

June 17, 2010 NEDB2UG

JDBC T4 Driver Primer

Bilung Lee, Ph.D., drlee@us.ibm.com
IBM Silicon Valley Laboratory

Information Management

IBM

June 17, 2010 NEDB2UG

© 2010 IBM Corporation

Things That May Also Interest You

- pureQuery
 - Provide a high-performance data access platform (APIs, runtime, tools, and monitoring)
 - Reduce CPU usage up to 42% (compared to dynamic JDBC)
 - Improve productivity up to 50% using developer friendly tooling
 - <http://www.ibm.com/software/data/optim/purequery-platform/>

Things That May Also Interest You (more)

3

- Extended Insight
 - Extend capabilities provided in Optim Performance Manager
 - Provide end-to-end database monitoring for Java technology applications
 - Provide even more capabilities for those running in WebSphere Application Server
 - Provide insight into any stop along the software stack
 - <http://www.ibm.com/software/data/optim/performance-manager-extended-edition/>

Outline

- JDBC Basics
- Driver Overview
- Package Management
- Property Configuration
- Defer Prepares/Send Data As Is
- Multiple Row Manipulation
- Multiple Row Query
- Progressive Streaming
- XML Capabilities

JDBC Basics

JDBC 1 Style Connection

- Load the driver (no need for JCC 4 driver)
`Class.forName("com.ibm.db2.jcc.DB2Driver");`

- Create a connection

```
String url = "jdbc:db2://server:446/sample";
```

```
Properties p = new Properties();
```

```
p.setProperty("user", "sysadm");
```

```
p.setProperty("password", "xxxxxx");
```

```
Connection c = DriverManager.getConnection(url, p);
```

JDBC 2 Style Connection

- Create a data source




```
DB2SimpleDataSource ds =  
    new DB2SimpleDataSource();  
ds.setDriverType(4);  
ds.setServerName("server");  
ds.setPortNumber(446);  
ds.setDatabaseName("sample");  
ds.setUser("sysadm");  
ds.setPassword("xxxxxx");
```

- Create a connection

```
Connection c = ds.getConnection();
```

JDBC Statements

- JDBC statements are the Java interfaces for performing SQL operations

	Input Parameters	Output Parameters
<code>java.sql.Statement</code>		
<code>java.sql.PreparedStatement</code>		
<code>java.sql.CallableStatement</code>		

Data Definition

- Create a table

```
Statement s = c.createStatement();
```

```
try {
```

```
    s.execute("DROP TABLE EMPLOYEE");
```

```
} catch (SQLException sqle) {
```

```
    if (sqle.getErrorCode() != -204)
```

```
        throw sqle;
```

```
}
```

```
s.execute("CREATE TABLE EMPLOYEE (" +  
    "ID INTEGER, NAME VARCHAR(10))");
```

Data Manipulation

- Insert data

```
PreparedStatement ps = c.prepareStatement(  
    "INSERT INTO EMPLOYEE VALUES (?, ?)");  
ps.setInt(1, 111);  
ps.setString(2, "John Doe");  
ps.execute();  
ps.setInt(1, 222);  
ps.setString(2, "Jane Doe");  
ps.execute();
```

Data Query

- Select data

```
ResultSet rs = s.executeQuery(  
    "SELECT ID, NAME FROM EMPLOYEE");  
while (rs.next()) {  
    int id = rs.getInt(1);  
    String name = rs.getString(2);  
}
```

Stored Procedure

- Create an SQL procedure

```
s.execute(
```

```
"CREATE PROCEDURE FINDNAME " +
```

```
"(IN ID_IN INTEGER, " +
```

```
" OUT NAME_OUT VARCHAR(10)) " +
```

```
"LANGUAGE SQL " +
```

```
"BEGIN " +
```

```
"SELECT NAME INTO NAME_OUT " +
```

```
" FROM EMPLOYEE WHERE ID = ID_IN; " +
```

```
"END");
```

Stored Procedure (continued)

- Invoke the SQL procedure

```
CallableStatement cs =
```

```
    c.prepareCall("CALL FINDNAME (?, ?)");
```

```
cs.setInt(1, 222);
```

```
cs.registerOutParameter(2, Types.VARCHAR);
```

```
cs.execute();
```

```
String name = cs.getString(2);
```

