Batch Processing in JDBC

Last modified: October 20, 2018

by baeldung

Java+
Persistence

I just announced the new *Spring Boot 2* material, coming in REST With Spring:

>> CHECK OUT THE COURSE

1. Introduction

Java Database Connectivity (JDBC) is a Java API used for interacting with databases. Batch processing groups multiple queries into one unit and passes it in a single network trip to a database.

In this article, we'll discover how JDBC can be used for batch processing of SQL queries.

For more on JDBC, you can check out our introduction article here.

2. Why Batch Processing?

Performance and data consistency are the primary motives to do batch processing.

2.1. Improved Performance

Some use cases require a large amount of data to be inserted into a database table. While using JDBC, one of the ways to achieve this without batch processing, is to execute multiple queries sequentially.

Let's see an example of sequential queries sent to database:

```
1
2  statement.execute("INSERT INTO EMPLOYEE(ID, NAME,
3  DESIGNATION) "
4
    + "VALUES ('1','EmployeeName1','Designation1')");

   statement.execute("INSERT INTO EMPLOYEE(ID, NAME,
   DESIGNATION) "
```

```
    + "VALUES ('2','EmployeeName2','Designation2')");
```

These sequential calls will increase the number of network trips to database resulting in poor performance.

By using batch processing, these queries can be sent to the database in one call, thus improving performance.

2.2. Data Consistency

In certain circumstances, data needs to be pushed into multiple tables. This leads to an interrelated transaction where the sequence of queries being pushed is important.

Any errors occurring during execution should result in a rollback of the data pushed by previous queries if any.

Let’s see an example of adding data to multiple tables:

```
1
2  statement.execute("INSERT INTO EMPLOYEE(ID, NAME,
3  DESIGNATION) "
4
    + "VALUES ('1','EmployeeName1','Designation1')");

   statement.execute("INSERT INTO EMP_ADDRESS(ID,
   EMP_ID, ADDRESS) "

    + "VALUES ('10','1','Address')");
```

A typical problem in the above approach arises when the first statement succeeds and the second statement fails. **In this situation there is no rollback of the data inserted by the first statement, leading to data inconsistency.**

We can achieve data consistency by spanning a transaction across multiple insert/updates and then committing the transaction at the end or performing a rollback in case of exceptions, but in this case, we're still hitting the database repeatedly for each statement.

3. How To Do Batch Processing

JDBC provides two classes, *Statement* and *PreparedStatement* to execute queries on the database. Both classes have their own implementation of the *addBatch()* and *executeBatch()* methods which provide us with the batch processing functionality.

3.1. Batch Processing Using *Statement*

With JDBC, the simplest way to execute queries on a database is via the *Statement* object.

First, using *addBatch()* we can add all SQL queries to a batch and then execute those SQL queries using *executeBatch()*.

The return type of *executeBatch()* is an *int* array indicating how many records were affected by the execution of each SQL statement.

Let's see an example of creating and executing a batch using Statement:

```
1
2  Statement statement = connection.createStatement();
3
4  statement.addBatch("INSERT INTO EMPLOYEE(ID, NAME,
5  DESIGNATION) "
6
```

```
 + "VALUES ('1','EmployeeName','Designation')");

statement.addBatch("INSERT INTO EMP_ADDRESS(ID,
EMP_ID, ADDRESS) "

 + "VALUES ('10','1','Address')");

statement.executeBatch();
```

In the above example, we are trying to insert records into the *EMPLOYEE* and *EMP_ADDRESS* tables using *Statement*. We can see how SQL queries are being added in the batch to be executed.

3.2. Batch Processing Using *PreparedStatement*

PreparedStatement is another class used to execute SQL queries. It enables reuse of SQL statements and requires us to set new parameters for each update/insert.

Let's see an example using *PreparedStatement.* First, we set up the statement using an SQL query encoded as a *String:*

```
1
2  String[] EMPLOYEES = new
3  String[]{"Zuck","Mike","Larry","Musk","Steve"};
4
5  String[] DESIGNATIONS = new
6  String[]{"CFO","CSO","CTO","CEO","CMO"};
```

```
String insertEmployeeSQL = "INSERT INTO EMPLOYEE(ID,
NAME, DESIGNATION) "
```

```
     + "VALUES (?,?,?)";

    PreparedStatement employeeStmt =
    connection.prepareStatement(insertEmployeeSQL);
```

Next, we loop through an array of *String* values and add a newly configured query to the batch.

