ResultSet

NOTE: The material in this chapter is based on *JDBC*tm *API Tutorial and Reference, Second Edition: Universal Data Access for the Java*tm *2 Platform*, published by Addison Wesley as part of the Java series, ISBN 0-201-43328-1.

5.1 ResultSet Overview

A `ResultSet` is a Java object that contains the results of executing an SQL query. In other words, it contains the rows that satisfy the conditions of the query. The data stored in a `ResultSet` object is retrieved through a set of `get` methods that allows access to the various columns of the current row. The `ResultSet.next` method is used to move to the next row of the `ResultSet`, making it the current row.

The general form of a result set is a table with column headings and the corresponding values returned by a query. For example, if your query is `SELECT a, b, c FROM Table1`, your result set will have the following form:

```
a           b               c
----------  ------------    -----------
12345       Cupertino       2459723.495
83472       Redmond         1.0
83492       Boston          35069473.43
```

The following code fragment is an example of executing an SQL statement that will return a collection of rows, with column `a` as an `int`, column `b` as a `String`, and column `c` as a float:

```
java.sql.Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (rs.next()) {
        // retrieve and print the values for the current row
        int i = rs.getInt("a");
        String s = rs.getString("b");
        float f = rs.getFloat("c");
        System.out.println("ROW = " + i + " " + s + " " + f);
}
```

5.1.1 Rows and Columns

A relational database is made up of tables, with each table consisting of rows and columns. A row in a relational database table can be thought of as representing an instance of the entity that the table represents. For example, if there is a table of employees, each row will contain information about a particular employee. Each piece of data about the employee is stored in a column, so, for instance, the table of employees could have columns for an identification number, a name, a salary, and a date of hire. The columns in a row would contain the ID number, name, salary, and date of hire for a particular employee.

A result set is also a table with rows and columns, but it contains only the column values from a database table that satisfy the conditions of a query. In other words, a result set row will contain a subset of the columns in the underlying database table (unless the query selects everything in the table, in which case the result set table will include all of the column values for every row in the database table). In the past, a column value in a relational database table (and consequently in a result set table) had to be atomic; that is, it could be only one indivisible value. For instance, an array could not be a column value because an array may be made up of multiple elements. With the advent of SQL3 data types, however, the permissible content of table columns has expanded dramatically. It is now possible for an array or even a user-defined structured type to be a column value. Because this new capability allows a relational database to store instances of complex types as column values, it makes a relational database more like an object database, blurring the distinction between relational and object databases. Programmers can take advantage of these new data types if they use a JDBC 2.0 driver that supports SQL3 types.

5.1.2 Cursors

A `ResultSet` object maintains a cursor, which points to its current row of data. The cursor moves down one row each time the method `next` is called. When a `ResultSet` object is first created, the cursor is positioned before the first row, so the first call to the `next` method puts the cursor on the first row, making it the current row. `ResultSet` rows can be retrieved in sequence from top to bottom as the cursor moves down one row with each successive call to the method `next`. This ability to move its cursor only forward is the default behavior for a `ResultSet` and is the only cursor movement possible with drivers that implement only the JDBC 1.0 API. This kind of result set has the type `ResultSet.TYPE_FORWARD_ONLY` and is referred to as a forward only result set.

If a driver implements the cursor movement methods in the JDBC 2.0 core API, its result sets can be scrollable. A scrollable result set's cursor can move both forward and backward as well as to a particular row. The following methods move the cursor backward, to the first row, to the last row, to a particular row number, to a specified number of rows from the current row, and so on: `previous`, `first`, `last`, `absolute`, `relative`, `afterLast`, and `beforeFirst`. An explanation and example of how to make a result set scrollable will be presented in the section "Creating Different Types of Result Sets" on page 55.

When a cursor is positioned on a row in a `ResultSet` object (not before the first row or after the last row), that row becomes the current row. This means that any methods called while the cursor is positioned on that row will (1) operate on values in that row (methods such as `getXXX` and `updateXXX`), (2) operate on the row as a whole (such as the methods `updateRow`, `insertRow`, `deleteRow`, `refresh-Row`), or (3) use that row as a starting point for moving to other rows (such as the method `relative`).

A cursor remains valid until the `ResultSet` object or its parent `Statement` object is closed.

