|  |  |
| --- | --- |
| |  | | --- | | JDBC | |

**JDBC drivers are divided into four types or levels. The different types of jdbc drivers are:**

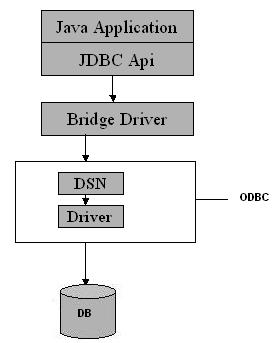
**Type 1:** JDBC-ODBC Bridge driver (Bridge)  
**Type 2:** Native-API/partly Java driver (Native)  
**Type 3:** AllJava/Net-protocol driver (Middleware)  
**Type 4:** All Java/Native-protocol driver (Pure)

**4 types of jdbc drivers** are elaborated in detail as shown below:

## Type 1 JDBC Driver

**JDBC-ODBC Bridge driver**

The Type 1 driver translates all JDBC calls into ODBC calls and sends them to the ODBC driver. ODBC is a generic API. The JDBC-ODBC Bridge driver is recommended only for experimental use or when no other alternative is available.

  
Type 1: JDBC-ODBC Bridge

Advantage

The JDBC-ODBC Bridge allows access to almost any database, since the database’s ODBC drivers are already available.

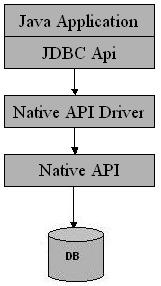
Disadvantages

1. Since the Bridge driver is not written fully in Java, Type 1 drivers are not portable.  
2. A performance issue is seen as a JDBC call goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process. They are the slowest of all driver types.  
3. The client system requires the ODBC Installation to use the driver.  
4. Not good for the Web.

## Type 2 JDBC Driver

**Native-API/partly Java driver**

The distinctive characteristic of type 2 jdbc drivers are that Type 2 drivers convert JDBC calls into database-specific calls i.e. this driver is specific to a particular database. Some distinctive characteristic of type 2 jdbc drivers are shown below. Example: Oracle will have oracle native api.

  
Type 2: Native api/ Partly Java Driver

Advantage

The distinctive characteristic of type 2 jdbc drivers are that they are typically offer better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less than that of Type  
1 and also it uses Native api which is Database specific.

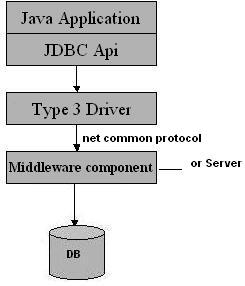
Disadvantage

1. Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet.  
2. Like Type 1 drivers, it’s not written in Java Language which forms a portability issue.  
3. If we change the Database we have to change the native api as it is specific to a database  
4. Mostly obsolete now  
5. Usually not thread safe.

### Type 3 JDBC Driver

**All Java/Net-protocol driver**

Type 3 database requests are passed through the network to the middle-tier server. The middle-tier then translates the request to the database. If the middle-tier server can in turn use Type1, Type 2 or Type 4 drivers.

  
Type 3: All Java/ Net-Protocol Driver

Advantage

1. This driver is server-based, so there is no need for any vendor database library to be present on client machines.  
2. This driver is fully written in Java and hence Portable. It is suitable for the web.  
3. There are many opportunities to optimize portability, performance, and scalability.  
4. The net protocol can be designed to make the client JDBC driver very small and fast to load.  
5. The type 3 driver typically provides support for features such as caching (connections, query results, and so on), load balancing, and advanced  
system administration such as logging and auditing.  
6. This driver is very flexible allows access to multiple databases using one driver.  
7. They are the most efficient amongst all driver types.

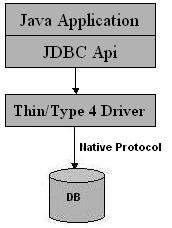
Disadvantage

It requires another server application to install and maintain. Traversing the recordset may take longer, since the data comes through the backend server.

### Type 4 JDBC Driver

**Native-protocol/all-Java driver**

The Type 4 uses java networking libraries to communicate directly with the database server.

  
Type 4: Native-protocol/all-Java driver

Advantage

1. The major benefit of using a type 4 jdbc drivers are that they are completely written in Java to achieve platform independence and eliminate deployment administration issues. It is most suitable for the web.  
2. Number of translation layers is very less i.e. type 4 JDBC drivers don’t have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server, performance is typically quite good.  
3. You don’t need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

Disadvantage

With type 4 drivers, the user needs a different driver for each database.