Once the loop is finished, we execute the batch:

```
1
2   for(int i = 0; i < EMPLOYEES.length; i++){
3
4       String employeeId = UUID.randomUUID().toString();
5
6       employeeStmt.setString(1,employeeId);
7
8       employeeStmt.setString(2,EMPLOYEES[i]);

        employeeStmt.setString(3,DESIGNATIONS[i]);

        employeeStmt.addBatch();

    }

    employeeStmt.executeBatch();
```

In the example shown above, we are inserting records into the *EMPLOYEE* table using *PreparedStatement.* We can see how values to be inserted are set in the query and then added to the batch to be executed.

4. Conclusion

In this article, we saw how batch processing of SQL queries are important while interacting with databases using JDBC.

As always, the code related to this article can be found over on Github.

https://github.com/eugenp/tutorials/tree/master/persistence-modules/core-java-persistence

==========

JDBC - Batch Processing

Previous Page
Next Page

Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.

When you send several SQL statements to the database at once, you reduce the amount of communication overhead, thereby improving performance.

- JDBC drivers are not required to support this feature. You should use the *DatabaseMetaData.supportsBatchUpdates()* method to determine if the target database supports batch update processing. The method returns true if your JDBC driver supports this feature.
- The addBatch() method of *Statement, PreparedStatement,* and *CallableStatement* is used to add individual statements to the batch. The executeBatch() is used to start the execution of all the statements grouped together.
- The executeBatch() returns an array of integers, and each element of the array represents the update count for the respective update statement.
- Just as you can add statements to a batch for processing, you can remove them with the clearBatch() method. This method removes all the statements you

added with the addBatch() method. However, you cannot selectively choose which statement to remove.

Batching with Statement Object

Here is a typical sequence of steps to use Batch Processing with Statement Object –

- Create a Statement object using either *createStatement()* methods.
- Set auto-commit to false using *setAutoCommit()*.
- Add as many as SQL statements you like into batch using *addBatch()*method on created statement object.
- Execute all the SQL statements using *executeBatch()* method on created statement object.
- Finally, commit all the changes using *commit()* method.

Example

The following code snippet provides an example of a batch update using Statement object –

```
// Create statement object
Statement stmt = conn.createStatement();

// Set auto-commit to false
conn.setAutoCommit(false);

// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
             "VALUES(200,'Zia', 'Ali', 30)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create one more SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
```

```
             "VALUES(201,'Raj', 'Kumar', 35)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create one more SQL statement
String SQL = "UPDATE Employees SET age = 35 " +
             "WHERE id = 100";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create an int[] to hold returned values
int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes
conn.commit();
```

For a better understanding, let us study the Batching - Example Code.

Batching with PrepareStatement Object

Here is a typical sequence of steps to use Batch Processing with PrepareStatement Object –

1. Create SQL statements with placeholders.
2. Create PrepareStatement object using either *prepareStatement()*methods.
3. Set auto-commit to false using *setAutoCommit()*.
4. Add as many as SQL statements you like into batch using *addBatch()*method on created statement object.
5. Execute all the SQL statements using *executeBatch()* method on created statement object.
6. Finally, commit all the changes using *commit()* method.

The following code snippet provides an example of a batch update using PrepareStatement object –

```
// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
```

```
              "VALUES(?, ?, ?, ?)";

// Create PrepareStatement object
PreparedStatemen pstmt = conn.prepareStatement(SQL);

//Set auto-commit to false
conn.setAutoCommit(false);

// Set the variables
pstmt.setInt( 1, 400 );
pstmt.setString( 2, "Pappu" );
pstmt.setString( 3, "Singh" );
pstmt.setInt( 4, 33 );
// Add it to the batch
pstmt.addBatch();

// Set the variables
pstmt.setInt( 1, 401 );
pstmt.setString( 2, "Pawan" );
pstmt.setString( 3, "Singh" );
pstmt.setInt( 4, 31 );
// Add it to the batch
pstmt.addBatch();

//add more batches
.
.
.
.
//Create an int[] to hold returned values
int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes
conn.commit();
```

For a better understanding, let us study the Batching - Example Code.

