4.2. Batch Processing

Complex systems often require both online and batch processing. Each kind of processing has very different requirements. Because online processing involves a user waiting on application processing order, the timing and performance of each statement execution in a process is important. Batch processing, on the other hand, occurs when a bunch of distinct transactions need to occur independently of user interaction. A bank's ATM machine is an example of a system of online processes. The monthly process that calculates and adds interest to your savings account is an example of a batch process.

JDBC 2.0 introduced new functionality to address the specific issues of batch processing. Using the JDBC 2.0 batch facilities, you can assign a series of SQL statements to a JDBC `Statement` (or one of its subclasses) to be submitted together for execution by the database. Using the techniques you have learned so far in this book, account interest-calculation processing occurs roughly in the following fashion:

1. Prepare statement.
2. Bind parameters.
3. Execute.
4. Repeat steps 2 and 3 for each account.

This style of processing requires a lot of "back and forth" between the Java application and the database. JDBC 2.0 batch processing provides a simpler, more efficient approach to this kind of processing:

1. Prepare statement.
2. Bind parameters.
3. Add to batch.
4. Repeat steps 2 and 3 until interest has been assigned for each account.
5. Execute.

Under batch processing, there is no "back and forth" between the database for each account. Instead, all Java-level processing—the binding of parameters—occurs before you send the statements to the database. Communication with the database occurs in one huge burst; the huge bottleneck of stop and go communication with the database is gone.

`Statement` and its children all support batch processing through an `addBatch( )` method. For `Statement` , `addBatch()` accepts a `String` that is the SQL to be executed as part of the batch. The `PreparedStatement` and `CallableStatement` classes, on the other hand, use `addBatch()` to bundle a set of parameters together as part of a single element in the batch. The following code shows how to use a `Statement` object to batch process interest calculation:

```
Statement stmt = conn.createStatement( );
int[] rows;

for(int i=0; i<accts.length; i++) {
    accts[i].calculateInterest( );
    stmt.addBatch("UPDATE account " +
                  "SET balance = " +
                  accts[i].getBalance( ) +
                  "WHERE acct_id = " + accts[i].getID( ));
}
rows = stmt.executeBatch( );
```

The `addBatch()` method is basically nothing more than a tool for assigning a bunch of SQL statements to a JDBC `Statement` object for execution together. Because it makes no sense to manage results in batch processing, the

statements you pass to addBatch( ) should be some form of an update: a CREATE, INSERT, UPDATE, or DELETE. Once you are done assigning SQL statements to the object, call executeBatch( ) to execute them. This method returns an array of row counts of modified rows. The first element, for example, contains the number of rows affected by the first statement in the batch. Upon completion, the list of SQL calls associated with the Statement instance is cleared.

This example uses the default auto-commit state in which each update is committed automatically.[*] If an error occurs somewhere in the batch, all accounts before the error will have their new balance stored in the database, and the subsequent accounts will not have had their interest calculated. The account where the error occurred will have an account object whose state is inconsistent with the database. You can use the getUpdateCounts( ) method in the BatchUpdateException thrown by executeBatch( ) to get the value executeBatch( ) should have otherwise returned. The size of this array tells you exactly how many statements executed successfully.

[*] Doing batch processing using a Statement results in the same inefficiencies you have already seen in Statement objects because the database must repeatedly rebuild the same query plan.

In a real-world batch process, you will not want to hold the execution of the batch until you are done with all accounts. If you do so, you will fill up the transaction log used by the database to manage its transactions and bog down database performance. You should therefore turn auto-commit off and commit changes every few rows while performing batch processing.

Using prepared statements and callable statements for batch processing is very similar to using regular statements. The main difference is that a batch prepared or callable statement represents a single SQL statement with a list of parameter groups, and the database should create a query plan only once. Calculating interest with a prepared statement would look like this:

```
PreparedStatement stmt = conn.prepareStatement(
    "UPDATE account SET balance = ? WHERE acct_id = ?");
int[] rows;

for(int i=0; i<accts.length; i++) {
    accts[i].calculateInterest( );
    stmt.setDouble(1, accts[i].getBalance( ));
    stmt.setLong(2, accts[i].getID( ));
    stmt.addBatch( );
}
rows = stmt.executeBatch( );
```

Example 4.1 provides the full example of a batch program that runs a monthly password-cracking program on people's passwords. The program sets a flag in the database for each bad password so a system administrator can act appropriately.

Example4.1. A Batch Process to Mark Users with Easy-to-Crack Passwords

```
import java.sql.*;
import java.util.ArrayList;
import java.util.Iterator;

public class Batch {
    static public void main(String[] args) {
        Connection conn = null;

        try {
            // we will store the bad UIDs in this list
            ArrayList breakable = new ArrayList( );
            PreparedStatement stmt;
            Iterator users;
            ResultSet rs;

            Class.forName(args[0]).newInstance( );
            conn = DriverManager.getConnection(args[1],
                                               args[2],
                                               args[3]);
            stmt = conn.prepareStatement("SELECT user_id, password " +
                                         "FROM user");
            rs = stmt.executeQuery( );
            while( rs.next( ) ) {
                String uid = rs.getString(1);
                String pw = rs.getString(2);

                // Assume PasswordCracker is some class that provides
                // a single static method called crack( ) that attempts
                // to run password cracking routines on the password
```

```
                if( PasswordCracker.crack(uid, pw) ) {
                    breakable.add(uid);
                }
            }
            stmt.close( );
            if( breakable.size( ) < 1 ) {
                return;
            }
            stmt = conn.prepareStatement("UPDATE user " +
                                         "SET bad_password = 'Y' " +
                                         "WHERE uid = ?");
            users = breakable.iterator( );
            // add each UID as a batch parameter
            while( users.hasNext( ) ) {
                String uid = (String)users.next( );

                stmt.setString(1, uid);
                stmt.addBatch( );
            }
            stmt.executeBatch( );
        }
        catch( Exception e ) {
            e.printStackTrace( );
        }
        finally {
            if( conn != null ) {
                try { conn.close( ); }
                catch( Exception e ) { }
            }
        }

              }
}
```