectionPoolDataSource` object. The third form of `getDB2TrustedPooledConnection` is for connections that use Kerberos security.

Parameter descriptions:

user

The DB2 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 527

Related reference

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

“DB2PooledConnection class” on page 356

DB2DatabaseMetaData interface

The `com.ibm.db2.jcc.DB2DatabaseMetaData` interface extends the `java.sql.DatabaseMetaData` interface.

`DB2DatabaseMetaData` implements the `java.sql.Wrapper` interface.

DB2DatabaseMetaData methods:

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

isIDSDatabaseAnsiCompliant

Format:

```
public boolean isIDSDatabaseAnsiCompliant();
```

Returns true if the current active IBM Informix Dynamic Server (IDS) 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 IDS data sources only. An `SQLException` is thrown if the data source is not an IDS data source.

isIDSDatabaseLogging

Format:

```
public boolean isIDSDatabaseLogging();
```

Returns true if the current active IDS database supports logging. Returns false otherwise.

An IDS 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 IDS data sources only. An `SQLException` is thrown if the data source is not an IDS data source.

isResetRequiredForDB2eWLM

Format:

```
public boolean isResetRequiredForDB2eWLM();
```

Returns true if the target database server requires clean reuse to support eWLM. Returns false otherwise.

supportsDB2ProgressiveStreaming

Format:

```
public boolean supportsDB2ProgressiveStreaming();
```

Returns true if the target data source supports progressive streaming. Returns false otherwise.

DB2Diagnosable interface

The `com.ibm.db2.jcc.DB2Diagnosable` interface provides a mechanism for getting DB2 diagnostics from an `SQLException`.

DB2Diagnosable methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getSqlca

Format:

```
public DB2Sqlca getSqlca();
```

Returns a `DB2Sqlca` object from a `java.sql.Exception` that is produced under a IBM Data Server Driver for JDBC and SQLJ.

getThrowable

Format:

```
public Throwable getThrowable();
```

Returns a `java.lang.Throwable` object from a `java.sql.Exception` that is produced under a IBM Data Server Driver for JDBC and SQLJ.

printTrace

Format:

```
static public void printTrace(java.io.PrintWriter printWriter,  
    String header);
```

Prints diagnostic information after a `java.sql.Exception` is thrown under a IBM Data Server Driver for JDBC and SQLJ.

Parameter descriptions:

printWriter

The destination for the diagnostic information.

header

User-defined information that is printed at the beginning of the output.

Related tasks

“Handling an `SQLException` under the IBM Data Server Driver for JDBC and SQLJ” on page 84

“Handling SQL warnings in an SQLJ application” on page 150

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 540

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.

getFileCcsid

Format:

```
public int getFileCcsid()
```

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 Database for Linux, UNIX, and Windows plug-in support. This class applies only to DB2 Database for Linux, UNIX, and Windows.

DB2JCCPlugin methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getTicket

Format:

```
public abstract byte[] getTicket(String user,  
    String password,  
    byte[] returnedToken)  
    throws org.ietf.jgss.GSSException
```

Retrieves a Kerberos ticket for a user.

Parameter descriptions:

user

The user ID for which the Kerberos ticket is to be retrieved.

password

The password for *user*.

returnedToken

DB2PooledConnection class

The `com.ibm.db2.jcc.DB2PooledConnection` class provides methods that an application server can use to switch users on a preexisting trusted connection.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 Database for Linux, UNIX, and Windows Version 9.5 or later, and DB2 for z/OS Version 9.1 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS Version 9.1 or later

DB2PooledConnection methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getConnection (untrusted or trusted reuse without reauthentication)

Format:

```
public DB2Connection getConnection()  
    throws java.sql.SQLException
```

This method is for *dirty reuse* of a connection. This means that the connection state is not reset when the object is reused from the pool. Special register settings and property settings remain in effect unless they are overridden by passed properties. Global temporary tables are not deleted. Properties that are not specified are not re-initialized. All JDBC standard transient properties, such as the isolation level, autocommit mode, and read-only mode are reset to their JDBC defaults. Certain properties, such as user, password, databaseName, serverName, portNumber, planName, and pkList remain unchanged.

getDB2Connection (trusted reuse)

Formats:

```
public DB2Connection getDB2Connection(byte[] cookie,  
    String user,  
    String password,  
    String userRegistry,  
    byte[] userSecToken,  
    String originalUser,  
    java.util.Properties properties)  
    throws java.sql.SQLException  
public Connection getDB2Connection(byte[] cookie,  
    org.ietf.GSSCredential gssCredential,  
    String usernameRegistry,  
    byte[] userSecToken,  
    String originalUser,  
    java.util.Properties properties)  
    throws java.sql.SQLException
```

Switches the user that is associated with a trusted connection without authentication.

The second form of `getDB2Connection` is supported only for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Parameter descriptions:

cookie

A unique cookie that the JDBC driver generates for the `Connection` instance. The cookie is known only to the application server and the underlying JDBC driver that established the initial trusted connection. The

application server passes the cookie that was created by the driver when the pooled connection instance was created. The JDBC driver checks that the supplied cookie matches the cookie of the underlying trusted physical connection to ensure that the request originated from the application server that established the trusted physical connection. If the cookies match, the connection can become available, with different properties, for immediate use by a new user .

user

The client identity that is used by the data source to establish the authorization ID for the database server. If the user was not authenticated by the application server, the application server must pass a user identity that represents an unauthenticated user.

password

The password for *user*.

gssCredential

If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

userNameRegistry

A name that identifies a mapping service that maps a workstation user ID to a z/OS RACF ID. An example of a mapping service is the Integrated Security Services Enterprise Identity Mapping (EIM). The mapping service is defined by a plugin. Valid values for *userNameRegistry* are defined by the plugin providers. If *userNameRegistry* is null, the connection does not use a mapping service.

userSecToken

The client's security tokens. This value is traced as part of DB2 for z/OS accounting data. The content of *userSecToken* is described by the application server and is referred to by the data source as an application server security token.

originalUser

The client identity that sends the original request to the application server. *originalUser* is included in DB2 for z/OS accounting data as the original user ID that was used by the application server.

properties

Properties for the reused connection. These properties override any properties that are already defined on the *DB2PooledConnection* instance.

getDB2Connection (untrusted reuse with reauthentication)

Formats:

```
public DB2Connection getDB2Connection(  
    String user,  
    String password,  
    java.util.Properties properties)  
    throws java.sql.SQLException  
public DB2Connection getDB2Connection(org.ietf.jgss.GSSCredential gssCredential,  
    java.util.Properties properties)  
    throws java.sql.SQLException
```

Switches the user that is associated with a untrusted connection, with authentication.

The first form *getDB2Connection* provides a user ID and password. The second form of *getDB2Connection* is for connections that use Kerberos security.

Parameter descriptions:

user

The user ID that is used by the data source to establish the authorization ID for the database server.

password

The password for *user*.

properties

Properties for the reused connection. These properties override any properties that are already defined on the `DB2PooledConnection` instance.

getDB2Connection (untrusted or trusted reuse without reauthentication)

Formats:

```
public java.sql.Connection getDB2Connection(
    java.util.Properties properties)
    throws java.sql.SQLException
```

Reuses an untrusted connection, without reauthentication.

This method is for *dirty reuse* of a connection. This means that the connection state is not reset when the object is reused from the pool. Special register settings and property settings remain in effect unless they are overridden by passed properties. Global temporary tables are not deleted. Properties that are not specified are not re-initialized. All JDBC standard transient properties, such as the isolation level, autocommit mode, and read-only mode are reset to their JDBC defaults. Certain properties, such as `user`, `password`, `databaseName`, `serverName`, `portNumber`, `planName`, and `pkList` remain unchanged.

Parameter descriptions:

properties

Properties for the reused connection. These properties override any properties that are already defined on the `DB2PooledConnection` instance.

Related concepts

Chapter 12, “JDBC and SQLJ connection pooling support,” on page 527

Related reference

“`DB2ConnectionPoolDataSource` class” on page 350

DB2PoolMonitor class

The `com.ibm.db2.jcc.DB2PoolMonitor` class provides methods for monitoring the global transport objects pool that is used for the connection concentrator and Sysplex workload balancing.

DB2PoolMonitor fields

The following fields are defined only for the IBM Data Server Driver for JDBC and SQLJ.

public static final int TRANSPORT_OBJECT = 1

This value is a parameter for the `DB2PoolMonitor.getPoolMonitor` method.

DB2PoolMonitor methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

agedOutObjectCount

Format:

```
public abstract int agedOutObjectCount()
```

Retrieves the number of objects that exceeded the idle time that was specified by `db2.jcc.maxTransportObjectIdleTime` and were deleted from the pool.

createdObjectCount

Format:

```
public abstract int createdObjectCount()
```

Retrieves the number of objects that the IBM Data Server Driver for JDBC and SQLJ created since the pool was created.

getMonitorVersion

Format:

```
public int getMonitorVersion()
```

Retrieves the version of the `DB2PoolMonitor` class that is shipped with the IBM Data Server Driver for JDBC and SQLJ.

getPoolMonitor

Format:

```
public static DB2PoolMonitor getPoolMonitor(int monitorType)
```

Retrieves an instance of the `DB2PoolMonitor` class.

Parameter descriptions:

monitorType

The monitor type. This value must be `DB2PoolMonitor.TRANSPORT_OBJECT`.

heavyWeightReusedObjectCount

Format:

```
public abstract int heavyWeightReusedObjectCount()
```

Retrieves the number of objects that were reused from the pool.

lightWeightReusedObjectCount

Format:

```
public abstract int lightWeightReusedObjectCount()
```

Retrieves the number of objects that were reused but were not in the pool. This can happen if a `Connection` object releases a transport object at a transaction boundary. If the `Connection` object needs a transport object later, and the original transport object has not been used by any other `Connection` object, the `Connection` object can use that transport object.

longestBlockedRequestTime

Format:

```
public abstract long longestBlockedRequestTime()
```

Retrieves the longest amount of time that a request was blocked, in milliseconds.

numberOfConnectionReleaseRefused

Format:

```
public abstract int numberOfConnectionReleaseRefused()
```

Retrieves the number of times that the release of a connection was refused.

numberOfRequestsBlocked

Format:

```
public abstract int numberOfRequestsBlocked()
```

Retrieves the number of requests that the IBM Data Server Driver for JDBC and SQLJ made to the pool that the pool blocked because the pool reached its maximum capacity. A blocked request might be successful if an object is returned to the pool before the `db2.jcc.maxTransportObjectWaitTime` is exceeded and an exception is thrown.

numberOfRequestsBlockedDataSourceMax

Format:

```
public abstract int numberOfRequestsBlockedDataSourceMax()
```

Retrieves the number of requests that the IBM Data Server Driver for JDBC and SQLJ made to the pool that the pool blocked because the pool reached the maximum for the `DataSource` object.

numberOfRequestsBlockedPoolMax

Format:

```
public abstract int numberOfRequestsBlockedPoolMax()
```

Retrieves the number of requests that the IBM Data Server Driver for JDBC and SQLJ made to the pool that the pool blocked because the maximum number for the pool was reached.

removedObjectCount

Format:

```
public abstract int removedObjectCount()
```

Retrieves the number of objects that have been deleted from the pool since the pool was created.

shortestBlockedRequestTime

Format:

```
public abstract long shortestBlockedRequestTime()
```

Retrieves the shortest amount of time that a request was blocked, in milliseconds.

successfulRequestsFromPool

Format:

```
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.

totalPoolObjects

Format:

```
public abstract int totalPoolObjects()
```

Retrieves the number of objects that are currently in the pool.

totalRequestsToPool

Format:

```
public abstract int totalRequestsToPool()
```

Retrieves the total number of requests that the IBM Data Server Driver for JDBC and SQLJ has made to the pool since the pool was created.

totalTimeBlocked

Format:

```
public abstract long totalTimeBlocked()
```

Retrieves the total time in milliseconds for requests that were blocked by the pool. This time can be much larger than the elapsed execution time of the application if the application uses multiple threads.

DB2PreparedStatement interface

The `com.ibm.db2.jcc.DB2PreparedStatement` interface extends the `com.ibm.db2.jcc.DB2Statement` and `java.sql.PreparedStatement` interfaces.

DB2PreparedStatement methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

executeDB2QueryBatch

Format:

```
public void executeDB2QueryBatch()
    throws java.sql.SQLException
```

Executes a statement batch that contains queries with parameters.

This method is not supported for connections to IBM Informix Dynamic Server 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.

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:

```
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:

```
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:

```
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.DB2XmlAsClobFileReference value that is assigned to the parameter marker.

setJccArrayAtName

Format:

```
public void setJccArrayAtName(String parameterMarkerName,  
    java.sql.Array x)  
    throws java.sql.SQLException
```

Assigns a java.sql.Array value to a named parameter marker.

If the data server does not support named parameter markers, this method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName

The name of the parameter marker to which a value is assigned.

x The java.sql.Array value that is assigned to the named parameter marker.

setJccAsciiStreamAtName

Formats:

Supported by the IBM Data Server Driver for JDBC and SQLJ version 3.57 and later:

```
public void setJccAsciiStreamAtName(String parameterMarkerName,  
    java.io.InputStream x, int length)  
    throws java.sql.SQLException
```

Supported by the IBM Data Server Driver for JDBC and SQLJ version 4.7 and later:

```
public void setJccAsciiStreamAtName(String parameterMarkerName,  
    java.io.InputStream x)  
    throws java.sql.SQLException  
public void setJccAsciiStreamAtName(String parameterMarkerName,  
    java.io.InputStream x, long length)  
    throws java.sql.SQLException
```

Assigns an ASCII value in a java.io.InputStream to a named parameter marker.

If the data server does not support named parameter markers, this method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName

The name of the parameter marker to which a value is assigned.

x The ASCII java.io.InputStream value that is assigned to the parameter marker.

length

The length in bytes of the java.io.InputStream value that is assigned to the named parameter marker.

setJccBigDecimalAtName

Format:

```
public void setJccBigDecimalAtName(String parameterMarkerName,  
    java.math.BigDecimal x)  
    throws java.sql.SQLException
```

Assigns a java.math.BigDecimal value to a named parameter marker.

If the data server does not support named parameter markers, this method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName

The name of the parameter marker to which a value is assigned.

x The java.math.BigDecimal value that is assigned to the named parameter marker.

setJccBinaryStreamAtName

Formats:

Supported by the IBM Data Server Driver for JDBC and SQLJ version 3.57 and later:

```
public void setJccBinaryStreamAtName(String parameterMarkerName,  
    java.io.InputStream x, int length)  
    throws java.sql.SQLException
```

Supported by the IBM Data Server Driver for JDBC and SQLJ version 4.7 and later:

```
public void setJccBinaryStreamAtName(String parameterMarkerName,  
    java.io.InputStream x)  
    throws java.sql.SQLException  
public void setJccBinaryStreamAtName(String parameterMarkerName,  
    java.io.InputStream x, long length)  
    throws java.sql.SQLException
```

Assigns a binary value in a java.io.InputStream to a named parameter marker.

If the data server does not support named parameter markers, this method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName

The name of the parameter marker to which a value is assigned.

x The binary java.io.InputStream value that is assigned to the parameter marker.

length

The number of bytes of the java.io.InputStream value that are assigned to the named parameter marker.

setJccBlobAtName

Formats:

Supported by the IBM Data Server Driver for JDBC and SQLJ version 3.57 and later:

```
public void setJccBlobAtName(String parameterMarkerName,
                             java.sql.Blob x)
    throws java.sql.SQLException
```

Supported by the IBM Data Server Driver for JDBC and SQLJ version 4.7 and later:

```
public void setJccBlobAtName(String parameterMarkerName,
                             java.io.InputStream x)
    throws java.sql.SQLException
public void setJccBlobAtName(String parameterMarkerName,
                             java.io.InputStream x, long length)
    throws java.sql.SQLException
```

Assigns a BLOB value to a named parameter marker.

If the data server does not support named parameter markers, this method can be called only if the `enableNamedParameterMarkers` property is set to `DB2BaseDataSource.YES (1)`.

Parameters:

parameterMarkerName

The name of the parameter marker to which a value is assigned.

x The `java.sql.Blob` value or `java.io.InputStream` value that is assigned to the parameter marker.

length

The number of bytes of the `java.io.InputStream` value that are assigned to the named parameter marker.

setJccBooleanAtName

Format:

```
public void setJccBooleanAtName(String parameterMarkerName,
                                boolean x)
    throws java.sql.SQLException
```

Assigns a boolean value to a named parameter marker.

If the data server does not support named parameter markers, 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.

If the data server does not support named parameter markers, 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.

If the data server does not support named parameter markers, 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.

If the data server does not support named parameter markers, 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.Blob 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.InputStream x)  
    throws java.sql.SQLException  
public void setJccClobAtName(String parameterMarkerName,  
    java.io.InputStream x, long length)  
    throws java.sql.SQLException
```

Assigns a CLOB value to a named parameter marker.

If the data server does not support named parameter markers, 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.

If the data server does not support named parameter markers, this method can be called only if the `enableNamedParameterMarkers` property is set to `DB2BaseDataSource.YES (1)`.

Parameters:

parameterMarkerName

The name of the parameter marker to which a value is assigned.

x The `java.sql.Date` value that is assigned to the named parameter marker.

cal The `java.util.Calendar` object that the IBM Data Server Driver for JDBC and SQLJ uses to construct the date.

setJccDB2BlobFileReferenceAtName

Format:

```
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.

If the data server does not support named parameter markers, 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:

```
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.

If the data server does not support named parameter markers, 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:

```
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.

If the data server does not support named parameter markers, 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:

```
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.

If the data server does not support named parameter markers, 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.

setJccDoubleAtName

Format:

```
public void setJccDoubleAtName(String parameterMarkerName,  
double x)  
throws java.sql.SQLException
```

Assigns a value of type double to a named parameter marker.

If the data server does not support named parameter markers, this method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName

The name of the parameter marker to which a value is assigned.

x The value of type double that is assigned to the parameter marker.

setJccFloatAtName

Format:

```
public void setJccFloatAtName(String parameterMarkerName,  
double x)  
throws java.sql.SQLException
```

Assigns a value of type float to a named parameter marker.

If the data server does not support named parameter markers, this method can be called only if the `enableNamedParameterMarkers` property is set to `DB2BaseDataSource.YES (1)`.

Parameters:

parameterMarkerName

The name of the parameter marker to which a value is assigned.

x The value of type float that is assigned to the parameter marker.

setJccIntAtName

Format:

```
public void setJccIntAtName(String parameterMarkerName,
                             int x)
    throws java.sql.SQLException
```

Assigns a value of type int to a named parameter marker.

If the data server does not support named parameter markers, this method can be called only if the `enableNamedParameterMarkers` property is set to `DB2BaseDataSource.YES (1)`.

Parameters:

parameterMarkerName

The name of the parameter marker to which a value is assigned.

x The value of type int that is assigned to the parameter marker.

setJccLongAtName

Format:

```
public void setJccLongAtName(String parameterMarkerName,
                              long x)
    throws java.sql.SQLException
```

Assigns a value of type long to a named parameter marker.

If the data server does not support named parameter markers, this method can be called only if the `enableNamedParameterMarkers` property is set to `DB2BaseDataSource.YES (1)`.

Parameters:

parameterMarkerName

The name of the parameter marker to which a value is assigned.

x The value of type long that is assigned to the parameter marker.

setJccNullAtName

Format:

```
public void setJccNullAtName(String parameterMarkerName,
                              int jdbcType)
    throws java.sql.SQLException
public void setJccNullAtName(String parameterMarkerName,
                              int jdbcType,
                              String typeName)
    throws java.sql.SQLException
```

Assigns the SQL NULL value to a named parameter marker.

If the data server does not support named parameter markers, this method can be called only if the `enableNamedParameterMarkers` property is set to `DB2BaseDataSource.YES (1)`.

Parameters:

parameterMarkerName

The name of the parameter marker to which a value is assigned.

jdbcType

The JDBC type code of the NULL value that is assigned to the parameter marker, as defined in `java.sql.Types`.

typeName

If *jdbcType* is `java.sql.Types.DISTINCT` or `java.sql.Types.REF`, the fully-qualified name of the SQL user-defined type of the NULL value that is assigned to the parameter marker.

setJccObjectAtName

Formats:

```
public void setJccObjectAtName(String parameterMarkerName,
                               java.sql.Object x)
    throws java.sql.SQLException
public void setJccObjectAtName(String parameterMarkerName,
                               java.sql.Object x,
                               int targetJdbcType)
    throws java.sql.SQLException
public void setJccObjectAtName(String parameterMarkerName,
                               java.sql.Object x,
                               int targetJdbcType,
                               int scale)
    throws java.sql.SQLException
```

Assigns a value with type `java.lang.Object` to a named parameter marker.

If the data server does not support named parameter markers, 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:

```
public void setJccShortAtName(String parameterMarkerName,
                              short x)
    throws java.sql.SQLException
```

Assigns a value of type `short` to a named parameter marker.

If the data server does not support named parameter markers, this method can be called only if the `enableNamedParameterMarkers` property is set to `DB2BaseDataSource.YES (1)`.

Parameters:

parameterMarkerName

The name of the parameter marker to which a value is assigned.

x The value of type short that is assigned to the parameter marker.

setJccSQLXMLAtName

Format:

```
public void setJccSQLXMLAtName(String parameterMarkerName,
                                java.sql.SQLXML x)
    throws java.sql.SQLException
```

Assigns a value of type `java.sql.SQLXML` to a named parameter marker.

If the data server does not support named parameter markers, this method can be called only if the `enableNamedParameterMarkers` property is set to `DB2BaseDataSource.YES (1)`.

This method is supported only for connections to DB2 Database for Linux, UNIX, and Windows Version 9.1 or later or DB2 for z/OS Version 9 or later.

Parameters:

parameterMarkerName

The name of the parameter marker to which a value is assigned.

x The value of type `java.sql.SQLXML` that is assigned to the parameter marker.

setJccStringAtName

Format:

```
public void setJccStringAtName(String parameterMarkerName,
                                String x)
    throws java.sql.SQLException
```

Assigns a value of type `String` to a named parameter marker.

If the data server does not support named parameter markers, 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.

If the data server does not support named parameter markers, 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.

If the data server does not support named parameter markers, 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.

If the data server does not support named parameter markers, 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.

Related tasks

“Making batch queries in JDBC applications” on page 35

Related reference

“DB2Statement interface” on page 376

DB2ResultSetMetaData interface

The `com.ibm.db2.jcc.DB2ResultSetMetaData` interface provides methods that provide information about a `ResultSet` object.

Before a `com.ibm.db2.jcc.DB2ResultSetMetaData` method can be used, a `java.sql.ResultSetMetaData` object that is returned from a `java.sql.ResultSet.getMetaData` call needs to be cast to `com.ibm.db2.jcc.DB2ResultSetMetaData`.

DB2ResultSetMetaData methods:

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

isDB2ColumnNameDerived

Format:

```
public boolean isDB2ColumnNameDerived (int column)
    throws java.sql.SQLException
```

Returns true if the name of a `ResultSet` column is in the SQL SELECT list that generated the `ResultSet`.

For example, suppose that a `ResultSet` is generated from the SQL statement `SELECT EMPNAME, SUM(SALARY) FROM EMP`. Column name `EMPNAME` is derived from the SQL SELECT list, but the name of the column in the `ResultSet` that corresponds to `SUM(SALARY)` is not derived from the SELECT list.

Parameter descriptions:

column

The name 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 Dynamic Server.

DB2RowID methods

The following method is defined only for the IBM Data Server Driver for JDBC and SQLJ.

getBytes

Format:

```
public byte[] getBytes()
```

Converts a `com.ibm.jcc.DB2RowID` object to bytes.

Related concepts

"ROWIDs in JDBC with the IBM Data Server Driver for JDBC and SQLJ" on page 56

"ROWIDs in SQLJ with the IBM Data Server Driver for JDBC and SQLJ" on page 137

DB2SimpleDataSource class

The `com.ibm.db2.jcc.DB2SimpleDataSource` class extends the `DB2BaseDataSource` class.

A `DB2BaseDataSource` object does not support connection pooling or distributed transactions. It contains all of the properties and methods that the `DB2BaseDataSource` class contains. In addition, `DB2SimpleDataSource` contains the following IBM Data Server Driver for JDBC and SQLJ-only properties.

`DB2SimpleDataSource` implements the `java.sql.Wrapper` interface.

DB2SimpleDataSource properties

The following property is defined only for the IBM Data Server Driver for JDBC and SQLJ. See "Properties for the IBM Data Server Driver for JDBC and SQLJ" for an explanation of this property.

String `com.ibm.db2.jcc.DB2SimpleDataSource.password`

DB2SimpleDataSource methods

The following method is defined only for the IBM Data Server Driver for JDBC and SQLJ.

setPassword

Format:

```
public 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

"Connecting to a data source using the `DataSource` interface" on page 15

"Creating and deploying `DataSource` objects" on page 19

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 an SQLException under the IBM Data Server Driver for JDBC and SQLJ” on page 84

“Handling SQL warnings in an SQLJ application” on page 150

Related reference

 Description of SQLCA fields (SQL Reference)

DB2Statement interface

The `com.ibm.db2.jcc.DB2Statement` interface extends the `java.sql.Statement` interface.

`DB2Statement` implements the `java.sql.Wrapper` interface.

DB2Statement methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getDB2ClientProgramId

Format:

```
public String getDB2ClientProgramId()  
    throws java.sql.SQLException
```

Returns the user-defined client program identifier for the connection, which is stored on the data source.

getDB2ClientProgramId does not apply to DB2 Database for Linux, UNIX, and Windows data servers.

setDB2ClientProgramId

Format:

```
public abstract void setDB2ClientProgramId(String program-ID)  
    throws java.sql.SQLException
```

Sets a user-defined program identifier for the connection on a data server. That program identifier is an 80-byte string that is used to identify the caller.

setDB2ClientProgramId does not apply to DB2 Database for Linux, UNIX, and Windows data servers.

The DB2 for z/OS server places the string in IFCID 316 trace records along with other statistics, so that you can identify which program is associated with a particular SQL statement.

getIDSBigSerial

Format:

```
public int getIDSBigSerial()  
    throws java.sql.SQLException
```

Retrieves an automatically generated key from a BIGSERIAL column after the automatically generated key was inserted by a previously executed INSERT statement.

The following conditions must be true for getIDSBigSerial to execute successfully:

- The INSERT statement is the last SQL statement that is executed before this method is called.
- The table into which the row is inserted contains a BIGSERIAL column.
- The form of the JDBC Connection.prepareStatement method or Statement.executeUpdate method that prepares or executes the INSERT statement does not have parameters that request automatically generated keys.

This method applies only to connections to IBM Informix Dynamic Server (IDS) databases.

getIDSSerial

Format:

```
public int getIDSSerial()  
    throws java.sql.SQLException
```

Retrieves an automatically generated key from a SERIAL column after the automatically generated key was inserted by a previously executed INSERT statement.

The following conditions must be true for `getIDSSerial` to execute successfully:

- The INSERT statement is the last SQL statement that is executed before this method is called.
- The table into which the row is inserted contains a SERIAL column.
- The form of the JDBC Connection.`prepareStatement` method or `Statement.executeUpdate` method that prepares or executes the INSERT statement does not have parameters that request automatically generated keys.

This method applies only to connections to IBM Informix Dynamic Server (IDS) databases.

`getIDSSerial8`

Format:

```
public long getIDSSerial8()
    throws java.sql.SQLException
```

Retrieves an automatically generated key from a SERIAL8 column after the automatically generated key was inserted by a previously executed INSERT statement.

The following conditions must be true for `getIDSSerial8` to execute successfully:

- The INSERT statement is the last SQL statement that is executed before this method is called.
- The table into which the row is inserted contains a SERIAL8 column.
- The form of the JDBC Connection.`prepareStatement` method or `Statement.executeUpdate` method that prepares or executes the INSERT statement does not have parameters that request automatically generated keys.

This method applies only to connections to IDS data sources.

`getIDSQLStatementOffset`

Format:

```
public int getIDSQLStatementOffset()
    throws java.sql.SQLException
```

After an SQL statement executes on an IDS data source, if the statement has a syntax error, `getIDSQLStatementOffset` returns the offset into the statement text of the syntax error.

`getIDSQLStatementOffset` returns:

- 0, if the statement does not have a syntax error.
- -1, if the data source is not IDS.

This method applies only to connections to IDS data sources.

Related reference

“DB2PreparedStatement interface” on page 361

DB2SystemMonitor interface

The `com.ibm.db2.jcc.DB2SystemMonitor` interface is used for collecting system monitoring data for a connection. Each connection can have one `DB2SystemMonitor` instance.

DB2SystemMonitor fields

The following fields are defined only for the IBM Data Server Driver for JDBC and SQLJ.

public final static int RESET_TIMES

public final static int ACCUMULATE_TIMES

These values are arguments for the `DB2SystemMonitor.start` method.

`RESET_TIMES` sets time counters to zero before monitoring starts.

`ACCUMULATE_TIMES` does not set time counters to zero.

DB2SystemMonitor methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

enable

Format:

```
public void enable(boolean on)
    throws java.sql.SQLException
```

Enables the system monitor that is associated with a connection. This method cannot be called during monitoring. All times are reset when `enable` is invoked.

getApplicationTimeMillis

Format:

```
public long getApplicationTimeMillis()
    throws java.sql.SQLException
```

Returns the sum of the application, JDBC driver, network I/O, and database server elapsed times. The time is in milliseconds.

A monitored elapsed time interval is the difference, in milliseconds, between these points in the JDBC driver processing:

Interval beginning

When `start` is called.

Interval end

When `stop` is called.

`getApplicationTimeMillis` returns 0 if system monitoring is disabled. Calling this method without first calling the `stop` method results in an `SQLException`.

getCoreDriverTimeMicros

Format:

```
public long getCoreDriverTimeMicros()
    throws java.sql.SQLException
```

Returns the sum of elapsed monitored API times that were collected while system monitoring was enabled. The time is in microseconds.

A monitored API is a JDBC driver method for which processing time is collected. In general, elapsed times are monitored only for APIs that might result in network I/O or database server interaction. For example, `PreparedStatement.setXXX` methods and `ResultSet.getXXX` methods are not monitored.

Monitored API elapsed time includes the total time that is spent in the driver for a method call. This time includes any network I/O time and database server elapsed time.

A monitored API elapsed time interval is the difference, in microseconds, between these points in the JDBC driver processing:

Interval beginning

When a monitored API is called by the application.

Interval end

Immediately before the monitored API returns control to the application.

`getCoreDriverTimeMicros` returns 0 if system monitoring is disabled. Calling this method without first calling the stop method, or calling this method when the underlying JVM does not support reporting times in microseconds results in an `SQLException`.

getNetworkIOTimeMicros

Format:

```
public long getNetworkIOTimeMicros()  
    throws java.sql.SQLException
```

Returns the sum of elapsed network I/O times that were collected while system monitoring was enabled. The time is in microseconds.

Elapsed network I/O time includes the time to write and read DRDA data from network I/O streams. A network I/O elapsed time interval is the time interval to perform the following operations in the JDBC driver:

- Issue a TCP/IP command to send a DRDA message to the database server. This time interval is the difference, in microseconds, between points immediately before and after a write and flush to the network I/O stream is performed.
- Issue a TCP/IP command to receive DRDA reply messages from the database server. This time interval is the difference, in microseconds, between points immediately before and after a read on the network I/O stream is performed.

Network I/O time intervals are captured for all send and receive operations, including the sending of messages for commits and rollbacks.

The time spent waiting for network I/O might be impacted by delays in CPU dispatching at the database server for low-priority SQL requests.

`getNetworkIOTimeMicros` returns 0 if system monitoring is disabled. Calling this method without first calling the stop method, or calling this method when the underlying JVM does not support reporting times in microseconds results in an `SQLException`.

getServerTimeMicros

Format:

```
public long getServerTimeMicros()  
    throws java.sql.SQLException
```

Returns the sum of all reported database server elapsed times that were collected while system monitoring was enabled. The time is in microseconds.

The database server reports elapsed times under these conditions:

- The database server supports returning elapsed time data to the client.
DB2 Database for Linux, UNIX, and Windows Version 9.5 and later and DB2 for z/OS support this function.
- The database server performs operations that can be monitored. For example, database server elapsed time is not returned for commits or rollbacks.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows, and IBM Data Server Driver for JDBC and SQLJ type 4 connectivity: The database server elapsed time is defined as the elapsed time to parse the request data stream, process the command, and generate the reply data stream at the database server. Network time to receive or send the data stream is not included. The database server elapsed time interval is the difference, in microseconds, between these points in the database server processing:

Interval beginning

When the operating system dispatches the database server to process a TCP/IP message that is received from the JDBC driver.

Interval end

When the database server is ready to issue the TCP/IP command to return the reply message to the client.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS: The database server elapsed time interval is the difference, in microseconds, between these points in the JDBC driver native processing:

Interval beginning

The z/OS Store Clock (STCK) value when a JDBC driver native method calls the RRS attachment facility to process an SQL request.

Interval end

The z/OS Store Clock (STCK) value when control returns to the JDBC driver native method following an RRS attachment facility call to process an SQL request.

getServerTimeMicros returns 0 if system monitoring is disabled. Calling this method without first calling the stop method results in an SQLException.

start

Format:

```
public void start (int lapMode)  
    throws java.sql.SQLException
```

If the system monitor is enabled, start begins the collection of system monitoring data for a connection. Valid values for *lapMode* are RESET_TIMES or ACCUMULATE_TIMES.

Calling this method with system monitoring disabled does nothing. Calling this method more than once without an intervening stop call results in an SQLException.

stop

Format:

```
public void stop()
    throws java.sql.SQLException
```

If the system monitor is enabled, stop ends the collection of system monitoring data for a connection. After monitoring is stopped, monitored times can be obtained with the getXXX methods of DB2SystemMonitor.

Calling this method with system monitoring disabled does nothing. Calling this method without first calling start, or calling this method more than once without an intervening start call results in an SQLException.

Related tasks

Chapter 17, "System monitoring for the IBM Data Server Driver for JDBC and SQLJ," on page 549

DB2TraceManager class

The com.ibm.db2.jcc.DB2TraceManager class controls the global log writer.

The global log writer is driver-wide, and applies to all connections. The global log writer overrides any other JDBC log writers. In addition to starting the global log writer, the DB2TraceManager class provides the ability to suspend and resume tracing of any type of log writer. That is, the suspend and resume methods of the DB2TraceManager class apply to all current and future DriverManager log writers, DataSource log writers, or IBM Data Server Driver for JDBC and SQLJ-only connection-level log writers.

DB2TraceManager methods

getTraceManager

Format:

```
static public DB2TraceManager getTraceManager()
    throws java.sql.SQLException
```

Gets an instance of the global log writer.

setLogWriter

Formats:

```
public abstract void setLogWriter(String traceDirectory,
    String baseTraceFileName, int traceLevel)
    throws java.sql.SQLException
public abstract void setLogWriter(String traceFile,
    boolean fileAppend, int traceLevel)
    throws java.sql.SQLException
public abstract void setLogWriter(java.io.PrintWriter logWriter,
    int traceLevel)
    throws java.sql.SQLException
```

Enables a global trace. After setLogWriter is called, all calls for DataSource or Connection traces are discarded until DB2TraceManager.unsetLogWriter is called.

When setLogWriter is called, all future Connection or DataSource traces are redirected to a trace file or PrintWriter, depending on the form of setLogWriter that you use. If the global trace is suspended when setLogWriter is called, the specified settings take effect when the trace is resumed.

Parameter descriptions:

traceDirectory

Specifies a directory into which global trace information is written. This setting overrides the settings of the traceDirectory and logWriter properties for a DataSource or DriverManager connection.

When the form of setLogWriter with the traceDirectory parameter is used, the JDBC driver sets the traceFileAppend property to false when setLogWriter is called, which means that the existing log files are overwritten. Each JDBC driver connection is traced to a different file in the specified directory. The naming convention for the files in that directory depends on whether a non-null value is specified for baseTraceFileName:

- If a null value is specified for baseTraceFileName, a connection is traced to a file named traceFile_global_*n*.
n is the *n*th JDBC driver connection.
- If a non-null value is specified for baseTraceFileName, a connection is traced to a file named *baseTraceFileName*_global_*n*.
baseTraceFileName is the value of the baseTraceFileName parameter.
n is the *n*th JDBC driver connection.

baseTraceFileName

Specifies the stem for the names of the files into which global trace information is written. The combination of baseTraceFileName and traceDirectory determines the full path name for the global trace log files.

traceFileName

Specifies the file into which global trace information is written. This setting overrides the settings of the traceFile and logWriter properties for a DataSource or DriverManager connection.

When the form of setLogWriter with the traceFileName parameter is used, only one log file is written.

traceFileName can include a directory path.

logWriter

Specifies a character output stream to which all global log records are written.

This value overrides the logWriter property on a DataSource or DriverManager connection.

traceLevel

Specifies what to trace.

You can specify one or more of the following traces with the traceLevel parameter:

- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_NONE (X'00')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTION_CALLS (X'01')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_STATEMENT_CALLS (X'02')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_CALLS (X'04')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRIVER_CONFIGURATION (X'10')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS (X'20')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS (X'40')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_META_DATA (X'80')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_PARAMETER_META_DATA (X'100')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DIAGNOSTICS (X'200')

- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SQLJ (X'400')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_XA_CALLS` (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity for DB2 Database for Linux, UNIX, and Windows only) (X'800')
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_META_CALLS (X'2000')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DATASOURCE_CALLS (X'4000')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_LARGE_OBJECT_CALLS (X'8000')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_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`

fileAppend

Specifies whether to append to or overwrite the file that is specified by the `traceFile` parameter. `true` means that the existing file is not overwritten.

unsetLogWriter

Format:

```
public abstract void unsetLogWriter()
    throws java.sql.SQLException
```

Disables the global log writer override for future connections.

suspendTrace

Format:

```
public void suspendTrace()
    throws java.sql.SQLException
```

Suspends all global, Connection-level, or DataSource-level traces for current and future connections. `suspendTrace` can be called when the global log writer is enabled or disabled.

resumeTrace

Format:

```
public void resumeTrace()
    throws java.sql.SQLException
```

Resumes all global, Connection-level, or DataSource-level traces for current and future connections. `resumeTrace` can be called when the global log writer is enabled or disabled. If the global log writer is disabled, `resumeTrace` resumes Connection-level or DataSource-level traces. If the global log writer is enabled, `resumeTrace` resumes the global trace.

getLogWriter

Format:

```
public abstract java.io.PrintWriter getLogWriter()
    throws java.sql.SQLException
```

Returns the `PrintWriter` for the global log writer, if it is set. Otherwise, `getLogWriter` returns null.

getTraceFile

Format:

```
public abstract String getTraceFile()  
    throws java.sql.SQLException
```

Returns the name of the destination file for the global log writer, if it is set. Otherwise, `getTraceFile` returns null.

getTraceDirectory

Format:

```
public abstract String getTraceDirectory()  
    throws java.sql.SQLException
```

Returns the name of the destination directory for global log writer files, if it is set. Otherwise, `getTraceDirectory` returns null.

getTraceLevel

Format:

```
public abstract int getTraceLevel()  
    throws java.sql.SQLException
```

Returns the trace level for the global trace, if it is set. Otherwise, `getTraceLevel` returns -1 (`TRACE_ALL`).

getTraceFileAppend

Format:

```
public abstract boolean getTraceFileAppend()  
    throws java.sql.SQLException
```

Returns true if the global trace records are appended to the trace file. Otherwise, `getTraceFileAppend` returns false.

Related reference

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

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_T2Z0S (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 *n*th JDBC driver connection.
- If a non-null value is specified for `baseTraceFileName`, a connection is traced to a file named `baseTraceFileName_global_n`.
baseTraceFileName is the value of the `baseTraceFileName` parameter.
n is the *n*th JDBC driver connection.

baseTraceFileName

Specifies the stem for the names of the files into which global trace information is written. The combination of `baseTraceFileName` and `traceDirectory` determines the full path name for the global trace log files.

traceLevel

Specifies what to trace.

You can specify one or more of the following traces with the `traceLevel` parameter:

- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_NONE (X'00')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTION_CALLS (X'01')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_STATEMENT_CALLS (X'02')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_CALLS (X'04')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRIVER_CONFIGURATION (X'10')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS (X'20')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS (X'40')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_META_DATA (X'80')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_PARAMETER_META_DATA (X'100')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DIAGNOSTICS (X'200')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SQLJ (X'400')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_META_CALLS (X'2000')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DATASOURCE_CALLS (X'4000')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_LARGE_OBJECT_CALLS (X'8000')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_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:

```
public abstract boolean getTraceFileAppend()
    throws java.sql.SQLException
```

Returns true if trace records that are generated by the trace controller are appended to the trace file. Otherwise, getTraceFileAppend returns false.

getTraceDirectory

Format:

```
public void getTraceDirectory()
    throws java.sql.SQLException
```

Returns the name of the destination directory for trace records that are generated by the trace controller, if it is set. Otherwise, getTraceDirectory returns null.

getTraceLevel

Format:

```
public void getTraceLevel()
    throws java.sql.SQLException
```

Returns the trace level for the trace records that are generated by the trace controller, if it is set. Otherwise, getTraceLevel returns -1 (TRACE_ALL).

unsetLogWriter

Format:

```
public abstract void unsetLogWriter()
    throws java.sql.SQLException
```

Disables the global log writer override for future connections.

suspendTrace

Format:

```
public void suspendTrace()
    throws java.sql.SQLException
```

Suspends all global, Connection-level, or DataSource-level traces for current and future connections. suspendTrace can be called when the global log writer is enabled or disabled.

resumeTrace

Format:

```
public void resumeTrace()
    throws java.sql.SQLException
```

Resumes all global, Connection-level, or DataSource-level traces for current and future connections. resumeTrace can be called when the global log writer is

enabled or disabled. If the global log writer is disabled, `resumeTrace` resumes Connection-level or DataSource-level traces. If the global log writer is enabled, `resumeTrace` resumes the global trace.

DB2Types class

The `com.ibm.db2.jcc.DB2Types` class provides fields that define IBM Data Server Driver for JDBC and SQLJ-only data types.

DB2Types fields

The following constants define types codes only for the IBM Data Server Driver for JDBC and SQLJ.

- `public final static int BLOB_FILE = -100002`
- `public final static int CLOB_FILE = -100003`
- `public final static int CURSOR = -100008`
- `public final static int DECFLOAT = -100001`
- `public final static int XML_AS_BLOB_FILE = -100004`
- `public final static int XML_AS_CLOB_FILE = -100005`

DB2XADataSource class

`DB2XADataSource` is a factory for `XADataSource` objects. An object that implements this interface is registered with a naming service that is based on the Java Naming and Directory Interface (JNDI).

The `com.ibm.db2.jcc.DB2XADataSource` class extends the `com.ibm.db2.jcc.DB2BaseDataSource` class, and implements the `javax.sql.XADataSource`, `java.io.Serializable`, and `javax.naming.Referenceable` interfaces.

DB2XADataSource methods

`getDB2TrustedXAConnection`

Formats:

```
public Object[] getDB2TrustedXAConnection(String user,
    String password,
    java.util.Properties properties)
    throws java.sql.SQLException
public Object[] getDB2TrustedXAConnection(
    java.util.Properties properties)
    throws java.sql.SQLException
public Object[] getDB2TrustedXAConnection(
    org.ietf.jgss.GSSCredential gssCredential,
    java.util.Properties properties)
    throws java.sql.SQLException
```

An application server using a system authorization ID uses this method to establish a trusted connection.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 Database for Linux, UNIX, and Windows Version 9.5 or later, and DB2 for z/OS Version 9.1 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to 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 of `getDB2TrustedXAConnection` provides a user ID and password. The second form of `getDB2TrustedXAConnection` uses the user ID and password of the `DB2XADataSource` object. The third form of `getDB2TrustedXAConnection` is for connections that use Kerberos security.

Parameter descriptions:

user

The authorization ID that is used to establish the trusted connection.

password

The password for the authorization ID that is used to establish the trusted connection.

gssCredential

If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

properties

Properties for the connection.

getDB2TrustedPooledConnection

Format:

```
public Object[] getDB2TrustedPooledConnection(java.util.Properties properties)
    throws java.sql.SQLException
```

An application server using a system authorization ID uses this method to establish a trusted connection, using the user ID and password for the `DB2XADataSource` object.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 Database for Linux, UNIX, and Windows Version 9.5 or later, and DB2 for z/OS Version 9.1 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to 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 530

Related tasks

“Creating and deploying DataSource objects” on page 19

DB2Xml interface

The `com.ibm.db2.jcc.DB2Xml` interface is used for declaring Java objects for use with the DB2 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 DB2 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 DB2 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:

```
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:

```
public boolean isDB2XmlClosed()  
    throws SQLException
```

Indicates whether a com.ibm.jcc.DB2Xml object has been closed.

Related concepts

"XML column updates in SQLJ applications" on page 141

"XML column updates in JDBC applications" on page 70

"XML data retrieval in JDBC applications" on page 72

"XML data retrieval in SQLJ applications" on page 143

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.

DB2XmlAsBlobFileReference

Format:

```
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 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 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.

JDBC differences between the current IBM Data Server Driver for JDBC and SQLJ and earlier DB2 JDBC drivers

Before you can upgrade your JDBC applications from older drivers to the IBM Data Server Driver for JDBC and SQLJ, you need to understand the differences between those drivers.

Important: The JDBC/SQLJ Driver for OS/390® and z/OS is no longer supported. This information is provided only to help you diagnose problems in your applications after you upgrade to the IBM Data Server Driver for JDBC and SQLJ.

Supported methods

For a list of methods that the IBM Data Server Driver for JDBC and SQLJ supports, see "Driver support for JDBC APIs".

Use of progressive streaming by the JDBC drivers

For IBM Data Server Driver for JDBC and SQLJ, Version 3.50 and later, use of progressive streaming is the default for LOB retrieval, for connections to DB2 Database for Linux, UNIX, and Windows Version 9.5 and later.

Progressive streaming is supported in the IBM Data Server Driver for JDBC and SQLJ Version 3.1 and later, but for IBM Data Server Driver for JDBC and SQLJ version 3.2 and later, use of progressive streaming is the default for LOB and XML retrieval, for connections to DB2 for z/OS Version 9.1 and later.

Previous versions of the IBM Data Server Driver for JDBC and SQLJ did not support progressive streaming.

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

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

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

To avoid errors that are due to this changed behavior, you need to take one of the following actions:

- Modify your applications.
Applications that retrieve LOB data into application variables can manipulate the data in those application variables only until the cursors that were used to retrieve the data are moved or closed.
- Disable progressive streaming by setting the `progressiveStreaming` property to `DB2BaseDataSource.NO (2)`.

ResultSetMetaData values for IBM Data Server Driver for JDBC and SQLJ version 4.0 and later

For the IBM Data Server Driver for JDBC and SQLJ version 4.0 and later, the default behavior of `ResultSetMetaData.getColumnNames` and `ResultSetMetaData.getColumnLabels` 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.getColumnNames` and `ResultSetMetaData.getColumnLabels` 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.

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.

Use of LOB locators by the JDBC drivers

The IBM Data Server Driver for JDBC and SQLJ uses LOB locators internally under the following circumstances:

- Always, for fetching data from scrollable cursors, when the data source or the JDBC driver does not support progressive streaming
- Never, for fetching data from stored procedure output parameters
- If the `fullyMaterializeLobData` connection property is set to `false`, for fetching data from non-scrollable cursors

The JDBC/SQLJ Driver for OS/390 and z/OS did not support LOB locators.

Support for scrollable and updatable ResultSets

The IBM Data Server Driver for JDBC and SQLJ supports scrollable and updatable `ResultSets`.

The JDBC/SQLJ driver for z/OS support only non-scrollable and non-updatable `ResultSets`.

Difference in URL syntax

The syntax of the `url` parameter in the `DriverManager.getConnection` method is different for each driver. See the following topics for more information:

- "Connect to a data source using the `DriverManager` interface with the IBM Data Server Driver for JDBC and SQLJ"

Difference in error codes and SQLSTATEs returned for driver errors

The IBM Data Server Driver for JDBC and SQLJ does not use existing SQLCODEs or SQLSTATEs for internal errors, as the other drivers do. See "Error codes issued by the IBM Data Server Driver for JDBC and SQLJ" and "SQLSTATEs issued by the IBM Data Server Driver for JDBC and SQLJ".

The JDBC/SQLJ driver for z/OS returns SQLSTATE FFFFFF when internal errors occur.

Handling of null SQLSTATEs

By default, the IBM Data Server Driver for JDBC and SQLJ returns a null SQLSTATE value when an SQLWarning or SQLException object contains a null SQLSTATE value. The JDBC/SQLJ driver for z/OS returns SQLSTATE "FFFFF". To override this incompatibility between the drivers, you can set the `db2.jcc.defaultSQLState` configuration property for the IBM Data Server Driver for JDBC and SQLJ. If you specify `db2.jcc.defaultSQLState` with no value, the IBM Data Server Driver for JDBC and SQLJ returns "FFFFF" when the SQLWarning or SQLException object has a null value for SQLSTATE.

Security mechanisms

The JDBC drivers have different security mechanisms.

For information on IBM Data Server Driver for JDBC and SQLJ security mechanisms, see "Security under the IBM Data Server Driver for JDBC and SQLJ".

How connection properties are set

With IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, you set properties for a connection by setting the properties for the associated `DataSource` or `Connection` object.

With IBM Data Server Driver for JDBC and SQLJ type 2 connectivity, you set properties for a connection in one of these ways:

- You can set properties only for a connection by setting the properties for the associated `DataSource` or `Connection` object.
- You can set driver-wide properties through an optional run-time properties file.

For the JDBC/SQLJ driver for z/OS, you set properties through the JDBC/SQLJ run-time properties file.

Support for read-only connections

With the IBM Data Server Driver for JDBC and SQLJ, you can make a connection read-only through the `readOnly` property for a `Connection` or `DataSource` object.

The JDBC/SQLJ driver for z/OS does not support read-only connections.

Results returned from `ResultSet.getString` for a BIT DATA column

The IBM Data Server Driver for JDBC and SQLJ returns data from a `ResultSet.getString` call for a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA column as a lowercase hexadecimal string.

The JDBC/SQLJ Driver for OS/390 and z/OS returns the data in the encoding scheme of the caller.

Compatibility of DATE SQL type and `java.lang.String` type

With the IBM Data Server Driver for JDBC and SQLJ, the SQL DATE data type is compatible only with the `java.sql.Date` data type.

With the JDBC/SQLJ driver for z/OS and the DB2 JDBC Type 2 Driver, the `java.lang.String` data type is also compatible with the SQL DATE data type.

Date and time adjustment for input and output values that do not correspond to real dates and times

During update or retrieval of data in SQL DATE, TIME, or TIMESTAMP columns, the IBM Data Server Driver for JDBC and SQLJ adjusts date and time values that do not correspond to real dates and times. For example, if you update a TIMESTAMP column with the value 2007-12-31 24:00:00.0, the IBM Data Server Driver for JDBC and SQLJ adjusts the value to 2008-01-01 00:00:00.0. If you update a TIMESTAMP column with the value 9999-12-31 24:00:00.0, the IBM Data Server Driver for JDBC and SQLJ throws an exception because the adjusted value, 10000-01-01 00:00:00.0, is invalid.

