4.3. Updatable Result Sets

If you remember scrollable result sets from Chapter 3, you may recall that one of the parameters you used to create a scrollable result set was something called the *result set concurrency* . So far, the statements in this book have used the default concurrency, `ResultSet.CONCUR_READ_ONLY`. In other words, you cannot make changes to data in the result sets you have seen without creating a new update statement based on the data from your result set. Along with scrollable result sets, JDBC 2.0 also introduces the concept of *updatable result sets*—result sets you can change.

An updatable result set enables you to perform in-place changes to a result set and have them take effect using the current transaction. I place this discussion after batch processing because the only place it really makes sense in an enterprise environment is in large-scale batch processing. An overnight interest-assignment process for a bank is an example of such a potential batch process. It would read in an accounts balance and interest rate and, while positioned at that row in the database, update the interest. You naturally gain efficiency in processing since you do everything at once. The downside is that you perform database access and business logic together.

JDBC 2.0 result sets have two types of concurrency: `ResultSet.CONCUR_READ_ONLY` and `ResultSet.CONCUR_UPDATABLE` . You already know how to create an updatable result set from the discussion of scrollable result sets in Chapter 3. You pass the concurrency type `ResultSet.CONCUR_UPDATABLE` as the second argument to `createStatement()`, or the third argument to `prepareStatement()` or `prepareCall()`:

```
PreparedStatement stmt = conn.prepareStatement(
          "SELECT acct_id, balance FROM account",
          ResultSet.TYPE_SCROLL_SENSITIVE,
          ResultSet.CONCUR_UPDATABLE);
```

The most important thing to remember about updatable result sets is that you must always select from a single table and include the primary key columns. If you don't, the concept of the result set being updatable is nonsensical. After all, updatable result set only constructs a hidden `UPDATE` for you. If it does not know what the unique identifier for the row in question is, there is no way it can construct a valid update.

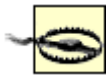
JDBC drivers are not required to support updatable result sets. The driver is, however, required to let you create result sets of any type you like. If you request `CONCUR_UPDATABLE` and the driver does not support it, it issues a `SQLWarning` and assigns the result set to a type it can support. It will not throw an exception until you try to use a feature of an unsupported result set type. Later in the chapter, I discuss the `DatabaseMetaData` class and how you can use it to determine if a specific type of concurrency is supported.

4.3.1. Updates

JDBC 2.0 introduces a set of `updateXXX( )` methods to match its `getXXX()` methods and enable you to update a result set. For example, `updateString(1, "violet")` enables your application to replace the current value for column 1 of the current row in the result set with a `string` that has the value `violet`. Once you are done modifying columns, call `updateRow( )` to make the changes permanent in the database. You naturally cannot make changes to primary key columns. Updates look like this:

```
while( rs.next( ) ) {
    long acct_id = rs.getLong(1);
    double balance = rs.getDouble(2);

    balance = balance + (balance * 0.03)/12;
    rs.updateDouble(2, balance);
    rs.updateRow( );
}
```



While this code does look simpler than batch processing, you should remember that it is a poor approach to enterprise-class problems. Specifically, imagine that you have been running a bank using this simple script run once a month to manage interest accumulation. After two years, you find that your business processes change—perhaps because of growth or a merger. Your new business processes introduce complex business rules pertaining to the accumulation of interest and general rules regarding balance changes. If this code is the only place where you have done direct data access, implementing interest accumulation and managing balance adjustments—a highly unlikely bit of luck—you could migrate to a more robust solution. On the other hand, your bank is probably like most systems and has code like this all over the place. You now have a total mess on your hands when it comes to managing the evolution of your business processes.

4.3.2. Deletes

Deletes are naturally much simpler than updates. Rather than setting values, you just have to call `deleteRow( )`. This method will delete the current row out from under you and out of the database.

4.3.3. Inserts

Inserting a new row into a result set is the most complex operation of updatable result sets because inserts introduce a few new steps into the process. The first step is to create a row for update via the method `moveToInsertRow( )`. This method creates a row that is basically a scratchpad for you to work in. This new row becomes your current row. As with other rows, you can call `getXXX()` and `updateXXX( )` methods on it to retrieve or modify its values. Once you are done making changes, call `insertRow( )` to make the changes permanent. Any values you fail to set are assumed to be `null`. The following code demonstrates the insertion of a new row using an updatable result set:

```
rs.moveToInsertRow( );
rs.updateString(1, "newuid");
rs.updateString(2, "newpass");
rs.insertRow( );
rs.moveToCurrentRow( );
```

The seemingly peculiar call to `moveToCurrentRow( )` returns you to the row you were on before you attempted to insert the new row.