==========

JDBC Batching with PrepareStatement Object

Previous Page

Next Page

Here is a typical sequence of steps to use Batch Processing with PrepareStatement Object –

- Create SQL statements with placeholders.
- Create PrepareStatement object using either *prepareStatement()*methods.
- Set auto-commit to false using *setAutoCommit()*.
- Add as many as SQL statements you like into batch using *addBatch()*method on created statement object.
- Execute all the SQL statements using *executeBatch()* method on created statement object.
- Finally, commit all the changes using *commit()* method.

This sample code has been written based on the environment and database setup done in the previous chapters.

Copy and past the following example in JDBCExample.java, compile and run as follows –

```
// Import required packages
import java.sql.*;

public class JDBCExample {
   // JDBC driver name and database URL
   static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
   static final String DB_URL = "jdbc:mysql://localhost/EMP";

   //  Database credentials
   static final String USER = "username";
   static final String PASS = "password";

   public static void main(String[] args) {
   Connection conn = null;
   PreparedStatement stmt = null;
   try{
      // Register JDBC driver
      Class.forName("com.mysql.jdbc.Driver");

      // Open a connection
      System.out.println("Connecting to database...");
```

```
      conn = DriverManager.getConnection(DB_URL,USER,PASS);

      // Create SQL statement
      String SQL = "INSERT INTO Employees(id,first,last,age) " +
                   "VALUES(?, ?, ?, ?)";

      // Create preparedStatemen
      System.out.println("Creating statement...");
      stmt = conn.prepareStatement(SQL);

      // Set auto-commit to false
      conn.setAutoCommit(false);

      // First, let us select all the records and display them.
      printRows( stmt );

      // Set the variables
      stmt.setInt( 1, 400 );
      stmt.setString( 2, "Pappu" );
      stmt.setString( 3, "Singh" );
      stmt.setInt( 4, 33 );
      // Add it to the batch
      stmt.addBatch();

      // Set the variables
      stmt.setInt( 1, 401 );
      stmt.setString( 2, "Pawan" );
      stmt.setString( 3, "Singh" );
      stmt.setInt( 4, 31 );
      // Add it to the batch
      stmt.addBatch();

      // Create an int[] to hold returned values
      int[] count = stmt.executeBatch();

      //Explicitly commit statements to apply changes
      conn.commit();

      // Again, let us select all the records and display them.
      printRows( stmt );

      // Clean-up environment
      stmt.close();
      conn.close();
   }catch(SQLException se){
      //Handle errors for JDBC
      se.printStackTrace();
   }catch(Exception e){
      //Handle errors for Class.forName
      e.printStackTrace();
```

```
   }finally{
      //finally block used to close resources
      try{
         if(stmt!=null)
            stmt.close();
      }catch(SQLException se2){
      }// nothing we can do
      try{
         if(conn!=null)
            conn.close();
      }catch(SQLException se){
         se.printStackTrace();
      }//end finally try
   }//end try
   System.out.println("Goodbye!");
}//end main

public static void printRows(Statement stmt) throws SQLException{
   System.out.println("Displaying available rows...");
   // Let us select all the records and display them.
   String sql = "SELECT id, first, last, age FROM Employees";
   ResultSet rs = stmt.executeQuery(sql);

   while(rs.next()){
      //Retrieve by column name
      int id  = rs.getInt("id");
      int age = rs.getInt("age");
      String first = rs.getString("first");
      String last = rs.getString("last");

      //Display values
      System.out.print("ID: " + id);
      System.out.print(", Age: " + age);
      System.out.print(", First: " + first);
      System.out.println(", Last: " + last);
   }
   System.out.println();
   rs.close();
}//end printRows()
}//end JDBCExample
```

Now let us compile above example as follows –

```
C:\>javac JDBCExample.java
C:\>
```

When you run JDBCExample, it produces the following result –

```
C:\>java JDBCExample
Connecting to database...
```

```
Creating statement...
Displaying available rows...
ID: 95, Age: 20, First: Sima, Last: Chug
ID: 100, Age: 35, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 30, First: Sumit, Last: Mittal
ID: 110, Age: 20, First: Sima, Last: Chug
ID: 200, Age: 30, First: Zia, Last: Ali
ID: 201, Age: 35, First: Raj, Last: Kumar

Displaying available rows...
ID: 95, Age: 20, First: Sima, Last: Chug
ID: 100, Age: 35, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 30, First: Sumit, Last: Mittal
ID: 110, Age: 20, First: Sima, Last: Chug
ID: 200, Age: 30, First: Zia, Last: Ali
ID: 201, Age: 35, First: Raj, Last: Kumar
ID: 400, Age: 33, First: Pappu, Last: Singh
ID: 401, Age: 31, First: Pawan, Last: Singh
Goodbye!
C:\>
```

==========

JDBC Batch Tutorial

In this tutorial, we will see "**how to execute multiple queries using a single JDBC Statement**".