Driver Overview

IBM Data Server Driver

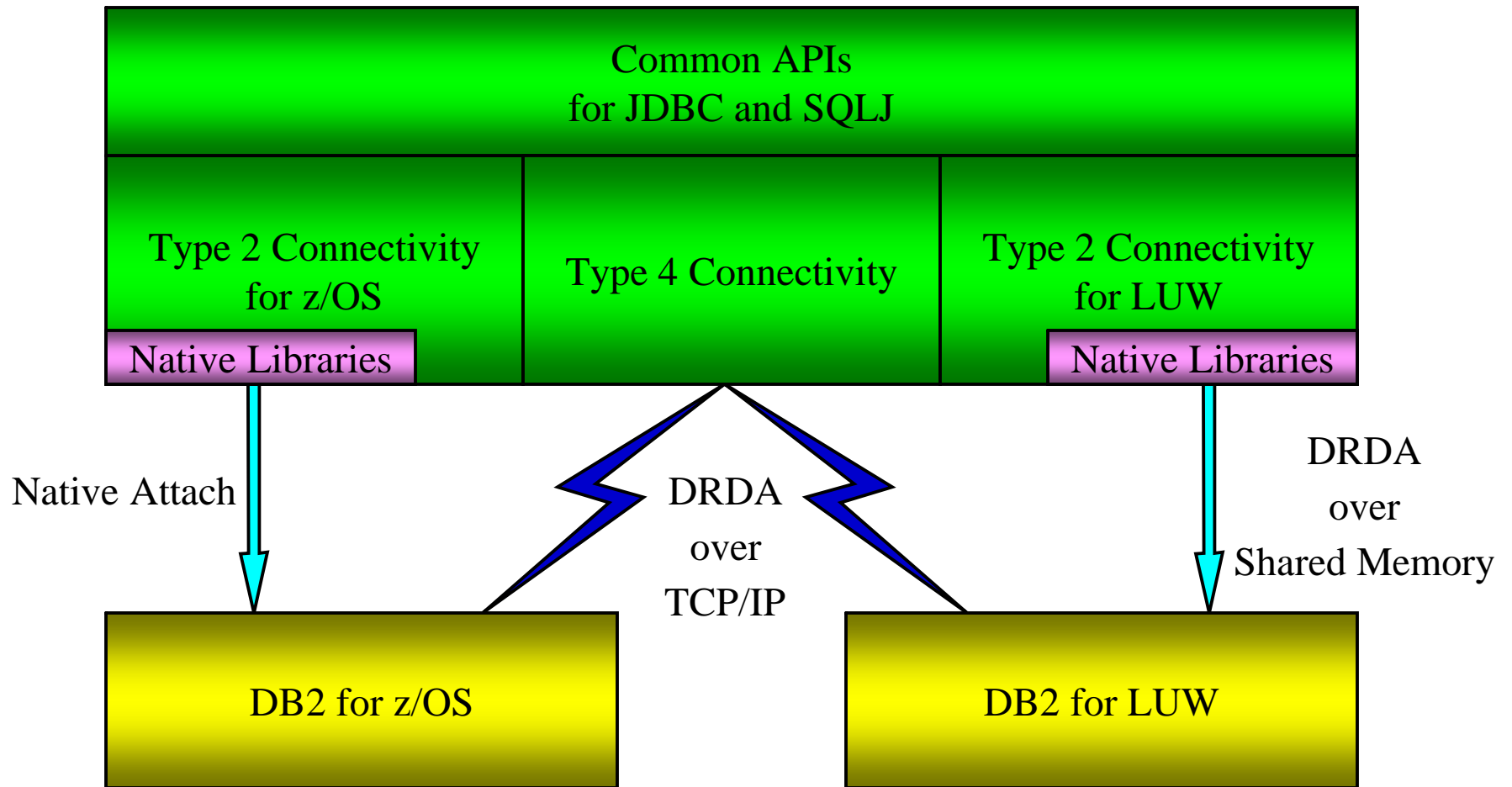
- A single driver (also known as JCC driver) enables Java access across DB2 product family

Driver	Driver class
IBM Data Server Driver	<code>com.ibm.db2.jcc.DB2Driver</code>
Legacy driver (DB2 for z/OS)	<code>COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver</code>
Legacy driver (DB2 for LUW)	<code>COM.ibm.db2.jdbc.app.DB2Driver</code> <code>COM.ibm.db2.jdbc.net.DB2Driver</code>

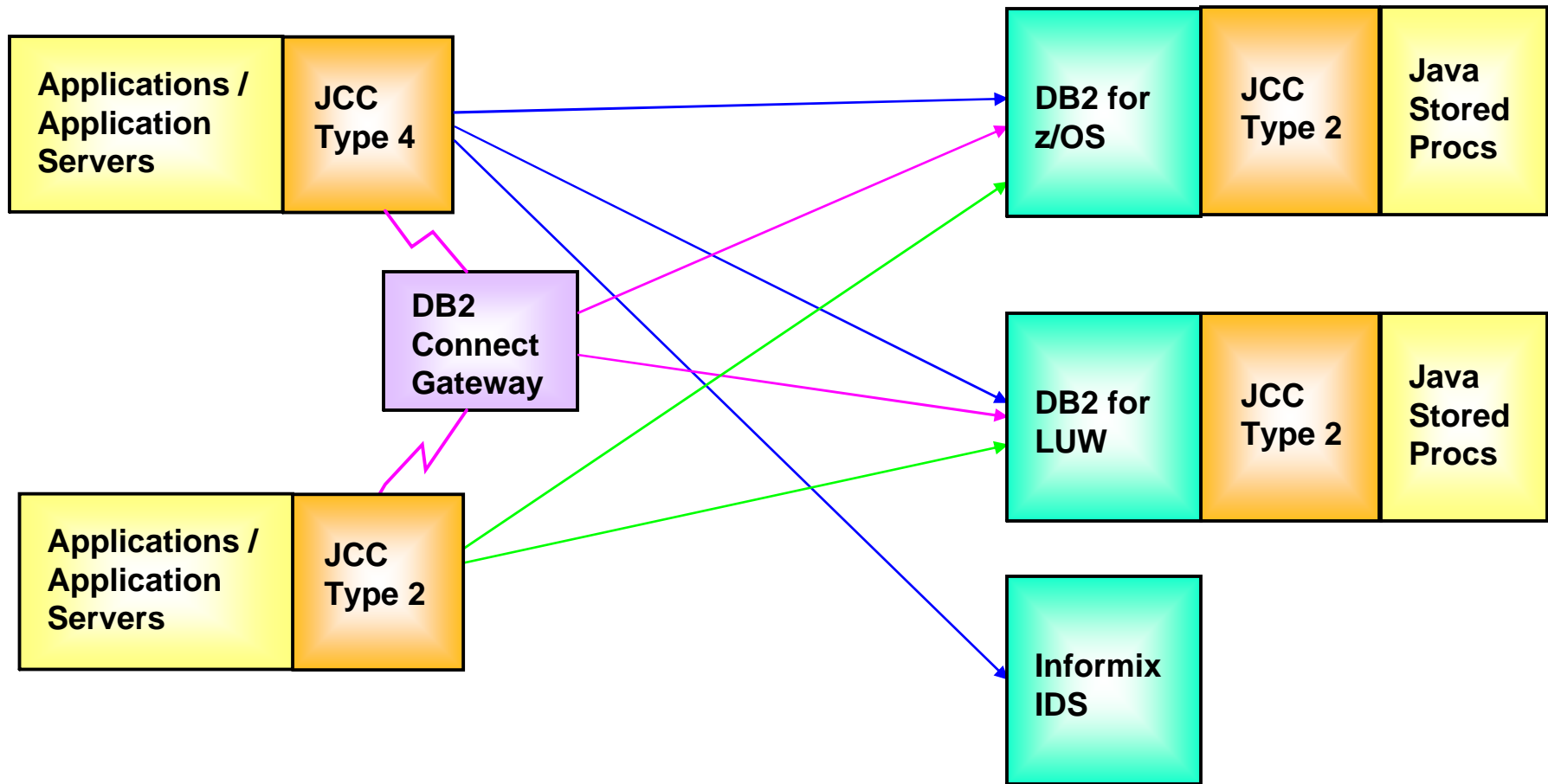
IBM Data Server Driver (continued)