In addition to requiring the result set to represent a single table in the database with no joins and fetch all the primary keys of the rows to be changed, inserts require that the result set has fetched—for each matching row—all non-`null` columns and all columns without default values.

4.3.4. Visibility of Changes

Chapter 3 mentioned two different types of scrollable result sets without diving into the details surrounding their differences. I ignored those differences specifically because they deal with the visibility of changes in updatable result sets. They determine how sensitive a result set is to changes to its underlying data. In other words, if you go back and retrieve values from a modified column, will you see the changes or the initial values? `ResultSet.TYPE_SCROLL_SENSITIVE` result sets are sensitive to changes in the underlying data, while `ResultSet.TYPE_SCROLL_INSENSITIVE` result sets are not. This may sound straightforward, but the devil is truly in the details.

How these two result set types manifest themselves is first dependent on something called *transaction isolation*. Transaction isolation identifies the visibility of your changes at a transaction level. In other words, what visibility do the actions of one transaction have to another? Can another transaction read your uncommitted database changes? Or, if another transaction does a select in the middle of your update transaction, will it see the old data?

Transactional parlance talks of several visibility issues that JDBC transaction isolation is designed to address. These issues are *dirty reads*, *repeatable reads*, and *phantom reads*. A dirty read means that one transaction can see uncommitted changes from another transaction. If the uncommitted changes are rolled back, the other transaction is said to have "dirty data"—thus the term dirty read.

A repeatable read occurs when one transaction always reads the same data from the same query no matter how many times the query is made or how many changes other transactions make to the rows read by the first transaction. In other words, a transaction that mandates repeatable reads will not see the committed changes made by another transaction. Your application needs to start a new transaction to see those changes.

The final issue, phantom reads, deals with changes occurring in other transactions that would result in new rows matching your where clause. Consider the situation in which you have a transaction reading all accounts with a balance less than $100. Your application logic makes two reads of that data. Between the two reads, another transaction adds a new account to the database with a balance of $0. That account will now match your query. If

your transaction isolation allows phantom reads, you will see that "phantom row." If it disallows phantom reads, then you will see the same result set you saw the first time.

The tradeoff in transaction isolations is performance versus consistency. Transaction isolation levels that avoid dirty, nonrepeatable, phantom reads will be consistent for the life of a transaction. Because the database has to worry about a lot of issues, however, transaction processing will be much slower. JDBC specifies the following transaction isolation levels:

`TRANSACTION_NONE`

: The database or the JDBC driver does not support transactions of any sort.

`TRANSACTION_READ_UNCOMMITTED`

: The transaction allows dirty reads, nonrepeatable reads, or phantom reads.

`TRANSACTION_READ_COMMITTED`

: Only data committed to the database can be read. It will, however, allow nonrepeatable reads and phantom reads.

`TRANSACTION_REPEATABLE_READ`

: Committed, repeatable reads as well as phantom reads are allowed. Nonrepeatable reads are not allowed.

`TRANSACTION_SERIALIZABLE`

: Only committed, repeatable reads are allowed. Phantom reads are specifically disallowed at this level.

You can find the transaction isolation of a connection by calling its `getTransactionIsolation( )` method. This visibility applies to updatable result sets as it does to other transaction components. Transaction isolation does not address the issue of one result set reading changes made by itself or other result sets in the same transaction. That visibility is determined by the result set type.

A `ResultSet.TYPE_SCROLL_INSENSITIVE` result set does not see any changes made by other transactions or other elements of the same transaction. `ResultSet.TYPE_SCROLL_SENSITIVE` result sets, on the other hand, see all updates to data made by other elements of the same transaction. Inserts and deletes may or may not be visible. You should note that any update that might affect the order of the result set—such as an update that modifies a column in an `ORDER BY` clause—acts like a `DELETE` followed by an `INSERT` and thus may or may not be visible.

4.3.5. Refreshing Data from the Database

In addition to all of these visibility issues, JDBC 2.0 provides a mechanism for getting up-to-the-second changes from the database. Not even a `TYPE_SCROLL_SENSITIVE` result set sees changes made by other transactions after it reads from the database. To go to the database and get the latest data for the current row, call the `refreshRow( )` method in your `ResultSet` instance.