**PreparedStatement Performance**

It takes time for a database to parse an SQL string, and create a query plan for it. A query plan is an analysis of how the database can execute the query in the most efficient way.

If you submit a new, full SQL statement for every query or update to the database, the database has to parse the SQL and for queries create a query plan. By reusing an existing PreparedStatement you can reuse both the SQL parsing and query plan for subsequent queries. This speeds up query execution, by decreasing the parsing and query planning overhead of each execution. There are two levels of potential reuse for a PreparedStatement.

1. Reuse of PreparedStatement by the JDBC driver.
2. Reuse of PreparedStatement by the database.

First of all, the JDBC driver can cache PreparedStatement objects internally, and thus reuse the PreparedStatement objects. This may save a little of the PreparedStatement creation time.

Second, the cached parsing and query plan could potentially be reused across Java applications, for instance application servers in a cluster, using the same database.

Here is a diagram illustrating the caching of statements in the database:

|  |
| --- |
|  |
| **The caching of PreparedStatement's in the database.** |

The diagram does not show the JDBC driver PreparedStatement cache. You will have to imagine that.

# JDBC: Batch Updates

A batch update is a batch of updates grouped together, and sent to the database in one "batch", rather than sending the updates one by one.

Sending a batch of updates to the database in one go, is faster than sending them one by one, waiting for each one to finish. There is less network traffic involved in sending one batch of updates (only 1 round trip), and the database might be able to execute some of the updates in parallel. The speed up compared to executing the updates one by one, can be quite big.

You can batch both SQL inserts, updates and deletes. It does not make sense to batch select statements.

There are two ways to execute batch updates:

1. Using a Statement
2. Using a PreparedStatement

This text explains both ways.

**Statement Batch Updates**

You can use a Statement object to execute batch updates. You do so using the addBatch() and executeBatch() methods. Here is an example:

Statement statement = null;

try{

statement = connection.createStatement();

statement.addBatch("update people set firstname='John' where id=123");

statement.addBatch("update people set firstname='Eric' where id=456");

statement.addBatch("update people set firstname='May' where id=789");

int[] recordsAffected = statement.executeBatch();

} finally {

if(statement != null) statement.close();

}

First you add the SQL statements to be executed in the batch, using the addBatch() method.

Then you execute the SQL statements using the executeBatch(). The int[] array returned by the executeBatch() method is an array of int telling how many records were affected by each executed SQL statement in the batch.

**PreparedStatement Batch Updates**

You can also use a PreparedStatement object to execute batch updates. The PreparedStatement enables you to reuse the same SQL statement, and just insert new parameters into it, for each update to execute. Here is an example:

String sql = "update people set firstname=? , lastname=? where id=?";

PreparedStatement preparedStatement = null;

try{

preparedStatement =

connection.prepareStatement(sql);

preparedStatement.setString(1, "Gary");

preparedStatement.setString(2, "Larson");

preparedStatement.setLong (3, 123);

preparedStatement.addBatch();

preparedStatement.setString(1, "Stan");

preparedStatement.setString(2, "Lee");

preparedStatement.setLong (3, 456);

preparedStatement.addBatch();

int[] affectedRecords = preparedStatement.executeBatch();

}finally {

if(preparedStatement != null) {

preparedStatement.close();

}

}

First a PreparedStatement is created from an SQL statement with question marks in, to show where the parameter values are to be inserted into the SQL.

Second, each set of parameter values are inserted into the preparedStatement, and the addBatch() method is called. This adds the parameter values to the batch internally. You can now add another set of values, to be inserted into the SQL statement. Each set of parameters are inserted into the SQL and executed separately, once the full batch is sent to the database.

Third, the executeBatch() method is called, which executes all the batch updates. The SQL statement plus the parameter sets are sent to the database in one go. The int[] array returned by the executeBatch() method is an array of int telling how many records were affected by each executed SQL statement in the batch.

**Batch Updates and Transactions**

It is important to keep in mind, that each update added to a Statement or PreparedStatement is executed separately by the database. That means, that some of them may succeed before one of them fails. All the statements that have succeeded are now applied to the database, but the rest of the updates may not be. This can result in an inconsistent data in the database.