- Both JDBC and SQLJ access
- Multiple platforms, including DB2 for Linux, Unix, and Windows, DB2 for z/OS, DB2 for iSeries, Informix IDS, and Cloudscape
- Different connectivity paths (local and remote)
 - Type 2: Local via native attach on z/OS
 - Type 2: Local via shared memory on LUW
 - Type 4: Remote via DRDA protocol (pure Java)
- Direct Access from remote (no need for gateway)

Common Architecture



Typical Usage



Connectivity Selection

- Considerations
 - T4 connectivity is binary portable
 - T4 connectivity can run with a security manager
 - T4 connectivity equips with XA support for distributed transactions
 - T2 connectivity performs better for local connections
 - T2 connectivity is used by Java Stored Procedures
- Applications may switch between different connectivity types

Providing Two JAR Files

JAR file	Driver version	Level of JDBC support	Minimum level of Java
db2jcc.jar	JCC 3	JDBC 3.0 and earlier	1.4
db2jcc4.jar*	JCC 4	JDBC 4.0 and earlier	6.0
* Available starting from DB2 for LUW version 9.5			

Package Management

JDBC Packages

- DB2Binder provides a command-line utility for the JDBC packages used by JCC

```
java com.ibm.db2.jcc.DB2Binder
```

```
-url "jdbc:db2://server:446/sample"
```

```
-user sysadm -password xxxxxx
```

- The JDBC packages consist of one static package and multiple dynamic packages
 - The static package is used for special operation, such as large object processing
 - The dynamic packages are used for normal operation, such as query and update

JDBC Dynamic Packages

- By default, three (six) dynamic packages are created per holdability and per isolation for DB2 for z/OS (DB2 for LUW)

	WITH HOLD	WITHOUT HOLD
UR	SYSLH100 (SYSSH100) SYSLH101 (SYSSH101) SYSLH102 (SYSSH102)	SYSLN100 (SYSSN100) SYSLN101 (SYSSN101) SYSLN102 (SYSSN102)
CS	SYSLH200 (SYSSH200) SYSLH201 (SYSSH201) SYSLH202 (SYSSH202)	SYSLN200 (SYSSN200) SYSLN201 (SYSSN201) SYSLN202 (SYSSN202)
RS	SYSLH300 (SYSSH300) SYSLH301 (SYSSH301) SYSLH302 (SYSSH302)	SYSLN300 (SYSSN300) SYSLN301 (SYSSN301) SYSLN302 (SYSSN302)
RR	SYSLH400 (SYSSH400) SYSLH401 (SYSSH401) SYSLH402 (SYSSH402)	SYSLN400 (SYSSN400) SYSLN401 (SYSSN401) SYSLN402 (SYSSN402)

Out of Package Diagnostics

- A package contains a number of sections, each needed for an SQL request
- An exception with SQL code -805 occurs when no more sections available in dynamic packages

***** Out of Package Error Occurred (2010-05-06 13:08:03.469) *****

Concurrently open statements:

1. SQL string: SELECT * FROM SYSIBM.SYSDUMMY1

Number of statements: 600

2. SQL string: INSERT INTO TB (COL1) VALUES(?)

Number of statements: 400

3. SQL string: SELECT USER FROM SYSIBM.SYSDUMMY1

Number of statements: 150

Out of Package Treatment

- Close unused resources (such as statements and result sets) whenever possible
- Use the `size` option of `bind` utility to increase the number of large packages to be bound

```
java com.ibm.db2.jcc.DB2Binder  
-url "jdbc:db2://server:446/sample"  
-user sysadm -password xxxxxx  
-size 6
```