1. Add allowMultiQueries=true to mysql url
2. Get the database connection
3. Create Statement
4. Use method statement.addBatch(sql) to add list of SQL statements.
5. invoke statement.executeBatch() to execute all SQL queries

JDBC Batch Tutorial java code:

```
String url = "jdbc:mysql://localhost:3306/database_name?autoReconnect=true&allowMultiQueries=true";
 String driver = "com.mysql.jdbc.Driver";
 String userName= "root";
```

```
String password= "abc";
Class.forName(driver).newInstance();
String sql1= "delete from table1 where id="+101;
String sql2= "delete from table2 where id="+101;
String sql3= "delete from table3 where id="+101;
String sql4= "insert into auditlog values(101, 'deleted from table1')";
Connection connection = null;
Statement statement = null;
try{
connection = DriverManager.getConnection(url, userName, password);
statement = connection.createStatement();
statement.addBatch(sql1);
statement.addBatch(sql2);
statement.addBatch(sql3);
statement.addBatch(sql4);
statement.executeBatch();
}catch(Exception e){
e.printStackTrace();
}
try {
statement.close();
} catch (SQLException e) {
e.printStackTrace();
}
try {
connection.close();
} catch (SQLException e) {
e.printStackTrace();
}
```

← Previous post

Next post →

==========

JDBC: Batch Updates

- Statement Batch Updates
- PreparedStatement Batch Updates

- Batch Updates and Transactions

Jakob Jenkov

Last update: 2014-06-23

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**. 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.
==========

JDBC Batch insert update MySQL Oracle

https://www.journaldev.com/2494/jdbc-batch-insert-update-mysql-oracle

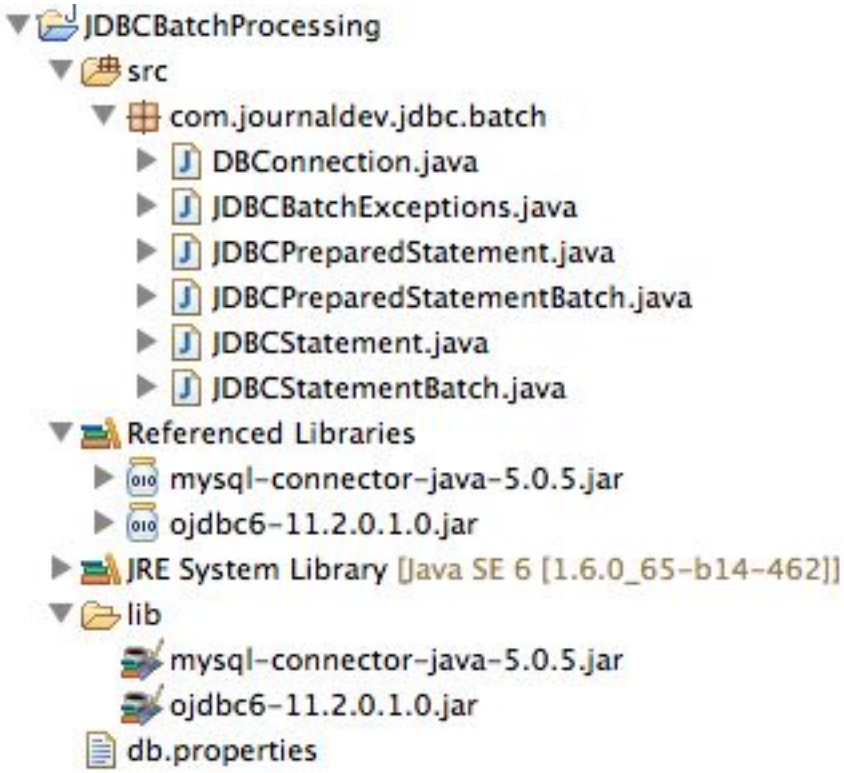
PANKAJ 18 COMMENTS

Today we will look into JDBC Batch insert and update examples in MySQL and Oracle databases. Sometimes we need to run bulk queries of similar kind for a database, for example loading data from CSV files to relational database tables. As we know that we have option to use `Statement` or `PreparedStatement`to execute queries. Apart from that JDBC provides Batch Processing feature through which we can execute bulk of queries in one go for a database.