5.1.3 Cursor Movement Examples

As stated in the previous section, the standard cursor movement for forward only result sets is to use the method `next` to iterate through each row of a result set once from top to bottom. With scrollable result sets, it is possible to revisit a row or to iterate through the result set multiple times. This is possible because the cursor can be moved before the first row at any time (with the method beforeFirst method (ResultSet interface)>`beforeFirst`). The cursor can begin another iteration through the result set with the method `next`. The following example positions the cursor before the first row and then iterates forward through the contents of the result set. The methods `getString` and `getFloat` retrieve the column values for each row until there are no more rows, at which time the method `next` returns the value `false`.

```
rs.beforeFirst();
while (rs.next()) {
        System.out.println(rs.getString("EMP_NO") +
                " " + rs.getFloat("SALARY");
}
```

It is also possible to iterate through a result set backwards, as is shown in the next example. The cursor is first moved to the very end of the result set (with the method `afterLast`), and then the method `previous` is invoked within a `while` loop to iterate through the contents of the result set by moving to the previous row with each iteration. The method `previous` returns `false` when there are no more rows, so the loop ends after all the rows have been visited.

```
rs.afterLast();
while (rs.previous()) {
        System.out.println(rs.getString("EMP_NO") +
                " " + rs.getFloat("SALARY");
}
```

The interface `ResultSet` offers still other ways to iterate through the rows of a scrollable result set. Care should be taken, however, to avoid incorrect alternatives such as the one illustrated in the following example:

```
// incorrect!
while (!rs.isAfterLast()) {
  rs.relative(1);
  System.out.println(
        rs.getString("EMP_NO") + " " + rs.getFloat("SALARY"));
}
```

This example attempts to iterate forward through a scrollable result set and is incorrect for several reasons. One error is that if `ResultSet.isAfterLast` is called when the result set is empty, it will return a value of `false` since there is no last row. The loop body will be executed, which is not what is wanted. An additional problem occurs when the cursor is positioned before the first row of a result set that contains data. In this case, calling `rs.relative(1)` is erroneous because there is no current row. The method `relative` moves the cursor the specified number of rows from the current row, and it must be invoked only while the cursor is on the current row.

The following code fragment fixes the problems in the previous example. Here a call to the method `ResultSet.first` is used to distinguish the case of an empty result set from one that contains data. Because `ResultSet.isAfterLast` is called only when the result set is non-empty, the loop control works correctly. Since `ResultSet.first method (ResultSet interface)>first` initially positions the cursor on the first row, the method `ResultSet.relative(1)` steps through the rows of the result set as expected.

```
if (rs.first()) {
  while (!rs.isAfterLast()) {
    System.out.println(
         rs.getString("EMP_NO") + " " + rs.getFloat("SALARY"));
    rs.relative(1);
    }
}
```

5.1.4 Determining the Number of Rows in a Result Set

With the new cursor movement methods, it is easy to see how many rows a scrollable `ResultSet` object contains. All that is necessary is to go to the last row of the result set and get the number of that row. In the following example, rs will have one row for each employee.

```
ResultSet rs = stmt.executeQuery(
                "SELECT LAST_NAME, FIRST_NAME FROM EMPLOYEES");
rs.last();
int numberOfRows = rs.getRow();
System.out.println("XYZ, Inc. has " + numberOfRows + " employees");
rs.beforeFirst();
while (next()) {
        . . . // retrieve first and last names of each employee
}
```

Though not as convenient, it is also possible to find out how many rows a nonscrollable result set has. The following example shows one way to determine the number of rows.

```
ResultSet rs = stmt.executeQuery("SELECT COUNT(*) FROM EMPLOYEES");
rs.next();
```

```
    int count = rs.getInt(1);
    System.out.println("XYZ, Inc. has " + count + " employees");

    ResultSet rs2 = stmt.executeQuery(
                    "SELECT LAST_NAME, FIRST_NAME FROM EMPLOYEES");
    while (rs2.next()) {
            . . . // retrieve first and last names of each employee
    }
```

With the scrollable result set, the cursor was just repositioned to start iterating through the same result set to retrieve its data. In the preceding example, however, one query is needed to get the count, and another query is needed to get a result set with the data that is desired. Both queries must, of course, produce result sets of the same size for the count to be accurate.

A second way to determine the number of rows in a forward-only result set is to iterate through the result set, incrementing a variable with each iteration, which is shown in the following example. Because an application can iterate through a forward-only result set just once, the same query needs to be executed twice. In the iteration through the first rs, the number of rows is counted; in the iteration through the second rs, the data is retrieved.

```
    ResultSet rs = stmt.executeQuery(
                    "SELECT LAST_NAME, FIRST_NAME FROM EMPLOYEES");
    int count = 0;
    while (rs.next()) {
            count++;
    }
    System.out.println("Company XYZ has " + count " employees.");
    rs = stmt.executeQuery(
                    "SELECT LAST_NAME, FIRST_NAME FROM EMPLOYEES");
    while (rs.next()) {
            . . . // retrieve first and last names of each employee
    }
```

5.1.5 Retrieving Column Values

The `ResultSet.getXXX` methods provide the means for retrieving column values from the current row. For maximum portability with forward only result sets, values should be retrieved from left to right, and column values should be read only once. With scrollable result sets, however, there are no such restrictions.