Property Configuration

Variety of Properties

- Pre-connect level
 - JDBC 1 style: Via properties to driver managers
`p.setProperty("traceFile", "/tmp/jcc.txt");`
`DriverManager.getConnection(url, p);`
 - JDBC 2 style: Via methods on data sources
`ds.setTraceFile("/tmp/jcc.txt");`
- Post-connect level
 - Via methods on connection
`c.setJccLogWriter("/tmp/jcc.txt", true, -1);`

Variety of Properties (continued)

- Global level
 - Two scopes: override (db2.jcc.override) and default (db2.jcc or db2.jcc.default)
 - Provided via Java system properties
`java -Ddb2.jcc.traceFile=/tmp/jcc.txt MyApp`
 - Provided via a Java properties file (specified by the system property db2.jcc.propertiesFile)

`db2.jcc.traceFile=/tmp/jcc.txt`

- Loaded only once when the driver is first loaded

Variety of Properties (continued)

- Not all properties are provided at all levels
- Most properties are only at pre-connect level
- The precedence for a property set at various levels is as follows
 - Global override level
 - Post-connect level
 - Pre-connect level
 - Global default level

Tracing w/o Changing Applications

- Stop applications
- Create a properties file, e.g. jcc.properties

```
db2.jcc.override.traceFile=/tmp/jcc.txt  
db2.jcc.override.traceLevel=-1
```

- Start applications with the properties file

```
java -Ddb2.jcc.propertiesFile=jcc.properties MyApp
```

Tracing w/o Stopping Applications

- Stop applications
- Create a properties file, e.g. jcc.properties

```
db2.jcc.tracePolling=true  
db2.jcc.tracePollingInterval=600
```

- Start applications with the properties file
- Enable trace by adding to the properties file

```
db2.jcc.tracePolling=true  
db2.jcc.tracePollingInterval=600  
db2.jcc.override.traceFile=/tmp/jcc.txt  
db2.jcc.override.traceLevel=-1
```

Defer Prepares/Send Data As Is

Properties

- deferPrepares
 - Hold prepare request until execute or open time
 - Matter whether describe information is available at the first execution or not
 - Enabled by default
 - Not applicable to type 2 for z/OS
- sendDataAsIs
 - Judicially guess data types
 - Always without describe information
 - Disabled by default
- Both are provided at pre-connect level

deferPrepares true sendDataAsIs false

```
s.execute("CREATE TABLE CATALOG (" +  
    "ID BIGINT, VIDEO BLOB)");  
PreparedStatement ps = c.prepareStatement(  
    "INSERT INTO CATALOG VALUES (?, ?)");  
ps.setInt(1, 111);  
ps.setBytes(2, video);  
ps.execute();
```

- 1) SQL PREPARE is flowed together with SQL EXECUTE
- 2) 1st (2nd) param is sent as INTEGER (VARCHAR FOR BIT DATA)
- 3) describe info is obtained but execution fails with data type mismatch (SQL code -301) on the 2nd param
- 4) SQL EXECUTE is flowed again
- 5) 1st (2nd) parameter is converted and sent as BIGINT (BLOB)
- 6) execution succeeds

deferPrepares false sendDataAsIs false

```
s.execute("CREATE TABLE CATALOG (" +
  "ID BIGINT, VIDEO BLOB)");
```

```
PreparedStatement ps = c.prepareStatement(
  "INSERT INTO CATALOG VALUES (?, ?)");
```

```
ps.setInt(1, 111);
```

```
ps.setBytes(2, video);
```

```
ps.execute();
```

2) 1st (2nd) param is converted to BIGINT (BLOB)

1) SQL PREPARE is flowed and describe info is obtained

3) SQL EXECUTE is flowed
4) 1st (2nd) param is sent as BIGINT (BLOB)
5) execution succeeds

deferPrepares true sendDataAsIs true

```
s.execute("CREATE TABLE CATALOG (" +  
    "ID BIGINT, VIDEO BLOB)");  
PreparedStatement ps = c.prepareStatement(  
    "INSERT INTO CATALOG VALUES (?, ?)");  
ps.setInt(1, 111);  
ps.setBytes(2, video);  
ps.execute();
```

- 1) SQL PREPARE is flowed together with SQL EXECUTE
- 2) 1st (2nd) param is sent as INTEGER (VARCHAR FOR BIT DATA)
- 3) execution fails with data type mismatch (SQL code -301) on the 2nd param

Summary

sendDataAsIs deferPrepares	false	true
true	No describes first time* No conversions first time* With retries first time*	No describes No conversions No retries
false	With describes With conversions No retries necessary	

* Except when addBatch() or setObject() is used

Data Conversion

- Under deferred prepares or as-is data
 - No describe information is available
 - Data conversion may occur on DB2
- Under other scenarios
 - Describe information is available
 - Data conversion may happen in JCC
- Different behaviors may result from data conversions at different places
- For example
 - `setString()` against `VARCHAR FOR BIT DATA`
 - `setBytes()` against `VARCHAR`

Multiple Row Manipulation

JDBC Batch

- Heterogeneous SQL statements

```
Statement s = c.createStatement();
```

```
s.addBatch("DROP TABLE EMPLOYEE");
```

```
s.addBatch("CREATE TABLE EMPLOYEE (" +  
"ID INTEGER, NAME VARCHAR(10))");
```

```
s.addBatch("INSERT INTO EMPLOYEE " +  
"VALUES (111, 'John Doe')");
```

```
s.executeBatch();
```