The JDBC/SQLJ Driver for OS/390 does no adjustment of date or time values that do not correspond to real dates or times. That driver passes the values to and from the database as they are. For example, if you update a TIMESTAMP column with the value 9999-12-31 24:00:00.0 under the JDBC/SQLJ Driver for OS/390, no exception is thrown. See the information on date, time, and timestamp values that can cause problems in JDBC and SQLJ applications for more information.

When an exception is thrown for `PreparedStatement.setXXXStream` with a length mismatch

When you use the `PreparedStatement.setBinaryStream`, `PreparedStatement.setCharacterStream`, or `PreparedStatement.setUnicodeStream` method, the *length* parameter value must match the number of bytes in the input stream.

If the numbers of bytes do not match, the IBM Data Server Driver for JDBC and SQLJ does not throw an exception until the subsequent `PreparedStatement.executeUpdate` method executes. Therefore, for the IBM Data Server Driver for JDBC and SQLJ, some data might be sent to the server when the lengths do not match. That data is truncated or padded by the server. The calling application needs to issue a rollback request to undo the database updates that include the truncated or padded data.

The JDBC/SQLJ Driver for OS/390 and z/OS throws an exception after the `PreparedStatement.setBinaryStream`, `PreparedStatement.setCharacterStream`, or `PreparedStatement.setUnicodeStream` method executes.

Default mappings for PreparedStatement.setXXXStream

With the IBM Data Server Driver for JDBC and SQLJ, when you use the `PreparedStatement.setBinaryStream`, `PreparedStatement.setCharacterStream`, or `PreparedStatement.setUnicodeStream` method, and no information about the data type of the target column is available, the input data is mapped to a BLOB or CLOB data type.

For the JDBC/SQLJ driver for z/OS, the input data is mapped to a VARCHAR FOR BIT DATA or VARCHAR data type.

Differences in character conversion

When character data is transferred between a client and a server, the data must be converted to a form that the receiver can process.

For the IBM Data Server Driver for JDBC and SQLJ, character data that is sent from the data source to the client is converted using Java's built-in character converters. The conversions that the IBM Data Server Driver for JDBC and SQLJ supports are limited to those that are supported by the underlying JRE implementation.

A IBM Data Server Driver for JDBC and SQLJ client using type 4 connectivity sends data to the data source as Unicode UTF-8.

The conversions that the JDBC/SQLJ driver for z/OS and the IBM Data Server Driver for JDBC and SQLJ with type 2 connectivity support are also limited to those that are supported by the underlying JRE implementation.

Those drivers use CCSID information from the data source if it is available. The drivers convert input parameter data to the CCSID of the data source before sending the data. If target CCSID information is not available, the drivers send the data as Unicode UTF-8.

Implicit or explicit data type conversion for input parameters

If you execute a `PreparedStatement.setXXX` method, and the resulting data type from the `setXXX` method does not match the data type of the table column to which the parameter value is assigned, the driver returns an error unless data type conversion occurs.

With the IBM Data Server Driver for JDBC and SQLJ, conversion to the correct SQL data type occurs implicitly if the target data type is known and if the `deferPrepares` and `sendDataAsIs` connection properties are set to false. In this case, the implicit values override any explicit values in the `setXXX` call. If the `deferPrepares` connection property or the `sendDataAsIs` connection property is set to true, you must use the `PreparedStatement.setObject` method to convert the parameter to the correct SQL data type.

For the JDBC/SQLJ driver for z/OS, if the data type of a parameter does not match its default SQL data type, you must use the `PreparedStatement.setObject` method to convert the parameter to the correct SQL data type.

Result of using `getBoolean` to retrieve a value from a CHAR column

With the IBM Data Server Driver for JDBC and SQLJ, when you execute `ResultSet.getBoolean` or `CallableStatement.getBoolean` to retrieve a Boolean value from a CHAR column, and the column contains the value "false" or "0", the value `false` is returned. If the column contains any other value, `true` is returned.

With the JDBC/SQLJ driver for z/OS, when you execute `ResultSet.getBoolean` or `CallableStatement.getBoolean` to retrieve a Boolean value from a CHAR column, and the column contains the value "0", the value `false` is returned. If the column contains any other value, `true` is returned.

Related concepts

"LOBs in JDBC applications with the IBM Data Server Driver for JDBC and SQLJ" on page 50

Related tasks

"Learning about a data source using `DatabaseMetaData` methods" on page 21

"Creating and modifying database objects using the `Statement.executeUpdate` method" on page 25

"Updating data in tables using the `PreparedStatement.executeUpdate` method" on page 26

"Making batch updates in JDBC applications" on page 28

"Calling stored procedures in JDBC applications" on page 46

Chapter 9, "Migrating from the JDBC/SQLJ Driver for OS/390 and z/OS to the IBM Data Server Driver for JDBC and SQLJ," on page 465

Related reference

"Data types that map to database data types in Java applications" on page 191

"Properties for the IBM Data Server Driver for JDBC and SQLJ" on page 201

"Driver support for JDBC APIs" on page 257

"Date, time, and timestamp values that can cause problems in JDBC and SQLJ applications" on page 198

JDBC differences between versions of the IBM Data Server Driver for JDBC and SQLJ

Before you can upgrade your JDBC applications from older to newer versions of the IBM Data Server Driver for JDBC and SQLJ, you need to understand the differences between those drivers.

Supported methods

For a list of methods that the IBM Data Server Driver for JDBC and SQLJ supports, see "Driver support for JDBC APIs".

Use of progressive streaming by the JDBC drivers

For IBM Data Server Driver for JDBC and SQLJ, Version 3.50 and later, progressive streaming, which is also known as dynamic data format, behavior is the default for LOB retrieval, for connections to DB2 Database for Linux, UNIX, and Windows Version 9.5 and later.

Progressive streaming is supported in the IBM Data Server Driver for JDBC and SQLJ Version 3.1 and later, but for IBM Data Server Driver for JDBC and SQLJ version 3.2 and later, progressive streaming behavior is the default for LOB and XML retrieval, for connections to DB2 for z/OS Version 9.1 and later.

Previous versions of the IBM Data Server Driver for JDBC and SQLJ did not support progressive streaming.

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

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

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

To avoid errors that are due to this changed behavior, you need to take one of the following actions:

- Modify your applications.
Applications that retrieve LOB data into application variables can manipulate the data in those application variables only until the cursors that were used to retrieve the data are moved or closed.
- Disable progressive streaming by setting the `progressiveStreaming` property to `DB2BaseDataSource.NO (2)`.

ResultSetMetaData values for IBM Data Server Driver for JDBC and SQLJ version 4.0 and later

For the IBM Data Server Driver for JDBC and SQLJ version 4.0 and later, the default behavior of `ResultSetMetaData.getColumnNames` and `ResultSetMetaData.getColumnLabels` 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.getColumnNames` and `ResultSetMetaData.getColumnLabels` 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.

Initial value of the CURRENT_CLIENT_ACCTNG special register

For a JDBC or SQLJ application that runs under the IBM Data Server Driver for JDBC and SQLJ version 2.6 or later, using type 4 connectivity, the initial value for the DB2 for z/OS `CURRENT_CLIENT_ACCTNG` special register is the concatenation of the DB2 for z/OS version and the value of the `clientWorkStation` property. For any other JDBC driver, version, and connectivity, the initial value is not set.

Properties that control the use of multi-row FETCH

Before version 3.7 and version 3.51 of the IBM Data Server Driver for JDBC and SQLJ, multi-row FETCH support was enabled and disabled through the `useRowsetCursor` property, and was available only for scrollable cursors, and for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS. Starting with version 3.7 and 3.51:

- For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, the IBM Data Server Driver for JDBC and SQLJ uses only the `enableRowsetSupport` property to determine whether to use multi-row FETCH for scrollable or forward-only cursors.
- For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows, or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 Database for Linux, UNIX, and Windows, the IBM Data Server Driver for JDBC and SQLJ uses the `enableRowsetSupport` property to determine whether to use multi-row FETCH for scrollable cursors, if `enableRowsetSupport` is set. If `enableRowsetSupport` is not set, the driver uses the `useRowsetCursor` property to determine whether to use multi-row FETCH.

JDBC 1 positioned updates and deletes and multi-row FETCH

Before version 3.7 and version 3.51 of the IBM Data Server Driver for JDBC and SQLJ, multi-row FETCH from DB2 for z/OS tables was controlled by the `useRowsetCursor` property. If an application contained JDBC 1 positioned update or delete operations, and multi-row FETCH support was enabled, the IBM Data Server Driver for JDBC and SQLJ permitted the update or delete operations, but unexpected updates or deletes might occur.

Starting with version 3.7 and 3.51 of the IBM Data Server Driver for JDBC and SQLJ, the `enableRowsetSupport` property enables or disables multi-row FETCH from DB2 for z/OS tables or DB2 Database for Linux, UNIX, and Windows tables. The `enableRowsetSupport` property overrides the `useRowsetCursor` property. If multi-row FETCH is enabled through the `enableRowsetSupport` property, and an application contains a JDBC 1 positioned update or delete operation, the IBM Data Server Driver for JDBC and SQLJ throws an `SQLException`.

Related concepts

“Examples of `ResultSetMetaData.getColumnNames` and `ResultSetMetaData.getColumnLabels` values”

Related tasks

“Upgrading the IBM Data Server Driver for JDBC and SQLJ to a new version” on page 460

Examples of `ResultSetMetaData.getColumnNames` and `ResultSetMetaData.getColumnLabels` values

For the IBM Data Server Driver for JDBC and SQLJ version 4.0 and later, the default behavior of `ResultSetMetaData.getColumnNames` and `ResultSetMetaData.getColumnLabels` differs from the default behavior for earlier JDBC drivers. You can use the `useJDBC4ColumnNameAndLabelSemantics` property to change this behavior.

The following examples show the values that are returned for IBM Data Server Driver for JDBC and SQLJ Version 4.0, and for previous JDBC drivers, when the useJDBC4ColumnNameAndLabelSemantics property is not set.

All queries use a table that is defined like this:

```
CREATE TABLE MYTABLE(INTCOL INT)
```

Example: The following query contains an AS CLAUSE, which defines a label for a column in the result set:

```
SELECT MYCOL AS MYLABEL FROM MYTABLE
```

The following table lists the ResultSetMetaData.getColumnNames and ResultSetMetaData.getColumnLabels values that are returned for the query:

Table 88. ResultSetMetaData.getColumnNames and ResultSetMetaData.getColumnLabels before and after IBM Data Server Driver for JDBC and SQLJ Version 4.0 for a query with an AS CLAUSE

Target data source	Behavior before IBM Data Server Driver for JDBC and SQLJ Version 4.0		Behavior for IBM Data Server Driver for JDBC and SQLJ Version 4.0 and later	
	getColumnNames value	getColumnLabels value	getColumnNames value	getColumnLabels value
DB2 Database for Linux, UNIX, and Windows	MYLABEL	MYLABEL	MYCOL	MYLABEL
IBM Informix Dynamic Server	MYLABEL	MYLABEL	MYCOL	MYLABEL
DB2 for z/OS Version 8 or later, and DB2 UDB for iSeries V5R3 and later	MYLABEL	MYLABEL	MYCOL	MYLABEL
DB2 for z/OS Version 7, and DB2 UDB for iSeries V5R2	MYLABEL	MYLABEL	MYLABEL	MYLABEL

Example: The following query contains no AS clause:

```
SELECT MYCOL FROM MYTABLE
```

The ResultSetMetaData.getColumnNames and ResultSetMetaData.getColumnLabels methods on the query return MYCOL, regardless of the target data source.

Example: On a DB2 for z/OS or DB2 for i data source, a LABEL ON statement is used to define a label for a column:

```
LABEL ON COLUMN MYTABLE.MYCOL IS 'LABELONCOL'
```

The following query contains an AS CLAUSE, which defines a label for a column in the ResultSet:

```
SELECT MYCOL AS MYLABEL FROM MYTABLE
```

The following table lists the ResultSetMetaData.getColumnNames and ResultSetMetaData.getColumnLabels values that are returned for the query.

Table 89. *ResultSetMetaData.getColumnNames* 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

Target data source	Behavior before IBM Data Server Driver for JDBC and SQLJ Version 4.0		Behavior for IBM Data Server Driver for JDBC and SQLJ Version 4.0 and later	
	getColumnNames value	getColumnLabel value	getColumnNames value	getColumnLabel value
DB2 for z/OS Version 8 or later, and DB2 UDB for iSeries V5R3 and later	MYLABEL	LABELONCOL	MYCOL	MYLABEL
DB2 for z/OS Version 7, and DB2 UDB for iSeries V5R2	MYLABEL	LABELONCOL	MYCOL	LABELONCOL

Example: On a DB2 for z/OS or DB2 for i data source, a LABEL ON statement is used to define a label for a column:

```
LABEL ON COLUMN MYTABLE.MYCOL IS 'LABELONCOL'
```

The following query contains no AS CLAUSE:

```
SELECT MYCOL FROM MYTABLE
```

The following table lists the *ResultSetMetaData.getColumnNames* and *ResultSetMetaData.getColumnLabel* values that are returned for the query.

Table 90. *ResultSetMetaData.getColumnNames* 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 no AS CLAUSE

Target data source	Behavior before IBM Data Server Driver for JDBC and SQLJ Version 4.0		Behavior for IBM Data Server Driver for JDBC and SQLJ Version 4.0	
	getColumnNames value	getColumnLabel value	getColumnNames value	getColumnLabel value
DB2 for z/OS Version 8 or later, and DB2 UDB for i5/OS V5R3 and later	MYCOL	LABELONCOL	MYCOL	MYCOL
DB2 for z/OS Version 7, and DB2 UDB for i5/OS V5R2	MYCOL	LABELONCOL	MYLABEL	LABELONCOL

Related reference

“JDBC differences between versions of the IBM Data Server Driver for JDBC and SQLJ” on page 400

SQLJ differences between the IBM Data Server Driver for JDBC and SQLJ and other DB2 JDBC drivers

There are a number of differences between the IBM Data Server Driver for JDBC and SQLJ and the older JDBC drivers. When you move to the IBM Data Server Driver for JDBC and SQLJ, you need to modify your SQLJ programs to account for those differences.

Important: The JDBC/SQLJ Driver for OS/390 and z/OS is no longer supported. This information is provided only to help you diagnose problems in your applications after migration to the IBM Data Server Driver for JDBC and SQLJ.

SQLJ support in the IBM Data Server Driver for JDBC and SQLJ differs from SQLJ support in the other DB2 JDBC drivers in the following areas:

Connection associated with the default connection context

With SQLJ, it is possible, although not recommended, to let the driver implicitly establish a connection to a data source, and to execute SQL under that implicitly established connection. An application does this by omitting code that obtains a connection and by omitting connection context objects from SQLJ executable clauses. For an application that is written for the JDBC/SQLJ Driver for OS/390 and z/OS, the result is unambiguous because there is only one type of connectivity (type 2), and there is a single default data source (the local location). However, with the IBM Data Server Driver for JDBC and SQLJ, there are multiple ways to make a connection. If you do not explicitly specify the connectivity type and the data source, the SQLJ runtime code cannot determine how to make the connection. One way to solve the problem, without modifying your applications, is to define a DataSource named jdbc/defaultDataSource and register that DataSource with a JNDI provider. That DataSource needs to have all the information that is required to make a connection. If you use WebSphere Application Server, you can use the JNDI service that is provided by WebSphere Application Server.

Production of DBRMs during SQLJ program preparation

The SQLJ program preparation process for the IBM Data Server Driver for JDBC and SQLJ does not produce DBRMs. Therefore, with the IBM Data Server Driver for JDBC and SQLJ, you can produce DB2 packages only by using the IBM Data Server Driver for JDBC and SQLJ utilities.

Difference in connection techniques

The connection techniques that are available, and the driver names and URLs that are used for those connection techniques, vary from driver to driver. See "Connect to a data source using SQLJ" for more information.

Support for scrollable and updatable iterators

SQLJ with the IBM Data Server Driver for JDBC and SQLJ supports scrollable and updatable iterators.

The JDBC/SQLJ driver for z/OS support only non-scrollable and non-updatable iterators.

Dynamic execution of SQL statements under WebSphere Application Server

For WebSphere Application Server for z/OS Version 5.0.2 and above, if you customize your SQLJ program, SQL statements are executed statically.

Related tasks

Chapter 9, “Migrating from the JDBC/SQLJ Driver for OS/390 and z/OS to the IBM Data Server Driver for JDBC and SQLJ,” on page 465

Error codes issued by the IBM Data Server Driver for JDBC and SQLJ

Error codes in the ranges +4200 to +4299, +4450 to +4499, -4200 to -4299, and -4450 to -4499 are reserved for the IBM Data Server Driver for JDBC and SQLJ.

When you call the `SQLException.getMessage` method after a IBM Data Server Driver for JDBC and SQLJ error occurs, a string is returned that includes:

- Whether the connection is a type 2 or type 4 connection
- Diagnostic information for IBM Software Support
- The level of the driver
- An explanatory message
- The error code
- The SQLSTATE

For example:

```
[jcc][t4][20128][12071][3.50.54] Invalid queryBlockSize specified: 1,048,576,012.  
Using default query block size of 32,767.  ERRORCODE=0, SQLSTATE=
```

Currently, the IBM Data Server Driver for JDBC and SQLJ issues the following error codes:

Table 91. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ

Error Code	Message text and explanation	SQLSTATE
+4204	Errors were encountered and tolerated as specified by the RETURN DATA UNTIL clause. Explanation: Tolerated errors include federated connection, authentication, and authorization errors. This warning applies only to connections to DB2 Database for Linux, UNIX, and Windows servers. It is issued only when a cursor operation, such as a <code>ResultSet.next</code> or <code>ResultSet.previous</code> call, returns false.	02506
+4222	<i>text-from-getMessage</i> Explanation: A warning condition occurred during connection to the data source. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
+4223	<i>text-from-getMessage</i> Explanation: A warning condition occurred during initialization. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
+4225	<i>text-from-getMessage</i> Explanation: A warning condition occurred when data was sent to a server or received from a server. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	

Table 91. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ (continued)

Error Code	Message text and explanation	SQLSTATE
+4226	<i>text-from-getMessage</i> Explanation: A warning condition occurred during customization or bind. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
+4228	<i>text-from-getMessage</i> Explanation: An warning condition occurred that does not fit in another category. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
+4450	Feature not supported: <i>feature-name</i>	
+4460	<i>text-from-getMessage</i> Explanation: The specified value is not a valid option. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
+4461	<i>text-from-getMessage</i> Explanation: The specified value is invalid or out of range. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
+4462	<i>text-from-getMessage</i> Explanation: A required value is missing. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
+4470	<i>text-from-getMessage</i> Explanation: The requested operation cannot be performed because the target resource is closed. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
+4471	<i>text-from-getMessage</i> Explanation: The requested operation cannot be performed because the target resource is in use. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
+4472	<i>text-from-getMessage</i> Explanation: The requested operation cannot be performed because the target resource is because the target resource is unavailable. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	

Table 91. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ (continued)

Error Code	Message text and explanation	SQLSTATE
+4474	<i>text-from-getMessage</i> Explanation: The requested operation cannot be performed because the target resource cannot be changed. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
-4200	Invalid operation: An invalid COMMIT or ROLLBACK has been called in an XA environment during a Global Transaction. Explanation: An application that was in a global transaction in an XA environment issued a commit or rollback. A commit or rollback operation in a global transaction is invalid.	2D521
-4201	Invalid operation: <code>setAutoCommit(true)</code> is not allowed during Global Transaction. Explanation: An application that was in a global transaction in an XA environment executed the <code>setAutoCommit(true)</code> statement. Issuing <code>setAutoCommit(true)</code> in a global transaction is invalid.	2D521
-4203	Error executing <i>function</i> . Server returned <i>rc</i> . : 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 <code>SQLException</code> that is thrown by the current SQL statement.	
-4210	Timeout getting a transport object from pool.	57033
-4211	Timeout getting an object from pool.	57033
-4212	Sysplex member unavailable.	
-4213	Timeout.	57033
-4214	<i>text-from-getMessage</i> Explanation: Authorization failed. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	28000
-4220	<i>text-from-getMessage</i> Explanation: An error occurred during character conversion. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
-4221	<i>text-from-getMessage</i> Explanation: An error occurred during encryption or decryption. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	

Table 91. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ (continued)

Error Code	Message text and explanation	SQLSTATE
-4222	<i>text-from-getMessage</i> Explanation: An error occurred during connection to the data source. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
-4223	<i>text-from-getMessage</i> Explanation: An error occurred during initialization. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
-4224	<i>text-from-getMessage</i> Explanation: An error occurred during resource cleanup. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
-4225	<i>text-from-getMessage</i> Explanation: An error occurred when data was sent to a server or received from a server. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
-4226	<i>text-from-getMessage</i> Explanation: An error occurred during customization or bind. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
-4227	<i>text-from-getMessage</i> Explanation: An error occurred during reset. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
-4228	<i>text-from-getMessage</i> Explanation: An error occurred that does not fit in another category. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
-4450	Feature not supported: <i>feature-name</i>	0A504
-4460	<i>text-from-getMessage</i> Explanation: The specified value is not a valid option. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	

Table 91. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ (continued)

Error Code	Message text and explanation	SQLSTATE
-4461	<i>text-from-getMessage</i> Explanation: The specified value is invalid or out of range. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	42815
-4462	<i>text-from-getMessage</i> Explanation: A required value is missing. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
-4463	<i>text-from-getMessage</i> Explanation: The specified value has a syntax error. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	42601
-4470	<i>text-from-getMessage</i> Explanation: The requested operation cannot be performed because the target resource is closed. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
-4471	<i>text-from-getMessage</i> Explanation: The requested operation cannot be performed because the target resource is in use. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
-4472	<i>text-from-getMessage</i> Explanation: The requested operation cannot be performed because the target resource is unavailable. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
-4473	<i>text-from-getMessage</i> Explanation: The requested operation cannot be performed because the target resource is no longer available. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	
-4474	<i>text-from-getMessage</i> Explanation: The requested operation cannot be performed because the target resource cannot be changed. User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.	

Table 91. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ (continued)

Error Code	Message text and explanation	SQLSTATE
-4475	<p><i>text-from-getMessage</i></p> <p>Explanation: The requested operation cannot be performed because access to the target resource is restricted.</p> <p>User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.</p>	
-4476	<p><i>text-from-getMessage</i></p> <p>Explanation: The requested operation cannot be performed because the operation is not allowed on the target resource.</p> <p>User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.</p>	
-4496	An SQL OPEN for a held cursor was issued on an XA connection. The JDBC driver does not allow a held cursor to be opened on the database server for an XA connection.	
-4497	The application must issue a rollback. The unit of work has already been rolled back in the DB2 server, but other resource managers involved in the unit of work might not have rolled back their changes. To ensure integrity of the application, all SQL requests are rejected until the application issues a rollback.	
-4498	<p>A connection failed but has been reestablished. Host name or IP address: <i>host-name</i>, service name or port number: <i>port</i>, special register modification indicator: <i>rc</i>.</p> <p>Explanation: <i>host-name</i> and <i>port</i> indicate the data source at which the connection is reestablished. <i>rc</i> indicates whether SQL statements that set special register values were executed again:</p> <ol style="list-style-type: none"> 1 SQL statements that set special register values were executed again. 2 SQL statements that set special register values might not have been executed again. <p>For client reroute against DB2 for z/OS servers, special register values that were set after the last commit point are not re-established.</p> <p>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.</p>	
-4499	<p><i>text-from-getMessage</i></p> <p>Explanation: A fatal error occurred that resulted in a disconnect from the data source. The existing connection has become unusable.</p> <p>User response: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.</p>	08001 or 58009
-30108	Client reroute exception for the Sysplex.	08506

Table 91. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ (continued)

Error Code	Message text and explanation	SQLSTATE
-99999	The IBM Data Server Driver for JDBC and SQLJ issued an error that does not yet have an error code.	

Related tasks

“Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ” on page 84

“Handling SQL errors in an SQLJ application” on page 149

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 92. SQLSTATES returned by the IBM Data Server Driver for JDBC and SQLJ

SQLSTATE class	SQLSTATE	Description
01xxx		Warning
02xxx		No data
	02501	The cursor position is not valid for a FETCH of the current row.
	02506	Tolerable error
08xxx		Connection exception
	08001	The application requester is unable to establish the connection.
	08003	A connection does not exist
	08004	The application server rejected establishment of the connection
	08506	Client reroute exception
0Axxx		Feature not supported
	0A502	The action or operation is not enabled for this database instance
	0A504	The feature is not supported by the driver
22xxx		Data exception
	22007	The string representation of a datetime value is invalid
	22021	A character is not in the coded character set
23xxx		Constraint violation
	23502	A value that is inserted into a column or updates a column is null, but the column cannot contain null values.
24xxx		Invalid cursor state
	24501	The identified cursor is not open
28xxx		Authorization exception
	28000	Authorization name is invalid.

Table 92. SQLSTATEs returned by the IBM Data Server Driver for JDBC and SQLJ (continued)

SQLSTATE class	SQLSTATE	Description
2Dxxx		Invalid transaction termination
	2D521	SQL COMMIT or ROLLBACK are invalid in the current operating environment.
34xxx		Invalid cursor name
	34000	Cursor name is invalid.
3Bxxx		Invalid savepoint
	3B503	A SAVEPOINT, RELEASE SAVEPOINT, or ROLLBACK TO SAVEPOINT statement is not allowed in a trigger or global transaction.
40xxx		Transaction rollback
42xxx		Syntax error or access rule violation
	42601	A character, token, or clause is invalid or missing
	42734	A duplicate parameter name, SQL variable name, cursor name, condition name, or label was detected.
	42807	The INSERT, UPDATE, or DELETE is not permitted on this object
	42808	A column identified in the insert or update operation is not updateable
	42815	The data type, length, scale, value, or CCSID is invalid
	42820	A numeric constant is too long, or it has a value that is not within the range of its data type
	42968	The connection failed because there is no current software license.
57xxx		Resource not available or operator intervention
	57033	A deadlock or timeout occurred without automatic rollback
58xxx		System error
	58008	Execution failed due to a distribution protocol error that will not affect the successful execution of subsequent DDM commands or SQL statements
	58009	Execution failed due to a distribution protocol error that caused deallocation of the conversation
	58012	The bind process with the specified package name and consistency token is not active
	58014	The DDM command is not supported
	58015	The DDM object is not supported
	58016	The DDM parameter is not supported
	58017	The DDM parameter value is not supported

“Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ” on page 84

“Handling SQL errors in an SQLJ application” on page 149

To determine the version of the IBM Data Server Driver for JDBC and SQLJ, as well as information about the environment in which the driver is running, run the DB2Jcc utility on the UNIX System Services command line.

```
» java com.ibm.db2.jcc.DB2Jcc -version -configuration -help
```

Specifies that the IBM Data Server Driver for JDBC and SQLJ displays its name and version.

Specifies that the IBM Data Server Driver for JDBC and SQLJ displays its name and version, and information about its environment, such as information about the Java runtime environment, operating system, path information, and license restrictions.

Specifies that the DB2Jcc utility describes each of the options that it supports. If any other options are specified with -help, they are ignored.

The following output is the result of invoking DB2Jcc with the `-configuration` parameter.

```
(myid@mymachine) /home/myusrid $ java com.ibm.db2.jcc.DB2Jcc -version
[jcc] Driver: IBM DB2 JDBC Universal Driver Architecture 3.50.137

(myid@mymachine) /home/myusrid $ java com.ibm.db2.jcc.DB2Jcc -configuration
[jcc] BEGIN TRACE_DRIVER_CONFIGURATION
[jcc] Driver: IBM DB2 JDBC Universal Driver Architecture 3.50.137
[jcc] Compatible JRE versions: { 1.4, 1.5 }
[jcc] Target server licensing restrictions: { z/OS: enabled; SQLDS: enabled; iSeries: enabled; DB2 for Unix/Windows: enabled; Cloudscape: enabled; Informix: enabled }
[jcc] Range checking enabled: true
[jcc] Bug check level: 0xff
[jcc] Default fetch size: 64
[jcc] Default isolation: 2
[jcc] Collect performance statistics: false
[jcc] No security manager detected.
[jcc] Detected local client host: lead.svl.ibm.com/9.30.10.102
[jcc] Access to package sun.io is permitted by security manager.
```

```

[jcc] JDBC 1 system property jdbc.drivers = null
[jcc] Java Runtime Environment version 1.4.2
[jcc] Java Runtime Environment vendor = IBM Corporation
[jcc] Java vendor URL = http://www.ibm.com/
[jcc] Java installation directory = /wsdb/v91/bldsupp/AIX5L64/jdk1.4.2_sr1/sh/..
/jre
[jcc] Java Virtual Machine specification version = 1.0
[jcc] Java Virtual Machine specification vendor = Sun Microsystems Inc.
[jcc] Java Virtual Machine specification name = Java Virtual Machine Specificati
on
[jcc] Java Virtual Machine implementation version = 1.4.2
[jcc] Java Virtual Machine implementation vendor = IBM Corporation
[jcc] Java Virtual Machine implementation name = Classic VM
[jcc] Java Runtime Environment specification version = 1.4
[jcc] Java Runtime Environment specification vendor = Sun Microsystems Inc.
[jcc] Java Runtime Environment specification name = Java Platform API Specificat
ion
[jcc] Java class format version number = 48.0
[jcc] Java class path = ../home2/myusrid/sqllib/java/db2java.zip:/lib/classes.z
ip:/home2/myusrid/sqllib/java/sqlj.zip:./test:/home2/myusrid/sqllib/java/db2jcc.
jar:/home2/myusrid/sqllib/java/db2jcc_license_cisuz.jar:...
[jcc] Java native library path = /wsdb/v91/bldsupp/AIX5L64/jdk1.4.2_sr1/sh/..jr
e/bin:/wsdb/v91/bldsupp/AIX5L64/jdk1.4.2_sr1/jre/bin/classic:/wsdb/v91/bldsupp/A
IX5L64/jdk1.4.2_sr1/jre/bin:/home2/myusrid/sqllib/lib:/local/cobol:/home2/myusri
d/sqllib/samples/c:/usr/lib
[jcc] Path of extension directory or directories = /wsdb/v91/bldsupp/AIX5L64/jdk
1.4.2_sr1/sh/..jre/lib/ext
[jcc] Operating system name = AIX
[jcc] Operating system architecture = ppc64
[jcc] Operating system version = 5.3
[jcc] File separator ("/" on UNIX) = /
[jcc] Path separator (":" on UNIX) = :
[jcc] User's account name = myusrid
[jcc] User's home directory = /home2/myusrid
[jcc] User's current working directory = /home2/myusrid
[jcc] Dumping all system properties: { java.assistive=ON, java.runtime.name=Java
(TM) 2 Runtime Environment, Standard Edition, sun.boot.library.path=/wsdb/v91/bl
dsupp/AIX5L64/jdk1.4.2_sr1/sh/..jre/bin, java.vm.version=1.4.2, java.vm.vendor=
IBM Corporation, java.vendor.url=http://www.ibm.com/, path.separator=:, java.vm.
name=Classic VM, file.encoding.pkg=sun.io, user.country=US, sun.os.patch.level=u
nknown, ... }
[jcc] Dumping all file properties: { }
[jcc] END TRACE_DRIVER_CONFIGURATION

```

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.

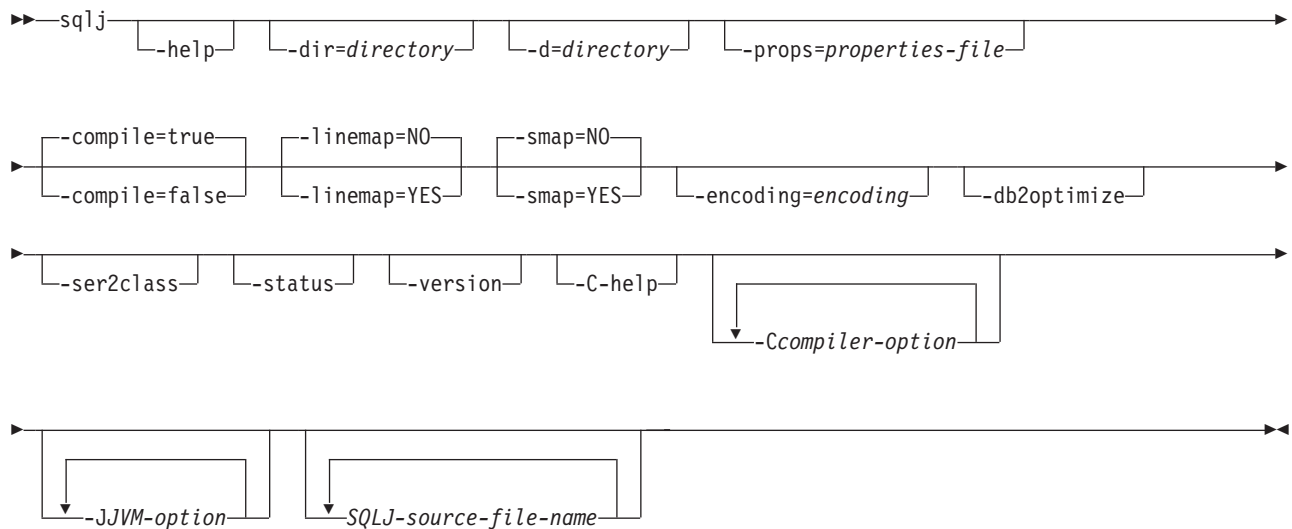
sqlj - SQLJ translator

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



Command parameters

-help

Specifies that the SQLJ translator describes each of the options that the translator supports. If any other options are specified with -help, they are ignored.

-dir=directory

Specifies the name of the directory into which SQLJ puts .java files that are generated by the translator and .class files that are generated by the compiler. The default is the directory that contains the SQLJ source files.

The translator uses the directory structure of the SQLJ source files when it puts the generated files in directories. For example, suppose that you want the translator to process two files:

- file1.sqlj, which is not in a Java package
- file2.sqlj, which is in Java package sqlj.test

Also suppose that you specify the parameter -dir=/src when you invoke the translator. The translator puts the Java source file for file1.sqlj in directory /src and puts the Java source file for file2.sqlj in directory /src/sqlj/test.

-d=directory

Specifies the name of the directory into which SQLJ puts the binary files that are generated by the translator and compiler. These files include the .ser files, the name_SJProfileKeys.class files, and the .class files that are generated by the compiler.

The default is the directory that contains the SQLJ source files.

The translator uses the directory structure of the SQLJ source files when it puts the generated files in directories. For example, suppose that you want the translator to process two files:

- file1.sqlj, which is not in a Java package
- file2.sqlj, which is in Java package sqlj.test

Also suppose that you specify the parameter -d=/src when you invoke the translator. The translator puts the serialized profiles for file1.sqlj in directory /src and puts the serialized profiles for file2.sqlj in directory /src/sqlj/test.

-compile=true | false

Specifies whether the SQLJ translator compiles the generated Java source into bytecodes.

true

The translator compiles the generated Java source code. This is the default.

false

The translator does not compile the generated Java source code.

-linemap=no | yes

Specifies whether line numbers in Java exceptions match line numbers in the SQLJ source file (the .sqlj file), or line numbers in the Java source file that is generated by the SQLJ translator (the .java file).

no Line numbers in Java exceptions match line numbers in the Java source file. This is the default.

yes

Line numbers in Java exceptions match line numbers in the SQLJ source file.

-smap=no | yes

Specifies whether the SQLJ translator generates a source map (SMAP) file for each SQLJ source file. An SMAP file is used by some Java language debug tools. This file maps lines in the SQLJ source file to lines in the Java source file that is generated by the SQLJ translator. The file is in the Unicode UTF-8 encoding scheme. Its format is described by Original Java Specification Request (JSR) 45, which is available from this web site:

<http://www.jcp.org>

no Do not generated SMAP files. This is the default.

yes

Generate SMAP files. An SMAP file name is *SQLJ-source-file-name.java.smap*. The SQLJ translator places the SMAP file in the same directory as the generated Java source file.

-encoding=encoding-name

Specifies the encoding of the source file. Examples are JIS or EUC. If this option is not specified, the default converter for the operating system is used.

-db2optimize

Specifies that the SQLJ translator generates code for a connection context class that is optimized for DB2. **-db2optimize** optimizes the code for the user-defined context but not the default context.

When you run the SQLJ translator with the **-db2optimize** option, if your applications use JDBC 3.0 or earlier functions, the IBM Data Server Driver for JDBC and SQLJ file *db2jcc.jar* must be in the CLASSPATH for compiling the generated Java application. If your applications use JDBC 4.0 or earlier functions, the IBM Data Server Driver for JDBC and SQLJ file *db2jcc4.jar* must be in the CLASSPATH for compiling the generated Java application.

-ser2class

Specifies that the SQLJ translator converts .ser files to .class files.

-status

Specifies that the SQLJ translator displays status messages as it runs.

-version

Specifies that the SQLJ translator displays the version of the IBM Data Server Driver for JDBC and SQLJ. The information is in this form:

IBM SQLJ xxxx.xxxx.xx

-C-help

Specifies that the SQLJ translator displays help information for the Java compiler.

-Ccompiler-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.

-JVM-option

Specifies an option that is to be passed to the Java virtual machine (JVM) in which the sqlj command runs. The option must be a valid JVM option that begins with a dash (-). Do not include spaces between -J and the JVM option. If you need to specify multiple JVM options, precede each compiler option with -J. For example:

-J-Xmx128m -J-Xmine2M

SQLJ-source-file-name

Specifies a list of SQLJ source files to be translated. This is a required parameter. All SQLJ source file names must have the extension .sqlj.

Output

For each source file, *program-name.sqlj*, the SQLJ translator produces the following files:

- The generated source program
The generated source file is named *program-name.java*.
- A serialized profile file for each connection context class that is used in an SQLJ executable clause
A serialized profile name is of the following form:
program-name_SJProfileIDNumber.ser
- If the SQLJ translator invokes the Java compiler, the class files that the compiler generates.

Examples

```
sqlj -encoding=UTF8 -C-0 MyApp.sqlj
```

Related tasks

“Program preparation for SQLJ programs” on page 181

Related reference

“db2sqljcustomize - SQLJ profile customizer”

“db2sqljprint - SQLJ profile printer” on page 437

“db2sqljbind - SQLJ profile binder” on page 431

db2sqljcustomize - SQLJ profile customizer

db2sqljcustomize processes an SQLJ profile, which contains embedded SQL statements.

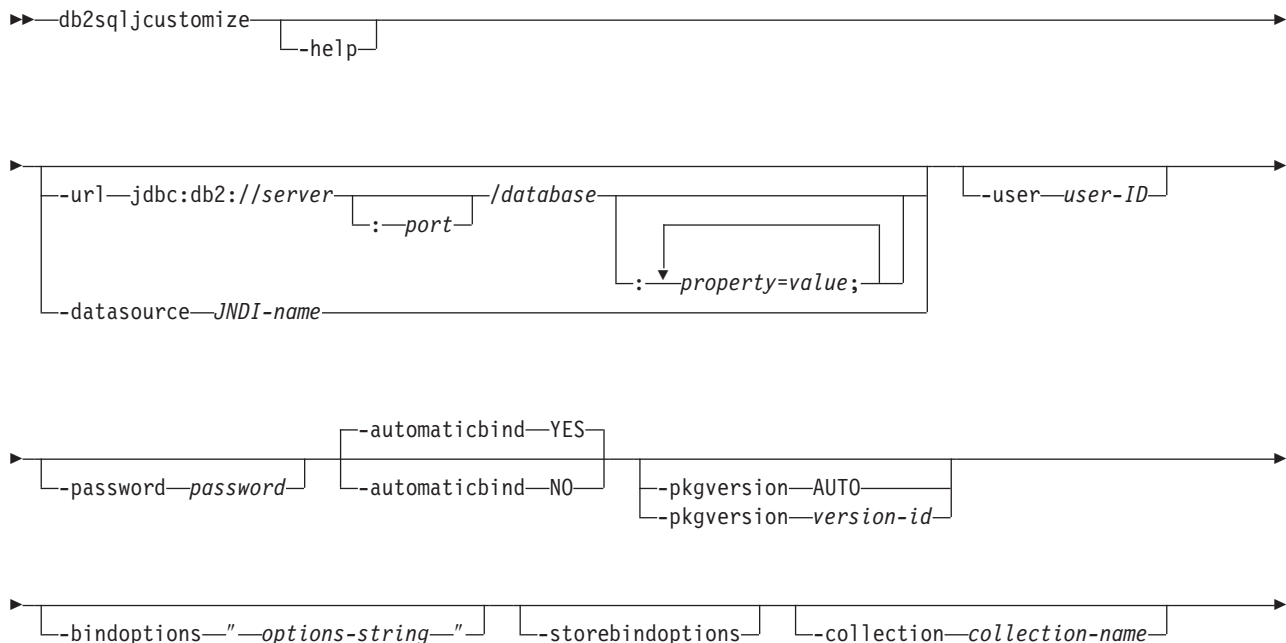
By default, db2sqljcustomize produces four DB2 packages: one for each isolation level. db2sqljcustomize augments the profile with DB2-specific information for use at run time.

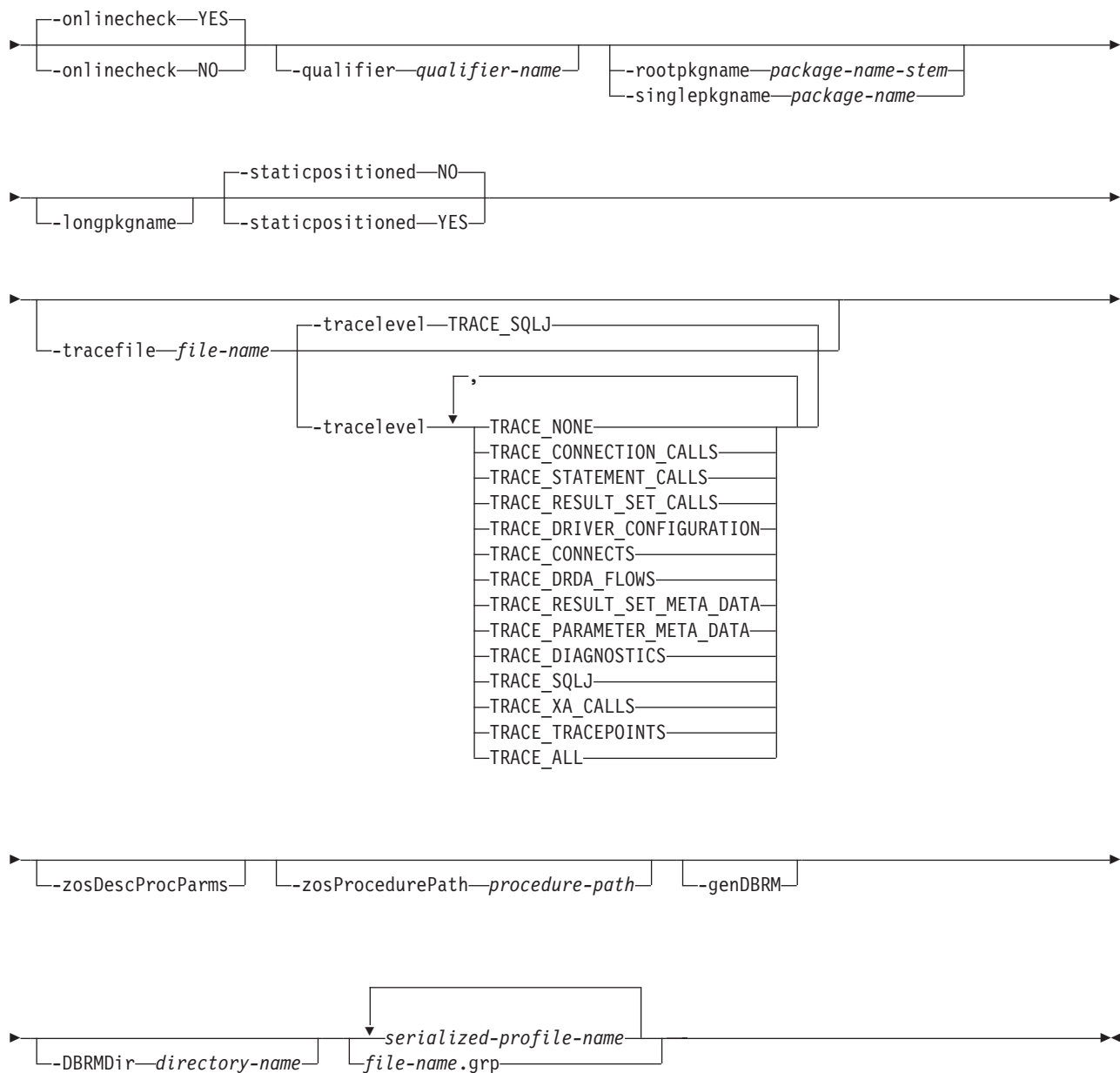
Authorization

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

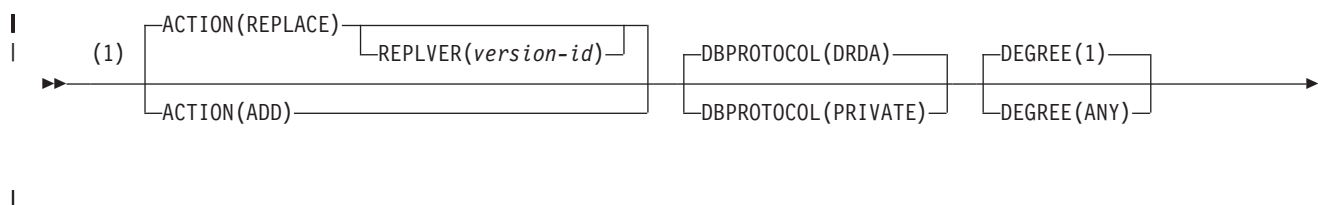


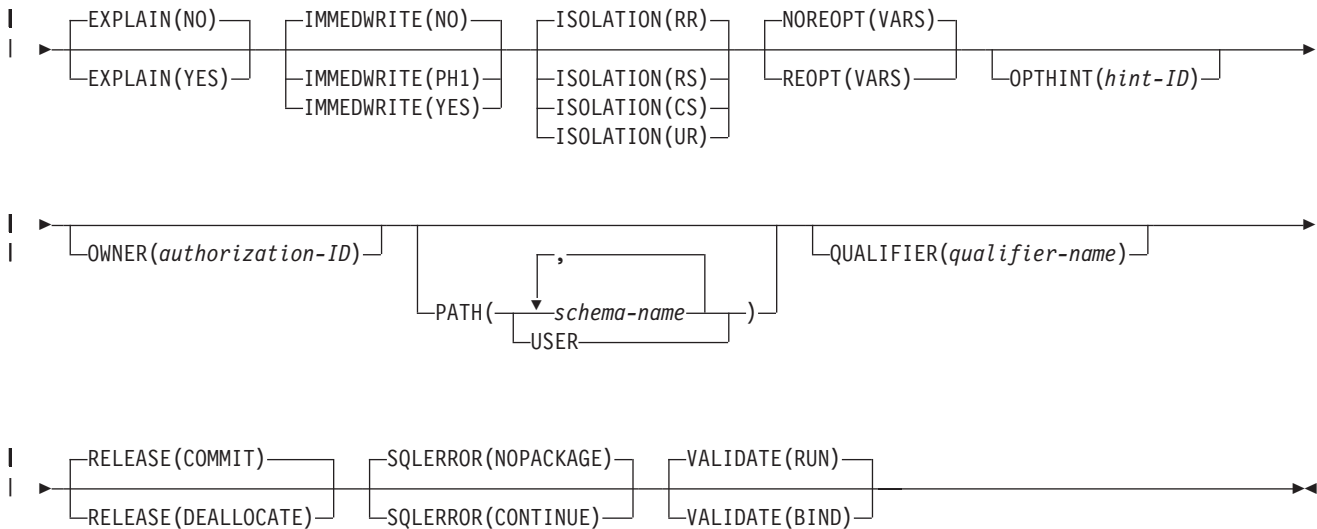


options-string:



DB2 for z/OS options:

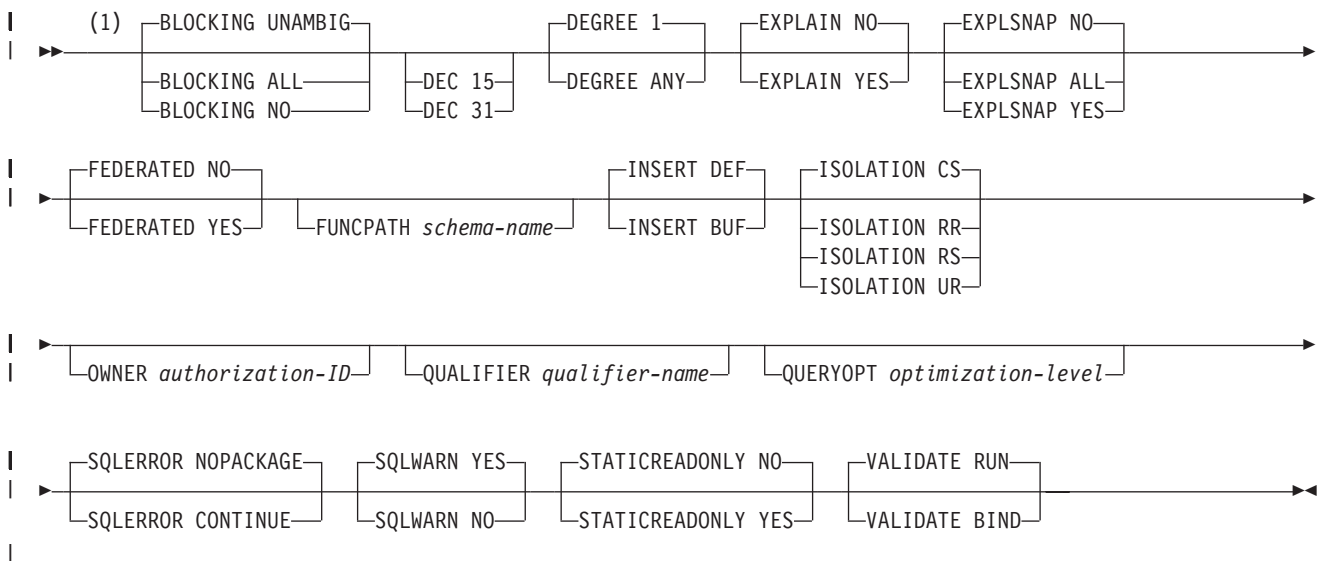




Notes:

- 1 These options can be specified in any order.

DB2 Database for Linux, UNIX, and Windows options



Notes:

- 1 These options can be specified in any order.