JDBC Batch

JDBC batch statements are processed through Statement and PreparedStatement `addBatch()` and `executeBatch()` methods. This tutorial is aimed to provide details about JDBC Batch insert example for MySQL and Oracle database.

We will look into different programs so we have a project with structure as below image.



Notice that I have MySQL and Oracle DB JDBC Driver jars in the project build path, so that we can run our application across MySQL and Oracle DB both.

Let's first create a simple table for our test programs. We will run bulk of JDBC insert queries and look at the performance with different approaches.

```
--Oracle DB
CREATE TABLE Employee (
  empId NUMBER NOT NULL,
  name varchar2(10) DEFAULT NULL,
  PRIMARY KEY (empId)
);

--MySQL DB
CREATE TABLE `Employee` (
  `empId` int(10) unsigned NOT NULL,
  `name` varchar(10) DEFAULT NULL,
  PRIMARY KEY (`empId`)
```

```
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

We will read the Database configuration details from property file, so that switching from one database to another is quick and easy.

db.properties

```
#mysql DB properties
DB_DRIVER_CLASS=com.mysql.jdbc.Driver
DB_URL=jdbc:mysql://localhost:3306/UserDB
#DB_URL=jdbc:mysql://localhost:3306/UserDB?rewriteBatchedStatements=
true
DB_USERNAME=pankaj
DB_PASSWORD=pankaj123

#Oracle DB Properties
#DB_DRIVER_CLASS=oracle.jdbc.driver.OracleDriver
#DB_URL=jdbc:oracle:thin:@localhost:1871:UserDB
#DB_USERNAME=scott
#DB_PASSWORD=tiger
```

Before we move into actual JDBC batch insert example to insert bulk data into Employee table, let's write a simple utility class to get the database connection.

DBConnection.java

```
package com.journaldev.jdbc.batch;

import java.io.FileInputStream;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

public class DBConnection {
```

```
	public static Connection getConnection() {
		Properties props = new Properties();
		FileInputStream fis = null;
		Connection con = null;
		try {
			fis = new FileInputStream("db.properties");
			props.load(fis);

			// load the Driver Class
			Class.forName(props.getProperty("DB_DRIVER_CLASS"));

			// create the connection now
			con =
DriverManager.getConnection(props.getProperty("DB_URL"),
					props.getProperty("DB_USERNAME"),
					props.getProperty("DB_PASSWORD"));
		} catch (IOException e) {
			e.printStackTrace();
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		} catch (SQLException e) {
			e.printStackTrace();
		}
		return con;
	}
}
```

Now let's look at different approach we can take for jdbc batch insert example.

1. Use Statement to execute one query at a time.
2. JDBCStatement.java
3.

```
package com.journaldev.jdbc.batch;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
```

```
public class JDBCStatement {

 public static void main(String[] args) {

        Connection con = null;
        Statement stmt = null;

        try {
                con = DBConnection.getConnection();
                stmt = con.createStatement();

                long start = System.currentTimeMillis();
                for(int i =0; i<10000;i++){
                        String query = "insert into Employee values
("+i+",'Name"+i+"')";
                        stmt.execute(query);
                }
                System.out.println("Time
Taken="+(System.currentTimeMillis()-start));

        } catch (SQLException e) {
                e.printStackTrace();
        }finally{
                try {
                        stmt.close();
                        con.close();
                } catch (SQLException e) {
                        e.printStackTrace();
                }
        }
 }

}
```

4. Use PreparedStatement to execute one query at a time.
5. JDBCPreparedStatement.java
6.