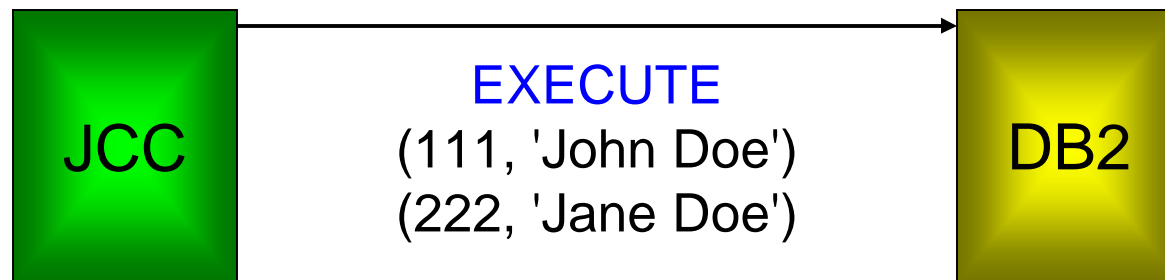
JDBC Batch (continued)

- An SQL statement with a set of parameters

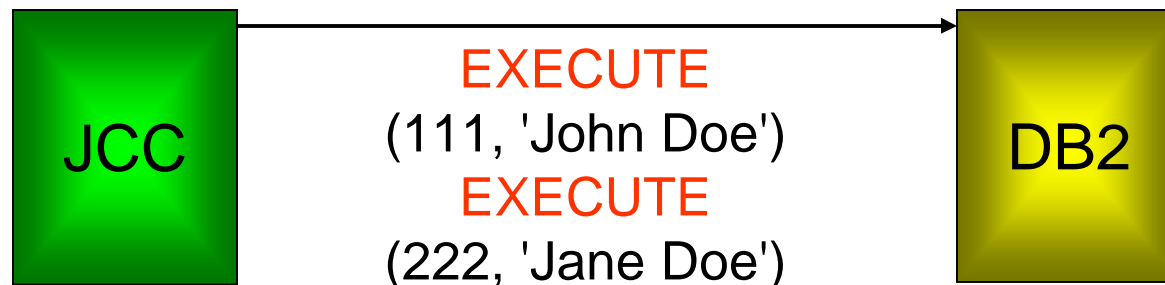
```
PreparedStatement ps = c.prepareStatement(  
    "INSERT INTO EMPLOYEE VALUES (?, ?)");  
ps.setInt(1, 111);  
ps.setString(2, "John Doe");  
ps.addBatch();  
ps.setInt(1, 222);  
ps.setString(2, "Jane Doe");  
ps.addBatch();  
ps.executeBatch();
```

Multiple Row Manipulation

- Multiple rows of data for one SQL execution
- Better performance than traditional approach



Multiple Row Execution



Single Row Execution

Server Specifics

- DB2 for z/OS
 - Known as multi-row insert
 - Apply only to INSERT/MERGE
 - Enabled by default when batch is used
 - Disabled via `enableMultirowInsertSupport false`
- DB2 for LUW
 - Known as array input
 - Apply to INSERT/MERGE/UPDATE/DELETE
 - Enabled when batch is used

Server Specifics (continued)

- No "array of array" input for multi-row insert
PreparedStatement ps = c.prepareStatement(
 "INSERT INTO EMPLOYEE VALUES (?, " +
 "**(SELECT NAME FROM TEMP WHERE ID = ?)**)");
ps.setInt(1, 111);
ps.setString(2, "John Doe");
ps.addBatch();
ps.setInt(1, 222);
ps.setString(2, "Jane Doe");
ps.addBatch();
ps.executeBatch();

Batch Atomicity

- Non-atomic operation by default: Insertion of each row is considered to be a separate execution

```
PreparedStatement ps = c.prepareStatement(
    "INSERT INTO EMPLOYEE VALUES (?, ?)");
ps.setInt(1, 111);
ps.setString(2, "John Doe");
ps.addBatch();
ps.setInt(1, 222);
ps.setString(2, "Jennifer Doe");
ps.addBatch();
ps.executeBatch();
```

Exceed table column
limit VARCHAR(10)

Fail with SQL code -302 due to 2nd row but
1st row (111, 'John Doe') is still inserted

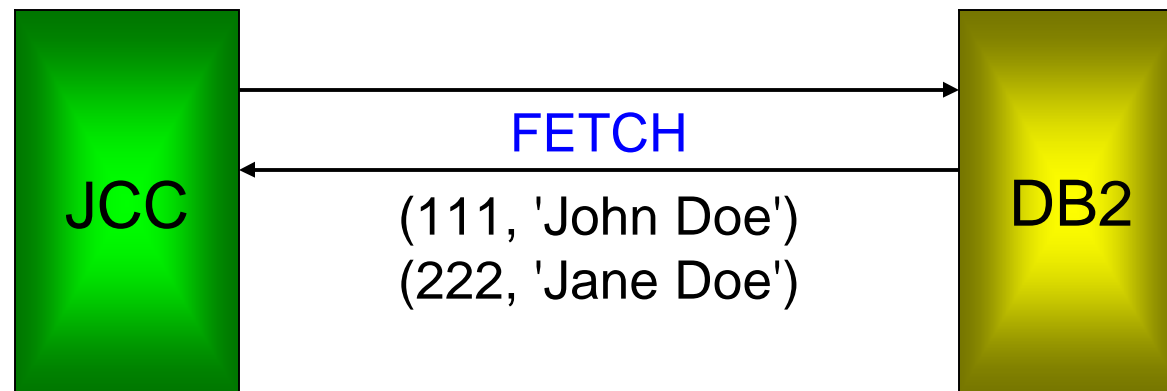
Batch Atomicity (continued)