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 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 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 SYSIBM.SYSDUMMY1;
```

If the connection is to a DB2 Database for Linux, UNIX, and Windows server, *database* is the database name that is defined during installation.

If the connection is to an IBM 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 DB2 for z/OS options. If you are preparing your program to run on a DB2 Database for Linux, UNIX, and Windows system, specify DB2 Database for Linux, UNIX, and Windows 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 93 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 93. Parts of a package name that is generated by `db2sqljcustomize`

Package name part	Number of bytes	Value
<i>Bytes-from-program-name</i>	$m = \min(\text{Length}(\text{program-name}), \text{maxlen} - 1 - \text{Length}(\text{IDNumber}))$	First <i>m</i> bytes of <i>program-name</i> , in uppercase
<i>IDNumber</i>	$\text{Length}(\text{IDNumber})$	<i>IDNumber</i>
<i>PkgIsolation</i>	1	1, 2, 3, or 4. This value represents the transaction isolation level for the package. See Table 94.

Table 94 shows the values of the *PkgIsolation* portion of a package name that is generated by `db2sqljcustomize`.

Table 94. *PkgIsolation* values and associated isolation levels

<i>PkgNumber</i> value	Isolation level for package
1	Uncommitted read (UR)
2	Cursor stability (CS)
3	Read stability (RS)
4	Repeatable read (RR)

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 data 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 database 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 DB2 database server finds a match, it obtains the information about the parameters for that stored procedure from the DB2 catalog. If the DB2 database server does not find a match, SQLJ sets the parameter data without any DB2 catalog information.

If -zosProcedurePath is not specified, the DB2 database 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 plans and 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 -rootpkgname SQLJSA0 is specified, and -genDBRM is also specified, the names of the four DBRM files are:

- SQLJSA01
- SQLJSA02
- SQLJSA03
- SQLJSA04

-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 -rootpkgname or -singlepkgname 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.

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
```

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 or plans 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 or plans, 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 and plans.

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.

DBPROTOCOL(DRDA)

Private protocol is deprecated, so you should use `DBPROTOCOL(DRDA)` for all applications. However, for SQLJ applications that use remote three-part table names, you must use `DBPROTOCOL(DRDA)`. Otherwise, those applications might fail.

Related concepts

“Security for preparing SQLJ applications with the IBM Data Server Driver for JDBC and SQLJ” on page 490

Related tasks

“Program preparation for SQLJ programs” on page 181

Related reference

“sqlj - SQLJ translator” on page 416

“db2sqljprint - SQLJ profile printer” on page 437

“db2sqljbind - SQLJ profile binder”

 [BIND and REBIND options \(DB2 Command Reference\)](#)

db2sqljbind - SQLJ profile binder

`db2sqljbind` binds DB2 packages for a serialized profile that was previously customized with the `db2sqljcustomize` command.

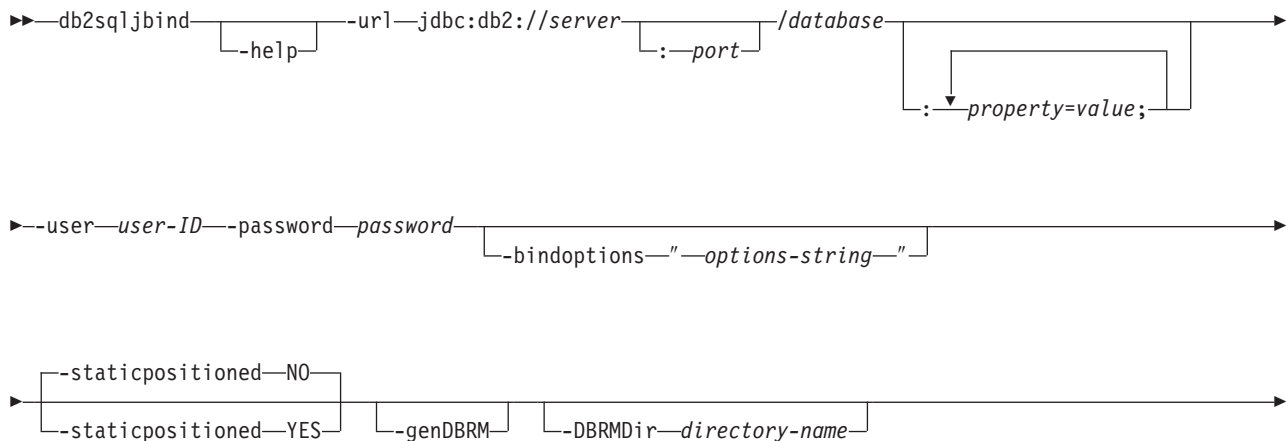
Applications that run with the IBM Data Server Driver for JDBC and SQLJ require packages but no plans. If the `db2sqljcustomize -automaticbind` option is specified as YES or defaults to YES, `db2sqljcustomize` binds packages for you at the data source that you specify in the `-url` parameter. However, 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.

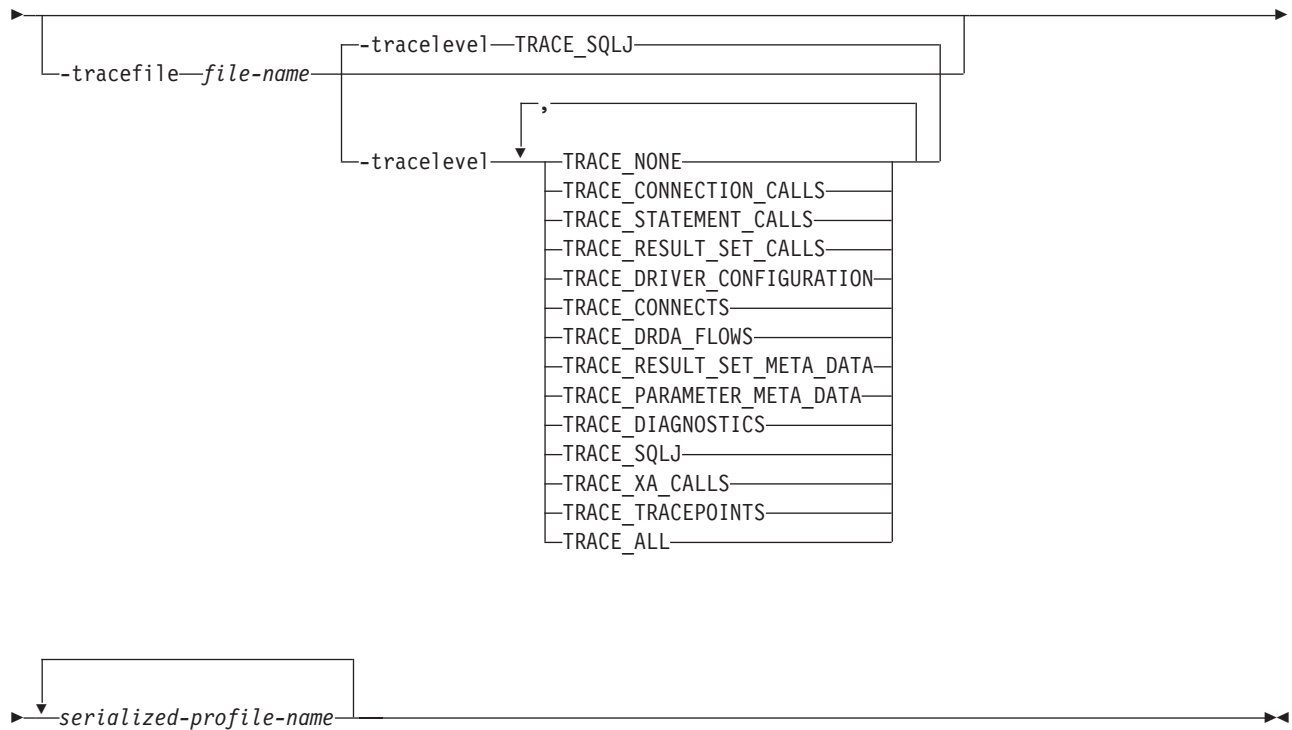
Authorization

The privilege set of the process must include one of the following authorities:

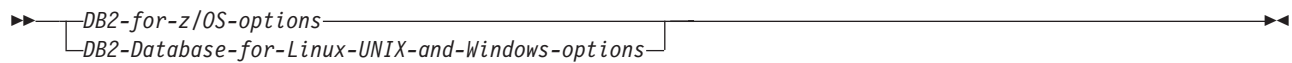
- SYSADM authority
- DBADM authority
- If the package does not exist, the BINDADD privilege, and one of the following privileges:
 - CREATEIN privilege
 - PACKADM authority on the collection or on all collections
- If the package exists, the BIND privilege on the package

Command syntax

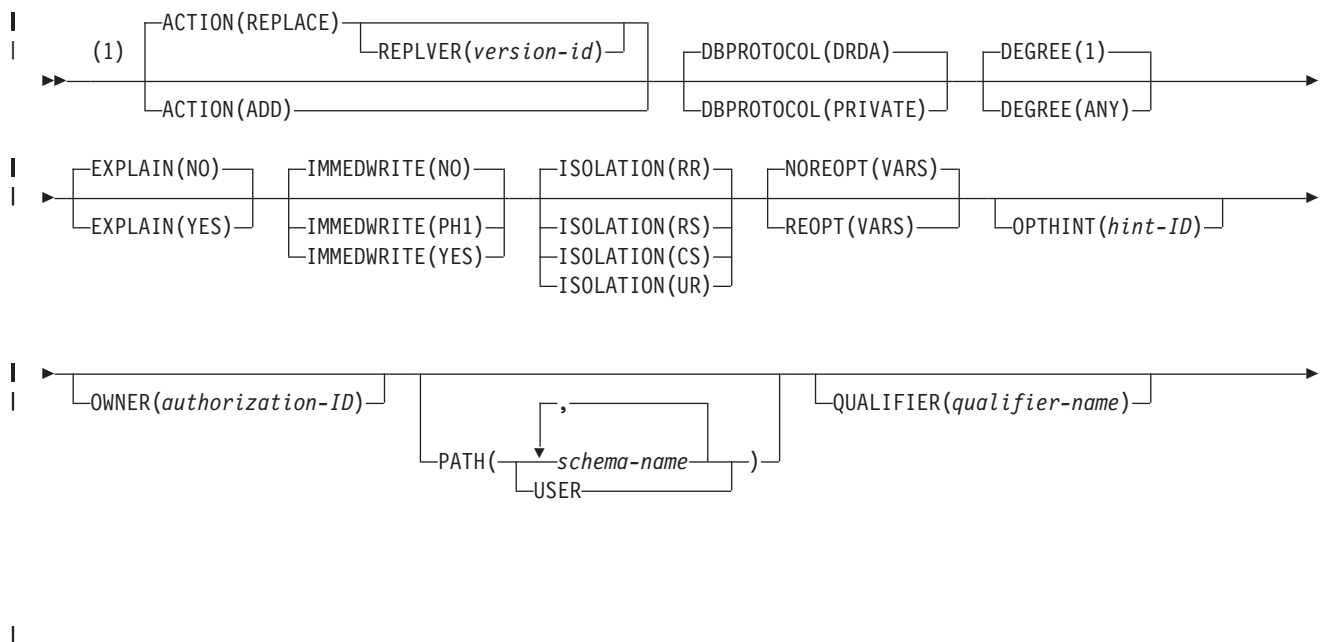


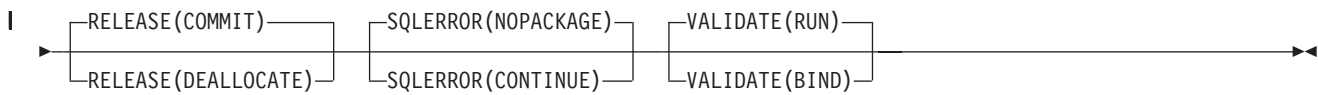


options-string:



DB2 for z/OS options:

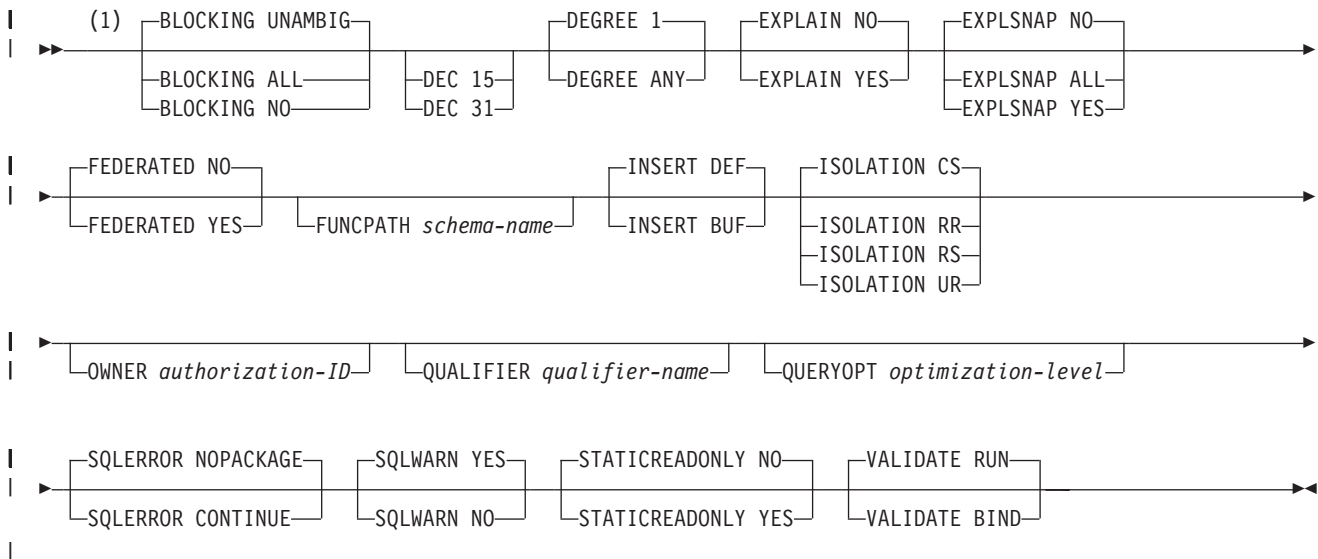




Notes:

- 1 These options can be specified in any order.

DB2 Database for Linux, UNIX, and Windows options



Notes:

- 1 These options can be specified in any order.

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 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 SYSIBM.SYSDUMMY1;
```

If the connection is to a DB2 Database for Linux, UNIX, and Windows server, *database* is the database name that is defined during installation.

If the connection is to an IBM 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.

-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 DB2 for z/OS options. If you are preparing your program to run on a DB2 Database for Linux, UNIX, and Windows system, specify DB2 Database for Linux, UNIX, and Windows options.

Notes on bind options:

- Specify VERSION only if the following conditions are true:
 - If you are binding a package at a DB2 Database for Linux, UNIX, and Windows 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

Specifies the name 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.

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.

Examples

```
db2sqljbind -user richler -password mordecai
            -url 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.

Related tasks

“Program preparation for SQLJ programs” on page 181

“Binding SQLJ applications to access multiple database servers” on page 183

Related reference

“sqlj - SQLJ translator” on page 416

“db2sqljcustomize - SQLJ profile customizer” on page 420

“db2sqljprint - SQLJ profile printer”

 BIND and REBIND options (DB2 Command Reference)

db2sqljprint - SQLJ profile printer

db2sqljprint prints the contents of the customized version of a profile as plain text.

Authorization

None

Command syntax

►—db2sqljprint—*profilename*—►

Command parameters

profilename

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
```

Related concepts

Chapter 15, “Problem diagnosis with the IBM Data Server Driver for JDBC and SQLJ,” on page 537

Related reference

“sqlj - SQLJ translator” on page 416

“db2sqljcustomize - SQLJ profile customizer” on page 420

“db2sqljbind - SQLJ profile binder” on page 431

Chapter 8. Installing the IBM Data Server Driver for JDBC and SQLJ

If you plan to run JDBC or SQLJ applications, 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

To use the IBM Data Server Driver for JDBC and SQLJ as a type 2 driver or a type 4 driver, you need to install the driver on your DB2 subsystem.

Prerequisites for the IBM Data Server Driver for JDBC and SQLJ:

- Java 2 Technology Edition, V1.4.2 service release 2 (SR2), or later.

The following functions require Java 2 Technology Edition, V5 or later.

- Accessing DB2 tables that include DECFLOAT columns
- Using Java support for XML schema registration and removal

JDBC 4.0 functions 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. 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.

If any Java 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 V2 R8/R9/R10 support for Unicode. Information APARs II13048 and II13049 contain additional information.

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 DSNTIPE. 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. *If you plan to use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to implement distributed transactions against DB2 UDB for OS/390 and z/OS Version 7 servers:* In z/OS UNIX System Services, run the DB2T4XAIndoubtUtil against each of those servers.
8. *If you plan to use LOB locators to access DBCLOB or CLOB columns in DB2 tables on DB2 for z/OS servers:* Create tables on the database servers that are needed for fetching data from DBCLOB or CLOB columns using LOB locators. Use one of the following techniques.
 - On the DB2 for z/OS servers, customize and run job DSNTIJMS. That job is located in data set *prefix.SDSNSAMP*.
 - On the client, in z/OS UNIX System Services, run the `com.ibm.db2.jcc.DB2LobTableCreator` utility against each of the DB2 for z/OS servers.
9. Verify the installation by running a simple JDBC application.

Related tasks



Connecting distributed database systems (DB2 Installation and Migration)



Connecting systems with TCP/IP (DB2 Installation and Migration)

Related reference



DESCRIBE FOR STATIC field (DESCSTAT subsystem parameter) (DB2 Installation and Migration)

Jobs for loading the IBM Data Server Driver for JDBC and SQLJ libraries

When you install DB2 for z/OS, include the steps for allocating the HFS or zFS directory structure and using 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.

DSNDDEF2

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.

DSNAPPL2

Includes the steps that perform the SMP/E APPLY CHECK and APPLY functions for the IBM Data Server Driver for JDBC and SQLJ libraries.

DSNACEP2

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

You need to set environment variables so that 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:

```
export STEPLIB=DSN910.SDSNEXIT:DSN910.SDSNLOAD:DSN910.SDSNLOD2:$STEPLIB
```

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/db2910_jdbc, modify PATH as follows:

```
export PATH=/usr/lpp/db2910_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/db2910_jdbc, modify LIBPATH as follows:

```
export LIBPATH=/usr/lpp/db2910_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/db2910_jdbc, modify CLASSPATH as follows:

For JDBC 3.0 and earlier support:

```
export CLASSPATH=/usr/lpp/db2910_jdbc/classes/db2jcc.jar: \
/usr/lpp/db2910_jdbc/classes/db2jcc_javax.jar: \
/usr/lpp/db2910_jdbc/classes/sqlj.zip: \
/usr/lpp/db2910_jdbc/classes/db2jcc_license_cisuz.jar: \
$CLASSPATH
```

For JDBC 4.0 and later, and JDBC 3.0 and earlier support:

```
export CLASSPATH=/usr/lpp/db2910_jdbc/classes/db2jcc4.jar: \
/usr/lpp/db2910_jdbc/classes/db2jcc_javax.jar: \
/usr/lpp/db2910_jdbc/classes/sqlj4.zip: \
/usr/lpp/db2910_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

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

“WLM application environment values for Java routines” on page 155

“Run-time environment for Java routines” on page 156

Related tasks

“Running JDBC and SQLJ programs” on page 190

“Installing the z/OS Application Connectivity to DB2 for z/OS feature” on page 461

Chapter 9, “Migrating from the JDBC/SQLJ Driver for OS/390 and z/OS to the IBM Data Server Driver for JDBC and SQLJ,” on page 465

“Upgrading the IBM Data Server Driver for JDBC and SQLJ to a new version” on page 460

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

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

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

property=value

You can set configuration properties in the following ways:

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

For stand-alone Java applications, you can set the configuration properties as Java system properties by specifying `-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.

Recommendation: Because support for `com/ibm/db2/jcc/DB2JccConfiguration.properties` as the default resource name for configuration properties is deprecated, use `DB2JccConfiguration.properties` instead.

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:

```
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:

```
jar -cvf jdbcProperties.jar DB2JccConfiguration.properties
```

Related tasks

“Installing the z/OS Application Connectivity to DB2 for z/OS feature” on page 461

Chapter 9, “Migrating from the JDBC/SQLJ Driver for OS/390 and z/OS to the IBM Data Server Driver for JDBC and SQLJ,” on page 465

Chapter 16, “Tracing IBM Data Server Driver for JDBC and SQLJ C/C++ native driver code,” on page 547

“Upgrading the IBM Data Server Driver for JDBC and SQLJ to a new version” on page 460

Related reference

“db2jcctrace - Format IBM Data Server Driver for JDBC and SQLJ trace data for C/C++ native driver code” on page 547

“IBM Data Server Driver for JDBC and SQLJ configuration properties” on page 243

Enabling the DB2-supplied stored procedures and defining the tables used by the IBM Data Server Driver for JDBC and SQLJ

Before you can use certain functions of the IBM Data Server Driver for JDBC and SQLJ on a DB2 for z/OS subsystem, you need to install a set of stored procedures and create a set of tables.

WLM must be installed on the z/OS system.

The stored procedures that you need to install are:

- SQLCOLPRIVILEGES
- SQLCOLUMNS
- SQLFOREIGNKEYS
- SQLFUNCTIONCOLUMNS
- SQLFUNCTIONS
- SQLGETTYPEINFO
- SQLPRIMARYKEYS
- SQLPROCEDURECOLS
- SQLPROCEDURES
- SQLSPECIALCOLUMNS
- SQLSTATISTICS
- SQLTABLEPRIVILEGES
- SQLTABLES
- SQLUDTS

- SQLCAMESSAGE

The tables that you need to create are:

- SYSIBM.SYSDUMMYU
- SYSIBM.SYSDUMMYA
- SYSIBM.SYSDUMMYE

Those tables ensure that character conversion does not occur when Unicode data is stored in DBCLOB or CLOB columns.

Follow these steps to install the stored procedures and create the tables:

1. Set up a WLM environment for running the stored procedures.
To set up a WLM application environment for these stored procedures, you need to define a JCL startup procedure for the WLM environment, and define the application environment to WLM.
2. Define the stored procedures to DB2, bind the stored procedure packages, and define the SYSIBM.SYSDUMMYU, SYSIBM.SYSDUMMYA, and SYSIBM.SYSDUMMYE tables.

To perform those tasks, use job DSNTIJSJ during installation or migration, or job DSNTIJTM after installation or migration.

Related tasks

“Installing the z/OS Application Connectivity to DB2 for z/OS feature” on page 461

Chapter 9, “Migrating from the JDBC/SQLJ Driver for OS/390 and z/OS to the IBM Data Server Driver for JDBC and SQLJ,” on page 465

“Upgrading the IBM Data Server Driver for JDBC and SQLJ to a new version” on page 460

Creating the WLM address space startup procedure for the IBM Data Server Driver for JDBC and SQLJ stored procedures

You can use the DSN8WLMP sample startup procedure as a model for your stored procedure address space startup procedure.

Make the following changes to that stored procedure:

1. Change the APPLENV value to match the definition name that you specify in the WLM Definition Menu.
2. Change the startup procedure name to match the procedure name that you specify in the WLM Create an Application Environment menu.
3. Change the DB2SSN value to the subsystem name of your DB2 for z/OS subsystem.
4. Edit the data set names to match your data set names.

Related concepts

“Values for the WLM environment for IBM Data Server Driver for JDBC and SQLJ stored procedures”

Values for the WLM environment for IBM Data Server Driver for JDBC and SQLJ stored procedures

You need to define the application environment that you use for DB2-supplied stored procedures for the IBM Data Server Driver for JDBC and SQLJ to WLM.

The following example shows a WLM Definition Menu with values for a Java stored procedure application environment.


```

File Utilities Notes Options Help
-----
Definition Menu      WLM Appl
Command ===> _____

Definition data set . . : none
Definition name . . . . WLMENV
Description . . . . . Environment for Java stored procedures
Select one of the
following options. . . 9 1. Policies
                        2. Workloads
                        3. Resource Groups
                        4. Service Classes
                        5. Classification Groups
                        6. Classification Rules
                        7. Report Classes
                        8. Service Coefficients/Options
                        9. Application Environments
                        10. Scheduling Environments

```

Definition name

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 a Java stored procedure application environment.

```

Application-Environment Notes Options Help
-----
Create an Application Environment
Command ===> _____

Application Environment Name . . : WLMENV
Description . . . . . Environment for Java stored procedures
Subsystem Type . . . . . DB2
Procedure Name . . . . . DSN8WLMF
Start Parameters . . . . . DB2SSN=DB2T,NUMTCB=3,APPLENV=WLMENV

Limit on starting server address spaces for a subsystem instance:
1 1. No limit.
   2. Single address space per system.
   3. Single address spaces per sysplex.

```

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 that you are

running. Specify a value between 5 and 8. 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).

Related tasks

“Creating the WLM address space startup procedure for the IBM Data Server Driver for JDBC and SQLJ stored procedures” on page 445

Jobs for creating IBM DB2 Driver for JDBC and SQLJ stored procedures and tables

DB2 provides JCL jobs that include statements that you can use to define the DB2-supplied stored procedures for JDBC, bind the stored procedure packages, and define the SYSIBM.SYSDUMMYU, SYSIBM.SYSDUMMYA, and SYSIBM.SYSDUMMYE tables.

DSNTIJSG

Use this job if you are defining the stored procedures and tables as part of installing or migrating a DB2 subsystem.

Before you run this job, you need to modify the WLM ENVIRONMENT parameter value for each stored procedure to match the Application Environment Name value that you specified in the WLM panels and the APPLENV name that you specified in the WLM address space startup procedure. Other customizations are made as part of the installation process.

DSNTIJMS

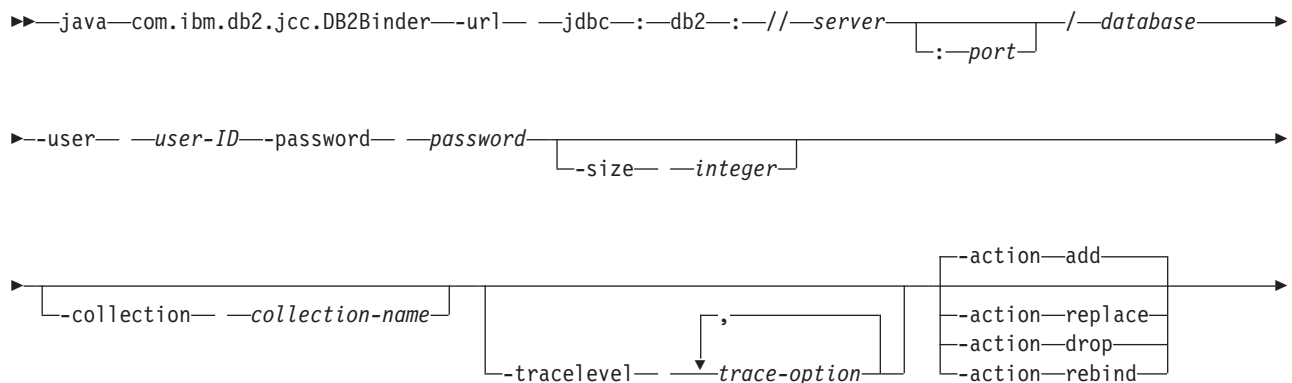
Use this job if you are defining the stored procedures and tables after you install or migrate a DB2 subsystem.

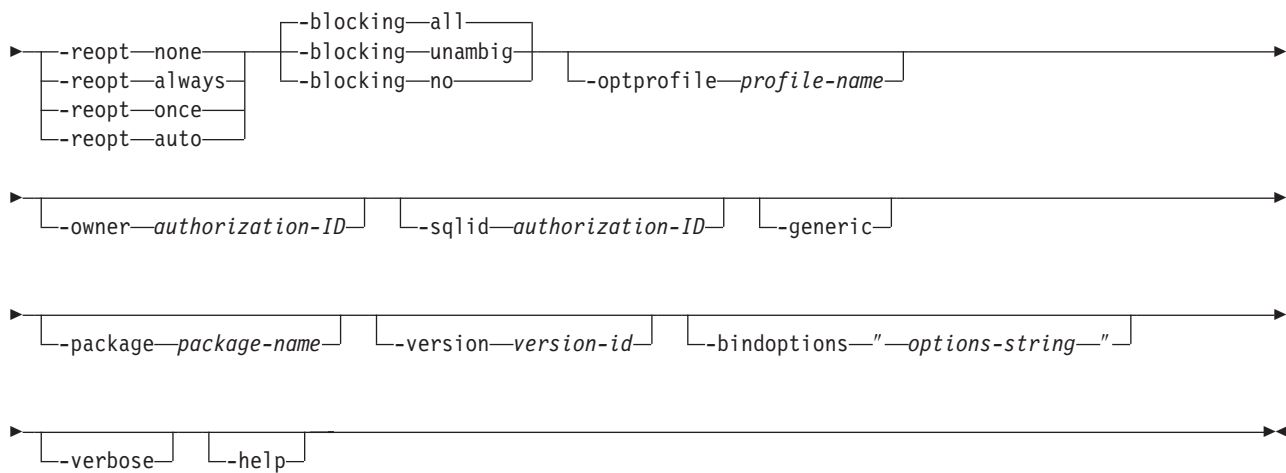
Before you run this job, you need to make the modifications that are described in the job prolog.

DB2Binder utility

The DB2Binder utility binds the DB2 packages that are used at the database 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 database server resides.

port

The TCP/IP server port number that is assigned to the database server. The default is 446.

database

The location name for the database server, as defined in the SYSIBM.LOCATIONS 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` is 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 database 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:

```
number-of-isolation-levels*
number-of-holdability-values*
integer+
number-of-packages-for-static-SQL
= 4*2*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 database 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.

-reopt

Specifies whether DB2 for z/OS database servers determine access paths at run time. This option is valid only for connections to DB2 for z/OS database servers. This option is not sent to the database server if it is not specified. In that case, the database server determines the reoptimization behavior.

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 database server.

-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 Database for Linux, UNIX, and Windows database servers.

-owner

Specifies the authorization ID of the owner of the packages. The default value is set by the database 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 database 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.

-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.

Possible parameters and values are:

bindObjectExistenceRequired

Specifies whether the database 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 database 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 database server processes

dynamic SQL statements using invoke behavior. Otherwise, the database 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 database server processes dynamic SQL statements using invoke behavior. Otherwise, the database 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 database server processes dynamic SQL statements using define behavior. Otherwise, the database 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 database server processes dynamic SQL statements using define behavior. Otherwise, the database 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 `-bindoptions` is 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 95. DB2Binder return codes when the target operating system is not Windows

Return code	Meaning
0	Successful execution.
1	An error occurred during DB2Binder execution.

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 96. DB2Binder return codes when the target operating system is Windows

Return code	Meaning
0	Successful execution.
-100	No bind options were specified.
-101	<code>-url</code> value was not specified.
-102	<code>-user</code> value was not specified.
-103	<code>-password</code> value was not specified.
-200	No valid bind options were specified.
-114	The <code>-package</code> option was not specified, but the <code>-generic</code> option was specified.
-201	<code>-url</code> value is invalid.
-204	<code>-action</code> value is invalid.
-205	<code>-blocking</code> value is invalid.
-206	<code>-collection</code> value is invalid.
-207	<code>-dbprotocol</code> value is invalid.
-208	<code>-keepdynamic</code> value is invalid.
-210	<code>-reopt</code> value is invalid.
-211	<code>-size</code> value is invalid.
-212	<code>-tracelevel</code> value is invalid.
-307	<code>-dbprotocol</code> value is not supported by the target database server.
-308	<code>-keepdynamic</code> value is not supported by the target database server.

Table 96. DB2Binder return codes when the target operating system is Windows (continued)

Return code	Meaning
-310	-reopt value is not supported by the target database server.
-313	-optprofile value is not supported by the target database server.
-401	The Binder class was not found.
-402	Connection to the database server failed.
-403	DatabaseMetaData retrieval for the database server failed.
-501	No more packages are available in the cluster.
-502	An existing package is not valid.
-503	The bind process returned an error.
-999	An error occurred during processing of an undocumented bind option.

Related tasks

“Installing the z/OS Application Connectivity to DB2 for z/OS feature” on page 461

Chapter 9, “Migrating from the JDBC/SQLJ Driver for OS/390 and z/OS to the IBM Data Server Driver for JDBC and SQLJ,” on page 465

“Upgrading the IBM Data Server Driver for JDBC and SQLJ to a new version” on page 460

DB2T4XAIndoubtUtil for distributed transactions with DB2 UDB for OS/390 and z/OS Version 7 servers

If you plan to implement distributed transactions using IBM Data Server Driver for JDBC and SQLJ type 4 connectivity that include DB2 UDB for OS/390 and z/OS Version 7 servers, you need to run the DB2T4XAIndoubtUtil utility against those servers.

DB2T4XAIndoubtUtil allows Version 7 servers, which do not have built-in support for distributed transactions that implement the XA specification, to emulate that support.

DB2T4XAIndoubtUtil performs one or both of the following tasks:

- Creates a table named SYSIBM.INDOUBT and an associated index
- Binds DB2 packages named T4XAIN01, T4XAIN02, T4XAIN03, and T4XAIN04

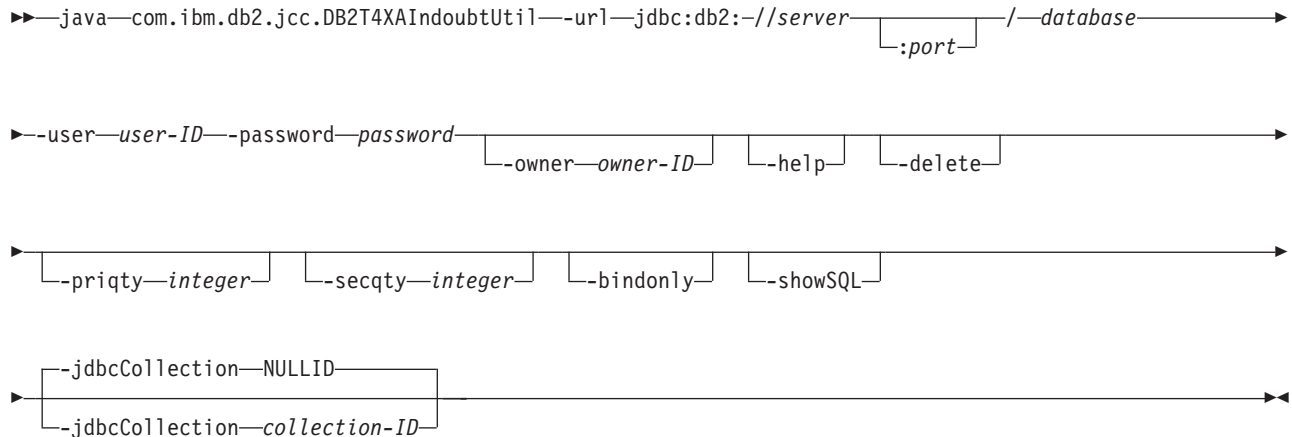
You should create and drop packages T4XAIN01, T4XAIN02, T4XAIN03, and T4XAIN04 only by running DB2T4XAIndoubtUtil. You can create and drop SYSTEM.INDOUBT and its index manually, but it is recommended that you use the utility. See DB2T4XAIndoubtUtil usage notes for instructions on how to create those objects manually.

DB2T4XAIndoubtUtil authorization

To run the DB2T4XAIndoubtUtil utility to create SYSTEM.INDOUBT and bind packages T4XAIN01, T4XAIN02, T4XAIN03, and T4XAIN04, you need SYSADM authority.

To run the DB2T4XAIndoubtUtil only to bind packages T4XAIN01, T4XAIN02, T4XAIN03, and T4XAIN04, you need BIND authority on the packages.

DB2T4XAIndoubtUtil syntax



DB2T4XAIndoubtUtil parameter descriptions

-url

Specifies the data source at which `DB2T4XAIndoubtUtil` 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.

database is the DB2 location name that is defined during installation. All characters in this value must be uppercase characters. You can determine the location name by executing the following SQL statement on the server:

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

-user

Specifies the user ID under which `DB2T4XAIndoubtUtil` is to run. This user must have `SYSADM` authority or must be a member of a RACF group that corresponds to a secondary authorization ID with `SYSADM` authority.

-password

Specifies the password for the user ID.

-owner

Specifies a secondary authorization ID that has `SYSADM` authority. Use the `-owner` parameter if the `-user` parameter value does not have `SYSADM` authority. The `-user` parameter value must be a member of a RACF group whose name is *owner-ID*.

When the `-owner` parameter is specified, `DB2T4XAIndoubtUtil` uses *owner-ID* as:

- The authorization ID for creating the `SYSIBM.INDOUBT` table.

- The authorization ID of the owner of the T4XAIN01, T4XAIN02, T4XAIN03, and T4XAIN04 packages. SQL statements in those packages are executed using the authority of *owner-ID*.

-help

Specifies that the DB2T4XAIndoubtUtil utility describes each of the options that it supports. If any other options are specified with -help, they are ignored.

-delete

Specifies that the DB2T4XAIndoubtUtil utility deletes the objects that were created when DB2T4XAIndoubtUtil was run previously.

-priqty

Specifies the primary space allocation, in kilobytes, for the table space that contains the SYSIBM.INDOUBT table. The default value for -priqty is 1000.

Important: The -priqty value divided by the page size for the table space in which SYSIBM.INDOUBT resides must be greater than the maximum number of indoubt transactions that are allowed at a given time. For example, for a 4 KB page size, the default -priqty value of 1000 allows about 250 concurrent indoubt transactions.

-secqty

Specifies the secondary space allocation, in kilobytes, for the table space that contains the SYSIBM.INDOUBT table. The default value for -secqty is 0.

Recommendation: Always use the default value of 0 for the -secqty value, and specify a -priqty value that is large enough to accommodate the maximum number of concurrent indoubt transactions.

-bindonly

Specifies that the DB2T4XAIndoubtUtil utility binds the T4XAIN01, T4XAIN02, T4XAIN03, and T4XAIN04 packages and grants permission to PUBLIC to execute the packages, but does not create the SYSIBM.INDOUBT table.

-showSQL

Specifies that the DB2T4XAIndoubtUtil utility displays the SQL statements that it executes.

-jdbcCollection *collection-name* | NULLID

Specifies the value of the -collection parameter that was used when the IBM Data Server Driver for JDBC and SQLJ packages were bound with the DB2Binder utility. The -jdbcCollection parameter *must* be specified if the explicitly or implicitly specified value of the -collection parameter was *not* NULLID.

The default is -jdbcCollection NULLID.

DB2T4XAIndoubtUtil usage notes

To create the SYSTEM.INDOUBT table and its index manually, use these SQL statements:

```
CREATE TABLESPACE INDBTTS
  USING STOGROUP
  LOCKSIZE ROW
  BUFFERPOOL BP0
  SEGSIZE 32
  CCSID EBCDIC;
```

```
CREATE TABLE SYSIBM.INDOUBT(indbtXid VARCHAR(140) FOR BIT DATA NOT NULL,
                             uowId VARCHAR(25) FOR BIT DATA NOT NULL,
                             pSyncLog VARCHAR(150) FOR BIT DATA,
```

```

                                cSyncLog VARCHAR(150) FOR BIT DATA)
IN INDBTTS;

CREATE UNIQUE INDEX INDBTIDX ON SYSIBM.INDOUBT(indbtXid, uowId);

```

DB2T4XAIndoubtUtil example

Run the DB2T4XAIndoubtUtil to allow a DB2 for OS/390 and z/OS Version 7 subsystem that has IP address mvs1, port number 446, and DB2 location name SJCEC1 to participate in XA distributed transactions.

```

java com.ibm.db2.jcc.DB2T4XAIndoubtUtil -url jdbc:db2://mvs1:446/SJCEC1 \
    -user SYSADM -password mypass

```

Related concepts

Chapter 13, “IBM Data Server Driver for JDBC and SQLJ type 4 connectivity JDBC and SQLJ distributed transaction support,” on page 529

Related tasks

“Installing the z/OS Application Connectivity to DB2 for z/OS feature” on page 461

Chapter 9, “Migrating from the JDBC/SQLJ Driver for OS/390 and z/OS to the IBM Data Server Driver for JDBC and SQLJ,” on page 465

“Upgrading the IBM Data Server Driver for JDBC and SQLJ to a new version” on page 460

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

```

▶▶—java—com.ibm.db2.jcc.DB2LobTableCreator—-url—jdbc:db2:—//server—[:port]—/—database—▶▶
▶—user—user-ID—-password—password—[-help]—▶▶

```

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.

database is the DB2 location name that is defined during installation. All characters in this value must be uppercase characters. You can determine the location name by executing the following SQL statement on the server:

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

-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.

Related tasks

"Installing the z/OS Application Connectivity to DB2 for z/OS feature" on page 461

Chapter 9, "Migrating from the JDBC/SQLJ Driver for OS/390 and z/OS to the IBM Data Server Driver for JDBC and SQLJ," on page 465

"Upgrading the IBM Data Server Driver for JDBC and SQLJ to a new version" on page 460

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 a simple JDBC application.

For example, you can compile and run this program to verify your installation:

```
/**
 * 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("\nLoading IBM Data Server Driver for JDBC and SQLJ");
            Class.forName("com.ibm.db2.jcc.DB2Driver");
            System.out.println(" Successful load. Driver version: " +
```

```

        com.ibm.db2.jcc.DB2Version.getVersion());

// Connect to the local DB2 for z/OS server
System.out.println("\nEstablishing 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("\nAcquiring DatabaseMetaData");
DatabaseMetaData dbmd = conn.getMetaData();
System.out.println(" DB2 version: " +
        dbmd.getDatabaseProductVersion());

// Create a Statement object for executing a query
System.out.println("\nCreating 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 SYSIBM.SYSTABLES " +
    "ORDER BY CREATOR, NAME";
System.out.println("\nPreparing 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("\nPreparing 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("\nPreparing to close ResultSet");
results.close();
System.out.println(" Successful close of ResultSet");

System.out.println("\nPreparing to close Statement");
stmt.close();
System.out.println(" Successful close of Statement");

System.out.println("\nPreparing to rollback Connection");
conn.rollback();
System.out.println(" Successful rollback");

System.out.println("\nPreparing to close Connection");
conn.close();
System.out.println(" Successful close of Connection");
}
// Handle errors
catch(ClassNotFoundException e)
{
    System.err.println("Unable to load IBM Data Server Driver " +
        "for JDBC and SQLJ, " + e);
}
catch(SQLException e)
{

```

```

        System.out.println("SQLException: " + e);
        e.printStackTrace();
    }
}
}

```

Related tasks

“Installing the z/OS Application Connectivity to DB2 for z/OS feature” on page 461

Chapter 9, “Migrating from the JDBC/SQLJ Driver for OS/390 and z/OS to the IBM Data Server Driver for JDBC and SQLJ,” on page 465

“Upgrading the IBM Data Server Driver for JDBC and SQLJ to a new version”

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.

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.
3. *If you plan to use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to implement distributed transactions against DB2 UDB for OS/390 and z/OS Version 7 servers:* In z/OS UNIX System Services, run the DB2T4XAIndoubtUtil against each of those servers.
4. *If you plan to use LOB locators to access DBCLOB or CLOB columns in DB2 tables on DB2 for z/OS servers:* Create tables on the database servers that are needed for fetching data from DBCLOB or CLOB columns using LOB locators. Use one of the following techniques.

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

“Environment variables for the IBM Data Server Driver for JDBC and SQLJ” on page 441

“Customization of IBM Data Server Driver for JDBC and SQLJ configuration properties” on page 442

“Verify the installation of the IBM Data Server Driver for JDBC and SQLJ” on page 458

“Run-time environment for Java routines” on page 156

Related tasks

“Enabling the DB2-supplied stored procedures and defining the tables used by the IBM Data Server Driver for JDBC and SQLJ” on page 444

Related reference

“IBM Data Server Driver for JDBC and SQLJ configuration properties” on page 243

“DB2Binder utility” on page 447

“DB2T4XAIndoubtUtil for distributed transactions with DB2 UDB for OS/390 and z/OS Version 7 servers” on page 454

“DB2LobTableCreator utility” on page 457

“JDBC differences between versions of the IBM Data Server Driver for JDBC and SQLJ” on page 400

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 that 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 for Linux, UNIX, and Windows servers.

Prerequisites for the IBM Data Server Driver for JDBC and SQLJ:

- Java 2 Technology Edition, V1.4.2 service release 2 (SR2), or later

The following functions require Java 2 Technology Edition, V5 or later.

- Accessing DB2 tables that include DECFLOAT columns
- Using Java support for XML schema registration and removal

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.

If any Java 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 V2 R8/R9/R10 support for Unicode. Information APARs II13048 and II13049 contain additional information.

- SYSIBM.SYSDUMMYA, SYSIBM.SYSDUMMYE, and SYSIBM.SYSDUMMYU catalog tables

If you plan to use LOB locators to retrieve CLOB or DBCLOB data from DB2 for z/OS servers, these tables must exist on all of those database servers.

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.

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. *If you plan to use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to implement distributed transactions against DB2 UDB for OS/390 and z/OS Version 7 servers:* In z/OS UNIX System Services, run the DB2T4XAIndoubtUtil against each of those servers.
7. *If you plan to use LOB locators to access DBCLOB or CLOB columns in DB2 tables on DB2 for z/OS servers:* Create tables on the database servers that are needed for fetching data from DBCLOB or CLOB columns using LOB locators. Use one of the following techniques.
 - On the DB2 for z/OS servers, customize and run job DSNTIJMS. That job is located in data set *prefix.SDSNSAMP*.
 - On the client, in z/OS UNIX System Services, run the `com.ibm.db2.jcc.DB2LobTableCreator` utility against each of the DB2 for z/OS servers.
8. Verify the installation by running a simple JDBC application.

Related concepts

“Environment variables for the IBM Data Server Driver for JDBC and SQLJ” on page 441

“Customization of IBM Data Server Driver for JDBC and SQLJ configuration properties” on page 442

“Verify the installation of the IBM Data Server Driver for JDBC and SQLJ” on page 458

“Run-time environment for Java routines” on page 156

Related tasks

“Enabling the DB2-supplied stored procedures and defining the tables used by the IBM Data Server Driver for JDBC and SQLJ” on page 444

 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 243

“DB2Binder utility” on page 447

“DB2T4XAIndoubtUtil for distributed transactions with DB2 UDB for OS/390 and z/OS Version 7 servers” on page 454

“DB2LobTableCreator utility” on page 457

 DESCRIBE FOR STATIC field (DESCSTAT subsystem parameter) (DB2 Installation and Migration)

Jobs for loading the z/OS Application Connectivity to DB2 for z/OS libraries

To allocate the HFS or zFS directory structure and use SMP/E to load the z/OS Application Connectivity to DB2 for z/OS libraries, you need to run a set of jobs.

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.

DDAALOC

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

Contains all JDBC classes and the SQLJ runtime classes for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

sqlj.zip

Contains the classes that are needed to prepare SQLJ applications for execution under the IBM Data Server Driver for JDBC and SQLJ.

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:

```
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
```

Related concepts

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

Chapter 9. Migrating from the JDBC/SQLJ Driver for OS/390 and z/OS to the IBM Data Server Driver for JDBC and SQLJ

You need to migrate to the IBM Data Server Driver for JDBC and SQLJ before you migrate to DB2 Version 9.1 for z/OS.

Prerequisites for migrating to the IBM Data Server Driver for JDBC and SQLJ:

- Java 2 Technology Edition, V1.4.2 service release 2 (SR2), or later.

The following functions require Java 2 Technology Edition, V5 or later.

- Accessing DB2 tables that include DECFLOAT columns
- Using Java support for XML schema registration and removal

JDBC 4.0 functions 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. 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.

If any Java 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.

- Latest IBM Data Server Driver for JDBC and SQLJ maintenance

Refer to *DB2 Program Directory* to ensure that you have the latest maintenance installed for FMID JDB9912.

Important: The JDBC/SQLJ Driver for OS/390 and z/OS is no longer supported. This information is provided only to help you diagnose problems in your applications after migration to the IBM Data Server Driver for JDBC and SQLJ.

To migrate to the IBM Data Server Driver for JDBC and SQLJ, follow these steps:

1. Back up **all** of the .sqlj, .ser, .java, .and .class files, packages, and DBRMs for applications that you created under the JDBC/SQLJ Driver for OS/390 and z/OS.
2. On DB2 for z/OS, set subsystem parameter DESCSTAT to YES. DESCSTAT corresponds to installation field DESCRIBE FOR STATIC on panel DSNTIPE. 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. Convert JDBC/SQLJ Driver for OS/390 and z/OS properties to IBM Data Server Driver for JDBC and SQLJ properties.
5. **Optional:** Customize the IBM Data Server Driver for JDBC and SQLJ configuration properties.

6. 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.
7. In z/OS UNIX System Services, run the DB2Binder utility to bind the packages for the IBM Data Server Driver for JDBC and SQLJ.
8. **Optional:** In DB2 for z/OS, bind the IBM Data Server Driver for JDBC and SQLJ packages into a plan.

One reason that you might continue to use a plan is that you want to continue to use plan-level authorization. For example, suppose that when you ran the DB2Binder utility, you specified a -collection parameter of JDBCCLID. You bind your SQLJ packages into a package collection named SQLJCLID. You can use a BIND command like this to bind the packages into a plan:

```
BIND PLAN(JDBCPLAN) -
QUALIFIER(JDBCUSER) -
PKLIST(SQLJCLID.*, -
JDBCCLID.*) -
ACTION(REPLACE) -
RETAIN -
VALIDATE(BIND)
```

After you bind the plan, you need to execute the SQL GRANT statement to give JDBC and SQLJ users authorization to execute the plan. For example:

```
GRANT EXECUTE ON PLAN JDBCPLAN TO USER1;
```

9. *If you plan to use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to implement distributed transactions against DB2 UDB for OS/390 and z/OS Version 7 servers:* In z/OS UNIX System Services, run the DB2T4XAIndoubtUtil against each of those servers.
10. *If you plan to use LOB locators to access DBCLOB or CLOB columns in DB2 tables on DB2 for z/OS servers:* Create tables on the database servers that are needed for fetching data from DBCLOB or CLOB columns using LOB locators. Use one of the following techniques.
 - On the DB2 for z/OS servers, customize and run job DSNTIJMS. That job is located in data set *prefix.SDSNSAMP*.
 - On the client, in z/OS UNIX System Services, run the com.ibm.db2.jcc.DB2LobTableCreator utility against each of the DB2 for z/OS servers.
11. Verify the installation by running a simple JDBC application.
12. For Java routines (stored procedures and user-defined functions):
 - a. In the JAVAENV data sets for your WLM environments, delete the DB2_HOME environment variable and add a JCC_HOME environment variable.
 JCC_HOME must specify the highest-level directory in the set of directories that contain the IBM Data Server Driver for JDBC and SQLJ code. For example, if you install the IBM Data Server Driver for JDBC and SQLJ in the default location of /usr/lpp/db2910/jcc, set JCC_HOME to /usr/lpp/db2910/jcc.
 - b. If you specify JDBC/SQLJ Driver for OS/390 and z/OS driver-wide properties for the stored procedure environment, you need to change to comparable properties for the IBM Data Server Driver for JDBC and SQLJ.
 For the IBM Data Server Driver for JDBC and SQLJ, you specify the driver-wide properties by specifying the -Ddb2.jcc.propertiesFile=*path* option in a file whose name you specify in the JVMPROPS environment variable. 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 in a JAVAENV data set.

- c. Check the routine definitions in the DB2 catalog.

If the value of COLLID in SYSIBM.SYSROUTINES is DSNJDBC, you need to take one of the following actions:

- Bind the IBM Data Server Driver for JDBC and SQLJ packages into a collection with the DSNJDBC collection ID.
- Redefine the routines with a COLLID value that matches the collection ID for the IBM Data Server Driver for JDBC and SQLJ packages. That value is the -collid value that you specified in the DB2Binder utility. If you did not specify a value, the collection ID is NULLID.