```
package com.journaldev.jdbc.batch;
```

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class JDBCPreparedStatement {

 public static void main(String[] args) {

        Connection con = null;
        PreparedStatement ps = null;
        String query = "insert into Employee (empId, name)
values (?,?)";
        try {
                con = DBConnection.getConnection();
                ps = con.prepareStatement(query);

                long start = System.currentTimeMillis();
                for(int i =0; i<10000;i++){
                        ps.setInt(1, i);
                        ps.setString(2, "Name"+i);
                        ps.executeUpdate();
                }
                System.out.println("Time
Taken="+(System.currentTimeMillis()-start));

        } catch (SQLException e) {
                e.printStackTrace();
        }finally{
                try {
                        ps.close();
                        con.close();
                } catch (SQLException e) {
                        e.printStackTrace();
                }
        }
 }

}
```

7. This approach is similar as using Statement but PreparedStatement provides performance benefits and avoid SQL injection attacks.
8. Using Statement Batch API for bulk processing.
9. JDBCStatementBatch.java
10.

```
package com.journaldev.jdbc.batch;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCStatementBatch {

 public static void main(String[] args) {

        Connection con = null;
        Statement stmt = null;

        try {
                con = DBConnection.getConnection();
                stmt = con.createStatement();

                long start = System.currentTimeMillis();
                for(int i =0; i<10000;i++){
                        String query = "insert into Employee values
("+i+",'Name"+i+"')";
                        stmt.addBatch(query);

                        //execute and commit batch of 1000 queries
                        if(i%1000 ==0) stmt.executeBatch();
                }
                //commit remaining queries in the batch
                stmt.executeBatch();

                System.out.println("Time
Taken="+(System.currentTimeMillis()-start));

        } catch (SQLException e) {
                e.printStackTrace();
```

```
        }finally{
            try {
                stmt.close();
                con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

}
```

11. We are processing 10,000 records with batch size of 1000 records. Once the batch size reaches, we are executing it and continue processing remaining queries.
12. Using PreparedStatement Batch Processing API for bulk queries.
13. JDBCPreparedStatementBatch.java
14.

```
package com.journaldev.jdbc.batch;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class JDBCPreparedStatementBatch {

    public static void main(String[] args) {

        Connection con = null;
        PreparedStatement ps = null;
        String query = "insert into Employee (empId, name)
values (?,?)";
        try {
            con = DBConnection.getConnection();
            ps = con.prepareStatement(query);

            long start = System.currentTimeMillis();
            for(int i =0; i<10000;i++){
                ps.setInt(1, i);
```

```
                ps.setString(2, "Name"+i);

                ps.addBatch();

                if(i%1000 == 0) ps.executeBatch();
            }
            ps.executeBatch();

            System.out.println("Time
Taken="+(System.currentTimeMillis()-start));

        } catch (SQLException e) {
            e.printStackTrace();
        }finally{
            try {
                ps.close();
                con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

}
```

Let’s see how our programs work with MySQL database, I have executed them separately multiple times and below table contains the results.

MYSQL DB	STATEMENT	PREPAREDSTATEMENT	STATEMENT BATCH	PREPAREDSTATEMENT BATCH

Time Taken (ms)	8256	8130	7129	7019

When I looked at the response time, I was not sure whether it's right because I was expecting some good response time improvements with Batch Processing. So I looked online for some explanation and found out that by default MySQL batch processing works in similar way like running without batch. To get the actual benefits of Batch Processing in MySQL, we need to pass `rewriteBatchedStatements` as TRUE while creating the DB connection. Look at the MySQL URL above in **db.properties** file for this.

With `rewriteBatchedStatements` as `true`, below table provides the response time for the same programs.

MYSQL DB	**STATEMENT**	**PREPAREDSTATEMENT**	**STATEMENT BATCH**	**PREPAREDSTATEMENT BATCH**
Time Taken (ms)	5676	5570	3716	394

As you can see that PreparedStatement Batch Processing is very fast when rewriteBatchedStatements is true. So if you have a lot of batch processing involved, you should use this feature for faster processing.

Oracle Batch Insert

When I executed above programs for Oracle database, the results were in line with MySQL processing results and PreparedStatement Batch processing was much faster than any other approach.

JDBC Batch Processing Exceptions

Let's see how batch programs behave incase one of the queries throw exceptions.