- Atomic operation via `atomicMultiRowInsert` YES
 - Insertion of all rows is considered to be a single operation
 - DB2 for z/OS
 - Apply only when multi-row insert is enabled
 - Apply only to INSERT
 - Cannot have more than 32767 rows
 - Cannot set multiple stream types against the same parameter
 - DB2 for LUW
 - Apply to INSERT/MERGE/UPDATE/DELETE

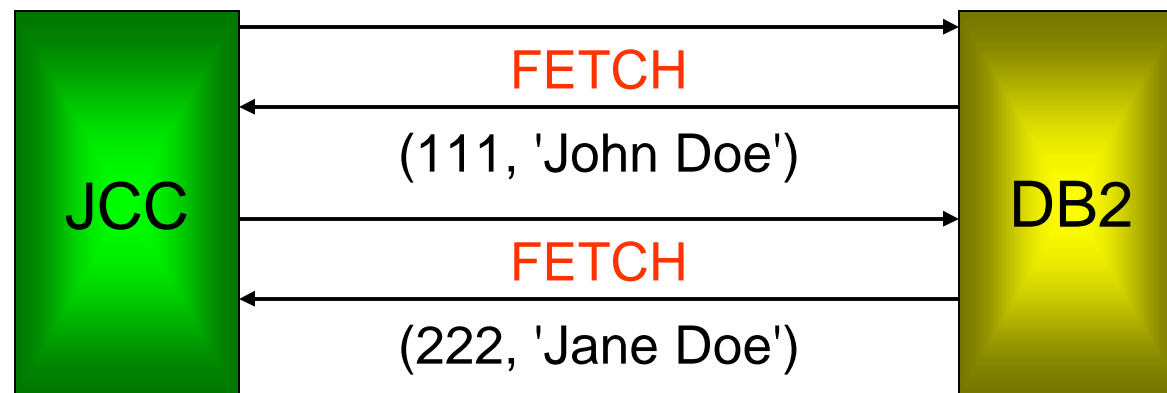
Multiple Row Query

Multiple Row Query

- Multiple rows of data for one SQL query



Multiple Row Query



Single Row Query

Cursor Blocking / Block Fetch

- Retrieve a block of rows that can fit into a buffer
- Influenced by the blocking option of bind utility
java com.ibm.db2.jcc.DB2Binder
 - url "jdbc:db2://server:446/sample"
 - user sysadm -password xxxxxx
 - blocking (all | no | unambig)
- Take hint from the queryDataSize property (in bytes)
 - Provided at pre-connect level
 - DB2 for LUW: 4096 - 65535
 - DB2 for z/OS: 32767 always

Cursor Blocking / Block Fetch (continued)

- For forward-only cursors that are read-only, the amount of data is determined by `queryDataSize`

```
Statement s = c.createStatement(  
    ResultSet.TYPE_FORWARD_ONLY,  
    ResultSet.CONCUR_READ_ONLY);  
ResultSet rs = s.executeQuery(  
    "SELECT ID, NAME FROM EMPLOYEE");  
while (rs.next()) {  
    int id = rs.getInt(1);  
    String name = rs.getString(2);  
}
```

DRDA Rowset

- Retrieve a specified number of rows in one network request
- Equivalent to performing the specified number of single-row fetches underneath in database
- More network-efficient
- Take hint from the `fetchSize` property (in rows)
 - Provided at pre-connect level
 - Can be overridden by the `setFetchSize` method of each statement

DRDA Rowset (continued)

- For scrollable cursors, the amount of data is determined by `fetchSize` (64 rows by default)

```
Statement s = c.createStatement(  
    ResultSet.TYPE_SCROLL_SENSITIVE,  
    ResultSet.CONCUR_UPDATABLE);  
ResultSet rs = s.executeQuery(  
    "SELECT ID, NAME FROM EMPLOYEE");  
System.out.println("Go to row number> ");  
rs.absolute(Integer.parseInt(input.readLine()));  
int id = rs.getInt(1);  
String name = rs.getString(2);
```

SQL Rowset

- Apply only to DB2 for z/OS
- Use rowset-positioned cursors
 - Return one or more rows for a single fetch
 - Positioned on the set of rows to be fetched
- Enabled by default when scrollable cursor is used
- Disabled via `enableRowsetSupport NO`
- Also take hint from the `fetchSize` property (in rows)

SQL Rowset (continued)

- JDBC 1 style positioned update may not work as expected for rowset-positioned cursors

PAYROLL	
ID	SALARY
1	70,000
2	65,000
3	60,000
4	55,000
5	50,000

```
rs = select.executeQuery("SELECT * FROM PAYROLL");
```

```
rs.next();
```

```
rs.next();
```

```
rs.next();
```

```
delete.execute("DELETE FROM PAYROLL " +  
"WHERE CURRENT OF "  
+ rs.getCursorName());
```

WHOLE rowset is deleted!