Either the column name or the column number can be used to designate the column from which to retrieve data. For example, if the second column of a `ResultSet` object rs is named `TITLE`, and it stores values as strings, either of the following will retrieve the value stored in that column:

```
    String s = rs.getString(2);
    String s = rs.getString("TITLE");
```

Note that columns are numbered from left to right starting with column 1. Also, column names used as input to `getXXX` methods are case insensitive.

The option of using the column name was provided so that a user who specifies column names in a query can use those same names as the arguments to `getXXX` methods. If, on the other hand, the `SELECT` statement does not specify column names (as in "`SELECT * FROM TABLE1`" or in cases where a column is derived), column numbers should be used. In such situations, there is no way for the user to know for sure what the column names are.

In some cases, it is possible for an SQL query to return a result set that has more than one column with the same name. If a column name is used as the parameter to a `getXXX` method, `getXXX` will return the value of the first matching column name. Thus, if there are multiple columns with the same name, one needs to use a column index to be sure that the correct column value is retrieved. It may also be slightly more efficient to use column numbers.

If the name of a column is known but not its index, the method findColumn can be used to find the column number.

Information about the columns in a ResultSet is available by calling the method ResultSet.getMetaData. The ResultSetMetaData object returned gives the number, types, and properties of its ResultSet object's columns.

5.1.6 Which getXXX Method to Use

JDBC drivers support type coercion. When a getXXX method is invoked, the driver attempts to convert the underlying data to the type XXX in the Java programming language and then returns a suitable value. For example, if the getXXX method is getString, and the data type of the data in the underlying database is VARCHAR, the JDBC Compliant driver will convert the VARCHAR value to a String object in the Java programming language. That String object will be the value returned by getString.

The JDBC 2.0 API adds new ResultSet.getXXX methods for retrieving the new SQL3 data types. These methods work the same way the getXXX methods in the JDBC 1.0 API work; that is, they map the SQL3 JDBC type to a type in the Java programming language and return that type. For example, the method getClob retrieves a JDBC CLOB value from the database and returns a Clob object, which is an instance of the java.sql.Clob interface.

The method getObject will retrieve any data type. This is possible because Object, being the type from which every other object type in the Java programming language is derived, is the most generic type. This is especially useful when the underlying data type is a database-specific type or when a generic application needs to be able to accept any data type. The method getObject, as would be expected from its name, returns a Java Object that must be narrowed if it is to be used as a more specific type. In other words, it must be cast from its generic Object type to its more derived type before it can be used as that derived type. The following code fragment illustrates using the method getObject to retrieve a Struct value from the column ADDRESS in the current row of the ResultSet object rs. The Object that getObject returns is narrowed to a Struct before assigning it to the variable address.

```
      Struct address = (Struct)rs.getObject("ADDRESS");
```

The method getObject is not only the one method capable of retrieving values of any data type but also the only ResultSet.getXXX method that does custom mapping. Therefore, to be custom mapped, a data type has to be retrieved with the method getObject. The two SQL data types that can be custom mapped are the user-defined types, SQL structured types and DISTINCT types. A JDBC DISTINCT value is normally retrieved with the getXXX method appropriate for its underlying type, but if it has a custom mapping, it must be retrieved by the method getObject in order to be custom mapped. A JDBC STRUCT can only be retrieved with the method getObject, guaranteeing that if there is a custom mapping for a JDBC STRUCT value, it will be used.

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	CLOB	BLOB	ARRAY	REF	STRUCT	JAVA OBJECT
getByte	X	x	x	x	x	x	x	x	x	x	x	x	x												
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x												
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x												
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x												
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x												
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x												
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x												
getBoolean	x	x	x	x	x	x	x	x	x	X	x	x	x												
getString	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x						
getBytes														X	X	x									
getDate											x	x	x				X		x						
getTime											x	x	x					X	x						
getTimestamp											x	x	x				x	x	X						
getAsciiStream											x	x	X	x	x	x									
getUnicodeStream											x	x	X	x	x	x									
getBinaryStream														x	x	X									
getClob																				X					
getBlob																					X				
getArray																						X			
getRef																							X		
getCharacterStream											x	x	X	x	x	x									
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	X	X

Table 5.1: Use of `ResultSet.getXXX` Methods to Retrieve JDBC Types [D]

An "x" indicates that the `getXXX` method may legally be used to retrieve the given JDBC type.