If the COLLID value in SYSIBM.SYSROUTINES is blank, the collection ID of the stored procedure is the same as the collection ID of the programs that call it. You need to take one of the following actions:

- Redefine the routines with a COLLID value that matches the collection ID for the IBM Data Server Driver for JDBC and SQLJ packages. That value is the -collid value that was specified in the DB2Binder utility.
 - Modify the invoking programs:
 - If the invoking programs use packages, bind those packages into a collection with a collection ID that matches the collection ID for the IBM Data Server Driver for JDBC and SQLJ packages.
 - If the invoking programs do not use packages, change the invoking programs to specify the package collections that include the routine procedure packages and the IBM Data Server Driver for JDBC and SQLJ packages. DB2 resolves the packages by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order).
 - Modify the routines to specify the package collections that include any packages that the routine invokes, and the IBM Data Server Driver for JDBC and SQLJ packages.
13. If differences between the JDBC/SQLJ Driver for OS/390 and z/OS and the IBM Data Server Driver for JDBC and SQLJ make application changes necessary, make those changes.
14. Choose one of the following methods to migrate your SQLJ serialized profiles and packages from the JDBC/SQLJ Driver for OS/390 and z/OS to the IBM Data Server Driver for JDBC and SQLJ:
- **Recommended method:** Translate, customize, and bind packages for all of your SQLJ applications that use the IBM Data Server Driver for JDBC and SQLJ. Use the IBM Data Server Driver for JDBC and SQLJ sqlj utility to translate the source code. Then use the db2sqljcustomize utility to customize the serialized profiles and bind the DB2 packages. (db2sqljcustomize binds DB2 packages as well as customizes the serialized profiles if you use the default setting of -automaticbind YES. If you set -automaticbind to NO, you need to run db2sqljbind to bind the DB2 packages.)
 - **Alternative method:** In z/OS UNIX System Services, run the db2sqljupgrade utility. If you choose this method, you do not need to bind new packages for your SQLJ applications.

Important: When the db2sqljupgrade utility creates serialized profiles for the IBM Data Server Driver for JDBC and SQLJ, db2sqljupgrade retrieves the information about input and output host variables and parameters, such as the data type, length, name, and encoding scheme, from the serialized profiles that were created under the JDBC/SQLJ Driver for OS/390 and z/OS. Information that is provided by the JDBC/SQLJ Driver for OS/390

and z/OS does not always fully describe host variables and parameters. Because the db2sqljupgrade utility is an offline utility, it cannot check host variable or parameter information against the corresponding table column or routine definition information. Therefore, you might experience problems due to data incompatibilities when you run your converted SQLJ applications.

Related concepts

“Environment variables for the IBM Data Server Driver for JDBC and SQLJ” on page 441

“Customization of IBM Data Server Driver for JDBC and SQLJ configuration properties” on page 442

“Verify the installation of the IBM Data Server Driver for JDBC and SQLJ” on page 458

Related tasks

“Enabling the DB2-supplied stored procedures and defining the tables used by the IBM Data Server Driver for JDBC and SQLJ” on page 444



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 243

“DB2Binder utility” on page 447

“DB2T4XAIndoubtUtil for distributed transactions with DB2 UDB for OS/390 and z/OS Version 7 servers” on page 454

“DB2LobTableCreator utility” on page 457

“JDBC differences between the current IBM Data Server Driver for JDBC and SQLJ and earlier DB2 JDBC drivers” on page 394

“SQLJ differences between the IBM Data Server Driver for JDBC and SQLJ and other DB2 JDBC drivers” on page 405



DESCRIBE FOR STATIC field (DESCSTAT subsystem parameter) (DB2 Installation and Migration)

Conversion of JDBC/SQLJ Driver for OS/390 and z/OS properties to IBM Data Server Driver for JDBC and SQLJ properties

Because the JDBC/SQLJ Driver for OS/390 and z/OS is no longer supported, you need to convert two types of JDBC/SQLJ Driver for OS/390 and z/OS properties to IBM Data Server Driver for JDBC and SQLJ properties: DataSource properties and driver-level properties.

DataSource properties

All of the JDBC/SQLJ Driver for OS/390 and z/OS DataSource properties are defined for the IBM Data Server Driver for JDBC and SQLJ, so you do not need to modify those properties in your applications when you migrate to the IBM Data Server Driver for JDBC and SQLJ.

Recommendation: Use the pkList property instead of the planName property after you migrate to the IBM Data Server Driver for JDBC and SQLJ. However, if you switch to the pkList property, you need to make additional changes.

When you use the planName property, you bind plans for your SQLJ programs and the JDBC driver. When you use the pkList property, you bind packages for your SQLJ programs and the JDBC driver. This means that if you move from using SQLJ and JDBC plans to packages, you need to grant execution privileges on the SQLJ and JDBC packages to users.

Driver-level properties

With the IBM Data Server Driver for JDBC and SQLJ, you set driver-level properties by setting configuration properties, rather than setting properties in an SQLJ/JDBC run-time properties file. There are a number of differences between those driver-level properties. The following table lists the driver-level JDBC/SQLJ Driver for OS/390 and z/OS properties and their IBM Data Server Driver for JDBC and SQLJ equivalents. If you specify any of the JDBC/SQLJ Driver for OS/390 and z/OS properties, you need to set the equivalent IBM Data Server Driver for JDBC and SQLJ configuration property when you migrate.

Table 97. JDBC/SQLJ Driver for OS/390 and z/OS properties and their IBM Data Server Driver for JDBC and SQLJ equivalents

JDBC/SQLJ Driver for OS/390 and z/OS property	IBM Data Server Driver for JDBC and SQLJ equivalent	Notes
DB2SQLJSSID	db2.jcc.ssid	Property values are the same in the JDBC/SQLJ Driver for OS/390 and z/OS and the IBM Data Server Driver for JDBC and SQLJ.
DB2SQLJPLANNAME	db2.jcc.planName	<ul style="list-style-type: none"> The default for DB2SQLJPLANNAME is DSNJDBC. The default for db2.jcc.planName is null. db2.jcc.pkList is the preferred alternative to db2.jcc.planName. Use of db2.jcc.pkList results in package-level authorization, rather than plan-level authorization. If you specify db2.jcc.planName, you need to bind the JDBC driver packages into a plan after you migrate to the IBM Data Server Driver for JDBC and SQLJ.
db2.sqlj.profile.caching	db2.jcc.disableSQLJProfileCaching	<ul style="list-style-type: none"> Both properties have possible values of YES and NO, but the meanings of the same value are opposite. A db2.sqlj.profile.caching value of YES is equivalent to a db2.jcc.disableSQLJProfileCaching value of NO. A db2.sqlj.profile.caching value of NO is equivalent to a db2.jcc.disableSQLJProfileCaching value of YES. The default for db2.sqlj.profile.caching is YES, and the default for db2.jcc.disableSQLJProfileCaching value of NO.
DB2ACCTINTERVAL	db2.jcc.accountingInterval	Property values are the same in the JDBC/SQLJ Driver for OS/390 and z/OS and the IBM Data Server Driver for JDBC and SQLJ.

Table 97. JDBC/SQLJ Driver for OS/390 and z/OS properties and their IBM Data Server Driver for JDBC and SQLJ equivalents (continued)

JDBC/SQLJ Driver for OS/390 and z/OS property	IBM Data Server Driver for JDBC and SQLJ equivalent	Notes
db2.sp.lob.output.parm.size	db2.jcc.lobOutputSize	Property values are the same in the JDBC/SQLJ Driver for OS/390 and z/OS and the IBM Data Server Driver for JDBC and SQLJ.
DB2SQLJDBRMLIB	None	Not needed for the IBM Data Server Driver for JDBC and SQLJ.
DB2SQLJ_LOCAL_LOCATION_NAME	None	Not needed for the IBM Data Server Driver for JDBC and SQLJ.
DB2SQLJJDBCPROGRAM	None	Not needed for the IBM Data Server Driver for JDBC and SQLJ.
db2.jdbc.profile.pathname	None	Not needed for the IBM Data Server Driver for JDBC and SQLJ.
DB2CURSORHOLD	None	If DB2CURSORHOLD was set to NO under the JDBC/SQLJ Driver for OS/390 and z/OS, programs might behave differently when they run under the IBM Data Server Driver for JDBC and SQLJ.
DB2SQLJ_TRACE_FILENAME	db2.jcc.override.traceFileName, db2.jcc.t2zosTraceFileName	<ul style="list-style-type: none"> The JDBC/SQLJ Driver for OS/390 and z/OS uses DB2SQLJ_TRACE_FILENAME to specify the output files for Java-side tracing as well as native-side tracing. The IBM Data Server Driver for JDBC and SQLJ uses db2.jcc.override.traceFileName to specify Java-side tracing and db2.jcc.t2zosTraceFileName to specify native-side tracing.
DB2SQLJ_DISABLE_JTRACE_TIMESTAMP	None	Not needed for the IBM Data Server Driver for JDBC and SQLJ.
DB2SQLJ_DISABLE_JTRACE	None	DB2SQLJ_DISABLE_JTRACE disables Java-side tracing under the JDBC/SQLJ Driver for OS/390 and z/OS. You can produce the same under the IBM Data Server Driver for JDBC and SQLJ by not specifying db2.jcc.override.traceFileName.
DB2SQLJ_TRACE_BUFFER_SIZE	db2.jcc.t2zosTraceBufferSize	Property values are the same in the JDBC/SQLJ Driver for OS/390 and z/OS and the IBM Data Server Driver for JDBC and SQLJ.
DB2SQLJ_TRACE_WRAP	db2.jcc.t2zosTraceWrap	Property values are the same in the JDBC/SQLJ Driver for OS/390 and z/OS and the IBM Data Server Driver for JDBC and SQLJ.
DB2SQLJ_TRACE_DUMP_FREQ	db2.jcc.t2zosTraceDumpFreq	Property values are the same in the JDBC/SQLJ Driver for OS/390 and z/OS and the IBM Data Server Driver for JDBC and SQLJ.
DB2SQLJMULTICONTEXT	None	Not needed for the IBM Data Server Driver for JDBC and SQLJ.
db2.connpool.max.size	None	Not needed for the IBM Data Server Driver for JDBC and SQLJ.
db2.connpool.idle.timeout	None	Not needed for the IBM Data Server Driver for JDBC and SQLJ.

Table 97. JDBC/SQLJ Driver for OS/390 and z/OS properties and their IBM Data Server Driver for JDBC and SQLJ equivalents (continued)

JDBC/SQLJ Driver for OS/390 and z/OS property	IBM Data Server Driver for JDBC and SQLJ equivalent	Notes
db2.connpool.connect.create.timeout	None	Not needed for the IBM Data Server Driver for JDBC and SQLJ.

Converting JDBC/SQLJ Driver for OS/390 and z/OS serialized profiles to IBM Data Server Driver for JDBC and SQLJ serialized profiles

Although only the DB2 Driver for JDBC and SQLJ is supported in DB2 Version 9.1 for z/OS, you can convert serialized profiles that you customized in previous releases of DB2 for z/OS to a format that is compatible with the IBM Data Server Driver for JDBC and SQLJ. To do that, run the db2sqljupgrade utility.

Before you can run the db2sqljupgrade utility, the IBM Data Server Driver for JDBC and SQLJ must be installed.

To convert your serialized profiles, follow these steps:

1. Determine the correct collection for a serialized profile that you plan to convert:
 - a. Run the db2profp utility.
 - b. Locate the program name in the db2profp output. The program name is the stem for each of the four DBRMs that the SQLJ customizer produces for a serialized profile.

For example, db2profp output from sample program Sample02.sqlj looks like this:

```
=====
printing contents of profile Sample02_SJProfile0
created 1137709347170 (Thu Jan 19 14:22:27 PST 2006)
DB2 consistency token is x'00000108E4C2F162'
DB2 program version string is null
DB2 program name is "SQLJ01"
associated context is Sample02ctx
profile loader is sqlj.runtime.profile.DefaultLoader@6a049a03
contains 1 customizations
COM.ibm.db2os390.sqlj.custom.DB2SQLJCustomizer@38f81a03
original source file: null
contains 2 entries
=====
```

The program name is SQLJ01, so the four DBRMs, and the packages into which they are bound, are SQLJ011, SQLJ012, SQLJ013, and SQLJ014.

- c. Query catalog table SYSIBM.SYSPACKAGE to determine the collection ID that is associated with the four packages.

For example:

```
SELECT NAME, COLLID
FROM SYSIBM.SYSPACKAGE
WHERE NAME IN ('SQLJ011', 'SQLJ012', 'SQLJ013', 'SQLJ014')
```

2. Run the db2sqljupgrade utility.

For example:

```
db2sqljupgrade -collection SQLJ01 Sample02_SJProfile0.ser
```

3. Set the db2.jcc.pkList or db2.jcc.planName configuration property.

If you do not plan to continue to use a plan for your SQLJ applications, specify the name of the package list for those applications in the db2.jcc.pkList configuration property.

If you plan to continue to use a plan for your SQLJ applications, specify that plan name in the db2.jcc.planName property.

4. Verify that the conversion was successful by running the application that is associated with the serialized profile.

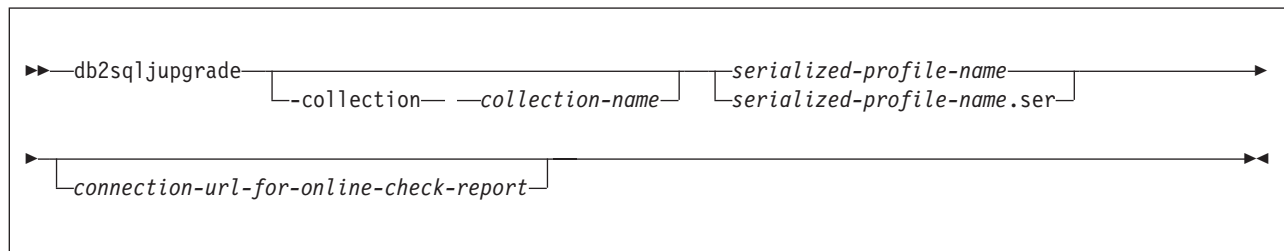
db2sqljupgrade utility

The db2sqljupgrade utility converts an SQLJ serialized profile that you customized under JDBC/SQLJ Driver for OS/390 and z/OS to a format that is compatible with the IBM Data Server Driver for JDBC and SQLJ.

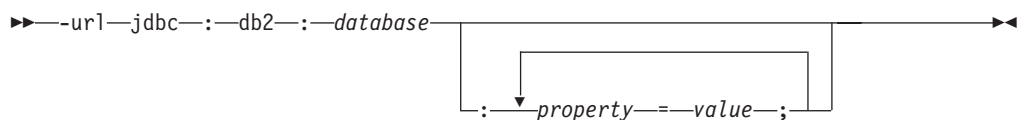
db2sqljupgrade environment

The db2sqljupgrade utility runs under the IBM Data Server Driver for JDBC and SQLJ.

db2sqljupgrade syntax



connection-url-for-online-check-report:



db2sqljupgrade parameter descriptions

-collection

Specifies the collection ID for the DB2 packages that were bound for the application that is associated with the JDBC/SQLJ Driver for OS/390 and z/OS serialized profile. This collection ID is stored in the converted serialized profile and is used as the qualifier for the DB2 packages for the application. The packages were created using the DB2 BIND command from DBRMs that were created when the db2profc command was run to create the serialized profile. The default is NULLID.

serialized-profile-name or serialized-profile-name.ser

Specifies the name of the JDBC/SQLJ Driver for OS/390 and z/OS serialized profile that is to be converted to the IBM Data Server Driver for JDBC and SQLJ format.

connection-url-for-online-check-report

Specifies a URL for IBM Data Server Driver for JDBC and SQLJ type 2

connectivity. The URL specifies a data source to which to connect, to produce a report that lists differences between the upgraded profile that is produced by db2sqljupgrade, and the profile that would have been produced if online checking were enabled during customization. Components of the URL are:

jdbc:db2:

Indicates that the connection is to a DB2 for z/OS server.

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, 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.

Recommendation: Specify the currentSchema and currentSQLID properties, which allow the IBM Data Server Driver for JDBC and SQLJ to do a successful PREPARE and DESCRIBE of the entries in the SQLJ profile. You also need to set the pkList property so that you can make a connection.

Output from db2sqljupgrade

db2sqljupgrade produces the following files:

- The original serialized profile, with the name *serialized-profile-name.ser_old*.
- The upgraded serialized profile, with the name *serialized-profile-name.ser*.
- If you specify the *connection-url-for-online-check-report* option, a file named *serialized-profile-name.rpt*.

Contents of the *serialized-profile-name.rpt* file

The *serialized-profile-name.rpt* file lists differences between the upgraded profile that is produced by db2sqljupgrade, and the profile that would have been produced if online checking were enabled during customization.

The report provides the following information:

- Significant differences in SQL type codes for result set columns or input variables

For example, the difference between VARCHAR and LONG VARCHAR is not considered to be a significant difference.

- For numeric result set columns or input variables, differences in:
 - Precision and scale, for decimal variables
 - Length, for other numeric variables
- Coded character set identifier (CCSID) differences for character or CLOB result set columns or input variables

This difference includes CCSID differences for character representations of datetime types that are sent to or retrieved from the data source.

The JDBC/SQLJ Driver for OS/390 and z/OS does not store the CCSIDs for columns in the serialized profile. It stores the encoding scheme. db2sqljupgrade derives the CCSID for the CCSID comparison from the encoding scheme and the CCSID values in the DSNHDECP module.

Restriction: The report does not provide information about the following differences:

- Differences in nullability of result set columns or input variables
- Differences in parameters of an SQL CALL statement
- Differences in data type information, when that information is overridden at run time.

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` property.

You can set this property in one of the following ways:

- If you use the `DriverManager` interface, set `securityMechanism` in a `java.util.Properties` object before you invoke the form of the `getConnection` method that includes the `java.util.Properties` parameter.
- If you use the `DataSource` interface, and you are creating and deploying your own `DataSource` objects, invoke the `DataSource.setSecurityMechanism` method after you create a `DataSource` object.

You can determine the security mechanism that is in effect for a connection by calling the `DB2Connection.getDB2SecurityMechanism` method.

The following table lists the security mechanisms that the IBM Data Server Driver for JDBC and SQLJ supports, and the data sources that support those security mechanisms.

Table 98. Database server support for IBM Data Server Driver for JDBC and SQLJ security mechanisms

Security mechanism	Supported by			
	DB2 Database for Linux, UNIX, and Windows	DB2 for z/OS	IBM Informix Dynamic Server	DB2 for i
User ID and password	Yes	Yes	Yes	Yes
User ID only	Yes	Yes	Yes	Yes
User ID and encrypted password	Yes	Yes	Yes	Yes ²
Encrypted user ID	Yes	Yes	No	No
Encrypted user ID and encrypted password	Yes	Yes	Yes	Yes ²
Encrypted user ID and encrypted security-sensitive data	No	Yes	No	No
Encrypted user ID, encrypted password, and encrypted security-sensitive data	Yes	Yes	No	No
Kerberos ¹	Yes	Yes	No	Yes
Plugin ¹	Yes	No	No	No

Note:

1. Available for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity only.
 2. The version of the data source must be DB2 for i V6R1 or later.
-

The following table lists the security mechanisms that the IBM Data Server Driver for JDBC and SQLJ supports, and the value that you need to specify for the `securityMechanism` property to specify each security mechanism.

The default security mechanism is `CLEAR_TEXT_PASSWORD_SECURITY`. If the server does not support `CLEAR_TEXT_PASSWORD_SECURITY` but supports `ENCRYPTED_USER_AND_PASSWORD_SECURITY`, the IBM Data Server Driver for JDBC and SQLJ driver updates the security mechanism to `ENCRYPTED_USER_AND_PASSWORD_SECURITY` and attempts to connect to the server. Any other mismatch in security mechanism support between the requester and the server results in an error.

Table 99. Security mechanisms supported by the IBM Data Server Driver for JDBC and SQLJ

Security mechanism	<code>securityMechanism</code> property value
User ID and password	<code>DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY</code>
User ID only	<code>DB2BaseDataSource.USER_ONLY_SECURITY</code>
User ID and encrypted password	<code>DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY</code>
Encrypted user ID	<code>DB2BaseDataSource.ENCRYPTED_USER_ONLY_SECURITY</code>
Encrypted user ID and encrypted password	<code>DB2BaseDataSource.ENCRYPTED_USER_AND_PASSWORD_SECURITY</code>
Encrypted user ID and encrypted security-sensitive data	<code>DB2BaseDataSource.ENCRYPTED_USER_AND_DATA_SECURITY</code>
Encrypted user ID, encrypted password, and encrypted security-sensitive data	<code>DB2BaseDataSource.ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY</code>
Kerberos	<code>DB2BaseDataSource.KERBEROS_SECURITY</code>
Plugin	<code>DB2BaseDataSource.PLUGIN_SECURITY</code>

Related reference

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

User ID and password security under the IBM Data Server Driver for JDBC and SQLJ

With the IBM Data Server Driver for JDBC and SQLJ, one of the available security methods is user ID and password security.

To specify user ID and password security for a JDBC connection, use one of the following techniques.

For the *DriverManager* interface: You can specify the user ID and password directly in the `DriverManager.getConnection` invocation. For example:

```
import java.sql.*;          // JDBC base
...
String id = "dbadm";        // Set user ID
String pw = "dbadm";        // Set password
String url = "jdbc: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:

```

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:

```

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:

```

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/sampledbs");
                                // Get DataSource object
String id = "dbadm";           // Set user ID
String pw = "dbadm";           // Set password
Connection con = ds.getConnection(id, pw);
                                // Create connection

```

Alternatively, if you create and deploy the DataSource object, you can set the user ID and password by invoking the DataSource.setUser and DataSource.setPassword methods after you create the DataSource object. Optionally, you can invoke the DataSource.setSecurityMechanism method property to indicate that you are using user ID and password security. For example:

```

...
com.ibm.db2.jcc.DB2SimpleDataSource ds = // Create DB2SimpleDataSource object
    new com.ibm.db2.jcc.DB2SimpleDataSource();
ds.setDriverType(4); // Set driver type
ds.setDatabaseName("san_jose"); // Set location
ds.setServerName("mvs1.sj.ibm.com"); // Set server name
ds.setPortNumber(5021); // Set port number
ds.setUser("dbadm"); // Set user ID
ds.setPassword("dbadm"); // Set password
ds.setSecurityMechanism(

```



```
com.ibm.db2.jcc.DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY);
// Set security mechanism to
// user ID and password
```

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.

Related tasks

“Connecting to a data source using the DataSource interface” on page 15

“Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ” on page 11

“Creating and deploying DataSource objects” on page 19

Related reference

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

User ID-only security under the IBM Data Server Driver for JDBC and SQLJ

With the IBM Data Server Driver for JDBC and SQLJ, one of the available security methods is user-ID only security.

To specify user ID security for a JDBC connection, use one of the following techniques.

For the DriverManager interface: Set the user ID and security mechanism by setting the user and securityMechanism properties in a Properties object, and then invoking the form of the getConnection method that includes the Properties object as a parameter. For example:

```
import java.sql.*;           // JDBC base
import com.ibm.db2.jcc.*;    // IBM Data Server Driver
                             // for JDBC and SQLJ
                             // implementation of JDBC
...
Properties properties = new Properties();
                             // Create a Properties object
properties.put("user", "db2adm"); // Set user ID for the connection
properties.put("securityMechanism",
    new String("" + com.ibm.db2.jcc.DB2BaseDataSource.USER_ONLY_SECURITY + ""));
                             // Set security mechanism to
                             // user ID only
String url = "jdbc: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:

```
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 DataSource interface” on page 15

“Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ” on page 11

“Creating and deploying DataSource objects” on page 19

Related reference

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

Encrypted password, user ID, or user ID and password security under the IBM Data Server Driver for JDBC and SQLJ

IBM Data Server Driver for JDBC and SQLJ supports encrypted password security, encrypted user ID security, or encrypted user ID and encrypted password security for accessing data sources.

The IBM Data Server Driver for JDBC and SQLJ supports 56-bit DES (weak) encryption or 256-bit AES (strong) encryption. AES encryption is available with IBM Data Server Driver for JDBC and SQLJ 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: <http://www.ibm.com/servers/eserver/zseries/software/java/j5jce.html>

For AES encryption, you need to get the unrestricted policy file for JCE. It is available at the following URL: <https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=jcesdk>

Connections to DB2 for i V6R1 or later servers can use encrypted password security or encrypted user ID and encrypted password security. For encrypted password security or encrypted user ID and encrypted password security, the IBM Java Cryptography Extension (ibmjceprovider.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, and only for DES encryption (encryptionAlgorithm value of 1).

DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows 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
- Result sets
- LOB data
- XML data
- Results of describe operations

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 user ID and encrypted password security mechanism, with AES encryption:

```
import java.sql.*;           // JDBC base
import com.ibm.db2.jcc.*;    // IBM Data Server Driver for JDBC
                               // and SQLJ implementation of JDBC

...
Properties properties = new Properties(); // Create a Properties object
properties.put("user", "dbadm");        // Set user ID for the connection
properties.put("password", "dbadm");    // Set password for the connection
properties.put("securityMechanism", "2");
    new String(" + com.ibm.db2.jcc.DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY +
    "");
                               // Set security mechanism to
                               // user ID and encrypted password
properties.put("encryptionAlgorithm", "2");
                               // Request AES security
String url = "jdbc: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:

```
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");
ds.setPortNumber(5021);
ds.setUser("db2adm");
ds.setPassword("db2adm");
ds.setSecurityMechanism(
    com.ibm.db2.jcc.DB2BaseDataSource.ENCrypted_PASSWORD_SECURITY);
ds.setEncryptionAlgorithm(2);

```

// Set the server name
// Set the port number
// Set the user ID
// Set the password
// Set security mechanism to
// User ID and encrypted password
// Request AES encryption

Related tasks

“Connecting to a data source using the DataSource interface” on page 15

“Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ” on page 11

“Creating and deploying DataSource objects” on page 19

Related reference

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

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
- Java Generic Security Service (JGSS)
- Java Authentication and Authorization Service (JAAS)

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:

```

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:

```

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:

```

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("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:

```

import java.sql.*;           // JDBC base
import com.ibm.db2.jcc.*;    // IBM Data Server Driver for JDBC
                               // and SQLJ implementation of 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:

```

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("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:

```

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(
    com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY);
// Set security mechanism to
// Kerberos

```

Related tasks

“Connecting to a data source using the *DataSource* interface” on page 15

“Connecting to a data source using the *DriverManager* interface with the IBM Data Server Driver for JDBC and SQLJ” on page 11

“Creating and deploying *DataSource* objects” on page 19

Related reference

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

IBM Data Server Driver for JDBC and SQLJ trusted context support

The IBM Data Server Driver for JDBC and SQLJ provides methods that allow you to establish and use trusted connections in Java programs.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 Database for Linux, UNIX, and Windows Version 9.5 or later, and DB2 for z/OS Version 9.1 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS Version 9.1 or later

A three-tiered application model consists of a database server, a middleware server such as WebSphere Application Server, and end users. With this model, the

middleware server is responsible for accessing the database server on behalf of end users. Trusted context support ensures that an end user's database identity and database privileges are used when the middleware server performs any database requests on behalf of that end user.

A trusted context is an object that the database administrator defines that contains a system authorization ID and a set of trust attributes. Currently, for DB2 database servers, a database connection is the only type of context that is supported. The trust attributes identify a set of characteristics of a connection that are required for the connection to be considered a trusted connection. The relationship between a database connection and a trusted context is established when the connection to the database server is first created, and that relationship remains for the life of the database connection.

After a trusted context is defined, and an initial trusted connection to the DB2 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 `DB2XADatasource` class provides several versions of the `getDB2TrustedXAConnection` method, which allow an application server to establish the initial trusted connection. You choose a method based on the types of connection properties that you pass and whether you use Kerberos security. When an application server calls one of these methods, the IBM Data Server Driver for JDBC and SQLJ returns an `Object[]` array with two elements:

- The first element contains a connection instance for the initial connection.
- The second element contains a unique cookie for the connection instance. The cookie is generated by the JDBC driver and is used for authentication during subsequent connection reuse.

The `DB2PooledConnection` class provides several versions of the `getDB2Connection` method, and the `DB2Connection` class provides several versions of the `reuseDB2Connection` method, which allow an application server to reuse an existing trusted connection on behalf of a new user. The application server uses the method to pass the following items to the new user:

- The cookie from the initial connection
- New connection properties for the reused connection

The JDBC driver checks that the supplied cookie matches the cookie of the underlying trusted physical connection, to ensure that the connection request originates from the application server that established the trusted physical connection. If the cookies match, the connection becomes available for immediate use by this new user, with the new properties.

Example: Obtain the initial trusted connection:

```
// Create a DB2ConnectionPoolDataSource instance
com.ibm.db2.jcc.DB2ConnectionPoolDataSource dataSource =
    new com.ibm.db2.jcc.DB2ConnectionPoolDataSource();
// Set properties for this instance
dataSource.setDatabaseName ("STLEC1");
dataSource.setServerName ("v7ec167.svl.ibm.com");
```



```

dataSource.setDriverType (4);
dataSource.setPortNumber(446);
java.util.Properties properties = new java.util.Properties();
// Set other properties using
// properties.put("property", "value");
// Supply the user ID and password for the connection
String user = "user";
String password = "password";
// Call getDB2TrustedPooledConnection to get the trusted connection
// instance and the cookie for the connection
Object[] objects = dataSource.getDB2TrustedPooledConnection(
    user,password, properties);

```

Example: Reuse an existing trusted connection:

```

// The first item that was obtained from the previous getDB2TrustedPooledConnection
// call is a connection object. Cast it to a PooledConnection object.
javax.sql.PooledConnection pooledCon =
    (javax.sql.PooledConnection)objects[0];
properties = new java.util.Properties();
// Set new properties for the reused object using
// properties.put("property", "value");
// The second item that was obtained from the previous getDB2TrustedPooledConnection
// call is the cookie for the connection. Cast it as a byte array.
byte[] cookie = ((byte[])(objects[1]));
// Supply the user ID for the new connection.
String newuser = "newuser";
// Supply the name of a mapping service that maps a workstation user
// ID to a z/OS RACF ID
String userRegistry = "registry";
// Do not supply any security token data to be traced.
byte[] userSecTkn = null;
// Do not supply a previous user ID.
String originalUser = null;
// Call getDB2Connection to get the connection object for the new
// user.
java.sql.Connection con =
    ((com.ibm.db2.jcc.DB2PooledConnection)pooledCon).getDB2Connection(
        cookie,newuser,password,userRegistry,userSecTkn,originalUser,properties);

```

Related tasks

“Connecting to a data source using the DataSource interface” on page 15

“Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ” on page 11

“Creating and deploying DataSource objects” on page 19

Related reference

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

IBM Data Server Driver for JDBC and SQLJ support for SSL

The IBM Data Server Driver for JDBC and SQLJ provides support for the Secure Sockets Layer (SSL) through the Java Secure Socket Extension (JSSE).

You can use SSL support in your Java applications if you use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS Version 9 or later, to DB2 Database for Linux, UNIX, and Windows Version 9.1, Fix Pack 2 or later, or to IBM Informix Dynamic Server (IDS) Version 11.50 or later.

If you use SSL support for a connection to a DB2 for z/OS data source, and the z/OS version is V1.8, V1.9, or V1.10, the appropriate PTF for APAR PK72201 must be applied to Communication Server for z/OS IP Services.

To use SSL connections, you need to:

- Configure connections to the data source to use SSL.
- Configure your Java Runtime Environment to use SSL.

Related tasks

“Connecting to a data source using the DataSource interface” on page 15

“Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ” on page 11

“Creating and deploying DataSource objects” on page 19

 [Configuring the DB2 server for SSL \(DB2 Administration Guide\)](#)

Related reference

“Properties for the IBM Data Server Driver for JDBC and SQLJ” on page 201

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`.

Prerequisite: Before a connection to a data source can use SSL, the port to which the application connects must be configured in the database server as the SSL listener port.

1. Set `DB2BaseDataSource.sslConnection` on a `Connection` or `DataSource` instance.
2. Optional: Set `DB2BaseDataSource.sslTrustStoreLocation` on a `Connection` or `DataSource` instance to identify the location of the truststore. Setting the `sslTrustStoreLocation` property is an alternative to setting the Java `javax.net.ssl.trustStore` property. If you set `DB2BaseDataSource.sslTrustStoreLocation`, `javax.net.ssl.trustStore` is not used.
3. Optional: Set `DB2BaseDataSource.sslTrustStorePassword` on a `Connection` or `DataSource` instance to identify the truststore password. Setting the `sslTrustStorePassword` property is an alternative to setting the Java `javax.net.ssl.trustStorePassword` property. If you set `DB2BaseDataSource.sslTrustStorePassword`, `javax.net.ssl.trustStorePassword` is not used.

The following example demonstrates how to set the `sslConnection` property on a `Connection` instance:

```
java.util.Properties properties = new java.util.Properties();
properties.put("user", "xxxx");
properties.put("password", "yyyy");
properties.put("sslConnection", "true");
java.sql.Connection con =
    java.sql.DriverManager.getConnection(url, properties);
```

Configuring the Java Runtime Environment to use SSL

Before you can use Secure Sockets Layer (SSL) connections in your JDBC and SQLJ applications, you need to configure the Java Runtime Environment to use SSL.

Before you can configure your Java Runtime Environment for SSL, you need to satisfy the following prerequisites:

- The Java Runtime Environment must include a Java security provider. The IBM JSSE provider or the Sun JSSE provider must be installed. The IBM JSSE provider is automatically installed with the IBM SDK for Java.

Restriction: You can only use the Sun JSSE provider only with a Sun Java Runtime Environment. The Sun JSSE provider does not work with an IBM Java Runtime Environment.

- SSL support must be configured on the database server.

To configure your Java Runtime Environment to use SSL, follow these steps.

1. Import a certificate from the database server to a Java truststore on the client.

Use the Java `keytool` utility to import the certificate into the truststore.

For example, suppose that the server certificate is stored in a file named `jcc.cacert`. Issue the following `keytool` utility statement to read the certificate from file `jcc.cacert`, and store it in a truststore named `cacerts`.

```
keytool -import -file jcc.cacert -keystore cacerts
```

2. Configure the Java Runtime Environment for the Java security providers by adding entries to the `java.security` file.

The format of a security provider entry is:

```
security.provider.n=provider-package-name
```

A provider with a lower value of *n* takes precedence over a provider with a higher value of *n*.

The Java security provider entries that you add depend on whether you use the IBM JSSE provider or the Sun JSSE provider.

- If you use the Sun JSSE provider, add entries for the Sun security providers to your `java.security` file.
- If you use the IBM JSSE provider, use one of the following methods:
 - **Use the IBMJSSE2 provider (supported for the IBM SDK for Java 1.4.2 and later):**

Recommendation: Use the IBMJSSE2 provider, and use it in FIPS mode.

- If you do not need to operate in FIPS-compliant mode:
 - For the IBM SDK for Java 1.4.2, add an entry for the IBMJSSE2Provider to the `java.security` file. Ensure that an entry for the IBMJCE provider is in the `java.security` file. The `java.security` file that is shipped with the IBM SDK for Java contains an entry for entries for IBMJCE.
 - For later versions of the IBM SDK for Java, ensure that entries for the IBMJSSE2Provider and the IBMJCE provider are in the `java.security` file. The `java.security` file that is shipped with the IBM SDK for Java contains entries for those providers.
- If you need to operate in FIPS-compliant mode:
 - Add an entry for the IBMJCEFIPS provider to your `java.security` file before the entry for the IBMJCE provider. Do not remove the entry for the IBMJCE provider.
 - Enable FIPS mode in the IBMJSSE2 provider. See step 3 on page 489.
- **Use the IBMJSSE provider (supported for the IBM SDK for Java 1.4.2 only):**
 - If you do not need to operate in FIPS-compliant mode, ensure that entries for the IBMJSSEProvider and the IBMJCE provider are in the

java.security file. The java.security file that is shipped with the IBM SDK for Java contains entries for those providers.

- If you need to operate in FIPS-compliant mode, add entries for the FIPS-approved provider IBMJSSEFIPSProvider and the IBMJCEFIPS provider to your java.security file, before the entry for the IBMJCE provider.

Restriction: If you use the IBMJSSE provider on the Solaris operating system, you need to include an entry for the SunJSSE provider before entries for the IBMJCE, IBMJCEFIPS, IBMJSSE, or IBMJSSE2 providers.

Example: Use a java.security file similar to this one if you need to run in FIPS-compliant mode, and you enable FIPS mode in the IBMJSSE2 provider:

```
# Set the Java security providers
security.provider.1=com.ibm.jsse2.IBMJSSEProvider2
security.provider.2=com.ibm.crypto.fips.provider.IBMJCEFIPS
security.provider.3=com.ibm.crypto.provider.IBMJCE
security.provider.4=com.ibm.security.jgss.IBMJGSSProvider
security.provider.5=com.ibm.security.cert.IBMCertPath
security.provider.6=com.ibm.security.sasl.IBMSASL
```

Example: Use a java.security file similar to this one if you need to run in FIPS-compliant mode, and you use the IBMJSSE provider:

```
# Set the Java security providers
security.provider.1=com.ibm.fips.jsse.IBMJSSEFIPSProvider
security.provider.2=com.ibm.crypto.fips.provider.IBMJCEFIPS
security.provider.3=com.ibm.crypto.provider.IBMJCE
security.provider.4=com.ibm.security.jgss.IBMJGSSProvider
security.provider.5=com.ibm.security.cert.IBMCertPath
security.provider.6=com.ibm.security.sasl.IBMSASL
```

Example: Use a java.security file similar to this one if you use the Sun JSSE provider:

```
# Set the Java security providers
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.rsajca.Provider
security.provider.3=com.sun.crypto.provider.SunJCE
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
```

3. If you plan to use the IBM Data Server Driver for JDBC and SQLJ in FIPS-compliant mode, you need to set the com.ibm.jsse2.JSSEFIPS Java system property:

```
com.ibm.jsse2.JSSEFIPS=true
```

Restriction: Non-FIPS-mode JSSE applications cannot run in a JVM that is in FIPS mode.

Restriction: When the IBMJSSE2 provider runs in FIPS mode, it cannot use hardware cryptography.

4. Configure the Java Runtime Environment for the SSL socket factory providers by adding entries to the java.security file.

The format of SSL socket factory provider entries are:

```
ssl.SocketFactory.provider=provider-package-name
ssl.ServerSocketFactory.provider=provider-package-name
```

Specify the SSL socket factory provider for the Java security provider that you are using.

Example: Include SSL socket factory provider entries like these in the java.security file when you enable FIPS mode in the IBMJSSE2 provider:

```
# Set the SSL socket factory provider
ssl.SocketFactory.provider=com.ibm.jsse2.SSLSocketFactoryImpl
ssl.ServerSocketFactory.provider=com.ibm.jsse2.SSLServerSocketFactoryImpl
```

Example: Include SSL socket factory provider entries like these in the java.security file when you enable FIPS mode in the IBMJSSE provider:

```
# Set the SSL socket factory provider
ssl.SocketFactory.provider=com.ibm.fips.jsse.JSSESocketFactory
ssl.ServerSocketFactory.provider=com.ibm.fips.jsse.JSSEServerSocketFactory
```

Example: Include SSL socket factory provider entries like these when you use the Sun JSSE provider:

```
# Set the SSL socket factory provider
ssl.SocketFactory.provider=com.sun.net.ssl.internal.ssl.SSLSocketFactoryImpl
ssl.ServerSocketFactory.provider=com.sun.net.ssl.internal.ssl.SSLServerSocketFactoryImpl
```

5. Configure Java system properties to use the truststore.

To do that, set the following Java system properties:

javax.net.ssl.trustStore

Specifies the name of the truststore that you specified with the -keystore parameter in the keytool utility in step 1 on page 488.

If the IBM Data Server Driver for JDBC and SQLJ property DB2BaseDataSource.sslTrustStoreLocation is set, its value overrides the javax.net.ssl.trustStore property value.

javax.net.ssl.trustStorePassword (optional)

Specifies the password for the truststore. You do not need to set a truststore password. However, if you do not set the password, you cannot protect the integrity of the truststore.

If the IBM Data Server Driver for JDBC and SQLJ property DB2BaseDataSource.sslTrustStorePassword is set, its value overrides the javax.net.ssl.trustStorePassword property value.

Example: One way that you can set Java system properties is to specify them as the arguments of the -D option when you run a Java application. Suppose that you want to run a Java application named MySSL.java, which accesses a data source using an SSL connection. You have defined a truststore named cacerts. The following command sets the truststore name when you run the application.

```
java -Djavax.net.ssl.trustStore=cacerts MySSL
```

Security for preparing SQLJ applications with the IBM Data Server Driver for JDBC and SQLJ

Two ways to provide security during SQLJ application preparation are to allow users to customize applications only, and to limit access to a specific set of tables during customization.

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 -compile=true option to create Java byte codes and serialized profiles for their programs. Then they run the db2sqljcustomize command with the -automaticbind 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.

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.

Related reference

“db2sqljcustomize - SQLJ profile customizer” on page 420

Chapter 11. Java client support for high availability on IBM data servers

Client applications that connect to DB2 Database for Linux, UNIX, and Windows, DB2 for z/OS, or IBM Informix Dynamic Server (IDS) 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 for IDS and DB2 for z/OS. Workload balancing ensures that work is distributed efficiently among servers in a high-availability cluster or data sharing group. A data server that is set up for workload balancing also has automatic client reroute capability.

The following table provides links to server-side information about these features.

Table 100. Server-side information on high availability

Data server	Related topics
DB2 Database for Linux, UNIX, and Windows	Automatic client reroute roadmap
IDS	Manage Cluster Connections with the Connection Manager
DB2 for z/OS	Communicating with data sharing groups

Important: For connections to DB2 for z/OS, this information discusses direct connections to DB2 for z/OS. For information about high availability for connections through DB2 Connect Server, see the DB2 Connect documentation.

Java client support for high availability for connections to DB2 Database for Linux, UNIX, and Windows servers

DB2 Enterprise Server Edition (ESE) with the database partitioning feature (DPF), WebSphere Replication Server, high availability cluster multiprocessor (HACMP™), or high availability disaster recovery (HADR) on DB2 Database for Linux, UNIX, and Windows servers provide high availability for client applications, through 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 DB2 Database for Linux, UNIX, and Windows high-availability support.

For non-Java clients, you need to use one of the following clients or client packages to take advantage of high-availability support:

- IBM Data Server Client
- IBM Data Server Runtime Client
- IBM Data Server Driver Package
- IBM Data Server Driver for ODBC and CLI

High availability support for connections to DB2 Database for Linux, UNIX, and Windows servers includes:

Automatic client reroute

This support enables a client to recover from a failure by attempting to reconnect to the database through an alternate server. Reconnection to another server is called *failover*. The DB2 Database for Linux, UNIX, and Windows database to which you connect needs to be configured to specify an alternate server before you can use this support.

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.

Client affinities

Client affinities is an automatic client reroute solution that is controlled completely by the client. It is intended for situations in which you need to connect to a particular primary server. If an outage occurs during the connection to the primary server, you use client affinities to enforce a specific order for failover to alternate servers.

Configuration of DB2 Database for Linux, UNIX, and Windows high availability support for Java clients

For connections to DB2 Database for Linux, UNIX, and Windows data servers that are set up for automatic client reroute, automatic client reroute capability is enabled at a Java client by default. However, you can set properties to control the

number of retries and the time between retries. You can also set properties to specify a primary and alternate server in case of a failure during the initial connection to a data source.

To configure automatic client reroute on a IBM Data Server Driver for JDBC and SQLJ client:

1. Set the appropriate properties to specify the primary and alternate server addresses to use if the first connection fails.
 - If your application is using the `DriverManager` interface for connections:
 - a. Specify the server name and port number of the primary server that you want to use in the connection URL.
 - b. Set the `clientRerouteAlternateServerName` and `clientRerouteAlternatePortNumber` properties to the server name and port number of the alternate server that you want to use.

Restriction: Automatic client reroute support for connections that are made with the `DriverManager` interface has the following restrictions:

- Alternate server information is shared between `DriverManager` connections only if you create the connections with the same URL and properties.
- You cannot set the `clientRerouteServerListJNDIName` property or the `clientRerouteServerListJNDIContext` properties for a `DriverManager` connection.
- Automatic client reroute is not enabled for default connections (`jdbc:default:connection`).
- If your application is using the `DataSource` interface for connections, use one or both of the following techniques:
 - Set the server names and port numbers in `DataSource` properties:
 - a. Set the `serverName` and `portNumber` properties to the server name and port number of the primary server that you want to use.
 - b. Set the `clientRerouteAlternateServerName` and `clientRerouteAlternatePortNumber` properties to the server name and port number of the alternate server that you want to use.
 - Configure JNDI for high availability by using a `DB2ClientRerouteServerList` instance to identify the primary server and alternate server.
 - a. Create an instance of `DB2ClientRerouteServerList`.
`DB2ClientRerouteServerList` is a serializable Java bean with the following properties:

Property name	Data type
<code>com.ibm.db2.jcc.DB2ClientRerouteServerList.alternateServerName</code>	<code>String[]</code>
<code>com.ibm.db2.jcc.DB2ClientRerouteServerList.alternatePortNumber</code>	<code>int[]</code>
<code>com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryServerName</code>	<code>String[]</code>
<code>com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryPortNumber</code>	<code>int[]</code>

`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` value for the
`DataSource` instance is used.

If you configure `DataSource` properties as well as configuring JNDI for high
availability, the `DataSource` properties have precedence over the JNDI
configuration.

- 2. Set properties to control the number of retries and the time between retries.
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, and
`retryIntervalForClientReroute` is set, the default is 0, which means that
client reroute processing does not occur.

retryIntervalForClientReroute

The number of seconds between consecutive connection retries. The default
is 0 if `maxRetriesForClientReroute` is set.

If neither `maxRetriesForClientReroute` nor `retryIntervalForClientReroute` is set,
automatic client reroute retries the connection to a database for up to 10 minutes,
with no wait between retries.

Example of enabling DB2 Database for Linux, UNIX, and Windows high availability support in Java applications

Java client setup for DB2 Database for Linux, UNIX, and Windows 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 DB2 Database for Linux, UNIX, and Windows high availability support.

Suppose that your installation has a primary server and an alternate server with the following server names and port numbers:

Server name	Port number
srv1.sj.ibm.com	50000
srv3.sj.ibm.com	50002

The following code sets up DataSource properties in an application so that the application connects to srv1.sj.ibm.com as the primary server, and srv3.sj.ibm.com as the alternative server. That is, if srv1.sj.ibm.com is down during the initial connection, the driver should connect to srv3.sj.ibm.com.

```
ds.setDriverType(4);
ds.setServerName("srv1.sj.ibm.com");
ds.setPortNumber("50000");
ds.setClientRerouteAlternateServerName("srv3.sj.ibm.com");
ds.setClientRerouteAlternatePortNumber("50002");
```

The following code configures JNDI for automatic client reroute. It creates an instance of DB2ClientRerouteServerList, binds that instance to the JNDI registry, and assigns the JNDI name of the DB2ClientRerouteServerList object to the clientRerouteServerListJNDIName property.