JDBCBatchExceptions.java

```
package com.journaldev.jdbc.batch;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Arrays;

public class JDBCBatchExceptions {

	public static void main(String[] args) {

		Connection con = null;
		PreparedStatement ps = null;
		String query = "insert into Employee (empId, name) values
(?,?)";
		try {
			con = DBConnection.getConnection();

			ps = con.prepareStatement(query);

			String name1 = "Pankaj";
			String name2="Pankaj Kumar"; //longer than column
length
			String name3="Kumar";

			ps.setInt(1, 1);
			ps.setString(2, name1);
```

```
                ps.addBatch();

                ps.setInt(1, 2);
                ps.setString(2, name2);
                ps.addBatch();

                ps.setInt(1, 3);
                ps.setString(2, name3);
                ps.addBatch();

                int[] results = ps.executeBatch();

                System.out.println(Arrays.toString(results));

            } catch (SQLException e) {
                e.printStackTrace();
            }finally{
                try {
                    ps.close();
                    con.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }

}
```

When I executed above program for MySQL database, I got below exception and none of the records were inserted in the table.

```
com.mysql.jdbc.MysqlDataTruncation: Data truncation: Data too long for column 'name' at row 2
        at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:2939)
        at com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:1623)
        at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:1715)
        at com.mysql.jdbc.Connection.execSQL(Connection.java:3249)
        at
```

```
com.mysql.jdbc.PreparedStatement.executeInternal(PreparedStatement.java:1268)
	at com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1541)
	at com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1455)
	at com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1440)
	at com.mysql.jdbc.PreparedStatement.executeBatchedInserts(PreparedStatement.java:1008)
	at com.mysql.jdbc.PreparedStatement.executeBatch(PreparedStatement.java:908)
	at com.journaldev.jdbc.batch.JDBCBatchExceptions.main(JDBCBatchExceptions.java:37)
```

When executed the same program for Oracle database, I got below exception.

```
java.sql.BatchUpdateException: ORA-12899: value too large for column "SCOTT"."EMPLOYEE"."NAME" (actual: 12, maximum: 10)

	at oracle.jdbc.driver.OraclePreparedStatement.executeBatch(OraclePreparedStatement.java:10070)
	at oracle.jdbc.driver.OracleStatementWrapper.executeBatch(OracleStatementWrapper.java:213)
	at com.journaldev.jdbc.batch.JDBCBatchExceptions.main(JDBCBatchExceptions.java:38)
```

But the rows before exception were inserted into the database successfully. Although the exception clearly says what the error is but it doesn't tell us which query is causing

the issue. So either we validate the data before adding them for batch processing or we should use JDBC Transaction Management to make sure all or none of the records are getting inserted incase of exceptions.

Same program with jdbc transaction management looks like below.

JDBCBatchExceptions.java

```
package com.journaldev.jdbc.batch;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Arrays;

public class JDBCBatchExceptions {

	public static void main(String[] args) {

		Connection con = null;
		PreparedStatement ps = null;
		String query = "insert into Employee (empId, name) values
(?,?)";
		try {
			con = DBConnection.getConnection();
			con.setAutoCommit(false);

			ps = con.prepareStatement(query);

			String name1 = "Pankaj";
			String name2="Pankaj Kumar"; //longer than column
length
			String name3="Kumar";

			ps.setInt(1, 1);
			ps.setString(2, name1);
			ps.addBatch();
```

```
                ps.setInt(1, 2);
                ps.setString(2, name2);
                ps.addBatch();

                ps.setInt(1, 3);
                ps.setString(2, name3);
                ps.addBatch();

                int[] results = ps.executeBatch();

                con.commit();
                System.out.println(Arrays.toString(results));

            } catch (SQLException e) {
                e.printStackTrace();
                try {
                    con.rollback();
                } catch (SQLException e1) {
                    e1.printStackTrace();
                }
            }finally{
                try {
                    ps.close();
                    con.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }

}
```

As you can see that I am rolling back the transaction if any SQL exception comes. If the batch processing is successful, I am explicitly committing the transaction.

That's all for JDBC Batch insert update example, make sure to experiment with your data to get the optimal value of batch size for bulk queries.

One of the limitation of jdbc batch processing is that we can't execute different type of queries in the batch.

==========

Batch Processing Example
Posted by: Chandan Singh in sql January 2nd, 2015 0 355 Views

In this example, we will see how we can use Batch Processing in Java Database Connectivity(i.e.JDBC).