To avoid this, you can execute the batch update inside a [transaction](http://tutorials.jenkov.com/jdbc/transaction.html). When executed inside a transaction you can make sure that either all updates are executed, or none are. Any successful updates can be rolled back, in case one of the updates fail.

**Key Terminology**

**Database**—The whole point of JDBC is to provide a uniform connection between Java code and a database. The database might be anything from Microsoft Access to IBM’s DB2.

**Driver**—The driver is the core component for JDBC. Drivers are written by vendors and must support the basic features of the JDBC specification.

**Connection**—A Connection is a Java interface defining a link to a database. The Connection is essentially a pipeline between your code and the database.

**Statement**—A Statement is a Java interface that represents messages sent from your code to the database. These statements can use database-specific SQL or a form of SQL-92 that is compatible across all database systems. Newer, nonstandard SQL features are supported through special “escape sequences” that enable a programmer to switch database platforms without changing the SQL code.

**ResultSet**—A ResultSet is a Java interface representing a set of data drawn from the database. Several kinds of SQL statements, including stored procedures and SELECT statements, may return one or more ResultSet objects.

**Why Should I Use Connection Pooling Rather Than Managing My Own Connections?**

Creating, opening, and closing a Connection involves a small but significant amount of overhead in both time and memory. Using connection pooling provides several advantages. First, the Connection objects are already established and waiting for use. This reduces the time and memory overhead for creating a new Connection each time you want to perform a database call. Second, an outside system handles the life cycle of the Connection, including creating, closing, and generating new connections. This means you can devote more time to your specific task and less time to the husbandry of connecting to a database. Third, fully developed connection pooling is often provided as a feature with EJB platforms such as BEA WebLogic and some JDBC 2.0 Type 3 drivers that support javax.sql.ConnectionPoolDataSource (e.g., Merant’s DataDirect; www.merant.com). These features can save you considerable time and effort in the development of your classes.

**JDBC Statements**

The JDBC standard provides for three basic types of statements: Statement, PreparedStatement, and CallableStatement. Each interface has a different purpose, although all three can be used for issuing SQL directives to a database.

**Statement**

The basic Statement object is the parent class for the other two Statement types. This interface allows you to submit basic SQL to the database and retrieve ResultSet objects.

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery( "SELECT \* FROM stocks");

To create a Statement, you need a valid Connection. Then you can use the Connection as a factory to generate new statements using the createStatement() method. Once you have a valid Statement object, you can use execute(), executeQuery(), or executeUpdate() to perform SQL actions.

**PreparedStatement**

In some cases, you will want to reuse an SQL statement. The PreparedStatement interface extends the Statement interface by generating a precompiled SQL statement that you can reuse. Because it is precompiled on the database, it will run faster for repeated calls. The disadvantage is that each PreparedStatement is locked into a single, specific SQL command.

Using the Statement method, you generate the SQL on the fly. With a PreparedStatement, on the other hand, you create the SQL and then insert the values as a second step.

//Create the statement

String mySQL = "INSERT INTO stocks (ticker, last\_bid ) " +

VALUES (?, ?)";

PreparedStatement prepStmt = conn.prepareStatement(mySQL);

//Insert the values to replace the ? placeholders

prepStmt.clearParameters();

prepStmt.setString(1, "MSFT");

prepStmt.setFloat(2, 60.5);

/\*\*Use executeUpdate() since this is an INSERT statement rowCount

will hold the number of rows added, which should be 1 \*\*/

int rowCount = prepStmt.executeUpdate();

**ExecuteUpdate vs ExecuteQuery vs Execute**

Depending upon what type of SQL you want to execute, you will generally need to use the executeUpdate() or executeQuery() methods. In cases where your SQL may return multiple ResultSets, or when you do not know the actual SQL command at compile-time, you can use the Execute() method, which supports both queries and updates.

**ExecuteUpdate**

The executeUpdate() command is used for SQL statements that do not directly return data from the database. This includes UPDATE, DELETE, INSERT, and ALTER statements. The executeUpdate() method will typically return an integer indicating the number of rows affected for UPDATE, DELETE, and INSERT commands. For other SQL statements, the executeUpdate() method returns a value of 0 for successful execution, although you should use the getWarnings() method to check for any nonfatal exceptions.

**ExecuteQuery**

If you are requesting data from the database, or expect data to be returned by your statement, you need to use executeQuery(). This method is used almost exclusively for SELECT statements or calls to stored procedures that have fixed arguments. The return value is always a ResultSet object.