```
// Create a starting context for naming operations
InitialContext registry = new InitialContext();
// Create a DB2ClientRerouteServerList object
DB2ClientRerouteServerList address = new DB2ClientRerouteServerList();

// Set the port number and server name for the primary server
address.setPrimaryPortNumber(50000);
address.setPrimaryServerName("srv1.sj.ibm.com");

// Set the port number and server name for the alternate server
int[] port = {50002};
String[] server = {"srv3.sj.ibm.com"};
address.setAlternatePortNumber(port);
address.setAlternateServerName(server);

registry.rebind("serverList", address);
// Assign the JNDI name of the DB2ClientRerouteServerList object to the
// clientRerouteServerListJNDIName property
datasource.setClientRerouteServerListJNDIName("serverList");
```

Operation of automatic client reroute for connections to DB2 Database for Linux, UNIX, and Windows from Java clients

When IBM Data Server Driver for JDBC and SQLJ client reroute support is enabled, a Java application that is connected to a DB2 Database for Linux, UNIX, and Windows server can continue to run when the primary server has a failure.

Automatic client reroute for a Java application that is connected to a DB2 Database for Linux, UNIX, and Windows server operates in the following way when support for client affinities is disabled:

1. During each connection to the data source, the IBM Data Server Driver for JDBC and SQLJ obtains primary and alternate server information.
 - For the first connection to a DB2 Database for Linux, UNIX, and Windows server:
 - 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.
 - 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.

If the connection is through the `DriverManager` interface, the IBM Data Server Driver for JDBC and SQLJ creates a `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 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 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, and the following conditions are satisfied, the driver retries the transaction on the new server, without notifying the application.
 - The `enableSeamlessFailover` property is set to `DB2BaseDataSource.YES` (1).
 - The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
 - There are no global temporary tables in use on the server.
 - There are no open, held cursors.
 - If seamless failover is not in effect, the driver throws an `SQLException` to the application with error code -4498, to indicate to the application that the connection was automatically reestablished and the transaction was implicitly rolled back. The application can then retry its transaction without doing an explicit rollback first.

A reason code that is returned with error code -4498 indicates whether any database server special registers that were modified during the original connection are reestablished in the failover connection.

You can determine whether alternate server information was used in establishing the initial connection by calling the `DB2Connection.alternateWasUsedOnConnect` method.
6. After failover, driver memory is updated with new primary and alternate server information from the new primary server.

Examples

Example: Automatic client reroute to a DB2 Database for Linux, UNIX, and Windows server when `maxRetriesForClientReroute` and `retryIntervalForClientReroute` are not set: Suppose that the following properties are set for a connection to a database:

Property	Value
<code>enableClientAffinitiesList</code>	<code>DB2BaseDataSource.NO</code> (2)
<code>serverName</code>	<code>host1</code>
<code>portNumber</code>	<code>port1</code>
<code>clientRerouteAlternateServerName</code>	<code>host2</code>
<code>clientRerouteAlternatePortNumber</code>	<code>port2</code>

The following steps demonstrate an automatic client reroute scenario for a connection to a DB2 Database for Linux, UNIX, and Windows server:

1. The IBM Data Server Driver for JDBC and SQLJ loads `host1:port1` into its memory as the primary server address, and `host2:port2` into its memory as the alternate server address.
2. On the initial connection, the driver tries to connect to `host1:port1`.
3. The connection to `host1:port1` fails, so the driver tries another connection to `host1:port1`.
4. The reconnection to `host1:port1` fails, so the driver tries to connect to `host2:port2`.
5. The connection to `host2:port2` succeeds.
6. The driver retrieves alternate server information that was received from server `host2:port2`, and updates its memory with that information.

Assume that the driver receives a server list that contains host2a:port2a, host2:port2. host2a:port2a is stored as the new primary server, and host2:port2 is stored as the new alternative 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 host2a:port2a as the new primary server. If that connection fails, the driver tries to connect to the new alternate server host2:port2.

7. A communication failure occurs during the connection to host2:port2.
8. The driver tries to connect to host2a:port2a.
9. The connection to host2a:port2a is successful.
10. The driver retrieves alternate server information that was received from server host2a:port2a, and updates its memory with that information.

Example: Automatic client reroute to a DB2 Database for Linux, UNIX, and Windows server when maxRetriesForClientReroute and retryIntervalForClientReroute are set for multiple retries: Suppose that the following properties are set for a connection to a database:

Property	Value
enableClientAffinitiesList	DB2BaseDataSource.NO (2)
serverName	host1
portNumber	port1
clientRerouteAlternateServerName	host2
clientRerouteAlternatePortNumber	port2
maxRetriesForClientReroute	3
retryIntervalForClientReroute	2

The following steps demonstrate an automatic client reroute scenario for a connection to a DB2 Database for Linux, UNIX, and Windows server:

1. The IBM Data Server Driver for JDBC and SQLJ loads host1:port1 into its memory as the primary server address, and host2:port2 into its memory as the alternate server address.
2. On the initial connection, the driver tries to connect to host1:port1.
3. The connection to host1:port1 fails, so the driver tries another connection to host1:port1.
4. The connection to host1:port1 fails again, so the driver tries to connect to host2:port2.
5. The connection to host2:port2 fails.
6. The driver waits two seconds.
7. The driver tries to connect to host1:port1 and fails.
8. The driver tries to connect to host2:port2 and fails.
9. The driver waits two seconds.
10. The driver tries to connect to host1:port1 and fails.
11. The driver tries to connect to host2:port2 and fails.
12. The driver waits two seconds.
13. The driver throws an SQLException with error code -4499.

Java application programming requirements for high availability for connections to DB2 Database for Linux, UNIX, and Windows servers

Failover for automatic client reroute can be seamless or non-seamless. If failover for connections to DB2 Database for Linux, UNIX, and Windows is not seamless, you need to add code to account for the errors that are returned when failover occurs.

If failover is non-seamless, and a connection is reestablished with the server, SQLCODE -4498 (for Java clients) or SQL30108N (for non-Java clients) is returned to the application. All work that occurred within the current transaction is rolled back. In the application, you need to:

- Check the reason code that is returned with the error. Determine whether special register settings on the failing data sharing member are carried over to the new (failover) data sharing member. Reset any special register values that are not current.
- Execute all SQL operations that occurred during the previous transaction.

The following conditions must be satisfied for failover for connections to DB2 Database for Linux, UNIX, and Windows to be seamless:

- The application programming language is Java, CLI, or .NET.
- The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
- All global session data is closed or dropped.
- There are no open, held cursors.
- If the application uses CLI, the application cannot perform actions that require the driver to maintain a history of previously called APIs in order to replay the SQL statement. Examples of such actions are specifying data at execution time, performing compound SQL, or using array input.
- The application is not a stored procedure.
- Autocommit is not enabled. Seamless failover can occur when autocommit is enabled. However, the following situation can cause problems: Suppose that SQL work is successfully executed and committed at the data server, but the connection or server goes down before acknowledgment of the commit operation is sent back to the client. When the client re-establishes the connection, it replays the previously committed SQL statement. The result is that the SQL statement is executed twice. To avoid this situation, turn autocommit off when you enable seamless failover.

Client affinities for DB2 Database for Linux, UNIX, and Windows 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.

Configuration of client affinities for Java clients for DB2 Database for Linux, UNIX, and Windows connections

To enable support for client affinities in Java applications, you set properties to indicate that you want to use client affinities, and to specify the primary and alternate servers.

The following table describes the property settings for enabling client affinities for Java applications.

Table 101. Property settings to enable client affinities for Java applications

IBM Data Server Driver for JDBC and SQLJ	
setting	Value
enableClientAffinitiesList	DB2BaseDataSource.YES (1)
clientRerouteAlternateServerName	A comma-separated list of the primary server and alternate servers
clientRerouteAlternatePortNumber	A comma-separated list of the port numbers for the primary server and alternate servers
enableSeamlessFailover	DB2BaseDataSource.YES (1) for seamless failover; DB2BaseDataSource.NO (2) or enableSeamlessFailover not specified for no seamless failover
maxRetriesForClientReroute	The number of times to retry the connection to each server, including the primary server, after a connection to the primary server fails. The default is 3.
retryIntervalForClientReroute	The number of seconds to wait between retries. The default is no wait.

Example of enabling client affinities in Java clients for DB2 Database for Linux, UNIX, and Windows connections

Before you can use client affinities for automatic client reroute in Java applications, you need to set properties to indicate that you want to use client affinities, and to identify the primary alternate servers.

The following example shows how to enable client affinities.

Suppose that you set the following properties for a connection to a database:

Property	Value
enableClientAffinitiesList	DB2BaseDataSource.YES (1)
clientRerouteAlternateServerName	host1,host2,host3
clientRerouteAlternatePortNumber	port1,port2,port3
maxRetriesForClientReroute	3
retryIntervalForClientReroute	2

Suppose that a communication failure occurs during a connection to the server that is identified by host1:port1. The following steps demonstrate automatic client reroute with client affinities.

1. The driver tries to connect to host1:port1.
2. The connection to host1:port1 fails.
3. The driver waits two seconds.
4. The driver tries to connect to host1:port1.
5. The connection to host1:port1 fails.
6. The driver waits two seconds.
7. The driver tries to connect to host1:port1.
8. The connection to host1:port1 fails.
9. The driver waits two seconds.
10. The driver tries to connect to host2:port2.
11. The connection to host2:port2 fails.
12. The driver waits two seconds.
13. The driver tries to connect to host2:port2.
14. The connection to host2:port2 fails.
15. The driver waits two seconds.
16. The driver tries to connect to host2:port2.
17. The connection to host2:port2 fails.
18. The driver waits two seconds.
19. The driver tries to connect to host3:port3.
20. The connection to host3:port3 fails.
21. The driver waits two seconds.
22. The driver tries to connect to host3:port3.
23. The connection to host3:port3 fails.
24. The driver waits two seconds.
25. The driver tries to connect to host3:port3.
26. The connection to host3:port3 fails.
27. The driver waits two seconds.
28. The driver throws an SQLException with error code -4499.

Java client support for high availability for connections to IDS servers

High-availability cluster support on IBM Informix Dynamic Server (IDS) 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 IDS 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 IDS 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 IDS 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 IDS 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, IDS 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 IDS. The connection concentrator reduces the resources that are required on IDS 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 IDS high-availability support for Java clients

To configure a IBM Data Server Driver for JDBC and SQLJ client application that connects to an IDS 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.

IBM Data Server Driver for JDBC and SQLJ client reroute support for DB2 Database for Linux, UNIX, and Windows and IBM Informix Dynamic Server works for connections that are obtained using the `javax.sql.DataSource`, `javax.sql.ConnectionPoolDataSource`, `javax.sql.XADataSource`, or `java.sql.DriverManager` interface.

Restriction: Automatic client reroute support for connections that are made with the `DriverManager` interface has the following restrictions:

- Alternate server information is shared between `DriverManager` connections only if you create the connections with the same URL and properties.
- You cannot set the `clientRerouteServerListJNDIName` property or the `clientRerouteServerListJNDIContext` properties for a `DriverManager` connection.
- Automatic client reroute is not enabled for default connections (`jdbc:default:connection`).

Before you can enable IBM Data Server Driver for JDBC and SQLJ for high availability for connections to IBM Informix Dynamic Server, 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 102. Basic settings to enable IDS high availability support in Java applications

IBM Data Server Driver for JDBC and SQLJ	
setting	Value
<code>enableSysplexWLB</code> property	true
<code>maxTransportObjects</code> property	The maximum number of connections that the requester can make to the high-availability cluster
Connection address:	
server	The IP address of a Connection Manager. See “Setting server and port properties for connecting to a Connection Manager” on page 506.

Table 102. Basic settings to enable IDS high availability support in Java applications (continued)

IBM Data Server Driver for JDBC and SQLJ setting	Value
port	The SQL port number for the Connection Manager. See “Setting server and port properties for connecting to a Connection Manager.”
database	The database name

If you want to enable the connection concentrator, but you do not want to enable workload balancing, you can use these properties.

Table 103. Settings to enable the IDS connection concentrator without workload balancing in Java applications

IBM Data Server Driver for JDBC and SQLJ setting	Value
enableSysplexWLB property	false
enableConnectionConcentrator property	true

If you want to fine-tune IDS 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 104. Properties for fine-tuning IDS high-availability support for connections from the IBM Data Server Driver for JDBC and SQLJ

IBM Data Server Driver for JDBC and SQLJ configuration property	Description
db2.jcc.maxTransportObjectIdleTime	Specifies the maximum elapsed time in number of seconds before an idle transport is dropped. The default is 60. The minimum supported value is 0.
db2.jcc.maxTransportObjectWaitTime	Specifies the number of seconds that the client will wait for a transport to become available. The default is -1 (unlimited). The minimum supported value is 0.
db2.jcc.minTransportObjects	Specifies the lower limit for the number of transport objects in a global transport object pool. The default value is 0. Any value that is less than or equal to 0 means that the global transport object pool can become empty.

Setting server and port properties for connecting to a Connection Manager

To set the server and port number for connecting to a Connection Manager, follow this process:

- If your high-availability cluster is using a single Connection Manager, and your application is using the DataSource interface for connections, set the serverName and portNumber properties to the server name and port number of the Connection Manager.

- If your high-availability cluster is using a single Connection Manager, and your application is using the DriverManager interface for connections, specify the server name and port number of the Connection manager in the connection URL.
- If your high-availability cluster is using more than one Connection manager, and your application is using the DriverManager interface for connections:
 1. Specify the server name and port number of the main Connection Manager that you want to use in the connection URL.
 2. Set the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties to the server names and port numbers of the alternative Connection Managers that you want to use.
- If your high-availability cluster is using more than one Connection Manager, and your application is using the DataSource interface for connections, use one of the following techniques:
 - Set the server names and port numbers in DataSource properties:
 1. Set the serverName and portNumber properties to the server name and port number of the main Connection Manager that you want to use.
 2. Set the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties to the server names and port numbers of the alternative Connection Managers that you want to use.
 - Configure JNDI for high availability by using a DB2ClientRerouteServerList instance to identify the main Connection Manager and alternative Connection Managers.
 1. Create an instance of DB2ClientRerouteServerList.
DB2ClientRerouteServerList is a serializable Java bean with the following properties:

Property name	Data type
com.ibm.db2.jcc.DB2ClientRerouteServerList.alternateServerName	String[]
com.ibm.db2.jcc.DB2ClientRerouteServerList.alternatePortNumber	int[]
com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryServerName	String[]
com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryPortNumber	int[]

getXXX and setXXX methods are defined for each property.

2. Set the com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryServerName and com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryPortNumber properties to the server name and port number of the main Connection Manager that you want to use.
3. Set the com.ibm.db2.jcc.DB2ClientRerouteServerList.alternateServerName and com.ibm.db2.jcc.DB2ClientRerouteServerList.alternatePortNumber properties to the server names and port numbers of the alternative Connection Managers that you want to use.
4. To make the DB2ClientRerouteServerList persistent:
 - a. Bind the DB2ClientRerouteServerList instance to the JNDI registry.
 - b. Assign the JNDI name of the DB2ClientRerouteServerList object to the IBM Data Server Driver for JDBC and SQLJ clientRerouteServerListJNDIName property.

- c. Assign the name of the JNDI context that is used for binding and lookup of the DB2ClientRerouteServerList instance to the clientRerouteServerListJNDIContext property.

When a DataSource is configured to use JNDI for storing automatic client reroute alternate information, the standard server and port properties of the DataSource are not used for a getConnection request. Instead, the primary server address is obtained from the transient clientRerouteServerList information. If the JNDI store is not available due to a JNDI bind or lookup failure, the IBM Data Server Driver for JDBC and SQLJ attempts to make a connection using the standard server and port properties of the DataSource. Warnings are accumulated to indicate that a JNDI bind or lookup failure occurred.

After a failover:

- The IBM Data Server Driver for JDBC and SQLJ attempts to propagate the updated server information to the JNDI store.
- primaryServerName and primaryPortNumber values that are specified in DB2ClientRerouteServerList are used for the connection. If primaryServerName is not specified, the serverName and portNumber values for the DataSource instance are used.

Example of enabling IDS high availability support in Java applications

Java client setup for IDS 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 IDS 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 the Sysplex workload balancing by following these steps:
 - a. Issue the following command in a command line window:

```
java com.ibm.db2.jcc.DB2Jcc -version
```
 - b. Find a line in the output like this, and check that *nnn* is 3.52 or later.
 - c.

```
[jcc] Driver: IBM Data Server Driver for JDBC and SQLJ Architecture nnn xxx
```
2. Set IBM Data Server Driver for JDBC and SQLJ Connection or DataSource properties to enable the connection concentrator or workload balancing:

Start with settings similar to these:

 - enableSysplexWLB
 - maxTransportObjects

Table 105. Example of Connection or DataSource property settings for workload balancing for IDS

Property	Setting
enableSysplexWLB	true
maxTransportObjects	80

Enabling workload balancing enables the connection concentrator by default.

The values that are specified are not intended to be recommended values. You need to determine values based on factors such as the number of transport objects that are available. The number of transport objects must be equal to or greater than the number of connection objects.

3. Set IBM Data Server Driver for JDBC and SQLJ configuration properties to fine-tune the connection concentrator 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.maxTransportObjectsStart with a setting similar to this one:
db2.jcc.maxTransportObjects=500
 - c. Include the directory that contains DB2JccConfiguration.properties in the CLASSPATH concatenation.

Operation of automatic client reroute for connections to IDS 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 Dynamic Server (IDS) 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 IDS 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 IDS:
 - 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 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 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 current 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 that is returned from the new primary server.

Examples

Example: Automatic client reroute to an IDS server when `maxRetriesForClientReroute` and `retryIntervalForClientReroute` are not set: Suppose that the following properties are set for a connection to a database:

Property	Value
<code>enableClientAffinitiesList</code>	<code>DB2BaseDataSource.NO (2)</code>
<code>serverName</code>	<code>host1</code>
<code>portNumber</code>	<code>port1</code>
<code>clientRerouteAlternateServerName</code>	<code>host2</code>
<code>clientRerouteAlternatePortNumber</code>	<code>port2</code>

The following steps demonstrate an automatic client reroute scenario for a connection to IDS:

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 IDS server when maxRetriesForClientReroute and retryIntervalForClientReroute are set for multiple retries: Suppose that the following properties are set for a connection to a database:

Property	Value
enableClientAffinitiesList	DB2BaseDataSource.NO (2)
serverName	host1
portNumber	port1
clientRerouteAlternateServerName	host2
clientRerouteAlternatePortNumber	port2
maxRetriesForClientReroute	3
retryIntervalForClientReroute	2

The following steps demonstrate an automatic client reroute scenario for a connection to IDS:

1. The IBM Data Server Driver for JDBC and SQLJ tries to connect to the Connection Manager that is identified by host1:port1.
2. The connection to host1:port1 fails, so the driver tries to connect to the Connection Manager that is identified by host2:port2.
3. The connection to host2:port2 succeeds.
4. The driver retrieves alternate server information from the connection manager that is identified by host2:port2, and updates its memory with that information. Assume that the Connection Manager identifies host1a:port1a as the new primary server, and host2a:port2a as the new alternate server.
5. The driver tries to connect to host1a:port1a.
6. The connection to host1a:port1a fails.
7. The driver tries to connect to host2a:port2a.
8. The connection to host2a:port2a fails.
9. The driver waits two seconds.
10. The driver tries to connect to host1a:port1a.
11. The connection to host1a:port1a fails.
12. The driver tries to connect to host2a:port2a.
13. The connection to host2a:port2a fails.
14. The driver waits two seconds.
15. The driver tries to connect to host1a:port1a.
16. The connection to host1a:port1a fails.
17. The driver tries to connect to host2a:port2a.
18. The connection to host2a:port2a fails.
19. The driver waits two seconds.
20. The driver throws an SQLException with error code -4499.

Operation of workload balancing for connections to IDS from Java clients

Workload balancing (also called transaction-level workload balancing) for connections to IBM Informix Dynamic Server (IDS) 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 IDS Connection Manager, and workload balancing is enabled:

1. When the client first establishes a connection using the IP address of the Connection Manager, the Connection Manager returns the server list and the connection details (IP address, port, and weight) for the servers in the cluster. The server list is cached by the client. The default lifespan of the cached server list is 30 seconds.
2. At the start of a new transaction, the client reads the cached server list to identify a server that has untapped capacity, and looks in the transport pool for an idle transport that is tied to the under-utilized server. (An idle transport is a transport that has no associated connection object.)
 - If an idle transport is available, the client associates the connection object with the transport.
 - If, after a user-configurable timeout, no idle transport is available in the transport pool and no new transport can be allocated because the transport pool has reached its limit, an error is returned to the application.
3. When the transaction runs, it accesses the server that is tied to the transport.
4. When the transaction ends, the client verifies with the server that transport reuse is still allowed for the connection object.
5. If transport reuse is allowed, the server returns a list of SET statements for special registers that apply to the execution environment for the connection object. The client caches these statements, which it replays in order to reconstruct the execution environment when the connection object is associated with a new transport.
6. The connection object is then dissociated from the transport, if the client determines that it needs to do so.
7. The client copy of the server list is refreshed when a new connection is made, or every 30 seconds, or 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 IDS servers

Failover for automatic client reroute can be seamless or non-seamless. If failover for connections to IDS 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 IDS databases:

- The application programming language is Java, CLI, or .NET.
- The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
- The data server must allow transport reuse at the end of the previous transaction.
- All global session data is closed or dropped.
- There are no open held cursors.
- If the application uses CLI, the application cannot perform actions that require the driver to maintain a history of previously called APIs in order to replay the SQL statement. Examples of such actions are specifying data at execution time, performing compound SQL, or using array input.
- The application is not a stored procedure.
- Autocommit is not enabled. Seamless failover can occur when autocommit is enabled. However, the following situation can cause problems: Suppose that SQL work is successfully executed and committed at the data server, but the connection or server goes down before acknowledgment of the commit operation is sent back to the client. When the client re-establishes the connection, it replays the previously committed SQL statement. The result is that the SQL statement is executed twice. To avoid this situation, turn autocommit off when you enable seamless failover.

In addition, seamless automatic client reroute might not be successful if the application has autocommit enabled. With autocommit enabled, a statement might be executed and committed multiple times.

Client affinities for connections to IDS 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.

Configuration of client affinities for Java clients for IDS 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 106. Property settings to enable client affinities for Java applications

IBM Data Server Driver for JDBC and SQLJ setting		Value
enableClientAffinitiesList		DB2BaseDataSource.YES (1)
clientRerouteAlternateServerName		A comma-separated list of the primary server and alternate servers
clientRerouteAlternatePortNumber		A comma-separated list of the port numbers for the primary server and alternate servers
enableSeamlessFailover		DB2BaseDataSource.YES (1) for seamless failover; DB2BaseDataSource.NO (2) or enableSeamlessFailover not specified for no seamless failover
maxRetriesForClientReroute		The number of times to retry the connection to each server, including the primary server, after a connection to the primary server fails. The default is 3.
retryIntervalForClientReroute		The number of seconds to wait between retries. The default is no wait.

Example of enabling client affinities in Java clients for IDS 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.

Suppose that you set the following properties for a connection to a database:

Property	Value
enableClientAffinitiesList	DB2BaseDataSource.YES (1)
clientRerouteAlternateServerName	host1,host2,host3
clientRerouteAlternatePortNumber	port1,port2,port3
maxRetriesForClientReroute	3

Property	Value
retryIntervalForClientReroute	2

Suppose that a communication failure occurs during a connection to the server that is identified by host1:port1. The following steps demonstrate automatic client reroute with client affinities.

1. The driver tries to connect to host1:port1.
2. The connection to host1:port1 fails.
3. The driver waits two seconds.
4. The driver tries to connect to host1:port1.
5. The connection to host1:port1 fails.
6. The driver waits two seconds.
7. The driver tries to connect to host1:port1.
8. The connection to host1:port1 fails.
9. The driver waits two seconds.
10. The driver tries to connect to host2:port2.
11. The connection to host2:port2 fails.
12. The driver waits two seconds.
13. The driver tries to connect to host2:port2.
14. The connection to host2:port2 fails.
15. The driver waits two seconds.
16. The driver tries to connect to host2:port2.
17. The connection to host2:port2 fails.
18. The driver waits two seconds.
19. The driver tries to connect to host3:port3.
20. The connection to host3:port3 fails.
21. The driver waits two seconds.
22. The driver tries to connect to host3:port3.
23. The connection to host3:port3 fails.
24. The driver waits two seconds.
25. The driver tries to connect to host3:port3.
26. The connection to host3:port3 fails.
27. The driver waits two seconds.
28. The driver throws an `SQLException` with error code -4499.

Java client 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), or non-Java clients (ODBC, CLI, .NET, OLE DB, PHP, Ruby, or embedded SQL).

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 107. Data sharing access methods and Sysplex workload balancing

Data sharing access method¹	Description	Sysplex workload balancing possible?
Group access	<p>A requester uses the group's dynamic virtual IP address (DVIPA) to make an initial connection to the DB2 for z/OS location. A connection to the data sharing group that uses the group IP address and SQL port is always successful if at least one member is started. The server list that is returned by the data sharing group contains:</p> <ul style="list-style-type: none"> • A list of members that are currently active and can perform work • The WLM weight for each member <p>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.</p>	Yes

Table 107. Data sharing access methods and Sysplex workload balancing (continued)

Data sharing access method ¹	Description	Sysplex workload balancing possible?
Member-specific access	<p>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:</p> <ul style="list-style-type: none"> • A list of members that are currently active, can perform work, and have been configured as an alias • The WLM weight for each member <p>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.</p>	Yes
Single-member access	Single-member access is used when requesters need to access only one member of a data sharing group. For single-member access, the connection uses the member-specific IP address.	No
Note:		
1. For more information on data sharing access methods, see http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db29.doc.dshare/db2z_tcpipaccessmethods.htm .		

Sysplex workload balancing includes automatic client reroute: Automatic client reroute support enables a client to recover from a failure by attempting to reconnect to the database through any available member of a Sysplex. Reconnection to another member is called *failover*.

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 support for high availability is incompatible with a DB2 Connect environment: Client support for high availability requires a DB2 Connect license. However, this support is incompatible with a DB2 Connect environment. Do not use a DB2 Connect server when you use client support for high availability.

Do not use client affinities: Client affinities should not be used as a high availability solution for direct connections to DB2 for z/OS. Client affinities is not applicable to a DB2 for z/OS data sharing environment, because all members of a data sharing group can access data concurrently. A major disadvantage of client affinities in a data sharing environment is that if failover occurs because a data sharing group member fails, the member that fails might have retained locks that can severely affect transactions on the member to which failover occurs.

Configuration of Sysplex workload balancing at a Java client

To configure a IBM Data Server Driver for JDBC and SQLJ client application that connects directly to DB2 for z/OS to use Sysplex workload balancing, you need to use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity. You also need to connect to an address that represents the data sharing group (for group access) or a subset of the data sharing group (for member-specific access), and set the properties that enable workload balancing and the maximum number of connections.

The following table describes the basic property settings for Java applications.

Table 108. Basic settings to enable Sysplex high availability support in Java applications

Data sharing access method	IBM Data Server Driver for JDBC and SQLJ setting	Value
Group access	enableSysplexWLB property	true
	Connection address:	
	server	The group IP address or domain name of the data sharing group
	port	The SQL port number for the DB2 location
	database	The DB2 location name that is defined during installation
Member-specific access	enableSysplexWLB property	true
	Connection address:	
	server	The group IP address or domain name of the data sharing group
	port	The port number for the DB2 location alias
	database	The name of the DB2 location alias that represents a subset of the members of the data sharing group

If you want to fine-tune Sysplex workload balancing, additional properties are available.

The following IBM Data Server Driver for JDBC and SQLJ Connection or DataSource properties control Sysplex workload balancing.

Table 109. Connection or DataSource properties for fine-tuning Sysplex workload balancing for direct connections from the IBM Data Server Driver for JDBC and SQLJ to DB2 for z/OS

IBM Data Server Driver for JDBC and SQLJ property	Description
blockingReadConnectionTimeout	Specifies the amount of time in seconds before a connection socket read times out. This property affects all requests that are sent to the data source after a connection is successfully established. The default is 0, which means that there is no timeout. Set this property to a value greater by a few seconds than the time that is required to execute the longest query in the application.

Table 109. Connection or DataSource properties for fine-tuning Sysplex workload balancing for direct connections from the IBM Data Server Driver for JDBC and SQLJ to DB2 for z/OS (continued)

IBM Data Server Driver for JDBC and SQLJ property	Description
loginTimeout	Specifies the maximum time in seconds to wait for a new connection to a data source. After the number of seconds that are specified by loginTimeout have elapsed, the driver closes the connection to the data source. The default is 0, which means that the timeout value is the default system timeout value. The recommended value is five seconds.
maxTransportObjects	Specifies the maximum number of connections that the requester can make to the data sharing group. The default is -1, which means an unlimited number.

The following IBM Data Server Driver for JDBC and SQLJ configuration properties also control Sysplex workload balancing.

Table 110. Configuration properties for fine-tuning Sysplex workload balancing for direct connections from the IBM Data Server Driver for JDBC and SQLJ to DB2 for z/OS

IBM Data Server Driver for JDBC and SQLJ configuration property	Description
db2.jcc.maxTransportObjectIdleTime	Specifies the maximum elapsed time in number of seconds before an idle transport is dropped. The default is 60. The minimum supported value is 0.
db2.jcc.maxTransportObjectWaitTime	Specifies the number of seconds that the client will wait for a transport to become available. The default is -1 (unlimited). The minimum supported value is 0.
db2.jcc.minTransportObjects	Specifies the lower limit for the number of transport objects in a global transport object pool. The default value is 0. Any value that is less than or equal to 0 means that the global transport object pool can become empty.

Example of enabling DB2 for z/OS Sysplex workload balancing in Java applications

Java client setup for Sysplex workload balancing includes setting several IBM Data Server Driver for JDBC and SQLJ properties.

The following examples demonstrate setting up Java client applications for Sysplex workload balancing for high availability.

Before you can set up the client, you need to configure the following server software:

- WLM for z/OS

For workload balancing to work efficiently, DB2 work needs to be classified. Classification applies to the first non-SET SQL statement in each transaction. Among the areas by which you need to classify the work are:

- Authorization ID
- Client info properties
- Stored procedure name

The stored procedure name is used for classification only if the first statement that is issued by the client in the transaction is an SQL CALL statement.

For a complete list of classification attributes, see the information on classification attributes at the following URL:

http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db29.doc.perf/db2z_classificationattributes.htm

- DB2 for z/OS, set up for data sharing

Example of setup with WebSphere Application Server

This example assumes that you are using WebSphere Application Server. The minimum level of WebSphere Application Server is Version 5.1.

Follow these steps to set up the client:

1. Verify that the IBM Data Server Driver for JDBC and SQLJ is at the correct level to support the Sysplex workload balancing by following these steps:

- a. Issue the following command 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.

```
[jcc] Driver: IBM Data Server Driver for JDBC and SQLJ Architecture nnn xxx
```

2. Set the IBM Data Server Driver for JDBC and SQLJ data source property `enableSysplexWLB` to enable the Sysplex workload balancing.

In the WebSphere Application Server administrative console, set the following properties for the DataSource that your application uses to connect to the data source. Start with settings similar to these:

Table 111. Example of data source property settings for IBM Data Server Driver for JDBC and SQLJ Sysplex workload balancing for DB2 for z/OS

Property	Setting
<code>enableSysplexWLB</code>	<code>true</code>
<code>maxTransportObjects</code>	<code>80</code>

Use property `maxTransportObjects` to limit the total number of connections to the DB2 for z/OS data sharing group.

Recommendation: Set `maxTransportObjects` to a value that is larger than the `MaxConnections` value for the WebSphere Application Server connection pool. Doing so allows workload balancing to occur between data sharing members without the need to open and close connections to DB2.

3. Set IBM Data Server Driver for JDBC and SQLJ configuration properties to fine-tune workload balancing for all DataSource or Connection instances that are created under the driver. Set the configuration properties in a `DB2JccConfiguration.properties` file by following these steps:

- a. Create a `DB2JccConfiguration.properties` file or edit the existing `DB2JccConfiguration.properties` file.

- b. Set the `db2.jcc.maxTransportObjects` configuration property only if multiple DataSource objects are defined that point to the same data sharing group, and the number of connections across the different DataSource objects needs to be limited.

Start with a setting similar to this one:

```
db2.jcc.maxTransportObjects=500
```

- c. Add the directory path for `DB2JccConfiguration.properties` to the WebSphere Application Server IBM Data Server Driver for JDBC and SQLJ classpath.

- d. Restart WebSphere Application Server.

Example of setup for DriverManager connections

This example assumes that you are using the DriverManager interface to establish a connection.

Follow these steps to set up the client:

1. Verify that the IBM Data Server Driver for JDBC and SQLJ is at the correct level to support the Sysplex workload balancing by following these steps:

- a. Issue the following command 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 for DriverManager connections.

- c.

```
[jcc] Driver: IBM Data Server Driver for JDBC and SQLJ Architecture nnn xxx
```

2. Set the IBM Data Server Driver for JDBC and SQLJ Connection property `enableSysplexWLB` to enable workload balancing. You can also use property `maxTransportObjects` to limit the total number of connections to the DB2 for z/OS data sharing group.

```
java.util.Properties properties = new java.util.Properties();
properties.put("user", "xxxx");
properties.put("password", "yyyy");
properties.put("enableSysplexWLB", "true");
properties.put("maxTransportObjects", "80");
java.sql.Connection con =
    java.sql.DriverManager.getConnection(url, properties);
```

3. Set IBM Data Server Driver for JDBC and SQLJ configuration properties to fine-tune workload balancing for all DataSource or Connection instances that are created under the driver. Set the configuration properties in a `DB2JccConfiguration.properties` file by following these steps:

- a. Create a `DB2JccConfiguration.properties` file or edit the existing `DB2JccConfiguration.properties` file.

- b. Set the `db2.jcc.maxTransportObjects` configuration property only if multiple DataSource objects are defined that point to the same data sharing group, and the number of connections across the different DataSource objects needs to be limited.

Start with a setting similar to this one:

```
db2.jcc.maxTransportObjects=500
```

- c. Include the directory that contains `DB2JccConfiguration.properties` in the CLASSPATH concatenation.

Operation of Sysplex workload balancing for connections from Java clients to DB2 for z/OS servers

Sysplex workload balancing (also called transaction-level workload balancing) for connections to DB2 for z/OS contributes to high availability by balancing work among members of a data sharing group at the start of a transaction.

The following overview describes the steps that occur when a client connects to a DB2 for z/OS Sysplex, and Sysplex workload balancing is enabled:

1. When the client first establishes a connection using the sysplex-wide IP address called the group IP address, or when a connection is reused by another connection object, the server returns member workload distribution information.

The default lifespan of the cached server list is 30 seconds.

2. At the start of a new transaction, the client reads the cached server list to identify a member that has untapped capacity, and looks in the transport pool for an idle transport that is tied to the under-utilized member. (An idle transport is a transport that has no associated connection object.)
 - If an idle transport is available, the client associates the connection object with the transport.
 - If, after a user-configurable timeout, no idle transport is available in the transport pool and no new transport can be allocated because the transport pool has reached its limit, an error is returned to the application.
3. When the transaction runs, it accesses the member that is tied to the transport.
4. When the transaction ends, the client verifies with the server that transport reuse is still allowed for the connection object.
5. If transport reuse is allowed, the server returns a list of SET statements for special registers that apply to the execution environment for the connection object.

The client caches these statements, which it replays in order to reconstruct the execution environment when the connection object is associated with a new transport.

6. The connection object is then disassociated from the transport.
7. The client copy of the server list is refreshed when a new connection is made, or every 30 seconds.
8. When workload balancing is required for a new transaction, the client uses the same process to associate the connection object with a transport.

Operation of automatic client reroute for connections from Java clients to DB2 for z/OS

Automatic client reroute support provides failover support when an IBM data server client loses connectivity to a member of a DB2 for z/OS Sysplex. Automatic client reroute enables the client to recover from a failure by attempting to reconnect to the database through any available member of the Sysplex.

Automatic client reroute is enabled by default when Sysplex workload balancing is enabled.

Client support for automatic client reroute is available in IBM data server clients that have a DB2 Connect license. The DB2 Connect server is not required to perform automatic client reroute.

Automatic client reroute for connections to DB2 for z/OS operates in the following way:

1. As part of the response to a COMMIT request from the client, the data server returns:
 - An indicator that specifies whether transports can be reused. Transports can be reused if there are no resources remaining, such as held cursors.
 - SET statements that the client can use to replay the connection state during transport reuse.

2. If the first SQL statement in a transaction fails, and transports can be reused:
 - No error is reported to the application.
 - The failing SQL statement is executed again.
 - The SET statements that are associated with the logical connection are replayed to restore the connection state.
3. If an SQL statement that is not the first SQL statement in a transaction fails, and transports can be reused:
 - The transaction is rolled back.
 - The application is reconnected to the data server.
 - The SET statements that are associated with the logical connection are replayed to restore the connection state.
 - SQL error -30108 (for Java) or SQL30108N (for non-Java clients) is returned to the application to notify it of the rollback and successful reconnection. The application needs to include code to retry the failed transaction.
4. If an SQL statement that is not the first SQL statement in a transaction fails, and transports cannot be reused:
 - The logical connection is returned to its initial, default state.
 - SQL error -30081 (for Java) or SQL30081N (for non-Java clients) is returned to the application to notify it that reconnection was unsuccessful. The application needs to reconnect to the data server, reestablish the connection state, and retry the failed transaction.
5. If connections to all members of the data sharing member list have been tried, and none have succeeded, a connection is tried using the URL that is associated with the data sharing group, to determine whether any members are now available.

Application programming requirements for high availability for connections from Java clients to DB2 for z/OS servers

Failover for automatic client reroute can be seamless or non-seamless. If failover for connections to DB2 for z/OS is not seamless, you need to add code to account for the errors that are returned when failover occurs.

If failover is not seamless, and a connection is reestablished with the server, SQLCODE -30108 (SQL30108N) is returned to the application. All work that occurred within the current transaction is rolled back. In the application, you need to:

- Check the reason code that is returned with the -30108 error to determine whether special register settings on the failing data sharing member are carried over to the new (failover) data sharing member. Reset any special register values that are not current.
- Execute all SQL operations that occurred since the previous commit operation.

The following conditions must be satisfied for seamless failover to occur for direct connections to DB2 for z/OS:

- The application language is Java, CLI, or .NET.
- The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
- The data server allows transport reuse at the end of the previous transaction. An exception to this condition is if transport reuse is not granted because the application was bound with KEEP_DYNAMIC(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.

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 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 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 reference

“IBM Data Server Driver for JDBC and SQLJ properties for DB2 for z/OS” on page 233

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 DB2 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 source, DB2 creates a new physical connection to the data source. When the application calls the `java.sql.Connection.close` method, DB2 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.

Related reference

"DB2ConnectionPoolDataSource class" on page 350

"DB2PooledConnection class" on page 356

Chapter 13. 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 for Linux, UNIX, and Windows, and DB2 for 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 112. Components of a distributed transaction system on DB2 for z/OS

Distributed transaction system component	Component function provided by
Resource manager	DB2 for z/OS or DB2 for Linux, UNIX, and Windows
Transaction manager	WebSphere Application Server or another application server
Transactional applications	JDBC or SQLJ applications

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 for Linux, UNIX, and Windows servers. DB2 UDB for OS/390 and z/OS Version 7 servers do not have native XA mode support, so the IBM Data Server Driver for JDBC and SQLJ emulates the XA mode support using the existing DB2 DRDA two-phase commit protocol. This XA mode emulation uses a DB2 table named SYSIBM.INDOUBT to store information about indoubt transactions. DB2 uses a package named T4XAIndbtPkg to perform SQL operations on SYSIBM.INDOUBT.

If your JDBC or SQLJ applications use distributed transactions, and those applications connect to DB2 UDB for OS/390 and z/OS Version 7 servers, you need to run the DB2T4XAIndoubtUtil utility at those servers to create the SYSIBM.INDOUBT table and the T4XAIndbtPkg package. You should never modify the SYSIBM.INDOUBT table manually.

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 14, “JDBC and SQLJ global transaction support,” on page 535

Related reference

“DB2T4XAIndoubtUtil for distributed transactions with DB2 UDB for OS/390 and z/OS Version 7 servers” on page 454

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 51 outlines code that executes local transactions.

```
con1.setAutoCommit(false); // Set autocommit off
// execute some SQL
...
con1.commit();             // Commit the transaction
// execute some more SQL
...
con1.rollback();           // Roll back the transaction
con1.setAutoCommit(true); // Enable commit after every SQL statement
...
// Execute some more SQL, which is automatically committed after
// every SQL statement.
```

Figure 51. 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 52 on page 531 demonstrates an application that uses distributed transactions. While the code in the example is running, the application server is also executing other EJBs that are part of this same distributed transaction. When all EJBs have called `utx.commit()`, the entire distributed transaction is committed by the application server. If any of the EJBs are unsuccessful, the application server rolls back all the work done by all EJBs that are associated with the distributed transaction.

```

javax.transaction.UserTransaction utx;
// Use the begin method on a UserTransaction object to indicate
// the beginning of a distributed transaction.
utx.begin();
...
// Execute some SQL with one Connection object.
// Do not call Connection methods commit or rollback.
...
// Use the commit method on the UserTransaction object to
// drive all transaction branches to commit and indicate
// the end of the distributed transaction.

utx.commit();
...

```

Figure 52. Example of a distributed transaction under an application server

Figure 53 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 53. Example of a distributed transaction that uses the JTA

```

class XASample
{
    javax.sql.XADataSource xaDS1;
    javax.sql.XADataSource xaDS2;
    javax.sql.XAConnection xaconn1;
    javax.sql.XAConnection xaconn2;
    javax.transaction.xa.XAResource xares1;
    javax.transaction.xa.XAResource xares2;
    java.sql.Connection conn1;
    java.sql.Connection conn2;

    public static void main (String args []) throws java.sql.SQLException
    {
        XASample xat = new XASample();
        xat.runThis(args);
    }
    // As the transaction manager, this program supplies the global
    // transaction ID and the branch qualifier. The global
    // transaction ID and the branch qualifier must not be
    // equal to each other, and the combination must be unique for
    // this transaction manager.
    public void runThis(String[] args)
    {
        byte[] gtrid = new byte[] { 0x44, 0x11, 0x55, 0x66 };
        byte[] bqqual = 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
xaes1 = xaconn1.getXAResource();
xaes2 = 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.
xaes1.start(xid1, javax.transaction.xa.XAResource.TMNOFLAGS);
xaes2.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.
xaes1.end(xid1, javax.transaction.xa.XAResource.TMSUCCESS);
xaes2.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 = xaes1.prepare(xid1);
    if(rc1 == javax.transaction.xa.XAResource.XA_OK)
    {
        // Prepare was successful. Prepare the second connection.
        rc2 = xaes2.prepare(xid1);
        if(rc2 == javax.transaction.xa.XAResource.XA_OK)
        {
            // Both connections prepared successfully and neither was read-only.
            xaes1.commit(xid1, false);
            xaes2.commit(xid1, false);
        }
        else if(rc2 == javax.transaction.xa.XAException.XA_RDONLY)
        {
            // The second connection is read-only, so just commit the
            // first connection.
            xaes1.commit(xid1, false);
        }
    }
}
else if(rc1 == javax.transaction.xa.XAException.XA_RDONLY)
{
    // SQL for the first connection is read-only (such as a SELECT).
    // The prepare committed it. Prepare the second connection.
    rc2 = xaes2.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.
        xaes2.commit(xid1, false);
    }
    else if(rc2 == javax.transaction.xa.XAException.XA_RDONLY)
    {
        // Both connections are read-only, and both already committed,
        // so there is nothing more to do.
    }
}

```

```

    }
    catch (javax.transaction.xa.XAException xae)
    { // Distributed transaction failed, so roll it back.
      // Report XAException on prepare/commit.
      System.out.println("Distributed transaction prepare/commit failed. " +
        "Rolling it back.");
      System.out.println("XAException error code = " + xae.errorCode);
      System.out.println("XAException message = " + xae.getMessage());
      xae.printStackTrace();
      try
      {
        xares1.rollback(xid1);
      }
      catch (javax.transaction.xa.XAException xae1)
      { // Report failure of rollback.
        System.out.println("distributed Transaction rollback xares1 failed");
        System.out.println("XAException error code = " + xae1.errorCode);
        System.out.println("XAException message = " + xae1.getMessage());
      }
      try
      {
        xares2.rollback(xid1);
      }
      catch (javax.transaction.xa.XAException xae2)
      { // Report failure of rollback.
        System.out.println("distributed Transaction rollback xares2 failed");
        System.out.println("XAException error code = " + xae2.errorCode);
        System.out.println("XAException message = " + xae2.getMessage());
      }
    }
  }
  try
  {
    conn1.close();
    xaconn1.close();
  }
  catch (Exception e)
  {
    System.out.println("Failed to close connection 1: " + e.toString());
    e.printStackTrace();
  }
  try
  {
    conn2.close();
    xaconn2.close();
  }
  catch (Exception e)
  {
    System.out.println("Failed to close connection 2: " + e.toString());
    e.printStackTrace();
  }
}
catch (java.sql.SQLException sqe)
{
  System.out.println("SQLException caught: " + sqe.getMessage());
  sqe.printStackTrace();
}
catch (javax.transaction.xa.XAException xae)
{
  System.out.println("XA error is " + xae.getMessage());
  xae.printStackTrace();
}
catch (javax.naming.NamingException nme)
{

```



```
        System.out.println(" Naming Exception: " + nme.getMessage());
    }
}
```

Recommendation: For better performance, complete a distributed transaction before you start another distributed or local transaction.

Related reference

“DB2XADataSource class” on page 389

Chapter 14. JDBC and SQLJ global transaction support

JDBC and SQLJ global transaction support lets Enterprise Java Beans (EJB) and Java servlets that run under WebSphere Application Server 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 for Linux, UNIX, and Windows.

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. As the following code shows, 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.

```
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
...
```

Figure 54. Example of a local transaction

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.

```

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();
...

```

Figure 55. Example of a global transaction

When you run a multi-threaded client under WebSphere, a transaction can span multiple threads. This situation might occur in a Java servlet. An application that runs in this environment needs to perform some SQL on each Connection object before the application passes the object to another thread. The following code illustrates this point.

```

javax.transaction.UserTransaction utx;
// Use the begin method on a UserTransaction object to indicate
// the beginning of a global transaction.
utx.begin();

...
// Obtain two JDBC Connections from DataSource ds
c1 = ds.getConnection();
c2 = ds.getConnection();

...
// Create a thread for each Connection object
ThreadClass1 tc1 = new ThreadClass1(c1);
ThreadClass2 tc2 = new ThreadClass1(c2);
Thread t1 = new Thread(tc1);
Thread t2 = new Thread(tc2);
// Execute some SQL on each Connection object to associate
// the threads with the global transaction

...
// Start the two threads that will use the Connection objects to do SQL
t1.start();
t2.start();

...
// 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();
...

```

Figure 56. Example of a global transaction in a multi-threaded environment

Related concepts

Chapter 13, “IBM Data Server Driver for JDBC and SQLJ type 4 connectivity JDBC and SQLJ distributed transaction support,” on page 529

Chapter 15. Problem diagnosis with the IBM Data Server Driver for JDBC and SQLJ

To obtain data for diagnosing SQLJ or JDBC problems with the IBM Data Server Driver for JDBC and SQLJ, collect trace data and run utilities that format the trace data.

You should run the trace and diagnostic utilities only under the direction of IBM software support.

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 "Properties for the IBM Data Server Driver for JDBC and SQLJ" for information on how to specify more than one type of tracing.
2. Invoke the `DB2BaseDataSource.setJccLogWriter` method to specify the trace destination and turn the trace on.

Procedure 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 "Properties for the IBM Data Server Driver for JDBC and SQLJ" for information on how to specify more than one type of tracing.

2. Invoke the `DriverManager.setLogWriter` method to specify the trace destination and turn the trace on.

After a connection is established, you can turn the trace off or back on, change the trace destination, or change the trace level with the `DB2Connection.setJccLogWriter` method. To turn the trace off, set the `logWriter` value to `null`.

The `logWriter` property is an object of type `java.io.PrintWriter`. If your application cannot handle `java.io.PrintWriter` objects, you can use the `traceFile` property to specify the destination of the trace output. To use the `traceFile` property, set the `logWriter` property to `null`, and set the `traceFile` property to the name of the file to which the driver writes the trace data. This file and the directory in which it resides must be writable. If the file already exists, the driver overwrites it.

Procedure 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 "Example of using configuration properties to start a JDBC trace".

Trace example program: For a complete example of a program for tracing under the IBM Data Server Driver for JDBC and SQLJ, see "Example of a trace program under the IBM Data Server Driver for JDBC and SQLJ".

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` 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

“DB2Connection interface” on page 333

“`db2sqljprint` - SQLJ profile printer” on page 437

Example of using configuration properties to start a JDBC trace

You can control tracing of JDBC applications without modifying those applications.

Suppose that you want to collect trace data for a program named `Test.java`, which uses IBM Data Server Driver for JDBC and SQLJ type 4 connectivity. `Test.java` does no tracing, and you do not want to modify the program, so you enable tracing using configuration properties. You want your trace output to have the following characteristics:

- Trace information for each connection on the same `DataSource` is written to a separate trace file. Output goes into a directory named `/Trace`.
- Each trace file name begins with `jccTrace1`.
- If trace files with the same names already exist, the trace data is appended to them.

Although `Test1.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.override.traceDirectory=/Trace`
- `db2.jcc.traceFile=jccTrace1`
- `db2.jcc.traceFileAppend=true`

You want the trace settings to apply only to your stand-alone program `Test1.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 `Test1.java`, you issue a command like this:

```
java -Ddb2.jcc.propertiesFile=/Test/jcc.properties Test1
```

Suppose that `Test1.java` creates two connections for one `DataSource`. The program does not define a `logWriter` object, so the driver creates a global `logWriter` object for the trace output. When the program completes, the following files contain the trace data:

- `/Trace/jccTrace1_global_0`
- `/Trace/jccTrace1_global_1`

Example of a trace program under the IBM Data Server Driver for JDBC and SQLJ

You might want to write a single class that includes methods for tracing under the DriverManager interface, as well as the DataSource interface.

The following example shows such a class. The example uses IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Figure 57. Example of tracing under the IBM Data Server Driver for JDBC and SQLJ

```
public class TraceExample
{
    public static void main(String[] args)
    {
        sampleConnectUsingSimpleDataSource();
        sampleConnectWithURLUsingDriverManager();
    }

    private static void sampleConnectUsingSimpleDataSource()
    {
        java.sql.Connection c = null;
        java.io.PrintWriter printWriter =
            new java.io.PrintWriter(System.out, true);
                                // Prints to console, true means
                                // auto-flush so you don't lose trace
        try {
            javax.sql.DataSource ds =
                new com.ibm.db2.jcc.DB2SimpleDataSource();
            ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setServerName("sysmvsl.stl.ibm.com");
            ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setPortNumber(5021);
            ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setDatabaseName("san_jose");
            ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setDriverType(4);

            ds.setLogWriter(printWriter);    // This turns on tracing

            // Refine the level of tracing detail
            ((com.ibm.db2.jcc.DB2BaseDataSource) ds).
                setTraceLevel(com.ibm.db2.jcc.DB2SimpleDataSource.TRACE_CONNECTS |
                    com.ibm.db2.jcc.DB2SimpleDataSource.TRACE_DRDA_FLOWS);

            // This connection request is traced using trace level
            // TRACE_CONNECTS | TRACE_DRDA_FLOWS
            c = ds.getConnection("myname", "mypass");

            // Change the trace level to TRACE_ALL
            // for all subsequent requests on the connection
            ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(printWriter,
                com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL);
            // The following INSERT is traced using trace level TRACE_ALL
            java.sql.Statement s1 = c.createStatement();
            s1.executeUpdate("INSERT INTO sampleTable(sampleColumn) VALUES(1)");
            s1.close();

            // This code disables all tracing on the connection
            ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(null);

            // The following INSERT statement is not traced
            java.sql.Statement s2 = c.createStatement();
            s2.executeUpdate("INSERT INTO sampleTable(sampleColumn) VALUES(1)");
            s2.close();

            c.close();
        }
    }
}
```

```

    }
    catch(java.sql.SQLException e) {
        com.ibm.db2.jcc.DB2ExceptionFormatter.printStackTrace(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.printStackTrace(e, printWriter,
                "[TraceExample]");
            printWriter.println("[TraceExample] " +
                "Unable to roll back the connection");
        }
        catch(java.lang.Throwable e) {
            printWriter.println("[TraceExample] Trapped the " +
                "following java.lang.Throwable while trying to roll back:");
            com.ibm.db2.jcc.DB2ExceptionFormatter.printStackTrace(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.printStackTrace(e, printWriter,
                "[TraceExample]");
        }
        catch(java.lang.Throwable e) {
            printWriter.println("[TraceExample] Throwable caught " +
                "while trying to close the connection");
        }
    }
}

```



```

        printWriter.println("[TraceExample] Deadlocks could " +
            "occur if the connection is not closed.");
        com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e, printWriter,
            "[TraceExample]");
    }
}
catch(java.lang.Throwable e) {
    printWriter.println("[TraceExample] Unable to " +
        "force the connection to close");
    printWriter.println("[TraceExample] Deadlocks " +
        "could occur if the connection is not closed.");
    com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e, printWriter,
        "[TraceExample]");
}
}
private static void sampleConnectWithURLUsingDriverManager()
{
    java.sql.Connection c = null;

    // This time, send the printWriter to a file.
    java.io.PrintWriter printWriter = null;
    try {
        printWriter =
            new java.io.PrintWriter(
                new java.io.BufferedOutputStream(
                    new java.io.FileOutputStream("/temp/driverLog.txt"), 4096), true);
    }
    catch(java.io.FileNotFoundException e) {
        java.lang.System.err.println("Unable to establish a print writer for trace");
        java.lang.System.err.flush();
        return;
    }

    try {
        Class.forName("com.ibm.db2.jcc.DB2Driver");
    }
    catch(ClassNotFoundException e) {
        printWriter.println("[TraceExample] " +
            "IBM Data Server Driver for JDBC and SQLJ type 4 connectivity " +
            "is not in the application classpath. Unable to load driver.");
        printWriter.flush();
        return;
    }

    // This URL describes the target data source for Type 4 connectivity.
    // The traceLevel property is established through the URL syntax,
    // and driver tracing is directed to file "/temp/driverLog.txt"
    // The traceLevel property has type int. The constants
    // com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS and
    // com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS represent
    // int values. Those constants cannot be used directly in the
    // first getConnection parameter. Resolve the constants to their
    // int values by assigning them to a variable. Then use the
    // variable as the first parameter of the getConnection method.
    String databaseURL =
        "jdbc: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();
}
}
}

```

Related reference

“DB2ExceptionFormatter class” on page 353

Techniques for monitoring IBM Data Server Driver for JDBC and SQLJ Sysplex support

To monitor IBM Data Server Driver for JDBC and SQLJ Sysplex support, you need to monitor the global transport objects pool.

You can monitor the global transport objects pool in either of the following ways:

- Using traces that you start by setting IBM Data Server Driver for JDBC and SQLJ configuration properties
- Using an application programming interface

Configuration properties for monitoring the global transport objects pool

The `db2.jcc.dumpPool`, `db2.jcc.dumpPoolStatisticsOnSchedule`, and `db2.jcc.dumpPoolStatisticsOnScheduleFile` configuration properties control tracing of the global transport objects pool.

For example, the following set of configuration property settings cause error messages and dump pool error messages to be written every 60 seconds to a file named `/home/WAS/logs/srv1/poolstats`:

```

db2.jcc.dumpPool=DUMP_SYSPLEX_MSG|DUMP_POOL_ERROR
db2.jcc.dumpPoolStatisticsOnSchedule=60
db2.jcc.dumpPoolStatisticsOnScheduleFile=/home/WAS/logs/srv1/poolstats

```

An entry in the pool statistics file looks like this:

```
time Scheduled PoolStatistics npr:2575 nsr:2575 lwroc:439 hwroc:1764 coc:372  
aoc:362 rmoc:362 nbr:2872 tbt:857520 tpo:10
```

The meanings of the fields are:

npr

The total number of requests that the IBM Data Server Driver for JDBC and SQLJ has made to the pool since the pool was created.

nsr

The number of successful requests that the IBM Data Server Driver for JDBC and SQLJ has made to the pool since the pool was created. A successful request means that the pool returned an object.

lwroc

The number of objects that were reused but were not in the pool. This can happen if a Connection object releases a transport object at a transaction boundary. If the Connection object needs a transport object later, and the original transport object has not been used by any other Connection object, the Connection object can use that transport object.

hwroc

The number of objects that were reused from the pool.

coc

The number of objects that the IBM Data Server Driver for JDBC and SQLJ created since the pool was created.

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.

Application programming interfaces for monitoring the global transport objects pool

You can write applications to gather statistics on the global transport objects pool. Those applications create objects in the `DB2PoolMonitor` class and invoke methods to retrieve information about the pool.

For example, the following code creates an object for monitoring the global transport objects pool:

```
import com.ibm.db2.jcc.DB2PoolMonitor;
DB2PoolMonitor transportObjectPoolMonitor =
    DB2PoolMonitor.getPoolMonitor (DB2PoolMonitor.TRANSPORT_OBJECT);
```

After you create the `DB2PoolMonitor` object, you can use methods in the `DB2PoolMonitor` class to monitor the pool.

Chapter 16. 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.

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.

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 442

Related reference

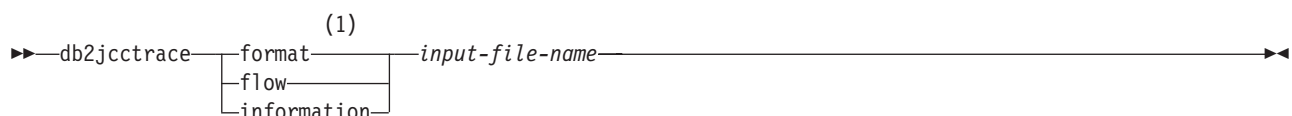
“`db2jcctrace` - Format IBM Data Server Driver for JDBC and SQLJ trace data for C/C++ native driver code”

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



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 `db2.jcc.t2zosTraceFile` global configuration parameter.

Related concepts

“Customization of IBM Data Server Driver for JDBC and SQLJ configuration properties” on page 442

Related tasks

Chapter 16, “Tracing IBM Data Server Driver for JDBC and SQLJ C/C++ native driver code,” on page 547

Chapter 17. System monitoring for the IBM Data Server Driver for JDBC and SQLJ

To assist you in monitoring the performance of your applications with the IBM Data Server Driver for JDBC and SQLJ, the driver provides two methods to collect information for a connection.

That information is:

Core driver time

The sum of elapsed monitored API times that were collected while system monitoring was enabled, in microseconds. In general, only APIs that might result in network I/O or database server interaction are monitored.

Network I/O time

The sum of elapsed network I/O times that were collected while system monitoring was enabled, in microseconds.

Server time

The sum of all reported database server elapsed times that were collected while system monitoring was enabled, in microseconds.

Currently, IBM Informix Dynamic Server databases do not support this function.

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.

For example, the following code demonstrates how to collect each type of elapsed time data. The numbers to the right of selected statements correspond to the previously described steps.