When multiple inserts are to be made to the table in a database, the trivial way is to execute a query per record.
However, this involves acquiring and releasing connection every time a record is inserted, which hampers application performance.

We overcome, this(acquiring and releasing connection every-time) by making use of Batch operations in JDBC.
Want to be a JDBC Master ?
Subscribe to our newsletter and download the JDBC Ultimate Guide right now!
In order to help you master database programming with JDBC, we have compiled a kick-ass guide with all the major JDBC features and use cases! Besides studying them online you may download the eBook in PDF format!
Download NOW!
We set the parameters in the java.sql.PreparedStatement and it to the batch using PreparedStatement.addBatch() method. Then when the batch size reaches a desired threshold, we execute the batch using PreparedStatement.executeBatch() method.
TIP:
Another way to avoid manually releasing/acquiring connection is to use Connection Pooling. However, when executing Queries in Batch, it is sometimes important to maintain the atomicity of the database. This can be a problem if one the commands in the batch throws some error since the commands after the exception will not be executed leaving the database in an inconsistent state. So we set the auto-commit to false and if all the records are executed successfully, we commit the transaction. This maintains the integrity of the the database.

We will try to understand points explained above with the help of an example :

BatchExample.java:

```
01
package com.javacodegeeks.examples;
02

03
import java.sql.Connection;
04
import java.sql.PreparedStatement;
05
import java.sql.SQLException;
06
import java.util.Arrays;
07

08

09
public class BatchExample
10
{
11
   public static void main(String[] args)
12
   {
13
      try (Connection connection = DBConnection.getConnection())
14
      {
15
         connection.setAutoCommit(false);
16

17
         try (PreparedStatement pstmt = connection.prepareStatement("Insert into txn_tbl
(txn_id,txn_amount, card_number, terminal_id) values (null,?,?,?)");)
18
         {
19
            pstmt.setString(1, "123.45");
20
            pstmt.setLong(2, 2345678912365L);
21
```

```
            pstmt.setLong(3, 1234567L);
22
            pstmt.addBatch();
23
24
            pstmt.setString(1, "456.00");
25
            pstmt.setLong(2, 567512198454L);
26
            pstmt.setLong(3, 1245455L);
27
            pstmt.addBatch();
28
29
            pstmt.setString(1, "7859.02");
30
            pstmt.setLong(2, 659856423145L);
31
            pstmt.setLong(3, 5464845L);
32
            pstmt.addBatch();
33
34
            int[] arr = pstmt.executeBatch();
35
            System.out.println(Arrays.toString(arr));
36
37
            connection.commit();
38
        }
39
        catch (SQLException e)
40
        {
41
            e.printStackTrace();
42
            connection.rollback();
```

```
43          }
44      }
45      catch (Exception e)
46      {
47          e.printStackTrace();
48      }
49    }
50 }
```

DBConnection.java:

```
01 package com.javacodegeeks.examples;
02
03
04 import java.sql.Connection;
05 import java.sql.DriverManager;
06 import java.sql.SQLException;
07
08
09 /**
10 * @author Chandan Singh
11 */
12 public class DBConnection
13
```

```
{
14
   public static Connection getConnection() throws SQLException, ClassNotFoundException
15
   {
16
      Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/jcg?rewriteBatchedStatements=true"
, "root", "toor");
17

18
      return connection;
19
   }
20

21
}
```

One important point to note here is the connection URL . The rewriteBatchedStatements=true is important since it nudges the JDBC to pack as many queries as possible into a single network data packet, thus lowering the traffic. Without that parameter, there will not be much performance improvement when using JDBC Batch.

NOTE:
We can use JDBC Batch, when using java.sql.Statement, in similar fashion.
The PreparedStatement.executeBatch() method retruns an int array. Depending upon the values, we can know the status of the each executed queries:

A value greater than zero usually indicates successful execution of query.
A value equal to Statement.SUCCESS_NO_INFO indicates, successful command execution but no record count is available.
A value equal to Statement.EXECUTE_FAILED indicates, command was not executed successfully. It shows up only if the driver continued to execute the commands after the failed command.
Summary:
Here we tried to understand what is JDBC Batch and how we can the use the same to reduce network traffic and improve our application performance.

Download
https://examples.javacodegeeks.com/core-java/sql/jdbc-batch-processing/