**Execute**

The execute() command is a generic method for executing SQL of unknown type. You should use this method when you don’t know whether the SQL will return a ResultSet or not. The execute() method returns a boolean value, true if a ResultSet was returned and false if an update count is waiting.

In order to access the data returned from an execute() command, you must use the getResultSet() or getUpdateCount() command. The following code fragment illustrates how this works:

/\*\* assume we built a Statement called stmt and then executed some

SQL of unknown type. It could be either a Query or an Update \*\*/

boolean rsCheck = stmt.execute(sqlString);

/\*\*if rsCheck is true, then it's a Query, otherwise it was an

update\*\*/

if (rsCheck) {

/\*\*The ResultSet is waiting in the Statement and we need to

use getResultSet to extract it. \*\*/

ResultSet rs1 = stmt.getResultSet();

} else {

/\*\*The SQL was an Update, so we need to use getUpdateCount to

find out how many rows were affected. Remember that for some

types of queries getUpdateCount will always be 0 \*\*/

int rowCount = stmt.getUpdateCount();

}

**CallableStatement**

The CallableStatement interface extends the PreparedStatement interface with features specific to the handling of stored procedures. CallableStatement is very similar to PreparedStatement in that you replace input (**IN**) and output (**OUT**) values with a “?” character.

/\*\*Assume we have a stored procedure with this declaration:

get\_name(in varchar username, out varchar fullName) \*\*/

String mySQL = "{call get\_name(?,?)}";

CallableStatement callStmt =conn.prepareCall(mySQL);

callStmt.setString(1, "lbrillo");

callStmt.registerOutParameter(2,Types.STRING);

callStmt.execute();

String myOut = callStmt.getString(2);

/\*\*Note that we extract the second parameter. Parameters arenumbered from left to right, regardless of whether they are in, out or inout \*\*/

There are a few key points to consider when working with **IN** and **OUT** parameters:

* All parameters are numbered from left to right, starting with 1. In a stored procedure that has a return value, the return value is number 1 and the first IN parameter is number 2.
* Input parameters are set the same way as for a PreparedStatement. Use the appropriate setXXXX() method for the datatype, then pass in the parameter number and the value.
* RegisterOutParameter() requires a parameter number and a java.sql.Types data type identifier. These types are formatted as Types.FLOAT, Types.STRING, Types.INTEGER, etc.

Once you have passed in all IN parameters and registered all OUT parameters, you will be able to execute the statement using the execute() command. After execution, you can access the results using getXXXX() methods. The getXXXX() methods are data-type-specific and require the parameter number.

// Assume conn is valid connection object

CallableStatement queryStatement = conn.prepareCall("{call get\_products\_created\_after(?, ?)}");

queryStatement.registerOutParameter(1,oracle.jdbc.driver.OracleTypes.CURSOR);

java.util.GregorianCalendar desiredDate =new java.util.GregorianCalendar(1980, 1, 1);

queryStatement.setDate(2,new java.sql.Date( desiredDate.getTime().getTime() ) );

queryStatement.execute();

ResultSet rs = ((oracle.jdbc.driver.OracleCallableStatement)stmt).getCursor(1);

while (rs.next()) {

// Display objects.

}

rs.close()

queryStatement.close();

***How Do I Use a Stored Procedure with INOUT Parameters?***

In the earlier section on CallableStatements, both IN (input) and OUT (output) parameters were demonstrated. A third type of parameter is both input and output (INOUT). When your database and drivers support INOUT parameters, you can use them by “overloading” a parameter with both a setXXXX() method and a registerOutParameter method. For example:

//my\_stored\_proc has a single INOUT parameter.

String mySQL = "{call my\_stored\_proc(?)}";

CallableStatement callStmt = conn.prepareCall(mySQL);

callStmt.setString(1, "This is an IN");

callStmt.registerOutParameter(1, Types.STRING);

callStmt.execute();

String myOut = cstmt.getString(1);

***Basic Patterns***

JDBC operations can usually be reduced to a simple pattern of boilerplate code. The steps are:

1. Create the Connection, or get one from a pool.

2. Create Statement, PreparedStatement, or CallableStatement.

3. Do the work (execute statement, process results).

4. Handle any exceptions.

5. Close the Statement and Connection (in a finally block).