```

import java.sql.*;
import com.ibm.db2.jcc.*;
public class TestSystemMonitor
{
    public static void main(String[] args)
    {
        String url = "jdbc: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 58. Example of using DB2SystemMonitor methods to collect system monitoring data

To collect system monitoring information using the trace method: Start a JDBC trace, using configuration properties or Connection or DataSource properties. Include TRACE_SYSTEM_MONITOR when you set the traceLevel property. For example:

```

String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose" +
    ":traceFile=/u/db2p/jcctrace;" +
    "traceLevel=" + com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SYSTEM_MONITOR + ";";

```

The trace records with system monitor information look similar to this:

```
[jcc][SystemMonitor:start]
```

```
...
```

```
[jcc][SystemMonitor:stop] core: 565.67ms | network: 211.695ms | server: 207.771ms
```

Related reference

“DB2SystemMonitor interface” on page 379

Information resources for DB2 for z/OS and related products

Many information resources are available to help you use DB2 for z/OS and many related products. A large amount of technical information about IBM products is now available online in information centers or on library Web sites.

Disclaimer: Any Web addresses that are included here are accurate at the time this information is being published. However, Web addresses sometimes change. If you visit a Web address that is listed here but that is no longer valid, you can try to find the current Web address for the product information that you are looking for at either of the following sites:

- <http://www.ibm.com/support/publications/us/library/index.shtml>, which lists the IBM information centers that are available for various IBM products
- <http://www.elink.ibm.link.ibm.com/public/applications/publications/cgibin/pbi.cgi>, which is the IBM Publications Center, where you can download online PDF books or order printed books for various IBM products

DB2 for z/OS product information

The primary place to find and use information about DB2 for z/OS is the Information Management Software for z/OS Solutions Information Center (<http://publib.boulder.ibm.com/infocenter/imzic>), which also contains information about IMS, QMF™, and many DB2 and IMS Tools products. This information center is also available as an installable information center that can run on a local system or on an intranet server. You can order the Information Management for z/OS Solutions Information Center DVD (SK5T-7377) for a low cost from the IBM Publications Center (www.ibm.com/shop/publications/order).

The majority of the DB2 for z/OS information in this information center is also available in the books that are identified in the following table. You can access these books at the DB2 for z/OS library Web site (<http://www.ibm.com/software/data/db2/zos/library.html>) or at the IBM Publications Center (<http://www.ibm.com/shop/publications/order>).

Table 113. DB2 Version 9.1 for z/OS book titles

Title	Publication number	Available in information center	Available in PDF	Available in BookManager® format	Available in printed book
<i>DB2 Version 9.1 for z/OS Administration Guide</i>	SC18-9840	X	X	X	X
<i>DB2 Version 9.1 for z/OS Application Programming & SQL Guide</i>	SC18-9841	X	X	X	X
<i>DB2 Version 9.1 for z/OS Application Programming Guide and Reference for Java</i>	SC18-9842	X	X	X	X
<i>DB2 Version 9.1 for z/OS Codes</i>	GC18-9843	X	X	X	X
<i>DB2 Version 9.1 for z/OS Command Reference</i>	SC18-9844	X	X	X	X
<i>DB2 Version 9.1 for z/OS Data Sharing: Planning and Administration</i>	SC18-9845	X	X	X	X

Table 113. DB2 Version 9.1 for z/OS book titles (continued)

Title	Publication number	Available in information center	Available in PDF	Available in BookManager® format	Available in printed book
<i>DB2 Version 9.1 for z/OS Diagnosis Guide and Reference</i> ¹	LY37-3218		X	X	X
<i>DB2 Version 9.1 for z/OS Diagnostic Quick Reference</i>	LY37-3219				X
<i>DB2 Version 9.1 for z/OS Installation Guide</i>	GC18-9846	X	X	X	X
<i>DB2 Version 9.1 for z/OS Introduction to DB2</i>	SC18-9847	X	X	X	X
<i>DB2 Version 9.1 for z/OS Licensed Program Specifications</i>	GC18-9848		X		X
<i>DB2 Version 9.1 for z/OS Messages</i>	GC18-9849	X	X	X	X
<i>DB2 Version 9.1 for z/OS ODBC Guide and Reference</i>	SC18-9850	X	X	X	X
<i>DB2 Version 9.1 for z/OS Performance Monitoring and Tuning Guide</i>	SC18-9851	X	X	X	X
<i>DB2 Version 9.1 for z/OS Optimization Service Center</i>		X			
<i>DB2 Version 9.1 for z/OS Program Directory</i>	GI10-8737		X		X
<i>DB2 Version 9.1 for z/OS RACF Access Control Module Guide</i>	SC18-9852	X	X	X	
<i>DB2 Version 9.1 for z/OS Reference for Remote DRDA Requesters and Servers</i>	SC18-9853	X	X	X	
<i>DB2 Version 9.1 for z/OS Reference Summary</i>	SX26-3854		X		
<i>DB2 Version 9.1 for z/OS SQL Reference</i>	SC18-9854	X	X	X	X
<i>DB2 Version 9.1 for z/OS Utility Guide and Reference</i>	SC18-9855	X	X	X	X
<i>DB2 Version 9.1 for z/OS What's New?</i>	GC18-9856	X	X		X
<i>DB2 Version 9.1 for z/OS XML Extender Administration and Programming</i>	SC18-9857	X	X	X	X
<i>DB2 Version 9.1 for z/OS XML Guide</i>	SC18-9858	X	X	X	X
<i>IRLM Messages and Codes for IMS and DB2 for z/OS</i>	GC19-2666	X	X	X	

Note:

1. *DB2 Version 9.1 for z/OS Diagnosis Guide and Reference* is available in PDF and BookManager formats on the DB2 Version 9.1 for z/OS Licensed Collection kit, LK3T-7195. You can order this License Collection kit on the IBM Publications Center site (<http://www.elink.ibm.link.ibm.com/public/applications/publications/cgibin/pbi.cgi>). This book is also available in online format in DB2 data set DSN910.SDSNIVPD(DSNDR).

Information resources for related products

In the following table, related product names are listed in alphabetic order, and the associated Web addresses of product information centers or library Web pages are indicated.

Table 114. Related product information resource locations

Related product	Information resources
C/C++ for z/OS	Library Web site: http://www.ibm.com/software/awdtools/czos/library/ This product is now called z/OS XL C/C++.
CICS Transaction Server for z/OS	Information center: http://publib.boulder.ibm.com/infocenter/cicsts/v3r1/index.jsp
COBOL	Information center: http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/index.jsp This product is now called Enterprise COBOL for z/OS.
DB2 Connect	Information center: http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp This resource is for DB2 Connect 9.
DB2 Database for Linux, UNIX, and Windows	Information center: http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp This resource is for DB2 9 for Linux, UNIX, and Windows.
DB2 Query Management Facility	Information center: http://publib.boulder.ibm.com/infocenter/imzic
DB2 Server for VSE & VM	One of the following locations: <ul style="list-style-type: none"> • For VSE: http://www.ibm.com/support/docview.wss?rs=66&uid=swg27003758 • For VM: http://www.ibm.com/support/docview.wss?rs=66&uid=swg27003759
DB2 Tools	One of the following locations: <ul style="list-style-type: none"> • Information center: http://publib.boulder.ibm.com/infocenter/imzic • Library Web site: http://www.ibm.com/software/data/db2imstools/library.html <p>These resources include information about the following products and others:</p> <ul style="list-style-type: none"> • DB2 Administration Tool • DB2 Automation Tool • DB2 Log Analysis Tool • DB2 Object Restore Tool • DB2 Query Management Facility • DB2 SQL Performance Analyzer
DB2® Universal Database™ for iSeries	Information center: http://www.ibm.com/systems/i/infocenter/
Debug Tool for z/OS	Information center: http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/index.jsp
Enterprise COBOL for z/OS	Information center: http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/index.jsp
Enterprise PL/I for z/OS	Information center: http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/index.jsp
IMS	Information center: http://publib.boulder.ibm.com/infocenter/imzic

Table 114. Related product information resource locations (continued)

Related product	Information resources
IMS Tools	<p>One of the following locations:</p> <ul style="list-style-type: none"> • Information center: http://publib.boulder.ibm.com/infocenter/imzic • Library Web site: http://www.ibm.com/software/data/db2imstools/library.html <p>These resources have information about the following products and others:</p> <ul style="list-style-type: none"> • IMS Batch Terminal Simulator for z/OS • IMS Connect • IMS HALDB Conversion and Maintenance Aid • IMS High Performance Utility products • IMS DataPropagator™ • IMS Online Reorganization Facility • IMS Performance Analyzer
Integrated Data Management products	<p>Information center: http://publib.boulder.ibm.com/infocenter/idm/v2r2/index.jsp</p> <p>This information center has information about the following products and others:</p> <ul style="list-style-type: none"> • IBM Data Studio • InfoSphere™ Data Architect • InfoSphere Warehouse • Optim Database Administrator • Optim Development Studio • Optim Query Tuner
PL/I	<p>Information center: http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/index.jsp</p> <p>This product is now called Enterprise PL/I for z/OS.</p>
System z®	http://publib.boulder.ibm.com/infocenter/eserver/v1r2/index.jsp
Tivoli OMEGAMONXE for DB2 Performance Expert on z/OS	<p>Information center: http://publib.boulder.ibm.com/infocenter/tivihelp/v15r1/index.jsp?topic=/com.ibm.ko2pe.doc/ko2welcome.htm</p> <p>In earlier releases, this product was called DB2 Performance Expert for z/OS.</p>
WebSphere Application Server	Information center: http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp
WebSphere Message Broker with Rules and Formatter Extension	<p>Information center: http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/index.jsp</p> <p>The product is also known as WebSphere MQ Integrator Broker.</p>
WebSphere MQ	<p>Information center: http://publib.boulder.ibm.com/infocenter/wmqv6/v6r0/index.jsp</p> <p>The resource includes information about MQSeries®.</p>
WebSphere Replication Server for z/OS	<p>Either of the following locations:</p> <ul style="list-style-type: none"> • Information center: http://publib.boulder.ibm.com/infocenter/imzic • Library Web site: http://www.ibm.com/software/data/db2imstools/library.html <p>This product is also known as DB2 DataPropagator.</p>
z/Architecture®	Library Center site: http://www.ibm.com/servers/eserver/zseries/zos/bkserv/

Table 114. Related product information resource locations (continued)

Related product	Information resources
z/OS	<p>Library Center site: http://www.ibm.com/servers/eserver/zseries/zos/bkserv/</p> <p>This resource includes information about the following z/OS elements and components:</p> <ul style="list-style-type: none"> • Character Data Representation Architecture • Device Support Facilities • DFSORT • Fortran • High Level Assembler • NetView® • SMP/E for z/OS • SNA • TCP/IP • TotalStorage® Enterprise Storage Server® • VTAM® • z/OS C/C++ • z/OS Communications Server • z/OS DCE • z/OS DFSMS™ • z/OS DFSMS Access Method Services • z/OS DFSMSdss™ • z/OS DFSMSHsm™ • z/OS DFSMSdftp™ • z/OS ICSF • z/OS ISPF • z/OS JES3 • z/OS Language Environment • z/OS Managed System Infrastructure • z/OS MVS • z/OS MVS JCL • z/OS Parallel Sysplex® • z/OS RMF™ • z/OS Security Server • z/OS UNIX System Services
z/OS XL C/C++	<p>http://www.ibm.com/software/awdtools/czos/library/</p>

The following information resources from IBM are not necessarily specific to a single product:

- The DB2 for z/OS Information Roadmap; available at: <http://www.ibm.com/software/data/db2/zos/roadmap.html>
- DB2 Redbooks® and Redbooks about related products; available at: <http://www.ibm.com/redbooks>
- IBM Educational resources:
 - Information about IBM educational offerings is available on the Web at: <http://www.ibm.com/software/sw-training/>

- A collection of glossaries of IBM terms in multiple languages is available on the IBM Terminology Web site at: <http://www.ibm.com/software/globalization/terminology/index.jsp>
- National Language Support information; available at the IBM Publications Center at: <http://www.elink.ibm.link.ibm.com/public/applications/publications/cgibin/pbi.cgi>
- *SQL Reference for Cross-Platform Development*; available at the following developerWorks® site: <http://www.ibm.com/developerworks/db2/library/techarticle/0206sqlref/0206sqlref.html>

The following information resources are not published by IBM but can be useful to users of DB2 for z/OS and related products:

- Database design topics:
 - *DB2 for z/OS and OS/390 Development for Performance Volume I*, by Gabrielle Wiorkowski, Gabrielle & Associates, ISBN 0-96684-605-2
 - *DB2 for z/OS and OS/390 Development for Performance Volume II*, by Gabrielle Wiorkowski, Gabrielle & Associates, ISBN 0-96684-606-0
 - *Handbook of Relational Database Design*, by C. Fleming and B. Von Halle, Addison Wesley, ISBN 0-20111-434-8
- Distributed Relational Database Architecture™ (DRDA) specifications; <http://www.opengroup.org>
- Domain Name System: *DNS and BIND*, Third Edition, Paul Albitz and Cricket Liu, O'Reilly, ISBN 0-59600-158-4
- Microsoft® Open Database Connectivity (ODBC) information; <http://msdn.microsoft.com/library/>
- Unicode information; <http://www.unicode.org>

How to obtain DB2 information

You can access the official information about the DB2 product in a number of ways.

- “DB2 on the Web”
- “DB2 product information”
- “DB2 education” on page 560
- “How to order the DB2 library” on page 560

DB2 on the Web

Stay current with the latest information about DB2 by visiting the DB2 home page on the Web:

www.ibm.com/software/db2zos

On the DB2 home page, you can find links to a wide variety of information resources about DB2. You can read news items that keep you informed about the latest enhancements to the product. Product announcements, press releases, fact sheets, and technical articles help you plan and implement your database management strategy.

DB2 product information

The official DB2 for z/OS information is available in various formats and delivery methods. IBM provides mid-version updates to the information in the information center and in softcopy updates that are available on the Web and on CD-ROM.

Information Management Software for z/OS Solutions Information Center

DB2 product information is viewable in the information center, which is the primary delivery vehicle for information about DB2 for z/OS, IMS, QMF, and related tools. This information center enables you to search across related product information in multiple languages for data management solutions for the z/OS environment and print individual topics or sets of related topics. You can also access, download, and print PDFs of the publications that are associated with the information center topics. Product technical information is provided in a format that offers more options and tools for accessing, integrating, and customizing information resources. The information center is based on Eclipse open source technology.

The Information Management Software for z/OS Solutions Information Center is viewable at the following Web site:

<http://publib.boulder.ibm.com/infocenter/imzic>

CD-ROMs and DVD

Books for DB2 are available on a CD-ROM that is included with your product shipment:

- DB2 V9.1 for z/OS Licensed Library Collection, LK3T-7195, in English

The CD-ROM contains the collection of books for DB2 V9.1 for z/OS in PDF and BookManager formats. Periodically, IBM refreshes the books on subsequent editions of this CD-ROM.

The books for DB2 for z/OS are also available on the following CD-ROM and DVD collection kits, which contain online books for many IBM products:

- IBM z/OS Software Products Collection , SK3T-4270, in English
- IBM z/OS Software Products DVD Collection , SK3T-4271, in English

PDF format

Many of the DB2 books are available in PDF (Portable Document Format) for viewing or printing from CD-ROM or the DB2 home page on the Web or from the information center. Download the PDF books to your intranet for distribution throughout your enterprise.

BookManager format

You can use online books on CD-ROM to read, search across books, print portions of the text, and make notes in these BookManager books. Using the IBM Softcopy Reader, appropriate IBM Library Readers, or the BookManager Read product, you can view these books in the z/OS, Windows, and VM environments. You can also view and search many of the DB2 BookManager books on the Web.

DB2 education

IBM Education and Training offers a wide variety of classroom courses to help you quickly and efficiently gain DB2 expertise. IBM schedules classes in cities all over the world. You can find class information, by country, at the IBM Learning Services Web site:

www.ibm.com/services/learning

IBM also offers classes at your location, at a time that suits your needs. IBM can customize courses to meet your exact requirements. For more information, including the current local schedule, contact your IBM representative.

How to order the DB2 library

To order books, visit the IBM Publication Center on the Web:

www.elink.ibm.link.ibm.com/public/applications/publications/cgibin/pbi.cgi

From the IBM Publication Center, you can go to the Publication Notification System (PNS). PNS users receive electronic notifications of updated publications in their profiles. You have the option of ordering the updates by using the publications direct ordering application or any other IBM publication ordering channel. The PNS application does not send automatic shipments of publications. You will receive updated publications and a bill for them if you respond to the electronic notification.

You can also order DB2 publications and CD-ROMs from your IBM representative or the IBM branch office that serves your locality. If your location is within the United States or Canada, you can place your order by calling one of the toll-free numbers:

- In the U.S., call 1-800-879-2755.
- In Canada, call 1-800-426-4968.

To order additional copies of licensed publications, specify the SOFTWARE option. To order additional publications or CD-ROMs, specify the PUBLICATIONS option.

Be prepared to give your customer number, the product number, and either the feature codes or order numbers that you want.

How to use the DB2 library

Titles of books in the library begin with DB2 Version 9.1 for z/OS. However, references from one book in the library to another are shortened and do not include the product name, version, and release. Instead, they point directly to the section that holds the information. The primary place to find and use information about DB2 for z/OS is the Information Management Software for z/OS Solutions Information Center (<http://publib.boulder.ibm.com/infocenter/imzic>).

If you are new to DB2 for z/OS, *Introduction to DB2 for z/OS* provides a comprehensive introduction to DB2 Version 9.1 for z/OS. Topics included in this book explain the basic concepts that are associated with relational database management systems in general, and with DB2 for z/OS in particular.

The most rewarding task associated with a database management system is asking questions of it and getting answers, the task called *end use*. Other tasks are also necessary—defining the parameters of the system, putting the data in place, and so on. The tasks that are associated with DB2 are grouped into the following major categories.

Installation

If you are involved with installing DB2, you will need to use a variety of resources, such as:

- *DB2 Program Directory*
- *DB2 Installation Guide*
- *DB2 Administration Guide*
- *DB2 Application Programming Guide and Reference for Java*
- *DB2 Codes*
- *DB2 Internationalization Guide*
- *DB2 Messages*
- *DB2 Performance Monitoring and Tuning Guide*
- *DB2 RACF Access Control Module Guide*
- *DB2 Utility Guide and Reference*

If you will be using data sharing capabilities you also need *DB2 Data Sharing: Planning and Administration*, which describes installation considerations for data sharing.

If you will be installing and configuring DB2 ODBC, you will need *DB2 ODBC Guide and Reference*.

If you are installing IBM Spatial Support for DB2 for z/OS, you will need *IBM Spatial Support for DB2 for z/OS User's Guide and Reference*.

If you are installing IBM OmniFind® Text Search Server for DB2 for z/OS, you will need *IBM OmniFind Text Search Server for DB2 for z/OS Installation, Administration, and Reference*.

End use

End users issue SQL statements to retrieve data. They can also insert, update, or delete data, with SQL statements. They might need an introduction to SQL, detailed instructions for using SPUFI, and an alphabetized reference to the types of SQL statements. This information is found in *DB2 Application Programming and SQL Guide*, and *DB2 SQL Reference*.

End users can also issue SQL statements through the DB2 Query Management Facility (QMF) or some other program, and the library for that licensed program might provide all the instruction or reference material they need. For a list of the titles in the DB2 QMF library, see the bibliography at the end of this book.

Application programming

Some users access DB2 without knowing it, using programs that contain SQL statements. DB2 application programmers write those programs. Because they write SQL statements, they need the same resources that end users do.

Application programmers also need instructions for many other topics:

- How to transfer data between DB2 and a host program—written in Java, C, or COBOL, for example
- How to prepare to compile a program that embeds SQL statements
- How to process data from two systems simultaneously, for example, DB2 and IMS or DB2 and CICS
- How to write distributed applications across operating systems
- How to write applications that use Open Database Connectivity (ODBC) to access DB2 servers
- How to write applications that use JDBC and SQLJ with the Java programming language to access DB2 servers
- How to write applications to store XML data on DB2 servers and retrieve XML data from DB2 servers.

The material needed for writing a host program containing SQL is in *DB2 Application Programming and SQL Guide*.

The material needed for writing applications that use JDBC and SQLJ to access DB2 servers is in *DB2 Application Programming Guide and Reference for Java*. The material needed for writing applications that use DB2 CLI or ODBC to access DB2 servers is in *DB2 ODBC Guide and Reference*. The material needed for working with XML data in DB2 is in *DB2 XML Guide*. For handling errors, see *DB2 Messages and DB2 Codes*.

If you are a software vendor implementing DRDA clients and servers, you will need *DB2 Reference for Remote DRDA Requesters and Servers*.

Information about writing applications across operating systems can be found in *IBM DB2 SQL Reference for Cross-Platform Development*.

System and database administration

Administration covers almost everything else. *DB2 Administration Guide* divides some of those tasks among the following sections:

- **DB2 concepts:** Introduces DB2 structures, the DB2 environment, and high availability.
- **Designing a database:** Discusses the decisions that must be made when designing a database and tells how to implement the design by creating and altering DB2 objects, loading data, and adjusting to changes.
- **Security and auditing:** Describes ways of controlling access to the DB2 system and to data within DB2, to audit aspects of DB2 usage, and to answer other security and auditing concerns.
- **Operation and recovery:** Describes the steps in normal day-to-day operation and discusses the steps one should take to prepare for recovery in the event of some failure.

DB2 Performance Monitoring and Tuning Guide explains how to monitor the performance of the DB2 system and its parts. It also lists things that can be done to make some parts run faster.

If you will be using the RACF access control module for DB2 authorization checking, you will need *DB2 RACF Access Control Module Guide*.

If you are involved with DB2 only to design the database, or plan operational procedures, you need *DB2 Administration Guide*. If you also want to carry out your own plans by creating DB2 objects, granting privileges, running utility jobs, and so on, you also need:

- *DB2 SQL Reference*, which describes the SQL statements you use to create, alter, and drop objects and grant and revoke privileges
- *DB2 Utility Guide and Reference*, which explains how to run utilities
- *DB2 Command Reference*, which explains how to run commands

If you will be using data sharing, you need *DB2 Data Sharing: Planning and Administration*, which describes how to plan for and implement data sharing.

Additional information about system and database administration can be found in *DB2 Messages* and *DB2 Codes*, which list messages and codes issued by DB2, with explanations and suggested responses.

Diagnosis

Diagnosticians detect and describe errors in the DB2 program. They might also recommend or apply a remedy. The documentation for this task is in *DB2 Diagnosis Guide and Reference*, *DB2 Messages*, and *DB2 Codes*.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
3-2-12, Roppongi, Minato-ku
Tokyo 106-8711
Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

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 Version 9.1 for z/OS servers. This book primarily documents General-use Programming Interface and Associated Guidance Information provided by DB2 Version 9.1 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 Version 9.1 for z/OS.

General-use Programming Interface and Associated Guidance Information is identified where it occurs by the following markings:



General-use Programming Interface and Associated Guidance Information...



Trademarks

IBM, the IBM logo, and [ibm.com](http://www.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 <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 of Sun Microsystems, Inc. in the United States, other countries, or both.

Glossary

abend See abnormal end of task.

abend reason code

A 4-byte hexadecimal code that uniquely identifies a problem with DB2.

abnormal end of task (abend)

Termination of a task, job, or subsystem because of an error condition that recovery facilities cannot resolve during execution.

access method services

The facility that is used to define, alter, delete, print, and reproduce VSAM key-sequenced data sets.

access path

The path that is used to locate data that is specified in SQL statements. An access path can be indexed or sequential.

active log

The portion of the DB2 log to which log records are written as they are generated. The active log always contains the most recent log records. See also archive log.

address space

A range of virtual storage pages that is identified by a number (ASID) and a collection of segment and page tables that map the virtual pages to real pages of the computer's memory.

address space connection

The result of connecting an allied address space to DB2. See also allied address space and task control block.

address space identifier (ASID)

A unique system-assigned identifier for an address space.

AFTER trigger

A trigger that is specified to be activated after a defined trigger event (an insert, update, or delete operation on the table that is specified in a trigger definition). Contrast with BEFORE trigger and INSTEAD OF trigger.

agent In DB2, the structure that associates all processes that are involved in a DB2 unit of work. See also allied agent and system agent.

aggregate function

An operation that derives its result by using values from one or more rows. Contrast with scalar function.

alias

An alternative name that can be used in SQL statements to refer to a table or view in the same or a remote DB2 subsystem. An alias can be qualified with a schema qualifier and can thereby be referenced by other users. Contrast with synonym.

allied address space

An area of storage that is external to DB2 and that is connected to DB2. An allied address space can request DB2 services. See also address space.

allied agent

An agent that represents work requests that originate in allied address spaces. See also system agent.

allied thread

A thread that originates at the local DB2 subsystem and that can access data at a remote DB2 subsystem.

allocated cursor

A cursor that is defined for a stored procedure result set by using the SQL ALLOCATE CURSOR statement.

ambiguous cursor

A database cursor for which DB2 cannot determine whether it is used for update or read-only purposes.

APAR See authorized program analysis report.

APF See authorized program facility.

API See application programming interface.

APPL A VTAM network definition statement that is used to define DB2 to VTAM as an application program that uses SNA LU 6.2 protocols.

application

A program or set of programs that performs a task; for example, a payroll application.

application plan

The control structure that is produced during the bind process. DB2 uses the

application plan to process SQL statements that it encounters during statement execution.

application process

The unit to which resources and locks are allocated. An application process involves the execution of one or more programs.

application programming interface (API)

A functional interface that is supplied by the operating system or by a separately orderable licensed program that allows an application program that is written in a high-level language to use specific data or functions of the operating system or licensed program.

application requester

The component on a remote system that generates DRDA requests for data on behalf of an application.

application server

The target of a request from a remote application. In the DB2 environment, the application server function is provided by the distributed data facility and is used to access DB2 data from remote applications.

archive log

The portion of the DB2 log that contains log records that have been copied from the active log. See also active log.

ASCII An encoding scheme that is used to represent strings in many environments, typically on PCs and workstations. Contrast with EBCDIC and Unicode.

ASID See address space identifier.

attachment facility

An interface between DB2 and TSO, IMS, CICS, or batch address spaces. An attachment facility allows application programs to access DB2.

attribute

A characteristic of an entity. For example, in database design, the phone number of an employee is an attribute of that employee.

authorization ID

A string that can be verified for connection to DB2 and to which a set of privileges is allowed. An authorization ID can represent an individual or an organizational group.

authorized program analysis report (APAR)

A report of a problem that is caused by a suspected defect in a current release of an IBM supplied program.

authorized program facility (APF)

A facility that allows an installation to identify system or user programs that can use sensitive system functions.

automatic bind

(More correctly *automatic rebind*.) A process by which SQL statements are bound automatically (without a user issuing a BIND command) when an application process begins execution and the bound application plan or package it requires is not valid.

automatic query rewrite

A process that examines an SQL statement that refers to one or more base tables or materialized query tables, and, if appropriate, rewrites the query so that it performs better.

auxiliary index

An index on an auxiliary table in which each index entry refers to a LOB or XML document.

auxiliary table

A table that contains columns outside the actual table in which they are defined. Auxiliary tables can contain either LOB or XML data.

backout

The process of undoing uncommitted changes that an application process made. A backout is often performed in the event of a failure on the part of an application process, or as a result of a deadlock situation.

backward log recovery

The final phase of restart processing during which DB2 scans the log in a backward direction to apply UNDO log records for all aborted changes.

base table

A table that is created by the SQL CREATE TABLE statement and that holds persistent data. Contrast with clone table, materialized query table, result table, temporary table, and transition table.

base table space

A table space that contains base tables.

basic row format	<ul style="list-style-type: none"> A row format in which values for columns are stored in the row in the order in which the columns are defined by the CREATE TABLE statement. Contrast with reordered row format.
basic sequential access method (BSAM)	An access method for storing or retrieving data blocks in a continuous sequence, using either a sequential-access or a direct-access device.
BEFORE trigger	A trigger that is specified to be activated before a defined trigger event (an insert, an update, or a delete operation on the table that is specified in a trigger definition). Contrast with AFTER trigger and INSTEAD OF trigger.
binary large object (BLOB)	A binary string data type that contains a sequence of bytes that can range in size from 0 bytes to 2 GB, less 1 byte. This string does not have an associated code page and character set. BLOBs can contain, for example, image, audio, or video data. In general, BLOB values are used whenever a binary string might exceed the limits of the VARBINARY type.
binary string	A sequence of bytes that is not associated with a CCSID. Binary string data type can be further classified as BINARY, VARBINARY, or BLOB.
bind	A process by which a usable control structure with SQL statements is generated; the structure is often called an access plan, an application plan, or a package. During this bind process, access paths to the data are selected, and some authorization checking is performed. See also automatic bind.
bit data	<ul style="list-style-type: none"> Data with character type CHAR or VARCHAR that is defined with the FOR BIT DATA clause. Note that using BINARY or VARBINARY rather than FOR BIT DATA is highly recommended. Data with character type CHAR or VARCHAR that is defined with the FOR BIT DATA clause.
	<ul style="list-style-type: none"> A form of character data. Binary data is generally more highly recommended than character-for-bit data.
BLOB	See binary large object.
block fetch	A capability in which DB2 can retrieve, or fetch, a large set of rows together. Using block fetch can significantly reduce the number of messages that are being sent across the network. Block fetch applies only to non-rowset cursors that do not update data.
bootstrap data set (BSDS)	A VSAM data set that contains name and status information for DB2 and RBA range specifications for all active and archive log data sets. The BSDS also contains passwords for the DB2 directory and catalog, and lists of conditional restart and checkpoint records.
BSAM	See basic sequential access method.
BSDS	See bootstrap data set.
buffer pool	An area of memory into which data pages are read, modified, and held during processing.
built-in data type	A data type that IBM supplies. Among the built-in data types for DB2 for z/OS are string, numeric, XML, ROWID, and datetime. Contrast with distinct type.
built-in function	A function that is generated by DB2 and that is in the SYSIBM schema. Contrast with user-defined function. See also function, cast function, external function, sourced function, and SQL function.
business dimension	A category of data, such as products or time periods, that an organization might want to analyze.
cache structure	A coupling facility structure that stores data that can be available to all members of a Sysplex. A DB2 data sharing group uses cache structures as group buffer pools.
CAF	See call attachment facility.

call attachment facility (CAF)

A DB2 attachment facility for application programs that run in TSO or z/OS batch. The CAF is an alternative to the DSN command processor and provides greater control over the execution environment. Contrast with Recoverable Resource Manager Services attachment facility.

call-level interface (CLI)

A callable application programming interface (API) for database access, which is an alternative to using embedded SQL.

cascade delete

A process by which DB2 enforces referential constraints by deleting all descendent rows of a deleted parent row.

CASE expression

An expression that is selected based on the evaluation of one or more conditions.

cast function

A function that is used to convert instances of a (source) data type into instances of a different (target) data type.

castout

The DB2 process of writing changed pages from a group buffer pool to disk.

castout owner

The DB2 member that is responsible for casting out a particular page set or partition.

catalog

In DB2, a collection of tables that contains descriptions of objects such as tables, views, and indexes.

catalog table

Any table in the DB2 catalog.

CCSID

See coded character set identifier.

CDB

See communications database.

CDRA

See Character Data Representation Architecture.

CEC

See central processor complex.

central electronic complex (CEC)

See central processor complex.

central processor complex (CPC)

A physical collection of hardware that consists of main storage, one or more central processors, timers, and channels.

central processor (CP)

The part of the computer that contains the sequencing and processing facilities for instruction execution, initial program load, and other machine operations.

CFRM See coupling facility resource management.

CFRM policy

The allocation rules for a coupling facility structure that are declared by a z/OS administrator.

character conversion

The process of changing characters from one encoding scheme to another.

Character Data Representation Architecture (CDRA)

An architecture that is used to achieve consistent representation, processing, and interchange of string data.

character large object (CLOB)

A character string data type that contains a sequence of bytes that represent characters (single-byte, multibyte, or both) that can range in size from 0 bytes to 2 GB, less 1 byte. In general, CLOB values are used whenever a character string might exceed the limits of the VARCHAR type.

character set

A defined set of characters.

character string

A sequence of bytes that represent bit data, single-byte characters, or a mixture of single-byte and multibyte characters. Character data can be further classified as CHARACTER, VARCHAR, or CLOB.

check constraint

A user-defined constraint that specifies the values that specific columns of a base table can contain.

check integrity

The condition that exists when each row in a table conforms to the check constraints that are defined on that table.

check pending

A state of a table space or partition that prevents its use by some utilities and by some SQL statements because of rows that violate referential constraints, check constraints, or both.

checkpoint

A point at which DB2 records status information on the DB2 log; the recovery process uses this information if DB2 abnormally terminates.

child lock

For explicit hierarchical locking, a lock that is held on either a table, page, row, or a large object (LOB). Each child lock has a parent lock. See also parent lock.

CI See control interval.

CICS Represents (in this information): CICS Transaction Server for z/OS: Customer Information Control System Transaction Server for z/OS.

CICS attachment facility

A facility that provides a multithread connection to DB2 to allow applications that run in the CICS environment to execute DB2 statements.

claim A notification to DB2 that an object is being accessed. Claims prevent drains from occurring until the claim is released, which usually occurs at a commit point. Contrast with drain.

claim class

A specific type of object access that can be one of the following isolation levels:

- Cursor stability (CS)
- Repeatable read (RR)
- Write

class of service

A VTAM term for a list of routes through a network, arranged in an order of preference for their use.

clause In SQL, a distinct part of a statement, such as a SELECT clause or a WHERE clause.

CLI See call-level interface.

client See requester.

CLOB See character large object.

clone object

An object that is associated with a clone table, including the clone table itself and check constraints, indexes, and BEFORE triggers on the clone table.

clone table

A table that is structurally identical to a base table. The base and clone table each

have separate underlying VSAM data sets, which are identified by their data set instance numbers. Contrast with base table.

closed application

An application that requires exclusive use of certain statements on certain DB2 objects, so that the objects are managed solely through the external interface of that application.

clustering index

An index that determines how rows are physically ordered (*clustered*) in a table space. If a clustering index on a partitioned table is not a partitioning index, the rows are ordered in cluster sequence within each data partition instead of spanning partitions.

CM See conversion mode.

CM* See conversion mode*.

C++ member

A data object or function in a structure, union, or class.

C++ member function

An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and to the member functions of objects in its class. Member functions are also called methods.

C++ object

A region of storage. An object is created when a variable is defined or a new function is invoked.

An instance of a class.

coded character set

A set of unambiguous rules that establish a character set and the one-to-one relationships between the characters of the set and their coded representations.

coded character set identifier (CCSID)

A 16-bit number that uniquely identifies a coded representation of graphic characters. It designates an encoding scheme identifier and one or more pairs that consist of a character set identifier and an associated code page identifier.

code page

A set of assignments of characters to code

points. Within a code page, each code point has only one specific meaning. In EBCDIC, for example, the character *A* is assigned code point X'C1', and character *B* is assigned code point X'C2'.

code point

In CDRA, a unique bit pattern that represents a character in a code page.

code unit

The fundamental binary width in a computer architecture that is used for representing character data, such as 7 bits, 8 bits, 16 bits, or 32 bits. Depending on the character encoding form that is used, each code point in a coded character set can be represented by one or more code units.

coexistence

During migration, the period of time in which two releases exist in the same data sharing group.

cold start

A process by which DB2 restarts without processing any log records. Contrast with warm start.

collection

A group of packages that have the same qualifier.

column

The vertical component of a table. A column has a name and a particular data type (for example, character, decimal, or integer).

column function

See aggregate function.

"come from" checking

An LU 6.2 security option that defines a list of authorization IDs that are allowed to connect to DB2 from a partner LU.

command

A DB2 operator command or a DSN subcommand. A command is distinct from an SQL statement.

command prefix

A 1- to 8-character command identifier. The command prefix distinguishes the command as belonging to an application or subsystem rather than to z/OS.

command recognition character (CRC)

A character that permits a z/OS console

operator or an IMS subsystem user to route DB2 commands to specific DB2 subsystems.

command scope

The scope of command operation in a data sharing group.

commit

The operation that ends a unit of work by releasing locks so that the database changes that are made by that unit of work can be perceived by other processes. Contrast with rollback.

commit point

A point in time when data is considered consistent.

common service area (CSA)

In z/OS, a part of the common area that contains data areas that are addressable by all address spaces. Most DB2 use is in the extended CSA, which is above the 16-MB line.

communications database (CDB)

A set of tables in the DB2 catalog that are used to establish conversations with remote database management systems.

comparison operator

A token (such as =, >, or <) that is used to specify a relationship between two values.

compatibility mode

See conversion mode.

compatibility mode* (CM*)

See conversion mode*.

composite key

An ordered set of key columns or expressions of the same table.

compression dictionary

The dictionary that controls the process of compression and decompression. This dictionary is created from the data in the table space or table space partition.

concurrency

The shared use of resources by more than one application process at the same time.

conditional restart

A DB2 restart that is directed by a user-defined conditional restart control record (CRCR).

connection

In SNA, the existence of a communication path between two partner LUs that allows information to be exchanged (for example, two DB2 subsystems that are connected and communicating by way of a conversation).

connection context

In SQLJ, a Java object that represents a connection to a data source.

connection declaration clause

In SQLJ, a statement that declares a connection to a data source.

connection handle

The data object containing information that is associated with a connection that DB2 ODBC manages. This includes general status information, transaction status, and diagnostic information.

connection ID

An identifier that is supplied by the attachment facility and that is associated with a specific address space connection.

consistency token

A timestamp that is used to generate the version identifier for an application. See also version.

constant

A language element that specifies an unchanging value. Constants are classified as string constants or numeric constants. Contrast with variable.

constraint

A rule that limits the values that can be inserted, deleted, or updated in a table. See referential constraint, check constraint, and unique constraint.

context

An application's logical connection to the data source and associated DB2 ODBC connection information that allows the application to direct its operations to a data source. A DB2 ODBC context represents a DB2 thread.

contracting conversion

A process that occurs when the length of a converted string is smaller than that of the source string. For example, this process occurs when an EBCDIC mixed-data string that contains DBCS characters is converted to ASCII mixed

data; the converted string is shorter because the shift codes are removed.

control interval (CI)

- A unit of information that VSAM transfers between virtual and auxiliary storage.
- In a key-sequenced data set or file, the set of records that an entry in the sequence-set index record points to.

conversation

Communication, which is based on LU 6.2 or Advanced Program-to-Program Communication (APPC), between an application and a remote transaction program over an SNA logical unit-to-logical unit (LU-LU) session that allows communication while processing a transaction.

conversion mode (CM)

The first stage of the version-to-version migration process. In a DB2 data sharing group, members in conversion mode can coexist with members that are still at the prior version level. Fallback to the prior version is also supported. When in conversion mode, the DB2 subsystem cannot use most new functions of the new version. Contrast with conversion mode*, enabling-new-function mode, enabling-new-function mode*, and new-function mode.

Previously known as compatibility mode (CM).

conversion mode* (CM*)

A stage of the version-to-version migration process that applies to a DB2 subsystem or data sharing group that was in enabling-new-function mode (ENFM), enabling-new-function mode* (ENFM*), or new-function mode (NFM) at one time. Fallback to a prior version is not supported. When in conversion mode*, a DB2 data sharing group cannot coexist with members that are still at the prior version level. Contrast with conversion mode, enabling-new-function mode, enabling-new-function mode*, and new-function mode.

Previously known as compatibility mode* (CM*).

coordinator

The system component that coordinates

the commit or rollback of a unit of work that includes work that is done on one or more other systems.

coprocessor

See SQL statement coprocessor.

copy pool

A collection of names of storage groups that are processed collectively for fast replication operations.

copy target

A named set of SMS storage groups that are to be used as containers for copy pool volume copies. A copy target is an SMS construct that lets you define which storage groups are to be used as containers for volumes that are copied by using FlashCopy functions.

copy version

A point-in-time FlashCopy copy that is managed by HSM. Each copy pool has a version parameter that specifies the number of copy versions to be maintained on disk.

correlated columns

A relationship between the value of one column and the value of another column.

correlated subquery

A subquery (part of a WHERE or HAVING clause) that is applied to a row or group of rows of a table or view that is named in an outer subselect statement.

correlation ID

An identifier that is associated with a specific thread. In TSO, it is either an authorization ID or the job name.

correlation name

An identifier that is specified and used within a single SQL statement as the exposed name for objects such as a table, view, table function reference, nested table expression, or result of a data change statement. Correlation names are useful in an SQL statement to allow two distinct references to the same base table and to allow an alternative name to be used to represent an object.

cost category

A category into which DB2 places cost estimates for SQL statements at the time the statement is bound. The cost category is externalized in the COST_CATEGORY

column of the DSN_STATEMENT_TABLE when a statement is explained.

coupling facility

A special PR/SM logical partition (LPAR) that runs the coupling facility control program and provides high-speed caching, list processing, and locking functions in a Parallel Sysplex.

coupling facility resource management (CFRM)

A component of z/OS that provides the services to manage coupling facility resources in a Parallel Sysplex. This management includes the enforcement of CFRM policies to ensure that the coupling facility and structure requirements are satisfied.

CP See central processor.

CPC See central processor complex.

CRC See command recognition character.

created temporary table

A persistent table that holds temporary data and is defined with the SQL statement CREATE GLOBAL TEMPORARY TABLE. Information about created temporary tables is stored in the DB2 catalog and can be shared across application processes. Contrast with declared temporary table. See also temporary table.

cross-system coupling facility (XCF)

A component of z/OS that provides functions to support cooperation between authorized programs that run within a Sysplex.

cross-system extended services (XES)

A set of z/OS services that allow multiple instances of an application or subsystem, running on different systems in a Sysplex environment, to implement high-performance, high-availability data sharing by using a coupling facility.

CS See cursor stability.

CSA See common service area.

CT See cursor table.

current data

Data within a host structure that is current with (identical to) the data within the base table.

current status rebuild

The second phase of restart processing during which the status of the subsystem is reconstructed from information on the log.

cursor A control structure that an application program uses to point to a single row or multiple rows within some ordered set of rows of a result table. A cursor can be used to retrieve, update, or delete rows from a result table.

cursor sensitivity

The degree to which database updates are visible to the subsequent FETCH statements in a cursor.

cursor stability (CS)

The isolation level that provides maximum concurrency without the ability to read uncommitted data. With cursor stability, a unit of work holds locks only on its uncommitted changes and on the current row of each of its cursors. See also read stability, repeatable read, and uncommitted read.

cursor table (CT)

The internal representation of a cursor.

cycle A set of tables that can be ordered so that each table is a descendent of the one before it, and the first table is a descendent of the last table. A self-referencing table is a cycle with a single member. See also referential cycle.

database

A collection of tables, or a collection of table spaces and index spaces.

database access thread (DBAT)

A thread that accesses data at the local subsystem on behalf of a remote subsystem.

database administrator (DBA)

An individual who is responsible for designing, developing, operating, safeguarding, maintaining, and using a database.

database alias

The name of the target server if it is different from the location name. The database alias is used to provide the name of the database server as it is known to the network.

database descriptor (DBD)

An internal representation of a DB2 database definition, which reflects the data definition that is in the DB2 catalog. The objects that are defined in a database descriptor are table spaces, tables, indexes, index spaces, relationships, check constraints, and triggers. A DBD also contains information about accessing tables in the database.

database exception status

In a data sharing environment, an indication that something is wrong with a database.

database identifier (DBID)

An internal identifier of the database.

database management system (DBMS)

A software system that controls the creation, organization, and modification of a database and the access to the data that is stored within it.

database request module (DBRM)

A data set member that is created by the DB2 precompiler and that contains information about SQL statements. DBRMs are used in the bind process.

database server

The target of a request from a local application or a remote intermediate database server.

data currency

The state in which the data that is retrieved into a host variable in a program is a copy of the data in the base table.

data dictionary

A repository of information about an organization's application programs, databases, logical data models, users, and authorizations.

data partition

A VSAM data set that is contained within a partitioned table space.

data-partitioned secondary index (DPSI)

A secondary index that is partitioned according to the underlying data. Contrast with nonpartitioned secondary index.

data set instance number

A number that indicates the data set that contains the data for an object.

data sharing

A function of DB2 for z/OS that enables applications on different DB2 subsystems to read from and write to the same data concurrently.

data sharing group

A collection of one or more DB2 subsystems that directly access and change the same data while maintaining data integrity.

data sharing member

A DB2 subsystem that is assigned by XCF services to a data sharing group.

data source

A local or remote relational or non-relational data manager that is capable of supporting data access via an ODBC driver that supports the ODBC APIs. In the case of DB2 for z/OS, the data sources are always relational database managers.

data type

An attribute of columns, constants, variables, parameters, special registers, and the results of functions and expressions.

data warehouse

A system that provides critical business information to an organization. The data warehouse system cleanses the data for accuracy and currency, and then presents the data to decision makers so that they can interpret and use it effectively and efficiently.

DBA See database administrator.

DBAT See database access thread.

DB2 catalog

A collection of tables that are maintained by DB2 and contain descriptions of DB2 objects, such as tables, views, and indexes.

DBCLOB

See double-byte character large object.

DB2 command

An instruction to the DB2 subsystem that a user enters to start or stop DB2, to display information on current users, to start or stop databases, to display information on the status of databases, and so on.

DBCS See double-byte character set.

DBD See database descriptor.

DB2I See DB2 Interactive.

DBID See database identifier.

DB2 Interactive (DB2I)

An interactive service within DB2 that facilitates the execution of SQL statements, DB2 (operator) commands, and programmer commands, and the invocation of utilities.

DBMS

See database management system.

DBRM

See database request module.

DB2 thread

The database manager structure that describes an application's connection, traces its progress, processes resource functions, and delimits its accessibility to the database manager resources and services. Most DB2 for z/OS functions execute under a thread structure.

DCLGEN

See declarations generator.