SQL Rowset (continued)

- To avoid unexpected updates,
 - Use JDBC 2 style positioned update
 - Use the FOR ROW *n* OF ROWSET clause

PAYROLL	
ID	SALARY
1	70,000
2	65,000
3	60,000
4	55,000
5	50,000

```
rs = select.executeQuery("SELECT * FROM PAYROLL");
```

```
rs.next();
```

```
rs.next();
```

```
rs.next();
```

```
rs.deleteRow ();
```

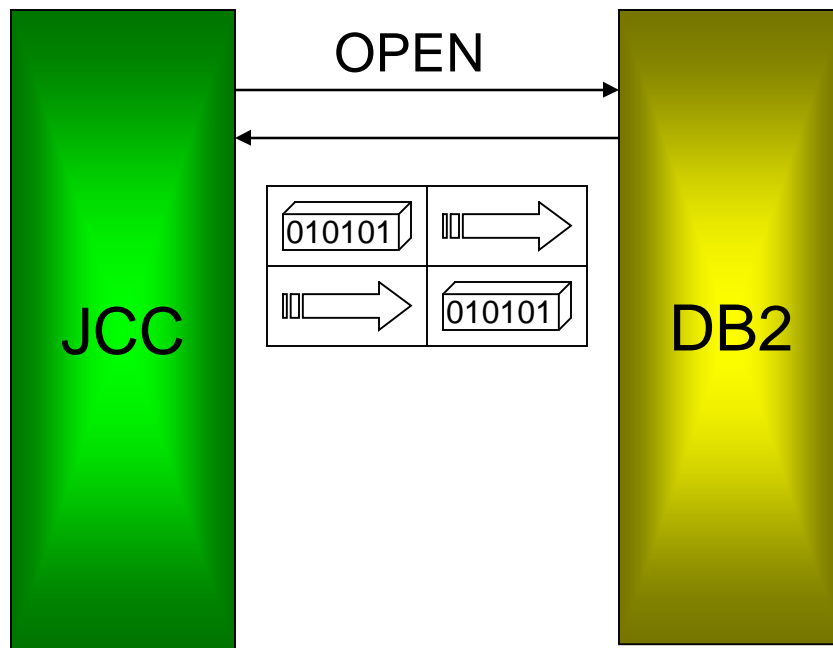
or

```
delete.execute ("DELETE FROM PAYROLL" +  
"WHERE CURRENT OF " + rs.getCursorName () +  
" FOR ROW 3 OF ROWSET");
```

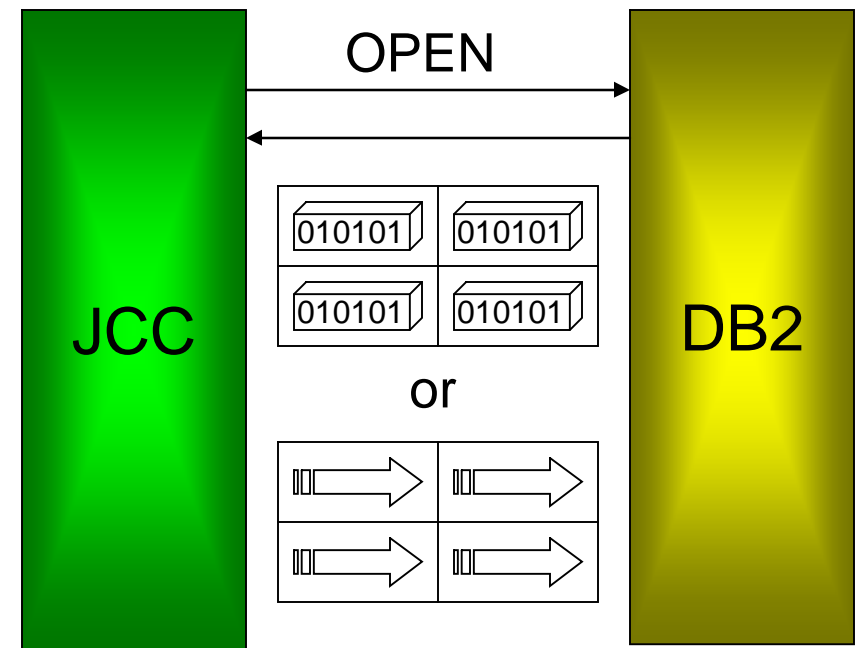
Progressive Streaming

Progressive Streaming

- Also known as dynamic data format for T4 driver
- Databases decide what to return for large objects
- Better performance and memory management



With Progressive Streaming



Without Progressive Streaming

Progressive Streaming (continued)

- What to return is based on the actual size
 - Materialized data (size \leq streamBufferSize)
 - Streaming data (size $>$ streamBufferSize)
 - streamBufferSize by default is 1M

```
ResultSet rs = s.executeQuery(  
    "SELECT VIDEO FROM CATALOG");  
while (rs.next()) {  
    Blob video = rs.getBlob("VIDEO");  
    byte[] videoBytes = video.getBytes(1, 1000);  
}
```

All materialized data
are returned here

Each streaming data is
requested as needed

Server Specifics

- DB2 for z/OS
 - Enabled by default (or via progressiveStreaming YES for some earlier JCC version)
 - Apply to version 9 and above
 - Apply to CLOB, BLOB, and XML
- DB2 for LUW
 - Enabled by default
 - Apply to version 9.5 and above
 - Apply to CLOB and BLOB