DDF See distributed data facility.

deadlock

Unresolvable contention for the use of a resource, such as a table or an index.

declarations generator (DCLGEN)

A subcomponent of DB2 that generates SQL table declarations and COBOL, C, or PL/I data structure declarations that conform to the table. The declarations are generated from DB2 system catalog information.

declared temporary table

A non-persistent table that holds temporary data and is defined with the SQL statement DECLARE GLOBAL TEMPORARY TABLE. Information about declared temporary tables is not stored in the DB2 catalog and can be used only by the application process that issued the DECLARE statement. Contrast with created temporary table. See also temporary table.

default value

A predetermined value, attribute, or option that is assumed when no other

value is specified. A default value can be defined for column data in DB2 tables by specifying the DEFAULT keyword in an SQL statement that changes data (such as INSERT, UPDATE, and MERGE).

deferred embedded SQL

SQL statements that are neither fully static nor fully dynamic. These statements are embedded within an application and are prepared during the execution of the application.

deferred write

The process of asynchronously writing changed data pages to disk.

degree of parallelism

The number of concurrently executed operations that are initiated to process a query.

delete hole

The location on which a cursor is positioned when a row in a result table is refetched and the row no longer exists on the base table. See also update hole.

delete rule

The rule that tells DB2 what to do to a dependent row when a parent row is deleted. Delete rules include CASCADE, RESTRICT, SET NULL, or NO ACTION.

delete trigger

A trigger that is defined with the triggering delete SQL operation.

delimited identifier

A sequence of characters that are enclosed within escape characters such as double quotation marks (").

delimiter token

A string constant, a delimited identifier, an operator symbol, or any of the special characters that are shown in DB2 syntax diagrams.

denormalization

The intentional duplication of columns in multiple tables to increase data redundancy. Denormalization is sometimes necessary to minimize performance problems. Contrast with normalization.

dependent

An object (row, table, or table space) that has at least one parent. The object is also said to be a dependent (row, table, or

table space) of its parent. See also parent row, parent table, and parent table space.

dependent row

A row that contains a foreign key that matches the value of a primary key in the parent row.

dependent table

A table that is a dependent in at least one referential constraint.

descendent

An object that is a dependent of an object or is the dependent of a descendent of an object.

descendent row

A row that is dependent on another row, or a row that is a descendent of a dependent row.

descendent table

A table that is a dependent of another table, or a table that is a descendent of a dependent table.

deterministic function

A user-defined function whose result is dependent on the values of the input arguments. That is, successive invocations with the same input values produce the same answer. Sometimes referred to as a *not-variant* function. Contrast with nondeterministic function (sometimes called a *variant function*).

dimension

A data category such as time, products, or markets. The elements of a dimension are referred to as members. See also dimension table.

dimension table

The representation of a dimension in a star schema. Each row in a dimension table represents all of the attributes for a particular member of the dimension. See also dimension, star schema, and star join.

directory

The DB2 system database that contains internal objects such as database descriptors and skeleton cursor tables.

disk

A direct-access storage device that records data magnetically.

distinct type

A user-defined data type that is represented as an existing type (its source

type), but is considered to be a separate and incompatible type for semantic purposes.

distributed data

Data that resides on a DBMS other than the local system.

distributed data facility (DDF)

A set of DB2 components through which DB2 communicates with another relational database management system.

Distributed Relational Database Architecture (DRDA)

A connection protocol for distributed relational database processing that is used by IBM relational database products. DRDA includes protocols for communication between an application and a remote relational database management system, and for communication between relational database management systems. See also DRDA access.

DNS See domain name server.

DOCID

See document ID.

document ID

A value that uniquely identifies a row that contains an XML column. This value is stored with the row and never changes.

domain

The set of valid values for an attribute.

domain name

The name by which TCP/IP applications refer to a TCP/IP host within a TCP/IP network.

domain name server (DNS)

A special TCP/IP network server that manages a distributed directory that is used to map TCP/IP host names to IP addresses.

double-byte character large object (DBCLOB)

A graphic string data type in which a sequence of bytes represent double-byte characters that range in size from 0 bytes to 2 GB, less 1 byte. In general, DBCLOB values are used whenever a double-byte character string might exceed the limits of the VARGRAPHIC type.

double-byte character set (DBCS)

A set of characters, which are used by

national languages such as Japanese and Chinese, that have more symbols than can be represented by a single byte. Each character is 2 bytes in length. Contrast with single-byte character set and multibyte character set.

double-precision floating point number

A 64-bit approximate representation of a real number.

DPSI See data-partitioned secondary index.

drain The act of acquiring a locked resource by quiescing access to that object. Contrast with claim.

drain lock

A lock on a claim class that prevents a claim from occurring.

DRDA

See Distributed Relational Database Architecture.

DRDA access

An open method of accessing distributed data that you can use to connect to another database server to execute packages that were previously bound at the server location.

DSN

- The default DB2 subsystem name.
- The name of the TSO command processor of DB2.
- The first three characters of DB2 module and macro names.

dynamic cursor

A named control structure that an application program uses to change the size of the result table and the order of its rows after the cursor is opened. Contrast with static cursor.

dynamic dump

A dump that is issued during the execution of a program, usually under the control of that program.

dynamic SQL

SQL statements that are prepared and executed at run time. In dynamic SQL, the SQL statement is contained as a character string in a host variable or as a constant, and it is not precompiled.

EA-enabled table space

A table space or index space that is enabled for extended addressability and

that contains individual partitions (or pieces, for LOB table spaces) that are greater than 4 GB.

EB See exabyte.

EBCDIC

Extended binary coded decimal interchange code. An encoding scheme that is used to represent character data in the z/OS, VM, VSE, and iSeries environments. Contrast with ASCII and Unicode.

embedded SQL

SQL statements that are coded within an application program. See static SQL.

enabling-new-function mode (ENFM)

A transitional stage of the version-to-version migration process during which the DB2 subsystem or data sharing group is preparing to use the new functions of the new version. When in enabling-new-function mode, a DB2 data sharing group cannot coexist with members that are still at the prior version level. Fallback to a prior version is not supported, and most new functions of the new version are not available for use in enabling-new-function mode. Contrast with conversion mode, conversion mode*, enabling-new-function mode*, and new-function mode.

enabling-new-function mode* (ENFM*)

A transitional stage of the version-to-version migration process that applies to a DB2 subsystem or data sharing group that was in new-function mode (NFM) at one time. When in enabling-new-function mode*, a DB2 subsystem or data sharing group is preparing to use the new functions of the new version but cannot yet use them. A data sharing group that is in enabling-new-function mode* cannot coexist with members that are still at the prior version level. Fallback to a prior version is not supported. Contrast with conversion mode, conversion mode*, enabling-new-function mode, and new-function mode.

enclave

In Language Environment, an independent collection of routines, one of which is designated as the main routine.

An enclave is similar to a program or run unit. See also WLM enclave.

encoding scheme

A set of rules to represent character data (ASCII, EBCDIC, or Unicode).

ENFM See enabling-new-function mode.

ENFM*

See enabling-new-function mode*.

entity A person, object, or concept about which information is stored. In a relational database, entities are represented as tables. A database includes information about the entities in an organization or business, and their relationships to each other.

enumerated list

A set of DB2 objects that are defined with a LISTDEF utility control statement in which pattern-matching characters (*, %;, _, or ?) are not used.

environment

A collection of names of logical and physical resources that are used to support the performance of a function.

environment handle

A handle that identifies the global context for database access. All data that is pertinent to all objects in the environment is associated with this handle.

equijoin

A join operation in which the join-condition has the form *expression* = *expression*. See also join, full outer join, inner join, left outer join, outer join, and right outer join.

error page range

A range of pages that are considered to be physically damaged. DB2 does not allow users to access any pages that fall within this range.

escape character

The symbol, a double quotation (") for example, that is used to enclose an SQL delimited identifier.

exabyte

A unit of measure for processor, real and virtual storage capacities, and channel volume that has a value of 1 152 921 504 606 846 976 bytes or 2⁶⁰.

exception

An SQL operation that involves the EXCEPT set operator, which combines two result tables. The result of an exception operation consists of all of the rows that are in only one of the result tables.

exception table

A table that holds rows that violate referential constraints or check constraints that the CHECK DATA utility finds.

exclusive lock

A lock that prevents concurrently executing application processes from reading or changing data. Contrast with share lock.

executable statement

An SQL statement that can be embedded in an application program, dynamically prepared and executed, or issued interactively.

execution context

In SQLJ, a Java object that can be used to control the execution of SQL statements.

exit routine

A user-written (or IBM-provided default) program that receives control from DB2 to perform specific functions. Exit routines run as extensions of DB2.

expanding conversion

A process that occurs when the length of a converted string is greater than that of the source string. For example, this process occurs when an ASCII mixed-data string that contains DBCS characters is converted to an EBCDIC mixed-data string; the converted string is longer because shift codes are added.

explicit hierarchical locking

Locking that is used to make the parent-child relationship between resources known to IRLM. This kind of locking avoids global locking overhead when no inter-DB2 interest exists on a resource.

explicit privilege

A privilege that has a name and is held as the result of an SQL GRANT statement and revoked as the result of an SQL REVOKE statement. For example, the SELECT privilege.

exposed name

A correlation name or a table or view name for which a correlation name is not specified.

expression

An operand or a collection of operators and operands that yields a single value.

Extended Recovery Facility (XRF)

A facility that minimizes the effect of failures in z/OS, VTAM, the host processor, or high-availability applications during sessions between high-availability applications and designated terminals. This facility provides an alternative subsystem to take over sessions from the failing subsystem.

Extensible Markup Language (XML)

A standard metalanguage for defining markup languages that is a subset of Standardized General Markup Language (SGML).

external function

A function that has its functional logic implemented in a programming language application that resides outside the database, in the file system of the database server. The association of the function with the external code application is specified by the EXTERNAL clause in the CREATE FUNCTION statement. External functions can be classified as external scalar functions and external table functions. Contrast with sourced function, built-in function, and SQL function.

external procedure

A procedure that has its procedural logic implemented in an external programming language application. The association of the procedure with the external application is specified by a CREATE PROCEDURE statement with a LANGUAGE clause that has a value other than SQL and an EXTERNAL clause that implicitly or explicitly specifies the name of the external application. Contrast with external SQL procedure and native SQL procedure.

external routine

A user-defined function or stored procedure that is based on code that is written in an external programming language.

external SQL procedure	orientation (for example, NEXT ROWSET, LAST ROWSET, or ROWSET STARTING AT ABSOLUTE <i>n</i>).
An SQL procedure that is processed using a generated C program that is a representation of the procedure. When an external SQL procedure is called, the C program representation of the procedure is executed in a stored procedures address space. Contrast with external procedure and native SQL procedure.	
failed member state	field procedure
A state of a member of a data sharing group in which the member's task, address space, or z/OS system terminates before the state changes from active to quiesced.	A user-written exit routine that is designed to receive a single value and transform (encode or decode) it in any way the user can specify.
fallback	file reference variable
The process of returning to a previous release of DB2 after attempting or completing migration to a current release. Fallback is supported only from a subsystem that is in conversion mode.	A host variable that is declared with one of the derived data types (BLOB_FILE, CLOB_FILE, DBCLOB_FILE); file reference variables direct the reading of a LOB from a file or the writing of a LOB into a file.
false global lock contention	filter factor
A contention indication from the coupling facility that occurs when multiple lock names are hashed to the same indicator and when no real contention exists.	A number between zero and one that estimates the proportion of rows in a table for which a predicate is true.
fan set	fixed-length string
A direct physical access path to data, which is provided by an index, hash, or link; a fan set is the means by which DB2 supports the ordering of data.	A character, graphic, or binary string whose length is specified and cannot be changed. Contrast with varying-length string.
federated database	FlashCopy
The combination of a DB2 server (in Linux, UNIX, and Windows environments) and multiple data sources to which the server sends queries. In a federated database system, a client application can use a single SQL statement to join data that is distributed across multiple database management systems and can view the data as if it were local.	A function on the IBM Enterprise Storage Server that can, in conjunction with the BACKUP SYSTEM utility, create a point-in-time copy of data while an application is running.
fetch orientation	foreign key
The specification of the desired placement of the cursor as part of a FETCH statement. The specification can be before or after the rows of the result table (with BEFORE or AFTER). The specification can also have either a single-row fetch orientation (for example, NEXT, LAST, or ABSOLUTE <i>n</i>) or a rowset fetch	A column or set of columns in a dependent table of a constraint relationship. The key must have the same number of columns, with the same descriptions, as the primary key of the parent table. Each foreign key value must either match a parent key value in the related parent table or be null.
	forest An ordered set of subtrees of XML nodes.
	forward log recovery
	The third phase of restart processing during which DB2 processes the log in a forward direction to apply all REDO log records.
	free space
	The total amount of unused space in a page; that is, the space that is not used to store records or control information is free space.

full outer join

The result of a join operation that includes the matched rows of both tables that are being joined and preserves the unmatched rows of both tables. See also join, equijoin, inner join, left outer join, outer join, and right outer join.

fullselect

A subselect, a fullselect in parentheses, or a number of both that are combined by set operators. Fullselect specifies a result table. If a set operator is not used, the result of the fullselect is the result of the specified subselect or fullselect.

fully escaped mapping

A mapping from an SQL identifier to an XML name when the SQL identifier is a column name.

function

A mapping, which is embodied as a program (the function body) that is invocable by means of zero or more input values (arguments) to a single value (the result). See also aggregate function and scalar function.

Functions can be user-defined, built-in, or generated by DB2. (See also built-in function, cast function, external function, sourced function, SQL function, and user-defined function.)

function definer

The authorization ID of the owner of the schema of the function that is specified in the CREATE FUNCTION statement.

function package

A package that results from binding the DBRM for a function program.

function package owner

The authorization ID of the user who binds the function program's DBRM into a function package.

function signature

The logical concatenation of a fully qualified function name with the data types of all of its parameters.

GB Gigabyte. A value of (1 073 741 824 bytes).

GBP See group buffer pool.

GBP-dependent

The status of a page set or page set partition that is dependent on the group

buffer pool. Either read/write interest is active among DB2 subsystems for this page set, or the page set has changed pages in the group buffer pool that have not yet been cast out to disk.

generalized trace facility (GTF)

A z/OS service program that records significant system events such as I/O interrupts, SVC interrupts, program interrupts, or external interrupts.

generic resource name

A name that VTAM uses to represent several application programs that provide the same function in order to handle session distribution and balancing in a Sysplex environment.

getpage

An operation in which DB2 accesses a data page.

global lock

A lock that provides concurrency control within and among DB2 subsystems. The scope of the lock is across all DB2 subsystems of a data sharing group.

global lock contention

Conflicts on locking requests between different DB2 members of a data sharing group when those members are trying to serialize shared resources.

governor

See resource limit facility.

graphic string

A sequence of DBCS characters. Graphic data can be further classified as GRAPHIC, VARGRAPHIC, or DBCLOB.

GRECP

See group buffer pool recovery pending.

gross lock

The *shared*, *update*, or *exclusive* mode locks on a table, partition, or table space.

group buffer pool duplexing

The ability to write data to two instances of a group buffer pool structure: a primary group buffer pool and a secondary group buffer pool. z/OS publications refer to these instances as the "old" (for primary) and "new" (for secondary) structures.

group buffer pool (GBP)

A coupling facility cache structure that is

used by a data sharing group to cache data and to ensure that the data is consistent for all members.

group buffer pool recovery pending (GRECP)

The state that exists after the buffer pool for a data sharing group is lost. When a page set is in this state, changes that are recorded in the log must be applied to the affected page set before the page set can be used.

group level

The release level of a data sharing group, which is established when the first member migrates to a new release.

group name

The z/OS XCF identifier for a data sharing group.

group restart

A restart of at least one member of a data sharing group after the loss of either locks or the shared communications area.

GTF See generalized trace facility.

handle

In DB2 ODBC, a variable that refers to a data structure and associated resources. See also statement handle, connection handle, and environment handle.

help panel

A screen of information that presents tutorial text to assist a user at the workstation or terminal.

heuristic damage

The inconsistency in data between one or more participants that results when a heuristic decision to resolve an indoubt LUW at one or more participants differs from the decision that is recorded at the coordinator.

heuristic decision

A decision that forces indoubt resolution at a participant by means other than automatic resynchronization between coordinator and participant.

histogram statistics

A way of summarizing data distribution. This technique divides up the range of possible values in a data set into intervals, such that each interval contains approximately the same percentage of the values. A set of statistics are collected for each interval.

hole A row of the result table that cannot be accessed because of a delete or an update that has been performed on the row. See also delete hole and update hole.

home address space

The area of storage that z/OS currently recognizes as *dispatched*.

host The set of programs and resources that are available on a given TCP/IP instance.

host expression

A Java variable or expression that is referenced by SQL clauses in an SQLJ application program.

host identifier

A name that is declared in the host program.

host language

A programming language in which you can embed SQL statements.

host program

An application program that is written in a host language and that contains embedded SQL statements.

host structure

In an application program, a structure that is referenced by embedded SQL statements.

host variable

In an application program written in a host language, an application variable that is referenced by embedded SQL statements.

host variable array

An array of elements, each of which corresponds to a value for a column. The dimension of the array determines the maximum number of rows for which the array can be used.

IBM System z9 Integrated Processor (zIIP)

A specialized processor that can be used for some DB2 functions.

IDCAMS

An IBM program that is used to process access method services commands. It can be invoked as a job or jobstep, from a TSO terminal, or from within a user's application program.

IDCAMS LISTCAT

A facility for obtaining information that is contained in the access method services catalog.

identity column

| A column that provides a way for DB2 to
| automatically generate a numeric value
| for each row. Identity columns are
| defined with the AS IDENTITY clause.
| Uniqueness of values can be ensured by
| defining a unique index that contains
| only the identity column. A table can
| have no more than one identity column.

IFCID See instrumentation facility component identifier.

IFI See instrumentation facility interface.

IFI call

An invocation of the instrumentation facility interface (IFI) by means of one of its defined functions.

image copy

An exact reproduction of all or part of a table space. DB2 provides utility programs to make full image copies (to copy the entire table space) or incremental image copies (to copy only those pages that have been modified since the last image copy).

IMS attachment facility

A DB2 subcomponent that uses z/OS subsystem interface (SSI) protocols and cross-memory linkage to process requests from IMS to DB2 and to coordinate resource commitment.

in-abort

A status of a unit of recovery. If DB2 fails after a unit of recovery begins to be rolled back, but before the process is completed, DB2 continues to back out the changes during restart.

in-commit

A status of a unit of recovery. If DB2 fails after beginning its phase 2 commit processing, it "knows," when restarted, that changes made to data are consistent. Such units of recovery are termed *in-commit*.

independent

An object (row, table, or table space) that is neither a parent nor a dependent of another object.

index A set of pointers that are logically ordered by the values of a key. Indexes can provide faster access to data and can enforce uniqueness on the rows in a table.

index-controlled partitioning

A type of partitioning in which partition boundaries for a partitioned table are controlled by values that are specified on the CREATE INDEX statement. Partition limits are saved in the LIMITKEY column of the SYSIBM.SYSINDEXPART catalog table.

index key

The set of columns in a table that is used to determine the order of index entries.

index partition

A VSAM data set that is contained within a partitioning index space.

index space

A page set that is used to store the entries of one index.

indicator column

A 4-byte value that is stored in a base table in place of a LOB column.

indicator variable

A variable that is used to represent the null value in an application program. If the value for the selected column is null, a negative value is placed in the indicator variable.

indoubt

A status of a unit of recovery. If DB2 fails after it has finished its phase 1 commit processing and before it has started phase 2, only the commit coordinator knows if an individual unit of recovery is to be committed or rolled back. At restart, if DB2 lacks the information it needs to make this decision, the status of the unit of recovery is *indoubt* until DB2 obtains this information from the coordinator. More than one unit of recovery can be *indoubt* at restart.

indoubt resolution

The process of resolving the status of an *indoubt* logical unit of work to either the committed or the rollback state.

inflight

A status of a unit of recovery. If DB2 fails before its unit of recovery completes phase 1 of the commit process, it merely

backs out the updates of its unit of recovery at restart. These units of recovery are termed *inflight*.

inheritance

The passing downstream of class resources or attributes from a parent class in the class hierarchy to a child class.

initialization file

For DB2 ODBC applications, a file containing values that can be set to adjust the performance of the database manager.

inline copy

A copy that is produced by the LOAD or REORG utility. The data set that the inline copy produces is logically equivalent to a full image copy that is produced by running the COPY utility with read-only access (SHRLEVEL REFERENCE).

inner join

The result of a join operation that includes only the matched rows of both tables that are being joined. See also join, equijoin, full outer join, left outer join, outer join, and right outer join.

inoperative package

A package that cannot be used because one or more user-defined functions or procedures that the package depends on were dropped. Such a package must be explicitly rebound. Contrast with invalid package.

insensitive cursor

A cursor that is not sensitive to inserts, updates, or deletes that are made to the underlying rows of a result table after the result table has been materialized.

insert trigger

A trigger that is defined with the triggering SQL operation, an insert.

install The process of preparing a DB2 subsystem to operate as a z/OS subsystem.

INSTEAD OF trigger

A trigger that is associated with a single view and is activated by an insert, update, or delete operation on the view and that can define how to propagate the insert, update, or delete operation on the view to the underlying tables of the view. Contrast with BEFORE trigger and AFTER trigger.

instrumentation facility component identifier (IFCID)

A value that names and identifies a trace record of an event that can be traced. As a parameter on the START TRACE and MODIFY TRACE commands, it specifies that the corresponding event is to be traced.

instrumentation facility interface (IFI)

A programming interface that enables programs to obtain online trace data about DB2, to submit DB2 commands, and to pass data to DB2.

Interactive System Productivity Facility (ISPF)

An IBM licensed program that provides interactive dialog services in a z/OS environment.

inter-DB2 R/W interest

A property of data in a table space, index, or partition that has been opened by more than one member of a data sharing group and that has been opened for writing by at least one of those members.

intermediate database server

The target of a request from a local application or a remote application requester that is forwarded to another database server.

internal resource lock manager (IRLM)

A z/OS subsystem that DB2 uses to control communication and database locking.

internationalization

The support for an encoding scheme that is able to represent the code points of characters from many different geographies and languages. To support all geographies, the Unicode standard requires more than 1 byte to represent a single character. See also Unicode.

intersection

An SQL operation that involves the INTERSECT set operator, which combines two result tables. The result of an intersection operation consists of all of the rows that are in both result tables.

invalid package

A package that depends on an object (other than a user-defined function) that is dropped. Such a package is implicitly rebound on invocation. Contrast with inoperative package.

IP address	A value that uniquely identifies a TCP/IP host.	columns, or an expression that is identified in the description of a table, index, or referential constraint. The same column or expression can be part of more than one key.
IRLM	See internal resource lock manager.	
isolation level	The degree to which a unit of work is isolated from the updating operations of other units of work. See also cursor stability, read stability, repeatable read, and uncommitted read.	
ISPF	See Interactive System Productivity Facility.	
iterator	In SQLJ, an object that contains the result set of a query. An iterator is equivalent to a cursor in other host languages.	
iterator declaration clause	In SQLJ, a statement that generates an iterator declaration class. An iterator is an object of an iterator declaration class.	
JAR	See Java Archive.	
Java Archive (JAR)	A file format that is used for aggregating many files into a single file.	
JDBC	A Sun Microsystems database application programming interface (API) for Java that allows programs to access database management systems by using callable SQL.	
join	A relational operation that allows retrieval of data from two or more tables based on matching column values. See also equijoin, full outer join, inner join, left outer join, outer join, and right outer join.	
KB	Kilobyte. A value of 1024 bytes.	
Kerberos	A network authentication protocol that is designed to provide strong authentication for client/server applications by using secret-key cryptography.	
Kerberos ticket	A transparent application mechanism that transmits the identity of an initiating principal to its target. A simple ticket contains the principal's identity, a session key, a timestamp, and other information, which is sealed using the target's secret key.	
key	A column, an ordered collection of	
		key-sequenced data set (KSDS) A VSAM file or data set whose records are loaded in key sequence and controlled by an index.
		KSDS See key-sequenced data set.
		large object (LOB) A sequence of bytes representing bit data, single-byte characters, double-byte characters, or a mixture of single- and double-byte characters. A LOB can be up to 2 GB minus 1 byte in length. See also binary large object, character large object, and double-byte character large object.
		last agent optimization An optimized commit flow for either presumed-nothing or presumed-abort protocols in which the last agent, or final participant, becomes the commit coordinator. This flow saves at least one message.
		latch A DB2 mechanism for controlling concurrent events or the use of system resources.
		LCID See log control interval definition.
		LDS See linear data set.
		leaf page An index page that contains pairs of keys and RIDs and that points to actual data. Contrast with nonleaf page.
		left outer join The result of a join operation that includes the matched rows of both tables that are being joined, and that preserves the unmatched rows of the first table. See also join, equijoin, full outer join, inner join, outer join, and right outer join.
		limit key The highest value of the index key for a partition.
		linear data set (LDS) A VSAM data set that contains data but no control information. A linear data set can be accessed as a byte-addressable string in virtual storage.

linkage editor

A computer program for creating load modules from one or more object modules or load modules by resolving cross references among the modules and, if necessary, adjusting addresses.

link-edit

The action of creating a loadable computer program using a linkage editor.

list

A type of object, which DB2 utilities can process, that identifies multiple table spaces, multiple index spaces, or both. A list is defined with the LISTDEF utility control statement.

list structure

A coupling facility structure that lets data be shared and manipulated as elements of a queue.

L-lock See logical lock.

load module

A program unit that is suitable for loading into main storage for execution. The output of a linkage editor.

LOB See large object.

LOB locator

A mechanism that allows an application program to manipulate a large object value in the database system. A LOB locator is a fullword integer value that represents a single LOB value. An application program retrieves a LOB locator into a host variable and can then apply SQL operations to the associated LOB value using the locator.

LOB lock

A lock on a LOB value.

LOB table space

A table space that contains all the data for a particular LOB column in the related base table.

local

A way of referring to any object that the local DB2 subsystem maintains. A *local table*, for example, is a table that is maintained by the local DB2 subsystem. Contrast with remote.

locale

The definition of a subset of a user's environment that combines a CCSID and characters that are defined for a specific language and country.

local lock

A lock that provides intra-DB2 concurrency control, but not inter-DB2 concurrency control; that is, its scope is a single DB2.

local subsystem

The unique relational DBMS to which the user or application program is directly connected (in the case of DB2, by one of the DB2 attachment facilities).

location

The unique name of a database server. An application uses the location name to access a DB2 database server. A database alias can be used to override the location name when accessing a remote server.

location alias

Another name by which a database server identifies itself in the network. Applications can use this name to access a DB2 database server.

lock

A means of controlling concurrent events or access to data. DB2 locking is performed by the IRLM.

lock duration

The interval over which a DB2 lock is held.

lock escalation

The promotion of a lock from a row, page, or LOB lock to a table space lock because the number of page locks that are concurrently held on a given resource exceeds a preset limit.

locking

The process by which the integrity of data is ensured. Locking prevents concurrent users from accessing inconsistent data. See also claim, drain, and latch.

lock mode

A representation for the type of access that concurrently running programs can have to a resource that a DB2 lock is holding.

lock object

The resource that is controlled by a DB2 lock.

lock promotion

The process of changing the size or mode of a DB2 lock to a higher, more restrictive level.

lock size

The amount of data that is controlled by a DB2 lock on table data; the value can be a row, a page, a LOB, a partition, a table, or a table space.

lock structure

A coupling facility data structure that is composed of a series of lock entries to support shared and exclusive locking for logical resources.

log

A collection of records that describe the events that occur during DB2 execution and that indicate their sequence. The information thus recorded is used for recovery in the event of a failure during DB2 execution.

log control interval definition

A suffix of the physical log record that tells how record segments are placed in the physical control interval.

logical claim

A claim on a logical partition of a nonpartitioning index.

logical index partition

The set of all keys that reference the same data partition.

logical lock (L-lock)

The lock type that transactions use to control intra- and inter-DB2 data concurrency between transactions. Contrast with physical lock (P-lock).

logically complete

A state in which the concurrent copy process is finished with the initialization of the target objects that are being copied. The target objects are available for update.

logical page list (LPL)

A list of pages that are in error and that cannot be referenced by applications until the pages are recovered. The page is in *logical error* because the actual media (coupling facility or disk) might not contain any errors. Usually a connection to the media has been lost.

logical partition

A set of key or RID pairs in a nonpartitioning index that are associated with a particular partition.

logical recovery pending (LRECP)

The state in which the data and the index keys that reference the data are inconsistent.

logical unit (LU)

An access point through which an application program accesses the SNA network in order to communicate with another application program. See also LU name.

logical unit of work identifier (LUWID)

A name that uniquely identifies a thread within a network. This name consists of a fully-qualified LU network name, an LUW instance number, and an LUW sequence number.

logical unit of work

The processing that a program performs between synchronization points.

log initialization

The first phase of restart processing during which DB2 attempts to locate the current end of the log.

log record header (LRH)

A prefix, in every log record, that contains control information.

log record sequence number (LRSN)

An identifier for a log record that is associated with a data sharing member. DB2 uses the LRSN for recovery in the data sharing environment.

log truncation

A process by which an explicit starting RBA is established. This RBA is the point at which the next byte of log data is to be written.

LPL See logical page list.

LRECP

See logical recovery pending.

LRH

See log record header.

LRSN

See log record sequence number.

LU

See logical unit.

LU name

Logical unit name, which is the name by which VTAM refers to a node in a network.

LUW

See logical unit of work.

LUWID

See logical unit of work identifier.

mapping table

A table that the REORG utility uses to map the associations of the RIDs of data records in the original copy and in the shadow copy. This table is created by the user.

mass delete

The deletion of all rows of a table.

materialize

- The process of putting rows from a view or nested table expression into a work file for additional processing by a query.
- The placement of a LOB value into contiguous storage. Because LOB values can be very large, DB2 avoids materializing LOB data until doing so becomes absolutely necessary.

materialized query table

A table that is used to contain information that is derived and can be summarized from one or more source tables. Contrast with base table.

MB Megabyte (1 048 576 bytes).

MBCS See multibyte character set.

member name

The z/OS XCF identifier for a particular DB2 subsystem in a data sharing group.

menu A displayed list of available functions for selection by the operator. A menu is sometimes called a *menu panel*.

metalanguage

A language that is used to create other specialized languages.

migration

The process of converting a subsystem with a previous release of DB2 to an updated or current release. In this process, you can acquire the functions of the updated or current release without losing the data that you created on the previous release.

mixed data string

A character string that can contain both single-byte and double-byte characters.

mode name

A VTAM name for the collection of

physical and logical characteristics and attributes of a session.

modify locks

An L-lock or P-lock with a MODIFY attribute. A list of these active locks is kept at all times in the coupling facility lock structure. If the requesting DB2 subsystem fails, that DB2 subsystem's modify locks are converted to retained locks.

multibyte character set (MBCS)

A character set that represents single characters with more than a single byte. UTF-8 is an example of an MBCS. Characters in UTF-8 can range from 1 to 4 bytes in DB2. Contrast with single-byte character set and double-byte character set. See also Unicode.

multidimensional analysis

The process of assessing and evaluating an enterprise on more than one level.

Multiple Virtual Storage (MVS)

An element of the z/OS operating system. This element is also called the Base Control Program (BCP).

multisite update

Distributed relational database processing in which data is updated in more than one location within a single unit of work.

multithreading

Multiple TCBs that are executing one copy of DB2 ODBC code concurrently (sharing a processor) or in parallel (on separate central processors).

MVS See Multiple Virtual Storage.

native SQL procedure

An SQL procedure that is processed by converting the procedural statements to a native representation that is stored in the database directory, as is done with other SQL statements. When a native SQL procedure is called, the native representation is loaded from the directory, and DB2 executes the procedure. Contrast with external procedure and external SQL procedure.

nested table expression

A fullselect in a FROM clause (surrounded by parentheses).

network identifier (NID)

The network ID that is assigned by IMS

or CICS, or if the connection type is RRSAF, the RRS unit of recovery ID (URID).

| **new-function mode (NFM)**

| The normal mode of operation that exists
| after successful completion of a
| version-to-version migration. At this
| stage, all new functions of the new
| version are available for use. A DB2 data
| sharing group cannot coexist with
| members that are still at the prior version
| level, and fallback to a prior version is
| not supported. Contrast with conversion
| mode, conversion mode*,
| enabling-new-function mode, and
| enabling-new-function mode*.

| **NFM** See new-function mode.

NID See network identifier.

| **node ID index**

| See XML node ID index.

nondeterministic function

A user-defined function whose result is not solely dependent on the values of the input arguments. That is, successive invocations with the same argument values can produce a different answer. This type of function is sometimes called a *variant* function. Contrast with deterministic function (sometimes called a *not-variant function*).

nonleaf page

A page that contains keys and page numbers of other pages in the index (either leaf or nonleaf pages). Nonleaf pages never point to actual data. Contrast with leaf page.

nonpartitioned index

An index that is not physically partitioned. Both partitioning indexes and secondary indexes can be nonpartitioned.

nonpartitioned secondary index (NPSI)

An index on a partitioned table space that is not the partitioning index and is not partitioned. Contrast with data-partitioned secondary index.

nonpartitioning index

See secondary index.

nonscrollable cursor

A cursor that can be moved only in a

forward direction. Nonscrollable cursors are sometimes called forward-only cursors or serial cursors.

normalization

A key step in the task of building a logical relational database design. Normalization helps you avoid redundancies and inconsistencies in your data. An entity is normalized if it meets a set of constraints for a particular normal form (first normal form, second normal form, and so on). Contrast with denormalization.

not-variant function

See deterministic function.

NPSI See nonpartitioned secondary index.

NUL The null character ('\0'), which is represented by the value X'00'. In C, this character denotes the end of a string.

null A special value that indicates the absence of information.

null terminator

In C, the value that indicates the end of a string. For EBCDIC, ASCII, and Unicode UTF-8 strings, the null terminator is a single-byte value (X'00'). For Unicode UTF-16 or UCS-2 (wide) strings, the null terminator is a double-byte value (X'0000').

ODBC

See Open Database Connectivity.

ODBC driver

A dynamically-linked library (DLL) that implements ODBC function calls and interacts with a data source.

| **OLAP** See online analytical processing.

| **online analytical processing (OLAP)**

| The process of collecting data from one or
| many sources; transforming and
| analyzing the consolidated data quickly
| and interactively; and examining the
| results across different dimensions of the
| data by looking for patterns, trends, and
| exceptions within complex relationships
| of that data.

Open Database Connectivity (ODBC)

A Microsoft database application programming interface (API) for C that allows access to database management systems by using callable SQL. ODBC

does not require the use of an SQL preprocessor. In addition, ODBC provides an architecture that lets users add modules called *database drivers*, which link the application to their choice of database management systems at run time. This means that applications no longer need to be directly linked to the modules of all the database management systems that are supported.

ordinary identifier

An uppercase letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. An ordinary identifier must not be a reserved word.

ordinary token

A numeric constant, an ordinary identifier, a host identifier, or a keyword.

originating task

In a parallel group, the primary agent that receives data from other execution units (referred to as *parallel tasks*) that are executing portions of the query in parallel.

outer join

The result of a join operation that includes the matched rows of both tables that are being joined and preserves some or all of the unmatched rows of the tables that are being joined. See also join, equijoin, full outer join, inner join, left outer join, and right outer join.

overloaded function

A function name for which multiple function instances exist.

package

An object containing a set of SQL statements that have been statically bound and that is available for processing. A package is sometimes also called an *application package*.

package list

An ordered list of package names that may be used to extend an application plan.

package name

The name of an object that is used for an application package or an SQL procedure package. An application package is a bound version of a database request module (DBRM) that is created by a

BIND PACKAGE or REBIND PACKAGE command. An SQL procedural language package is created by a CREATE or ALTER PROCEDURE statement for a native SQL procedure. The name of a package consists of a location name, a collection ID, a package ID, and a version ID.

page

A unit of storage within a table space (4 KB, 8 KB, 16 KB, or 32 KB) or index space (4 KB, 8 KB, 16 KB, or 32 KB). In a table space, a page contains one or more rows of a table. In a LOB or XML table space, a LOB or XML value can span more than one page, but no more than one LOB or XML value is stored on a page.

page set

Another way to refer to a table space or index space. Each page set consists of a collection of VSAM data sets.

page set recovery pending (PSRCP)

A restrictive state of an index space. In this case, the entire page set must be recovered. Recovery of a logical part is prohibited.

panel

A predefined display image that defines the locations and characteristics of display fields on a display surface (for example, a *menu panel*).

parallel complex

A cluster of machines that work together to handle multiple transactions and applications.

parallel group

A set of consecutive operations that execute in parallel and that have the same number of parallel tasks.

parallel I/O processing

A form of I/O processing in which DB2 initiates multiple concurrent requests for a single user query and performs I/O processing concurrently (in *parallel*) on multiple data partitions.

parallelism assistant

In Sysplex query parallelism, a DB2 subsystem that helps to process parts of a parallel query that originates on another DB2 subsystem in the data sharing group.

parallelism coordinator

In Sysplex query parallelism, the DB2 subsystem from which the parallel query originates.

Parallel Sysplex

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.

parallel task

The execution unit that is dynamically created to process a query in parallel. A parallel task is implemented by a z/OS service request block.

parameter marker

A question mark (?) that appears in a statement string of a dynamic SQL statement. The question mark can appear where a variable could appear if the statement string were a static SQL statement.

parameter-name

An SQL identifier that designates a parameter in a routine that is written by a user. Parameter names are required for SQL procedures and SQL functions, and they are used in the body of the routine to refer to the values of the parameters. Parameter names are optional for external routines.

parent key

A primary key or unique key in the parent table of a referential constraint. The values of a parent key determine the valid values of the foreign key in the referential constraint.

parent lock

For explicit hierarchical locking, a lock that is held on a resource that might have child locks that are lower in the hierarchy. A parent lock is usually the table space lock or the partition intent lock. See also child lock.

parent row

A row whose primary key value is the foreign key value of a dependent row.

parent table

A table whose primary key is referenced by the foreign key of a dependent table.

parent table space

A table space that contains a parent table. A table space containing a dependent of that table is a dependent table space.

participant

An entity other than the commit coordinator that takes part in the commit process. The term participant is synonymous with agent in SNA.

partition

A portion of a page set. Each partition corresponds to a single, independently extendable data set. The maximum size of a partition depends on the number of partitions in the partitioned page set. All partitions of a given page set have the same maximum size.

partition-by-growth table space

A table space whose size can grow to accommodate data growth. DB2 for z/OS manages partition-by-growth table spaces by automatically adding new data sets when the database needs more space to satisfy an insert operation. Contrast with range-partitioned table space. See also universal table space.

partitioned data set (PDS)

A data set in disk storage that is divided into partitions, which are called members. Each partition can contain a program, part of a program, or data. A program library is an example of a partitioned data set.

partitioned index

An index that is physically partitioned. Both partitioning indexes and secondary indexes can be partitioned.

partitioned page set

A partitioned table space or an index space. Header pages, space map pages, data pages, and index pages reference data only within the scope of the partition.

partitioned table space

A table space that is based on a single table and that is subdivided into partitions, each of which can be processed independently by utilities. Contrast with segmented table space and universal table space.

partitioning index

An index in which the leftmost columns

	are the partitioning columns of the table. The index can be partitioned or nonpartitioned.				decimal number (called the <i>size</i> in the C language). In the C language, the number of digits to the right of the decimal point (called the scale in SQL). The DB2 information uses the SQL terms.
partner logical unit	An access point in the SNA network that is connected to the local DB2 subsystem by way of a VTAM conversation.	precompilation			A processing of application programs containing SQL statements that takes place before compilation. SQL statements are replaced with statements that are recognized by the host language compiler. Output from this precompilation includes source code that can be submitted to the compiler and the database request module (DBRM) that is input to the bind process.
path	See SQL path.	predicate			An element of a search condition that expresses or implies a comparison operation.
PDS	See partitioned data set.	prefix			A code at the beginning of a message or record.
physical consistency	The state of a page that is not in a partially changed state.	preformat			The process of preparing a VSAM linear data set for DB2 use, by writing specific data patterns.
physical lock (P-lock)	A type of lock that DB2 acquires to provide consistency of data that is cached in different DB2 subsystems. Physical locks are used only in data sharing environments. Contrast with logical lock (L-lock).	prepare			The first phase of a two-phase commit process in which all participants are requested to prepare for commit.
physically complete	The state in which the concurrent copy process is completed and the output data set has been created.	prepared SQL statement			A named object that is the executable form of an SQL statement that has been processed by the PREPARE statement.
piece	A data set of a nonpartitioned page set.	primary authorization ID			The authorization ID that is used to identify the application process to DB2.
plan	See application plan.	primary group buffer pool			For a duplexed group buffer pool, the structure that is used to maintain the coherency of cached data. This structure is used for page registration and cross-invalidation. The z/OS equivalent is <i>old</i> structure. Compare with secondary group buffer pool.
plan allocation	The process of allocating DB2 resources to a plan in preparation for execution.	primary index			An index that enforces the uniqueness of a primary key.
plan member	The bound copy of a DBRM that is identified in the member clause.	primary key			In a relational database, a unique, nonnull key that is part of the definition of a
plan name	The name of an application plan.				
P-lock	See physical lock.				
point of consistency	A time when all recoverable data that an application accesses is consistent with other data. The term point of consistency is synonymous with sync point or commit point.				
policy	See CFRM policy.				
postponed abort UR	A unit of recovery that was inflight or in-abort, was interrupted by system failure or cancellation, and did not complete backout during restart.				
precision	In SQL, the total number of digits in a				

table. A table cannot be defined as a parent unless it has a unique key or primary key.

principal

An entity that can communicate securely with another entity. In Kerberos, principals are represented as entries in the Kerberos registry database and include users, servers, computers, and others.

principal name

The name by which a principal is known to the DCE security services.

privilege

The capability of performing a specific function, sometimes on a specific object. See also explicit privilege.

privilege set

- For the installation SYSADM ID, the set of all possible privileges.
- For any other authorization ID, including the PUBLIC authorization ID, the set of all privileges that are recorded for that ID in the DB2 catalog.

process

In DB2, the unit to which DB2 allocates resources and locks. Sometimes called an application process, a process involves the execution of one or more programs. The execution of an SQL statement is always associated with some process. The means of initiating and terminating a process are dependent on the environment.

program

A single, compilable collection of executable statements in a programming language.

program temporary fix (PTF)

A solution or bypass of a problem that is diagnosed as a result of a defect in a current unaltered release of a licensed program. An authorized program analysis report (APAR) fix is corrective service for an existing problem. A PTF is preventive service for problems that might be encountered by other users of the product. A PTF is *temporary*, because a permanent fix is usually not incorporated into the product until its next release.

protected conversation

A VTAM conversation that supports two-phase commit flows.

PSRCP

See page set recovery pending.

PTF

See program temporary fix.

QSAM

See queued sequential access method.

query

A component of certain SQL statements that specifies a result table.

query block

The part of a query that is represented by one of the FROM clauses. Each FROM clause can have multiple query blocks, depending on DB2 processing of the query.

query CP parallelism

Parallel execution of a single query, which is accomplished by using multiple tasks. See also Sysplex query parallelism.

query I/O parallelism

Parallel access of data, which is accomplished by triggering multiple I/O requests within a single query.

queued sequential access method (QSAM)

An extended version of the basic sequential access method (BSAM). When this method is used, a queue of data blocks is formed. Input data blocks await processing, and output data blocks await transfer to auxiliary storage or to an output device.

quiesce point

A point at which data is consistent as a result of running the DB2 QUIESCE utility.

RACF

Resource Access Control Facility. A component of the z/OS Security Server.

range-partitioned table space

A type of universal table space that is based on partitioning ranges and that contains a single table. Contrast with partition-by-growth table space. See also universal table space.

RBA

See relative byte address.

RCT

See resource control table.

RDO

See resource definition online.

read stability (RS)

An isolation level that is similar to repeatable read but does not completely isolate an application process from all other concurrently executing application

	processes. See also cursor stabilityrepeatable read, and uncommitted read.	
rebind	The creation of a new application plan for an application program that has been bound previously. If, for example, you have added an index for a table that your application accesses, you must rebind the application in order to take advantage of that index.	
rebuild	The process of reallocating a coupling facility structure. For the shared communications area (SCA) and lock structure, the structure is repopulated; for the group buffer pool, changed pages are usually cast out to disk, and the new structure is populated only with changed pages that were not successfully cast out.	
record	The storage representation of a row or other data.	
record identifier (RID)	A unique identifier that DB2 uses to identify a row of data in a table. Compare with row identifier.	
record identifier (RID) pool	An area of main storage that is used for sorting record identifiers during list-prefetch processing.	
record length	The sum of the length of all the columns in a table, which is the length of the data as it is physically stored in the database. Records can be fixed length or varying length, depending on how the columns are defined. If all columns are fixed-length columns, the record is a fixed-length record. If one or more columns are varying-length columns, the record is a varying-length record.	
Recoverable Resource Manager Services attachment facility (RRSAF)	A DB2 subcomponent that uses Resource Recovery Services to coordinate resource commitment between DB2 and all other resource managers that also use RRS in a z/OS system.	
recovery	The process of rebuilding databases after a system failure.	
		recovery log A collection of records that describes the events that occur during DB2 execution and indicates their sequence. The recorded information is used for recovery in the event of a failure during DB2 execution.
		recovery manager A subcomponent that supplies coordination services that control the interaction of DB2 resource managers during commit, abort, checkpoint, and restart processes. The recovery manager also supports the recovery mechanisms of other subsystems (for example, IMS) by acting as a participant in the other subsystem's process for protecting data that has reached a point of consistency. A coordinator or a participant (or both), in the execution of a two-phase commit, that can access a recovery log that maintains the state of the logical unit of work and names the immediate upstream coordinator and downstream participants.
		recovery pending (RECP) A condition that prevents SQL access to a table space that needs to be recovered.
		recovery token An identifier for an element that is used in recovery (for example, NID or URID).
		RECP See recovery pending.
		redo A state of a unit of recovery that indicates that changes are to be reapplied to the disk media to ensure data integrity.
		reentrant code Executable code that can reside in storage as one shared copy for all threads. Reentrant code is not self-modifying and provides separate storage areas for each thread. See also threadsafe.
		referential constraint The requirement that nonnull values of a designated foreign key are valid only if they equal values of the primary key of a designated table.
		referential cycle A set of referential constraints such that each base table in the set is a descendent of itself. The tables that are involved in a referential cycle are ordered so that each

| table is a descendent of the one before it,
| and the first table is a descendent of the
| last table.

referential integrity

The state of a database in which all values of all foreign keys are valid. Maintaining referential integrity requires the enforcement of referential constraints on all operations that change the data in a table on which the referential constraints are defined.

referential structure

A set of tables and relationships that includes at least one table and, for every table in the set, all the relationships in which that table participates and all the tables to which it is related.

refresh age

The time duration between the current time and the time during which a materialized query table was last refreshed.

registry

See registry database.

registry database

A database of security information about principals, groups, organizations, accounts, and security policies.

relational database

A database that can be perceived as a set of tables and manipulated in accordance with the relational model of data.

relational database management system (RDBMS)

A collection of hardware and software that organizes and provides access to a relational database.

| **relational schema**

| See SQL schema.

relationship

A defined connection between the rows of a table or the rows of two tables. A relationship is the internal representation of a referential constraint.

relative byte address (RBA)

The offset of a data record or control interval from the beginning of the storage space that is allocated to the data set or file to which it belongs.

remigration

The process of returning to a current release of DB2 following a fallback to a previous release. This procedure constitutes another migration process.

remote

Any object that is maintained by a remote DB2 subsystem (that is, by a DB2 subsystem other than the local one). A *remote view*, for example, is a view that is maintained by a remote DB2 subsystem. Contrast with local.

remote subsystem

Any relational DBMS, except the local subsystem, with which the user or application can communicate. The subsystem need not be remote in any physical sense, and might even operate on the same processor under the same z/OS system.

reoptimization

The DB2 process of reconsidering the access path of an SQL statement at run time; during reoptimization, DB2 uses the values of host variables, parameter markers, or special registers.

| **reordered row format**

| A row format that facilitates improved
| performance in retrieval of rows that have
| varying-length columns. DB2 rearranges
| the column order, as defined in the
| CREATE TABLE statement, so that the
| fixed-length columns are stored at the
| beginning of the row and the
| varying-length columns are stored at the
| end of the row. Contrast with basic row
| format.

REORG pending (REORP)

A condition that restricts SQL access and most utility access to an object that must be reorganized.

REORP

See REORG pending.

repeatable read (RR)

The isolation level that provides maximum protection from other executing application programs. When an application program executes with repeatable read protection, rows that the program references cannot be changed by other programs until the program reaches

a commit point. See also cursor stability, read stability, and uncommitted read.

repeating group

A situation in which an entity includes multiple attributes that are inherently the same. The presence of a repeating group violates the requirement of first normal form. In an entity that satisfies the requirement of first normal form, each attribute is independent and unique in its meaning and its name. See also normalization.

replay detection mechanism

A method that allows a principal to detect whether a request is a valid request from a source that can be trusted or whether an untrustworthy entity has captured information from a previous exchange and is replaying the information exchange to gain access to the principal.

request commit

The vote that is submitted to the prepare phase if the participant has modified data and is prepared to commit or roll back.

requester

The source of a request to access data at a remote server. In the DB2 environment, the requester function is provided by the distributed data facility.

resource

The object of a lock or claim, which could be a table space, an index space, a data partition, an index partition, or a logical partition.

resource allocation

The part of plan allocation that deals specifically with the database resources.

resource control table

A construct of previous versions of the CICS attachment facility that defines authorization and access attributes for transactions or transaction groups. Beginning in CICS Transaction Server Version 1.3, resources are defined by using resource definition online instead of the resource control table. See also resource definition online.

resource definition online (RDO)

The recommended method of defining resources to CICS by creating resource definitions interactively, or by using a utility, and then storing them in the CICS

definition data set. In earlier releases of CICS, resources were defined by using the resource control table (RCT), which is no longer supported.

resource limit facility (RLF)

A portion of DB2 code that prevents dynamic manipulative SQL statements from exceeding specified time limits. The resource limit facility is sometimes called the governor.

resource limit specification table (RLST)

A site-defined table that specifies the limits to be enforced by the resource limit facility.

resource manager

- A function that is responsible for managing a particular resource and that guarantees the consistency of all updates made to recoverable resources within a logical unit of work. The resource that is being managed can be physical (for example, disk or main storage) or logical (for example, a particular type of system service).
- A participant, in the execution of a two-phase commit, that has recoverable resources that could have been modified. The resource manager has access to a recovery log so that it can commit or roll back the effects of the logical unit of work to the recoverable resources.

restart pending (RESTP)

A restrictive state of a page set or partition that indicates that restart (backout) work needs to be performed on the object.

RESTP

See restart pending.

result set

The set of rows that a stored procedure returns to a client application.

result set locator

A 4-byte value that DB2 uses to uniquely identify a query result set that a stored procedure returns.

result table

The set of rows that are specified by a SELECT statement.

retained lock

A MODIFY lock that a DB2 subsystem

was holding at the time of a subsystem failure. The lock is retained in the coupling facility lock structure across a DB2 for z/OS failure.

RID See record identifier.

RID pool

See record identifier pool.

right outer join

The result of a join operation that includes the matched rows of both tables that are being joined and preserves the unmatched rows of the second join operand. See also join, equijoin, full outer join, inner join, left outer join, and outer join.

RLF See resource limit facility.

RLST See resource limit specification table.

role A database entity that groups together one or more privileges and that can be assigned to a primary authorization ID or to PUBLIC. The role is available only in a trusted context.

rollback

The process of restoring data that was changed by SQL statements to the state at its last commit point. All locks are freed. Contrast with commit.

root page

The index page that is at the highest level (or the beginning point) in an index.

routine

A database object that encapsulates procedural logic and SQL statements, is stored on the database server, and can be invoked from an SQL statement or by using the CALL statement. The main classes of routines are procedures and functions.

row The horizontal component of a table. A row consists of a sequence of values, one for each column of the table.

row identifier (ROWID)

A value that uniquely identifies a row. This value is stored with the row and never changes.

row lock

A lock on a single row of data.

row-positioned fetch orientation

The specification of the desired placement

of the cursor as part of a FETCH statement, with respect to a single row (for example, NEXT, LAST, or ABSOLUTE *n*). Contrast with rowset-positioned fetch orientation.

rowset

A set of rows for which a cursor position is established.

rowset cursor

A cursor that is defined so that one or more rows can be returned as a rowset for a single FETCH statement, and the cursor is positioned on the set of rows that is fetched.

rowset-positioned fetch orientation

The specification of the desired placement of the cursor as part of a FETCH statement, with respect to a rowset (for example, NEXT ROWSET, LAST ROWSET, or ROWSET STARTING AT ABSOLUTE *n*). Contrast with row-positioned fetch orientation.

row trigger

A trigger that is defined with the trigger granularity FOR EACH ROW.

RRSAF

See Recoverable Resource Manager Services attachment facility.

RS

See read stability.

savepoint

A named entity that represents the state of data and schemas at a particular point in time within a unit of work.

SBCS See single-byte character set.

SCA See shared communications area.

scalar function

An SQL operation that produces a single value from another value and is expressed as a function name, followed by a list of arguments that are enclosed in parentheses.

scale

In SQL, the number of digits to the right of the decimal point (called the precision in the C language). The DB2 information uses the SQL definition.

schema

The organization or structure of a database.

A collection of, and a way of qualifying, database objects such as tables, views, routines, indexes or triggers that define a database. A database schema provides a logical classification of database objects.

scrollability

The ability to use a cursor to fetch in either a forward or backward direction. The FETCH statement supports multiple fetch orientations to indicate the new position of the cursor. See also fetch orientation.

scrollable cursor

A cursor that can be moved in both a forward and a backward direction.

search condition

A criterion for selecting rows from a table. A search condition consists of one or more predicates.

secondary authorization ID

An authorization ID that has been associated with a primary authorization ID by an authorization exit routine.

secondary group buffer pool

For a duplexed group buffer pool, the structure that is used to back up changed pages that are written to the primary group buffer pool. No page registration or cross-invalidation occurs using the secondary group buffer pool. The z/OS equivalent is *new* structure.

secondary index

A nonpartitioning index that is useful for enforcing a uniqueness constraint, for clustering data, or for providing access paths to data for queries. A secondary index can be partitioned or nonpartitioned. See also data-partitioned secondary index (DPSI) and nonpartitioned secondary index (NPSI).

section

The segment of a plan or package that contains the executable structures for a single SQL statement. For most SQL statements, one section in the plan exists for each SQL statement in the source program. However, for cursor-related statements, the DECLARE, OPEN, FETCH, and CLOSE statements reference the same section because they each refer to the SELECT statement that is named in the DECLARE CURSOR statement. SQL

statements such as COMMIT, ROLLBACK, and some SET statements do not use a section.

| **security label**

| A classification of users' access to objects
| or data rows in a multilevel security
| environment."

segment

A group of pages that holds rows of a single table. See also segmented table space.

segmented table space

A table space that is divided into equal-sized groups of pages called segments. Segments are assigned to tables so that rows of different tables are never stored in the same segment. Contrast with partitioned table space and universal table space.

self-referencing constraint

A referential constraint that defines a relationship in which a table is a dependent of itself.

self-referencing table

A table with a self-referencing constraint.

sensitive cursor

A cursor that is sensitive to changes that are made to the database after the result table has been materialized.

sequence

A user-defined object that generates a sequence of numeric values according to user specifications.

sequential data set

A non-DB2 data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape. Several of the DB2 database utilities require sequential data sets.

sequential prefetch

A mechanism that triggers consecutive asynchronous I/O operations. Pages are fetched before they are required, and several pages are read with a single I/O operation.

serialized profile

A Java object that contains SQL statements and descriptions of host variables. The SQLJ translator produces a serialized profile for each connection context.

server The target of a request from a remote requester. In the DB2 environment, the server function is provided by the distributed data facility, which is used to access DB2 data from remote applications.

service class
An eight-character identifier that is used by the z/OS Workload Manager to associate user performance goals with a particular DDF thread or stored procedure. A service class is also used to classify work on parallelism assistants.

service request block
A unit of work that is scheduled to execute.

session
A link between two nodes in a VTAM network.

session protocols
The available set of SNA communication requests and responses.

set operator
The SQL operators UNION, EXCEPT, and INTERSECT corresponding to the relational operators union, difference, and intersection. A set operator derives a result table by combining two other result tables.

shared communications area (SCA)
A coupling facility list structure that a DB2 data sharing group uses for inter-DB2 communication.

share lock
A lock that prevents concurrently executing application processes from changing data, but not from reading data. Contrast with exclusive lock.

shift-in character
A special control character (X'0F') that is used in EBCDIC systems to denote that the subsequent bytes represent SBCS characters. See also shift-out character.

shift-out character
A special control character (X'0E') that is used in EBCDIC systems to denote that the subsequent bytes, up to the next shift-in control character, represent DBCS characters. See also shift-in character.

sign-on
A request that is made on behalf of an individual CICS or IMS application

process by an attachment facility to enable DB2 to verify that it is authorized to use DB2 resources.

simple page set
A nonpartitioned page set. A simple page set initially consists of a single data set (page set piece). If and when that data set is extended to 2 GB, another data set is created, and so on, up to a total of 32 data sets. DB2 considers the data sets to be a single contiguous linear address space containing a maximum of 64 GB. Data is stored in the next available location within this address space without regard to any partitioning scheme.

simple table space
A table space that is neither partitioned nor segmented. Creation of simple table spaces is not supported in DB2 Version 9.1 for z/OS. Contrast with partitioned table space, segmented table space, and universal table space.

single-byte character set (SBCS)
A set of characters in which each character is represented by a single byte. Contrast with double-byte character set or multibyte character set.

single-precision floating point number
A 32-bit approximate representation of a real number.

SMP/E
See System Modification Program/Extended.

SNA See Systems Network Architecture.

SNA network
The part of a network that conforms to the formats and protocols of Systems Network Architecture (SNA).

socket A callable TCP/IP programming interface that TCP/IP network applications use to communicate with remote TCP/IP partners.

sourced function
A function that is implemented by another built-in or user-defined function that is already known to the database manager. This function can be a scalar function or an aggregate function; it returns a single value from a set of values

(for example, MAX or AVG). Contrast with built-in function, external function, and SQL function.

source program

A set of host language statements and SQL statements that is processed by an SQL precompiler.

source table

A table that can be a base table, a view, a table expression, or a user-defined table function.

source type

An existing type that DB2 uses to represent a distinct type.

space A sequence of one or more blank characters.

special register

A storage area that DB2 defines for an application process to use for storing information that can be referenced in SQL statements. Examples of special registers are SESSION_USER and CURRENT DATE.

specific function name

A particular user-defined function that is known to the database manager by its specific name. Many specific user-defined functions can have the same function name. When a user-defined function is defined to the database, every function is assigned a specific name that is unique within its schema. Either the user can provide this name, or a default name is used.

SPUFI See SQL Processor Using File Input.

SQL See Structured Query Language.

SQL authorization ID (SQL ID)

The authorization ID that is used for checking dynamic SQL statements in some situations.

SQLCA

See SQL communication area.

SQL communication area (SQLCA)

A structure that is used to provide an application program with information about the execution of its SQL statements.

SQL connection

An association between an application process and a local or remote application server or database server.

SQLDA

See SQL descriptor area.

SQL descriptor area (SQLDA)

A structure that describes input variables, output variables, or the columns of a result table.

SQL escape character

The symbol that is used to enclose an SQL delimited identifier. This symbol is the double quotation mark ("). See also escape character.

SQL function

A user-defined function in which the CREATE FUNCTION statement contains the source code. The source code is a single SQL expression that evaluates to a single value. The SQL user-defined function can return the result of an expression. See also built-in function, external function, and sourced function.

SQL ID

See SQL authorization ID.

SQLJ Structured Query Language (SQL) that is embedded in the Java programming language.

SQL path

An ordered list of schema names that are used in the resolution of unqualified references to user-defined functions, distinct types, and stored procedures. In dynamic SQL, the SQL path is found in the CURRENT PATH special register. In static SQL, it is defined in the PATH bind option.

SQL procedure

A user-written program that can be invoked with the SQL CALL statement. An SQL procedure is written in the SQL procedural language. Two types of SQL procedures are supported: external SQL procedures and native SQL procedures. See also external procedure and native SQL procedure.

SQL processing conversation

Any conversation that requires access of DB2 data, either through an application or by dynamic query requests.

SQL Processor Using File Input (SPUFI)

A facility of the TSO attachment subcomponent that enables the DB2I user

to execute SQL statements without embedding them in an application program.

SQL return code

Either SQLCODE or SQLSTATE.

SQL routine

A user-defined function or stored procedure that is based on code that is written in SQL.

SQL schema

A collection of database objects such as tables, views, indexes, functions, distinct types, schemas, or triggers that defines a database. An SQL schema provides a logical classification of database objects.

SQL statement coprocessor

An alternative to the DB2 precompiler that lets the user process SQL statements at compile time. The user invokes an SQL statement coprocessor by specifying a compiler option.

SQL string delimiter

A symbol that is used to enclose an SQL string constant. The SQL string delimiter is the apostrophe ('), except in COBOL applications, where the user assigns the symbol, which is either an apostrophe or a double quotation mark (").

SRB See service request block.

stand-alone

An attribute of a program that means that it is capable of executing separately from DB2, without using DB2 services.

star join

A method of joining a dimension column of a fact table to the key column of the corresponding dimension table. See also join, dimension, and star schema.

star schema

The combination of a fact table (which contains most of the data) and a number of dimension tables. See also star join, dimension, and dimension table.

statement handle

In DB2 ODBC, the data object that contains information about an SQL statement that is managed by DB2 ODBC. This includes information such as dynamic arguments, bindings for dynamic arguments and columns, cursor information, result values, and status

information. Each statement handle is associated with the connection handle.

statement string

For a dynamic SQL statement, the character string form of the statement.

statement trigger

A trigger that is defined with the trigger granularity FOR EACH STATEMENT.

static cursor

A named control structure that does not change the size of the result table or the order of its rows after an application opens the cursor. Contrast with dynamic cursor.

static SQL

SQL statements, embedded within a program, that are prepared during the program preparation process (before the program is executed). After being prepared, the SQL statement does not change (although values of variables that are specified by the statement might change).

storage group

A set of storage objects on which DB2 for z/OS data can be stored. A storage object can have an SMS data class, a management class, a storage class, and a list of volume serial numbers.

stored procedure

A user-written application program that can be invoked through the use of the SQL CALL statement. Stored procedures are sometimes called procedures.

string See binary string, character string, or graphic string.

strong typing

A process that guarantees that only user-defined functions and operations that are defined on a distinct type can be applied to that type. For example, you cannot directly compare two currency types, such as Canadian dollars and U.S. dollars. But you can provide a user-defined function to convert one currency to the other and then do the comparison.

structure

- A name that refers collectively to different types of DB2 objects, such as tables, databases, views, indexes, and table spaces.
- A construct that uses z/OS to map and manage storage on a coupling facility. See also cache structure, list structure, or lock structure.

Structured Query Language (SQL)

A standardized language for defining and manipulating data in a relational database.

structure owner

In relation to group buffer pools, the DB2 member that is responsible for the following activities:

- Coordinating rebuild, checkpoint, and damage assessment processing
- Monitoring the group buffer pool threshold and notifying castout owners when the threshold has been reached

subcomponent

A group of closely related DB2 modules that work together to provide a general function.

subject table

The table for which a trigger is created. When the defined triggering event occurs on this table, the trigger is activated.

subquery

A SELECT statement within the WHERE or HAVING clause of another SQL statement; a nested SQL statement.

subselect

That form of a query that includes only a SELECT clause, FROM clause, and optionally a WHERE clause, GROUP BY clause, HAVING clause, ORDER BY clause, or FETCH FIRST clause.

substitution character

A unique character that is substituted during character conversion for any characters in the source program that do not have a match in the target coding representation.

subsystem

A distinct instance of a relational database management system (RDBMS).

surrogate pair

A coded representation for a single character that consists of a sequence of

two 16-bit code units, in which the first value of the pair is a high-surrogate code unit in the range U+D800 through U+DBFF, and the second value is a low-surrogate code unit in the range U+DC00 through U+DFFF. Surrogate pairs provide an extension mechanism for encoding 917 476 characters without requiring the use of 32-bit characters.

SVC dump

A dump that is issued when a z/OS or a DB2 functional recovery routine detects an error.

sync point

See commit point.

syncpoint tree

The tree of recovery managers and resource managers that are involved in a logical unit of work, starting with the recovery manager, that make the final commit decision.

synonym

In SQL, an alternative name for a table or view. Synonyms can be used to refer only to objects at the subsystem in which the synonym is defined. A synonym cannot be qualified and can therefore not be used by other users. Contrast with alias.

Sysplex

See Parallel Sysplex.

Sysplex query parallelism

Parallel execution of a single query that is accomplished by using multiple tasks on more than one DB2 subsystem. See also query CP parallelism.

system administrator

The person at a computer installation who designs, controls, and manages the use of the computer system.

system agent

A work request that DB2 creates such as prefetch processing, deferred writes, and service tasks. See also allied agent.

system authorization ID

The primary DB2 authorization ID that is used to establish a trusted connection. A system authorization ID is derived from the system user ID that is provided by an external entity, such as a middleware server.

system conversation

The conversation that two DB2 subsystems must establish to process system messages before any distributed processing can begin.

System Modification Program/Extended (SMP/E)

A z/OS tool for making software changes in programming systems (such as DB2) and for controlling those changes.

Systems Network Architecture (SNA)

The description of the logical structure, formats, protocols, and operational sequences for transmitting information through and controlling the configuration and operation of networks.

table A named data object consisting of a specific number of columns and some number of unordered rows. See also base table or temporary table. Contrast with auxiliary table, clone table, materialized query table, result table, and transition table.

table-controlled partitioning

A type of partitioning in which partition boundaries for a partitioned table are controlled by values that are defined in the CREATE TABLE statement.

table function

A function that receives a set of arguments and returns a table to the SQL statement that references the function. A table function can be referenced only in the FROM clause of a subselect.

table locator

| A mechanism that allows access to trigger
| tables in SQL or from within user-defined
| functions. A table locator is a fullword
| integer value that represents a transition
| table.

table space

A page set that is used to store the records in one or more tables. See also partitioned table space, segmented table space, and universal table space.

table space set

A set of table spaces and partitions that should be recovered together for one of the following reasons:

- Each of them contains a table that is a parent or descendent of a table in one of the others.

- The set contains a base table and associated auxiliary tables.

A table space set can contain both types of relationships.

task control block (TCB)

A z/OS control block that is used to communicate information about tasks within an address space that is connected to a subsystem. See also address space connection.

TB Terabyte. A value of 1 099 511 627 776 bytes.

TCB See task control block.

TCP/IP

A network communication protocol that computer systems use to exchange information across telecommunication links.

TCP/IP port

A 2-byte value that identifies an end user or a TCP/IP network application within a TCP/IP host.

template

A DB2 utilities output data set descriptor that is used for dynamic allocation. A template is defined by the TEMPLATE utility control statement.

temporary table

A table that holds temporary data. Temporary tables are useful for holding or sorting intermediate results from queries that contain a large number of rows. The two types of temporary table, which are created by different SQL statements, are the created temporary table and the declared temporary table. Contrast with result table. See also created temporary table and declared temporary table.

thread See DB2 thread.

threadsafe

A characteristic of code that allows multithreading both by providing private storage areas for each thread, and by properly serializing shared (global) storage areas.

three-part name

The full name of a table, view, or alias. It

consists of a location name, a schema name, and an object name, separated by a period.

time A three-part value that designates a time of day in hours, minutes, and seconds.

timeout Abnormal termination of either the DB2 subsystem or of an application because of the unavailability of resources. Installation specifications are set to determine both the amount of time DB2 is to wait for IRLM services after starting, and the amount of time IRLM is to wait if a resource that an application requests is unavailable. If either of these time specifications is exceeded, a timeout is declared.

Time-Sharing Option (TSO)

An option in z/OS that provides interactive time sharing from remote terminals.

timestamp A seven-part value that consists of a date and time. The timestamp is expressed in years, months, days, hours, minutes, seconds, and microseconds.

trace A DB2 facility that provides the ability to monitor and collect DB2 monitoring, auditing, performance, accounting, statistics, and serviceability (global) data.

transaction An atomic series of SQL statements that make up a logical unit of work. All of the data modifications made during a transaction are either committed together as a unit or rolled back as a unit.

transaction lock A lock that is used to control concurrent execution of SQL statements.

transaction program name In SNA LU 6.2 conversations, the name of the program at the remote logical unit that is to be the other half of the conversation.

transition table A temporary table that contains all the affected rows of the subject table in their state before or after the triggering event occurs. Triggered SQL statements in the trigger definition can reference the table of changed rows in the old state or the

new state. Contrast with auxiliary table, base table, clone table, and materialized query table.

transition variable

A variable that contains a column value of the affected row of the subject table in its state before or after the triggering event occurs. Triggered SQL statements in the trigger definition can reference the set of old values or the set of new values.

tree structure

A data structure that represents entities in nodes, with a most one parent node for each node, and with only one root node.

trigger

A database object that is associated with a single base table or view and that defines a rule. The rule consists of a set of SQL statements that run when an insert, update, or delete database operation occurs on the associated base table or view.

trigger activation

The process that occurs when the trigger event that is defined in a trigger definition is executed. Trigger activation consists of the evaluation of the triggered action condition and conditional execution of the triggered SQL statements.

trigger activation time

An indication in the trigger definition of whether the trigger should be activated before or after the triggered event.

trigger body

The set of SQL statements that is executed when a trigger is activated and its triggered action condition evaluates to true. A trigger body is also called triggered SQL statements.

trigger cascading

The process that occurs when the triggered action of a trigger causes the activation of another trigger.

triggered action

The SQL logic that is performed when a trigger is activated. The triggered action consists of an optional triggered action condition and a set of triggered SQL statements that are executed only if the condition evaluates to true.

triggered action condition

An optional part of the triggered action. This Boolean condition appears as a WHEN clause and specifies a condition that DB2 evaluates to determine if the triggered SQL statements should be executed.

triggered SQL statements

The set of SQL statements that is executed when a trigger is activated and its triggered action condition evaluates to true. Triggered SQL statements are also called the trigger body.

trigger granularity

In SQL, a characteristic of a trigger, which determines whether the trigger is activated:

- Only once for the triggering SQL statement
- Once for each row that the SQL statement modifies

triggering event

The specified operation in a trigger definition that causes the activation of that trigger. The triggering event is comprised of a triggering operation (insert, update, or delete) and a subject table or view on which the operation is performed.

triggering SQL operation

The SQL operation that causes a trigger to be activated when performed on the subject table or view.

trigger package

A package that is created when a CREATE TRIGGER statement is executed. The package is executed when the trigger is activated.

trust attribute

An attribute on which to establish trust. A trusted relationship is established based on one or more trust attributes.

trusted connection

A database connection whose attributes match the attributes of a unique trusted context defined at the DB2 database server.

trusted connection reuse

The ability to switch the current user ID on a trusted connection to a different user ID.

trusted context

A database security object that enables the establishment of a trusted relationship between a DB2 database management system and an external entity.

trusted context default role

A role associated with a trusted context. The privileges granted to the trusted context default role can be acquired only when a trusted connection based on the trusted context is established or reused.

trusted context user

A user ID to which switching the current user ID on a trusted connection is permitted.

trusted context user-specific role

A role that is associated with a specific trusted context user. It overrides the trusted context default role if the current user ID on the trusted connection matches the ID of the specific trusted context user.

trusted relationship

A privileged relationship between two entities such as a middleware server and a database server. This relationship allows for a unique set of interactions between the two entities that would be impossible otherwise.

TSO See Time-Sharing Option.

TSO attachment facility

A DB2 facility consisting of the DSN command processor and DB2I. Applications that are not written for the CICS or IMS environments can run under the TSO attachment facility.

typed parameter marker

A parameter marker that is specified along with its target data type. It has the general form:
CAST(? AS data-type)

type 2 indexes

Indexes that are created on a release of DB2 after Version 7 or that are specified as type 2 indexes in Version 4 or later.

UCS-2 Universal Character Set, coded in 2 octets, which means that characters are represented in 16-bits per character.

UDF See user-defined function.

UDT User-defined data type. In DB2 for z/OS,

the term distinct type is used instead of user-defined data type. See distinct type.

uncommitted read (UR)

The isolation level that allows an application to read uncommitted data. See also cursor stability, read stability, and repeatable read.

underlying view

The view on which another view is directly or indirectly defined.

undo A state of a unit of recovery that indicates that the changes that the unit of recovery made to recoverable DB2 resources must be backed out.

Unicode

A standard that parallels the ISO-10646 standard. Several implementations of the Unicode standard exist, all of which have the ability to represent a large percentage of the characters that are contained in the many scripts that are used throughout the world.

union An SQL operation that involves the UNION set operator, which combines the results of two SELECT statements. Unions are often used to merge lists of values that are obtained from two tables.

unique constraint

An SQL rule that no two values in a primary key, or in the key of a unique index, can be the same.

unique index

An index that ensures that no identical key values are stored in a column or a set of columns in a table.

unit of recovery (UOR)

A recoverable sequence of operations within a single resource manager, such as an instance of DB2. Contrast with unit of work.

unit of work (UOW)

A recoverable sequence of operations within an application process. At any time, an application process is a single unit of work, but the life of an application process can involve many units of work as a result of commit or rollback operations. In a multisite update operation, a single unit of work can include several *units of recovery*. Contrast with unit of recovery.

universal table space

A table space that is both segmented and partitioned. Contrast with partitioned table space, segmented table space, partition-by-growth table space, and range-partitioned table space.

unlock

The act of releasing an object or system resource that was previously locked and returning it to general availability within DB2.

untyped parameter marker

A parameter marker that is specified without its target data type. It has the form of a single question mark (?).

updatability

The ability of a cursor to perform positioned updates and deletes. The updatability of a cursor can be influenced by the SELECT statement and the cursor sensitivity option that is specified on the DECLARE CURSOR statement.

update hole

The location on which a cursor is positioned when a row in a result table is fetched again and the new values no longer satisfy the search condition. See also delete hole.

update trigger

A trigger that is defined with the triggering SQL operation update.

UR See uncommitted read.

user-defined data type (UDT)

See distinct type.

user-defined function (UDF)

A function that is defined to DB2 by using the CREATE FUNCTION statement and that can be referenced thereafter in SQL statements. A user-defined function can be an external function, a sourced function, or an SQL function. Contrast with built-in function.

user view

In logical data modeling, a model or representation of critical information that the business requires.

UTF-8 Unicode Transformation Format, 8-bit encoding form, which is designed for ease of use with existing ASCII-based systems. The CCSID value for data in UTF-8

format is 1208. DB2 for z/OS supports UTF-8 in mixed data fields.

UTF-16

Unicode Transformation Format, 16-bit encoding form, which is designed to provide code values for over a million characters and a superset of UCS-2. The CCSID value for data in UTF-16 format is 1200. DB2 for z/OS supports UTF-16 in graphic data fields.

value The smallest unit of data that is manipulated in SQL.

variable

A data element that specifies a value that can be changed. A COBOL elementary data item is an example of a host variable. Contrast with constant.

variant function

See nondeterministic function.

varying-length string

A character, graphic, or binary string whose length varies within set limits. Contrast with fixed-length string.

version

A member of a set of similar programs, DBRMs, packages, or LOBs.

- **A version of a program** is the source code that is produced by precompiling the program. The program version is identified by the program name and a timestamp (consistency token).
- **A version of an SQL procedural language routine** is produced by issuing the CREATE or ALTER PROCEDURE statement for a native SQL procedure.
- **A version of a DBRM** is the DBRM that is produced by precompiling a program. The DBRM version is identified by the same program name and timestamp as a corresponding program version.
- **A version of an application package** is the result of binding a DBRM within a particular database system. The application package version is identified by the same program name and consistency token as the DBRM.
- **A version of a LOB** is a copy of a LOB value at a point in time. The version number for a LOB is stored in the auxiliary index entry for the LOB.

- **A version of a record** is a copy of the record at a point in time.

view

A logical table that consists of data that is generated by a query. A view can be based on one or more underlying base tables or views, and the data in a view is determined by a SELECT statement that is run on the underlying base tables or views.

Virtual Storage Access Method (VSAM)

An access method for direct or sequential processing of fixed- and varying-length records on disk devices.

Virtual Telecommunications Access Method (VTAM)

An IBM licensed program that controls communication and the flow of data in an SNA network (in z/OS).

volatile table

A table for which SQL operations choose index access whenever possible.

VSAM

See Virtual Storage Access Method.

VTAM

See Virtual Telecommunications Access Method.

warm start

The normal DB2 restart process, which involves reading and processing log records so that data that is under the control of DB2 is consistent. Contrast with cold start.

WLM application environment

A z/OS Workload Manager attribute that is associated with one or more stored procedures. The WLM application environment determines the address space in which a given DB2 stored procedure runs.

WLM enclave

A construct that can span multiple dispatchable units (service request blocks and tasks) in multiple address spaces, allowing them to be reported on and managed by WLM as part of a single work request.

write to operator (WTO)

An optional user-coded service that allows a message to be written to the system console operator informing the

operator of errors and unusual system conditions that might need to be corrected (in z/OS).

WTO See write to operator.

WTOR

Write to operator (WTO) with reply.

XCF See cross-system coupling facility.

XES See cross-system extended services.

XML See Extensible Markup Language.

XML attribute

A name-value pair within a tagged XML element that modifies certain features of the element.

XML column

A column of a table that stores XML values and is defined using the data type XML. The XML values that are stored in XML columns are internal representations of well-formed XML documents.

XML data type

A data type for XML values.

XML element

A logical structure in an XML document that is delimited by a start and an end tag. Anything between the start tag and the end tag is the content of the element.

XML index

An index on an XML column that provides efficient access to nodes within an XML document by providing index keys that are based on XML patterns.

XML lock

A column-level lock for XML data. The operation of XML locks is similar to the operation of LOB locks.

XML node

The smallest unit of valid, complete structure in a document. For example, a node can represent an element, an attribute, or a text string.

XML node ID index

An implicitly created index, on an XML table that provides efficient access to XML documents and navigation among multiple XML data rows in the same document.

XML pattern

A slash-separated list of element names, an optional attribute name (at the end), or

kind tests, that describe a path within an XML document in an XML column. The pattern is a restrictive form of path expressions, and it selects nodes that match the specifications. XML patterns are specified to create indexes on XML columns in a database.

XML publishing function

A function that returns an XML value from SQL values. An XML publishing function is also known as an XML constructor.

XML schema

In XML, a mechanism for describing and constraining the content of XML files by indicating which elements are allowed and in which combinations. XML schemas are an alternative to document type definitions (DTDs) and can be used to extend functionality in the areas of data typing, inheritance, and presentation.

XML schema repository (XSR)

A repository that allows the DB2 database system to store XML schemas. When registered with the XSR, these objects have a unique identifier and can be used to validate XML instance documents.

XML serialization function

A function that returns a serialized XML string from an XML value.

XML table

An auxiliary table that is implicitly created when an XML column is added to a base table. This table stores the XML data, and the column in the base table points to it.

XML table space

A table space that is implicitly created when an XML column is added to a base table. The table space stores the XML table. If the base table is partitioned, one partitioned table space exists for each XML column of data.

X/Open

An independent, worldwide open systems organization that is supported by most of the world's largest information systems suppliers, user organizations, and software companies. X/Open's goal is to increase the portability of applications by combining existing and emerging standards.

- XRF** See Extended Recovery Facility.
- | **XSR** See XML schema repository.
- | **zIIP** See IBM System z9 Integrated Processor.
- z/OS** An operating system for the System z product line that supports 64-bit real and virtual storage.
- z/OS Distributed Computing Environment (z/OS DCE)** A set of technologies that are provided by the Open Software Foundation to implement distributed computing.

Index

A

- accessibility
 - keyboard x
 - shortcut keys x
- ALLOW DEBUG MODE
 - clause of CREATE PROCEDURE 161
- application development
 - high availability
 - connections to DB2 Database for Linux, UNIX, and Windows servers 501
 - connections to IDS 513
 - direct connections to DB2 for z/OS servers 524
 - JDBC
 - application programming 7
 - SQLJ 93
- assignment-clause
 - SQLJ 296
- auto-generated keys
 - retrieving in JDBC application 60
- autocommit modes
 - default JDBC 81
- automatic client reroute
 - client applications 493
 - DB2 for z/OS 523
 - IDS servers 509
- automatically generated keys
 - retrieving
 - JDBC applications 60

B

- batch queries
 - JDBC 35
- batch updates
 - JDBC 28
 - SQLJ 113
- BatchUpdateException exception
 - retrieving information 89
- binding SQLJ applications
 - access multiple servers 183

C

- CallableStatement class 46
- CD-ROM, books on 559
- client affinities
 - .NET 514
 - CLI 514
 - configuring 502
 - enabling
 - Java clients example 502
 - IBM Data Server Driver for JDBC and SQLJ 514, 515
 - overview 501
- client affinities, example of enabling
 - Java clients 515
- client applications
 - automatic client reroute 493
 - high availability 493
 - transaction-level load balancing 493

- client info properties
 - IBM Data Server Driver for JDBC and SQLJ 66, 67
- client reroute
 - IBM Data Server Driver for JDBC and SQLJ 498
- clients
 - automatic client reroute
 - connections to DB2 for z/OS 523
 - connections to IDS 509
 - high-availability support configuration
 - DB2 Database for Linux, UNIX, and Windows 495
 - IDS 505
 - Sysplex support
 - configuration 519
- commands
 - db2sqljbind 432
 - db2sqljprint 437
 - sqlj 416
 - SQLJ 416
- comments
 - SQLJ applications 104
- commits
 - SQLJ transactions 148
 - transactions
 - JDBC 81
- configuration
 - JDBC 442
 - SQLJ 442
- configuration properties
 - customizing 442
 - details 243
 - parameters 442
- connection context
 - class 95
 - closing 151
 - default 95
 - object 95
- connection declaration clause
 - SQLJ 289
- connection pooling
 - overview 527
- connections
 - closing
 - importance 92, 151
 - data sources using SQLJ 95
 - DataSource interface 15
 - existing 101
 - high availability 501
- context clause
 - SQLJ 292, 293
- conversion
 - JDBC/SQLJ Driver for OS/390 and z/OS properties to IBM Data Server Driver for JDBC and SQLJ properties 468
 - JDBC/SQLJ Driver for OS/390 and z/OS serialized profiles to IBM Data Server Driver for JDBC and SQLJ serialized profiles 471

D

- data
 - retrieving
 - JDBC 32

- data sources
 - connecting to
 - DriverManager 11
 - JDBC 9
 - JDBC DataSource 15
- data type mappings
 - Java types to other types 191
- DatabaseMetaData methods 21
- databases
 - compatibility
 - IBM Data Server Driver for JDBC and SQLJ 4
- DataSource interface
 - SQLJ
 - connection technique 3 99
 - connection technique 4 100
- DataSource objects
 - creating 19
 - deploying 19
- date value adjustment
 - JDBC applications 198
 - SQLJ applications 198
- DB2 books online 559
- DB2 for Linux, UNIX, and Windows
 - associated IBM Data Server Driver for JDBC and SQLJ level 5
- DB2 for z/OS
 - binding plans and packages 420
 - direct connections 522, 524
 - Sysplex support
 - client configuration 519
 - overview 516
- DB2 Information Center for z/OS solutions 559
- DB2BaseDataSource class 325
- DB2Binder utility 447
- DB2BlobFileReference class 331
- DB2ClientRerouteServerList class 331
- DB2ClobFileReference class 332
- DB2Connection interface 333
- DB2ConnectionPoolDataSource class 350
- DB2DatabaseMetaData interface 352
- DB2Diagnosable class
 - retrieving the SQLCA 149
- DB2Diagnosable interface 353
- DB2ExceptionFormatter class 354
- DB2FileReference class 354
- DB2JCCPlugin interface 355
- db2jcctrace command 547
- DB2LobTableCreator utility 457
- DB2PooledConnection interface 356
- DB2PoolMonitor class 358
- DB2PreparedStatement interface 361
- DB2ResultSetMetaData interface 374
- DB2RowID interface 374
- DB2SimpleDataSource class
 - definition 19
 - details 375
- DB2Sqlca class 375
- db2sqljbind command 432
- db2sqljcustomize command 420
- db2sqljprint
 - formation JCC customized profile 539
- db2sqljprint command
 - details 437
 - formatting information about SQLJ customized profile 537
- db2sqljupgrade
 - JDBC/SQLJ Driver for OS/390 and z/OS conversion of serialized profiles 472
- DB2Statement interface 376
- DB2SystemMonitor interface 379
- DB2T4XAIndoubtUtil 454
- DB2TraceManager class 382
- DB2TraceManagerMXBean interface 385
- DB2Types class 389
- DB2XADatasource class 389
- DB2Xml interface 391
- DB2XmlAsBlobFileReference class 393
- DB2XmlAsClobFileReference class 394
- DBBatchUpdateException interface 324
- DBINFO clause
 - CREATE FUNCTION statement 161
 - CREATE PROCEDURE statement 161
- default connection context 102
- deregisterDB2XMLObject method 75
- disability x
- DISABLE DEBUG MODE
 - clause of CREATE PROCEDURE 161
- DISALLOW DEBUG MODE
 - clause of CREATE PROCEDURE 161
- distinct types
 - JDBC applications 58
 - SQLJ applications 139
- distributed transactions
 - example 530
- DriverManager interface
 - SQLJ
 - SQLJ connection technique 1 96
 - SQLJ connection technique 2 97
- drivers
 - determining IBM Data Server Driver for JDBC and SQLJ version 415
- dynamic data format 130

E

- encryption
 - IBM Data Server Driver for JDBC and SQLJ 479
- environment
 - Java stored procedure 153
 - Java user-defined function 153
- environment variables
 - JDBC 442
 - settings for Java routine 156
 - SQLJ 442
 - z/OS Application Connectivity to DB2 for z/OS feature 464
- errors
 - SQLJ 149
- examples
 - deregisterDB2XMLObject 75
 - registerDB2XMLSchema 75
- exceptions
 - IBM Data Server Driver for JDBC and SQLJ 82
- executable clause 291
- executeUpdate methods 28
- extended client information 65
- EXTERNAL
 - clause of CREATE FUNCTION statement 161
 - clause of CREATE PROCEDURE statement 161

F

- failover
 - IBM Data Server Driver for JDBC and SQLJ type 2 connectivity 525
- FINAL CALL clause
 - CREATE FUNCTION statement 161

G

- general-use programming information, described 569
- getCause method 82
- getDatabaseProductName method 23
- getDatabaseProductVersion method 23
- global transaction
 - JDBC and SQLJ 535
- GUI symbols 569

H

- high availability
 - client applications 493
 - clients
 - configuring 495
 - connections to DB2 Database for Linux, UNIX, and Windows 494
 - IBM Data Server Driver for JDBC and SQLJ 497
 - IDS 494, 504
- host expressions
 - SQLJ 103, 286

I

- IBM data server clients
 - automatic client reroute
 - DB2 for z/OS 523
 - IDS 509
- IBM Data Server Driver for JDBC and SQLJ
 - associated DB2 for Linux, UNIX, and Windows level 5
 - client info properties 66
 - compatibility with databases 4
 - connecting to data sources 11
 - connection concentrator monitoring 543
 - DB2T4XAIndoubtUtil utility 454
 - errors 407
 - exceptions 82
 - extended client information 65
 - high-availability examples
 - DB2 Database for Linux, UNIX, and Windows 497
 - IDS 508
 - JDBC extensions 322
 - Kerberos security 481
 - LOB support
 - JDBC 51, 53
 - SQLJ 130
 - properties 201
 - security
 - details 475
 - encrypted password 479
 - encrypted user ID 479
 - user ID and password 476
 - user ID-only 478
 - SQLExceptions 84
 - SQLSTATES 413
 - Sysplex examples 520
 - trace program example 540

- IBM Data Server Driver for JDBC and SQLJ *(continued)*
 - tracing with configuration parameters example 539
 - trusted context support 484
 - type 2 connectivity
 - DB2 for z/OS failover support 525
 - overview 17
 - type 4 connectivity 17
 - version determination 415
 - warnings 82
 - XML support 140
- IBM Data Server Driver for JDBC and SQLJ stored procedures
 - WLM environment definition 445
- IBM Data Server Driver for JDBC and SQLJ-only fields
 - DB2Types class 389
- IBM Data Server Driver for JDBC and SQLJ-only methods
 - DB2BaseDataSource class 325
 - DB2BlobFileReference class 331
 - DB2ClientRerouteServerList class 331
 - DB2ClobFileReference class 332
 - DB2Connection interface 333
 - DB2ConnectionPoolDataSource class 350
 - DB2DatabaseMetaData interface 352
 - DB2Diagnosable interface 353
 - DB2ExceptionFormatter class 354
 - DB2FileReference class 354
 - DB2JCCPlugin interface 355
 - DB2PooledConnection interface 356
 - DB2PoolMonitor class 358
 - DB2PreparedStatement interface 361
 - DB2ResultSetMetaData interface 374
 - DB2RowID interface 374
 - DB2SimpleDataSource class 375
 - DB2sqlca class 375
 - DB2Statement interface 376
 - DB2SystemMonitor interface 379
 - DB2TraceManager class 382
 - DB2TraceManagerMXBean interface 385
 - DB2XADatasource class 389
 - DB2Xml interface 391
 - DB2XmlAsBlobFileReference class 393
 - DB2XmlAsClobFileReference class 394
 - DBBatchUpdateException interface 324
- IBM Data Server Driver for JDBC and SQLJ-only properties
 - DB2BaseDataSource class 325
 - DB2ClientRerouteServerList class 331
 - DB2ConnectionPoolDataSource class 350
 - DB2SimpleDataSource class 375
- IBM data server drivers
 - automatic client reroute
 - DB2 for z/OS 523
 - IDS 509
- IDS
 - high availability
 - application programming 513
 - client configuration 505
 - cluster support 504
 - example 508
 - workload balancing 513
- implements clause
 - SQLJ 286
- isolation levels
 - JDBC 80
 - SQLJ 148
- iterator conversion clause
 - SQLJ 297
- iterator declaration clause
 - SQLJ 290

- iterators
 - obtaining JDBC result sets from 133
 - positioned DELETE 107
 - positioned UPDATE 107

J

- JAR file
 - creating for JDBC routine 189
 - defining to DB2 160, 165
- Java
 - applications
 - overview 1
 - environment
 - customization 442
- Java routine 154
 - environment variable settings 156
- Java routine with no SQLJ
 - program preparation 185, 186
- Java routine with SQLJ
 - program preparation 186, 189
- Java stored procedure
 - defining to DB2 160
 - differences from Java program 175
 - differences from other stored procedures 175
 - parameters specific to 161
 - writing 174
- Java user-defined function
 - defining to DB2 160
 - differences from Java program 175
 - differences from other user-defined functions 175
 - parameters specific to 161
 - writing 174
- JAVAENV data set
 - characteristics 156
- JDBC
 - 4.0
 - getColumnLabel change 404
 - columnName change 404
 - accessing packages 21
 - APIs 257
 - applications
 - 24 as hour value 198
 - data retrieval 32
 - example 7
 - invalid Gregorian date 198
 - programming overview 7
 - transaction control 80
 - variables 23
 - batch errors 89
 - batch queries 35
 - batch updates 28
 - configuring 442
 - connections 18
 - data type mappings 191
 - drivers
 - details 3
 - differences 394, 400, 406
 - environment variables 442
 - executeUpdate methods 28
 - executing SQL 24
 - extensions 322
 - file reference variables 78
 - input file reference variables 78
 - isolation levels 80
 - migration, IBM Data Server Driver for JDBC and SQLJ 465

- JDBC (*continued*)
 - named parameter markers 63, 64
 - objects
 - creating 25
 - modifying 25
 - problem diagnosis 537
 - ResultSet holdability 39
 - ResultSets
 - delete holes 44
 - holdability 37
 - inserting row 44, 45
 - running a program 190
 - sample program 458
 - scrollable ResultSet 37, 39
 - SQLWarning 88
 - transactions
 - committing 81
 - default autocommit modes 81
 - rolling back 81
 - updatable ResultSet 37, 39
- JDBC program
 - program preparation 181
- JDBC/SQLJ Driver for OS/390 and z/OS properties
 - conversion to IBM Data Server Driver for JDBC and SQLJ properties 468
- JDBC/SQLJ Driver for OS/390 and z/OS serialized profiles
 - conversion to IBM Data Server Driver for JDBC and SQLJ serialized profiles 471
- JDBC/SQLJ Driver for OS/390 and z/OS serialized profiles with db2sqljupgrade
 - conversion to IBM Data Server Driver for JDBC and SQLJ serialized profiles 472

K

- Kerberos authentication protocol
 - IBM Data Server Driver for JDBC and SQLJ 481

L

- LANGUAGE
 - clause of CREATE FUNCTION statement 161
 - clause of CREATE PROCEDURE statement 161
- large objects (LOBs)
 - compatible Java data types
 - JDBC applications 54
 - SQLJ applications 131
 - IBM Data Server Driver for JDBC and SQLJ 51, 53, 130
- locators
 - IBM Data Server Driver for JDBC and SQLJ 52, 53
 - SQLJ 130
- library 559

M

- memory
 - IBM Data Server Driver for JDBC and SQLJ 91
- migration
 - IBM Data Server Driver for JDBC and SQLJ 465
- monitoring
 - system
 - IBM Data Server Driver for JDBC and SQLJ 549
- multi-row operations 42
- multiple result sets
 - retrieving from stored procedure in JDBC application keeping result sets open 50

- multiple result sets (*continued*)
 - retrieving from stored procedure in JDBC application (*continued*)
 - known number 48
 - overview 48
 - unknown number 49
 - retrieving from stored procedure in SQLJ application 129

N

- named iterators
 - passed as variables 111
 - result set iterator 117
- named parameter markers
 - CallableStatement objects 64
 - JDBC 63
 - PreparedStatement objects 63
- NO SQL
 - clause of CREATE FUNCTION statement 161
 - clause of CREATE PROCEDURE statement 161

O

- online 559
- online books 559

P

- packages
 - JDBC 21
 - SQLJ 102
- PARAMETER STYLE
 - clause of CREATE FUNCTION statement 161
 - clause of CREATE PROCEDURE statement 161
- ParameterMetaData methods 31
- positioned deletes
 - SQLJ 107
- positioned iterators
 - passed as variables 111
 - result set iterators 119
- positioned updates
 - SQLJ 107
- PreparedStatement methods
 - SQL statements with no parameter markers 26
 - SQL statements with parameter markers 26, 34
- problem determination
 - JDBC 537
 - SQLJ 537
- program preparation
 - example, Java routine with SQLJ 186
 - Java routine with no SQLJ 185, 186
 - Java routine with SQLJ 186, 189
 - JDBC program 181
- PROGRAM TYPE clause
 - CREATE FUNCTION statement 161
 - CREATE PROCEDURE statement 161
- programming interface information, described 569
- progressive streaming
 - IBM Data Server Driver for JDBC and SQLJ 51, 53
 - JDBC 130
- properties
 - IBM Data Server Driver for JDBC and SQLJ
 - customizing 442
 - for all database products 203
 - for DB2 Database for Linux, UNIX, and Windows 227, 229, 230

- properties (*continued*)
 - IBM Data Server Driver for JDBC and SQLJ (*continued*)
 - for DB2 for z/OS 233
 - for DB2 servers 219
 - for IDS 227, 229, 238
 - overview 201

R

- reference information
 - Java 191
- registerDB2XMLSchema method 75
- resources
 - releasing
 - closing connections 92, 151
- restrictions
 - SQLJ variable names 104
- result set iterator
 - public declaration in separate file 134
- result set iterators
 - details 116
 - generating JDBC ResultSets from SQLJ iterators 133
 - named 117
 - positioned 119
 - retrieving data from JDBC result sets using SQLJ iterators 133
- ResultSet
 - holdability 37
 - inserting row 44
 - testing for delete hole 44
 - testing for inserted row 45
- ResultSet holdability
 - JDBC 39
- ResultSetMetaData methods
 - ResultSetMetaData.getColumnLabel change in value 404
 - ResultSetMetaData.getColumnName change in value 404
 - retrieving result set information 36
- retrieving data
 - JDBC
 - data source information 21
 - PreparedStatement.executeQuery method 34
 - result set information 36
 - tables 32
 - SQLJ 116, 122, 123
- retrieving parameter information
 - JDBC 31
- retrieving SQLCA
 - DB2Diagnosable class 149
- return codes
 - IBM Data Server Driver for JDBC and SQLJ errors 407
- rollbacks
 - JDBC transactions 81
 - SQLJ transactions 148
- ROWID 137
- RUN OPTIONS clause
 - CREATE FUNCTION statement 161
 - CREATE PROCEDURE statement 161
- running a program
 - SQLJ and JDBC 190

S

- sample program
 - JDBC 458
- savepoints
 - JDBC applications 59

- savepoints (*continued*)
 - SQLJ applications 139
- SCRATCHPAD clause
 - CREATE FUNCTION statement 161
- scrollable iterators
 - SQLJ 124
- scrollable ResultSet
 - JDBC 39
- scrollable ResultSets
 - JDBC 37
- SDKs
 - version 1.5 145
- security
 - IBM Data Server Driver for JDBC and SQLJ
 - encrypted security-sensitive data 479
 - encrypted user ID or encrypted password 479
 - Kerberos 481
 - security mechanisms 475
 - user ID and password 476
 - user ID only 478
 - SQLJ program preparation 490
- SECURITY
 - clause of CREATE FUNCTION 161
 - clause of CREATE PROCEDURE 161
- SET TRANSACTION clause 295
- shortcut keys
 - keyboard x
- softcopy publications 559
- SQL statements
 - error handling
 - SQLJ applications 149
 - executing
 - JDBC interfaces 24
 - SQLJ applications 105, 136
- SQLException
 - IBM Data Server Driver for JDBC and SQLJ 84
- SQLJ
 - accessing packages for 102
 - applications
 - 24 as hour value 198
 - examples 93
 - invalid Gregorian date 198
 - programming 93
 - transaction control 148
 - assignment clause 296
 - batch updates 113
 - binding applications to access multiple servers 183
 - calling stored procedures 128
 - clauses 285
 - collecting trace data 537
 - comments 104
 - connecting to data source 95
 - connecting using default context 102
 - connection declaration clause 289
 - context clause 292, 293
 - DataSource interface 99, 100
 - DB2 tables
 - creating 106
 - modifying 106
 - DriverManager interface 96, 97
 - drivers 3
 - environment variables 442
 - error handling 149
 - executable clauses 291
 - executing SQL 105
 - execution context 136
 - execution control 136
- SQLJ (*continued*)
 - existing connections 101
 - host expressions 103, 286
 - implements clause 286
 - installing runtime environment 442
 - isolation levels 148
 - iterator conversion clause 297
 - iterator declaration clause 290
 - migration, IBM Data Server Driver for JDBC and SQLJ 465
 - multiple instances of iterator 123
 - multiple iterators on table 122
 - problem diagnosis 537
 - Profile Binder command 432
 - Profile Printer command 437
 - program preparation 416
 - result set iterator 116
 - retrieving SQLCA 149
 - running a program 190
 - scrollable iterators 124
 - SDK for Java Version 5 functions 145
 - security 490
 - SET TRANSACTION clause 295
 - SQLWarning 150
 - statement reference 285
 - transactions 148
 - translator command 416
 - variable names 103
 - with-clause 287
- sqlj command 416
- SQLJ variable names
 - restrictions 104
- SQLJ.ALTER_JAVA_PATH stored procedure 172
- SQLJ.DB2_INSTALL_JAR stored procedure 166
- SQLJ.DB2_REPLACE_JAR stored procedure 169
- SQLJ.INSTALL_JAR stored procedure 165
- SQLJ.REMOVE_JAR stored procedure 171
- SQLJ.REPLACE_JAR stored procedure 168
- sqlj.runtime package 297
- sqlj.runtime.ASCIIStream 310, 320
- sqlj.runtime.BinaryStream 310
- sqlj.runtime.CharacterStream 311
- sqlj.runtime.ConnectionContext 298
- sqlj.runtime.ExecutionContext 312
- sqlj.runtime.ForUpdate 303
- sqlj.runtime.NamedIterator 303
- sqlj.runtime.PositionedIterator 304
- sqlj.runtime.ResultSetIterator 304
- sqlj.runtime.Scrollable 307
- sqlj.runtime.SQLNullException 320
- sqlj.runtime.UnicodeStream 321
- SQLSTATE
 - IBM Data Server Driver for JDBC and SQLJ errors 413
- SQLWarning
 - IBM Data Server Driver for JDBC and SQLJ 88
 - SQLJ applications 150
- SSID
 - IBM Data Server Driver for JDBC and SQLJ 243
- SSL
 - configuring
 - Java Runtime Environment 487
 - IBM Data Server Driver for JDBC and SQLJ 486
 - sslConnection property 487
 - sslConnection property 487
 - Statement.executeQuery 32
 - stored procedure
 - access to z/OS UNIX System Services 161

- stored procedure (*continued*)
 - Java 153
 - returning result set 177
- stored procedures
 - calling
 - CallableStatement class 46
 - SQLJ applications 128
 - DB2 for z/OS 46
 - keeping result sets open in JDBC applications 50
 - retrieving result sets
 - known number (JDBC) 48
 - multiple (JDBC) 48
 - multiple (SQLJ) 129
 - unknown number (JDBC) 49
- syntax diagram
 - how to read xi
- Sysplex
 - direct connections to DB2 for z/OS 522
 - enablement examples
 - IBM Data Server Driver for JDBC and SQLJ 520
 - support 516

T

- time value adjustment
 - JDBC applications 198
 - SQLJ applications 198
- trace C/C++ native driver code
 - db2jcttrace 547
- traces
 - IBM Data Server Driver for JDBC and SQLJ 537, 539, 540
- transaction control
 - JDBC 80
 - SQLJ 148
- transaction-level load balancing
 - client applications 493
- trusted contexts
 - JDBC support 484

U

- updatable ResultSet
 - inserting row 44
 - JDBC 37, 39
 - testing for delete hole 44
 - testing for inserted row 45
- updates
 - data
 - PreparedStatement.executeUpdate method 26
- URL format
 - DB2BaseDataSource class 13, 14
- user ID and password security
 - IBM Data Server Driver for JDBC and SQLJ 476
- user ID-only security
 - IBM Data Server Driver for JDBC and SQLJ 478
- user-defined function
 - access to z/OS UNIX System Services 161
 - Java 153

W

- warnings
 - IBM Data Server Driver for JDBC and SQLJ 82
- with clause
 - SQLJ 287

- WLM environment
 - definition for IBM Data Server Driver for JDBC and SQLJ
 - stored procedures 445
- WLM ENVIRONMENT
 - clause of CREATE FUNCTION statement 161
 - clause of CREATE PROCEDURE statement 161
- WLM setup 154
 - Java routine 154
- workload balancing
 - IDS
 - operation 513

X

- XML
 - IBM Data Server Driver for JDBC and SQLJ 140
- XML data
 - Java applications 69
 - updating
 - tables in Java applications 70, 141
- XML data retrieval
 - Java applications 72, 143
- XML schemas
 - registering 75
 - removing 75
- XMLCAST
 - SQLJ applications 145

Z

- z/OS Application Connectivity to DB2 for z/OS
 - SMP/E jobs for loading 463
- z/OS Application Connectivity to DB2 for z/OS feature
 - environment variables 464
- z/OS UNIX System Services
 - authority to access 161



Program Number: 5635-DB2

Printed in USA

SC18-9842-05



Spine information:

DB2 Version 9.1 for z/OS

Application Programming Guide and Reference for Java™