Comparison

- With progressive streaming
 - Databases determine what to return
 - A variety of data depending on the actual size
 - Cursor based (with less resource hogging)
 - Sequential-access based
- Without progressive streaming
 - Applications explicitly request what to return
 - Either all locators or all materialized data
 - Transaction based (with longer life span)
 - Random-access based

Potential Issue

- Large objects are no longer available after the cursor is moved or closed

```
Map map = new HashMap();
```

```
ResultSet rs = s.executeQuery(
```

```
    "SELECT ID, VIDEO FROM CATALOG");
```

```
while (rs.next())
```

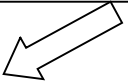
```
    map.put(rs.getInt("ID"), rs.getBlob("VIDEO"));
```

```
...
```

```
Blob video = (Blob)map.get(id);
```

```
byte[] videoBytes = video.getBytes(1, 1000);
```

Fail because blobs
are all closed already



Potential Issue (continued)

- Random access on streaming data may have negative performance impact

do {

```
System.out.println("END> 0 : PREV> 1 : NEXT> 2");
```

```
int direction = Integer.parseInt(input.readLine());
```

```
if (direction == PREV)
```

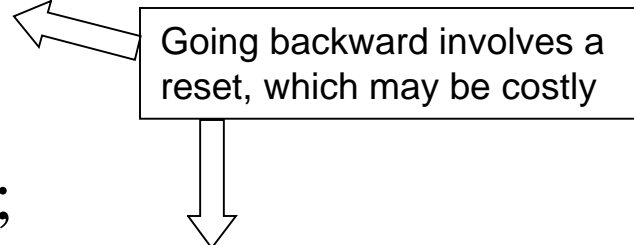
```
    position = position - 1000;
```

```
else if (direction == NEXT)
```

```
    position = position + 1000;
```

```
videoBytes = video.getBytes(position, 1000);
```

```
} while (direction != END);
```



Going backward involves a reset, which may be costly

Legacy Compatibility

- Disable progressive streaming via property setting NO, i.e. a constant 2
 - Pre-connect level
 - `properties.put("progressiveStreaming", "2")`
 - `dataSource.setProgressiveStreaming(2)`
 - Post-connect level
 - `connection.setDBProgressiveStreaming(2)`
 - Global level
 - Override: `db2.jcc.override.progressiveStreaming = 2`
 - Default: `db2.jcc.progressiveStreaming = 2`

XML Capabilities

Data Type

- A built-in data type on DB2 since version 9

	JCC 3 driver (db2jcc.jar)	JCC 4 driver (db2jcc4.jar)
Java class	com.ibm.db2.jcc.DB2Xml	java.sql.SQLXML
Metadata type	java.sql.Types.OTHER	java.sql.Types.SQLXML
Metadata type name	"XML"	"XML"

Data Query

- An XML data type can be mapped to string, bytes, stream, DB2Xml, and SQLXML in JCC

```
ResultSet rs = s.executeQuery(  
    "SELECT ORDER FROM BOOK");  
rs.next();  
String xmlString = rs.getString(1);  
byte[] xmlBytes = rs.getBytes(1);  
InputStream xmlStream = rs.getBinaryStream(1);  
DB2Xml db2xml = (DB2Xml)rs.getObject(1);  
SQLXML sqlxml = rs.getSQLXML(1);
```

Data Query (continued)

- Usage of DB2Xml interface

```
DB2Xml db2xml = (DB2Xml)rs.getObject(1);
```

```
byte[] db2xmlBytes =
```

```
    db2xml.getDB2XmlBytes("Cp930");
```

```
FileOutputStream fos =
```

```
    new FileOutputStream("Order.Cp930.xml");
```

```
fos.write(db2xmlBytes);
```

```
fos.flush();
```

Data Query (continued)

- Usage of SQLXML interface

```
SQLXML sqlxml = rs.getSQLXML(1);
```

```
SAXSource source =
```

```
    sqlxml.getSource(SAXSource.class);
```

```
XMLReader reader = source.getXMLReader();
```

```
ContentHandler myHandler = ...;
```

```
reader.setContentHandler(myHandler);
```

```
reader.parse(source.getInputSource());
```

Data Manipulation

- Various methods for XML columns

```
PreparedStatement ps = c.prepareStatement(  
    "INSERT INTO BOOK (ORDER) VALUES (?)");  
ps.setString(1, xmlString);  
ps.execute();  
ps.setBytes(1, xmlBytes);  
ps.execute();  
ps.setBinaryStream(1, xmlStream);  
ps.execute();
```

Data Manipulation (continued)

```
ps.setObject(1, db2xml);
```

```
ps.execute();
```

```
ps.setSQLXML(1, sqlxml);
```

```
ps.execute();
```

```
InputStreamReader isr =
```

```
    new InputStreamReader(
```

```
        new FileInputStream("Order.Cp930.xml"), "Cp930");
```

```
ps.setCharacterStream(1, isr);
```

```
ps.execute();
```

Data Manipulation (continued)

- Usage of SQLXML interface

```
SQLXML sqlxml = c.createSQLXML();
```

```
SAXResult result = sqlxml.setResult(SAXResult.class);
```

```
ContentHandler handler = result.getHandler();
```

```
handler.startDocument();
```

```
... // write elements and attributes to the result
```

```
handler.endDocument();
```

Questions/Comments