An "**X**" indicates that the `getXXX` method is *recommended* for retrieving the given JDBC type.

5.1.7 Types of Result Sets

Results sets may have different levels of functionality. For example, they may be scrollable or nonscrollable. A scrollable result set has a cursor that moves both forward and backward and can be moved to a particular row. Also, result sets may be sensitive or insensitive to changes made while they are open; that is, they may or may not reflect changes to column values that are modified in the database. A developer should always keep in mind the fact that adding capabilities to a `ResultSet` object incurs additional overhead, so it should be done only as necessary.

Based on the capabilities of scrollability and sensitivity to changes, there are three types of result sets available with the JDBC 2.0 core API. The following constants, defined in the `ResultSet` interface, are used to specify these three types of result sets:

1. TYPE_FORWARD_ONLY
    - The result set is nonscrollable; its cursor moves forward only, from top to bottom.
    - The view of the data in the result set depends on whether the DBMS materializes results incrementally.
2. TYPE_SCROLL_INSENSITIVE
    - The result set is scrollable: Its cursor can move forward or backward and can be moved to a particular row or to a row whose position is relative to its current position.
    - The result set generally does not show changes to the underlying database that are made while it is open. The membership, order, and column values of rows are typically fixed when the result set is created.
3. TYPE_SCROLL_SENSITIVE
    - The result set is scrollable; its cursor can move forward or backward and can be moved to a particular row or to a row whose position is relative to its current position.
    - The result set is sensitive to changes made while it is open. If the underlying column values are modified, the new values are visible, thus providing a dynamic view of the underlying data. The membership and ordering of rows in the result set may be fixed or not, depending on the implementation.

5.1.8 Concurrency Types

A result set may have different update capabilities. As with scrollability, making a `ResultSet` object updatable increases overhead and should be done only when necessary. That said, it is often more convenient to make updates programmatically, and that can only be done if a result set is made updatable. The JDBC 2.0 core API offers two update capabilities, specified by the following constants in the `ResultSet` interface:

1. CONCUR_READ_ONLY
    - Indicates a result set that *cannot* be updated programmatically
    - The one concurrency type available to drivers that implement only the JDBC 1.0 API
    - Offers the highest level of concurrency (allows the largest number of simultaneous users). When a `ResultSet` object with read-only concurrency needs to set a lock, it uses a read-only lock. This allow users to read data but not to change it. Because there is no limit to the number of read-only locks that may be held on data at one time, there is no limit to the number of concurrent users unless the DBMS or driver imposes one.
2. CONCUR_UPDATABLE
    - Indicates a result set that *can* be updated programmatically
    - Available to drivers that implement the JDBC 2.0 core API
    - Reduces the level on concurrency. Updatable results sets may use write-only locks so that only one user at a time has access to a data item. This eliminates the possibility that two or more users might change the same data, thus ensuring database consistency. However, the price for this consistency is a reduced level of concurrency.

To allow a higher level of concurrency, an updatable result set may be implemented so that it uses an optimistic concurrency control scheme. This implementation assumes that conflicts will be rare and avoids using write-only locks, thereby permitting more users concurrent access to data. Before committing any updates, it determines whether a conflict has occurred by comparing rows either by value or by a version number. If there has been an update conflict between two transactions, one of the transactions will be aborted in order to maintain consistency. Optimistic concurrency control implementations can increase concurrency; however, if there are too many conflicts, they may actually reduce performance.

5.1.9 Providing Performance Hints

Many DBMSs and drivers are optimized to give the best performance under various circumstances, which means that generally a database programmer is best advised to use their default settings. However, for programmers who may want to fine tune database performance for a particular application, the JDBC 2.0 API provides methods that give hints to the driver for making access to result set data more efficient. These performance hints are exactly that, just hints; a JDBC Compliant driver may choose to ignore them.

The following two hints give the driver suggestions for improving performance:

1. The number of rows that should be fetched from the database each time new rows are needed The number of rows to be fetched is called the *fetch size*, and it can be set by two different methods: `Statement.setFetchSize` and `ResultSet.setFetchSize`. The statement that creates a `ResultSet` object sets the default fetch size for that `ResultSet` object, using the `Statement` method `setFetchSize`. The following code fragment sets the fetch size for the `ResultSet` object rs to 25. Until the fetch size is changed, any result set created by the `Statement` object stmt will automatically have a fetch size of 25.

   ```
   Statement stmt = con.createStatement();
   stmt.setFetchSize(25);
   ResultSet rs = stmt.executeQuery(SELECT * FROM EMPLOYEES);
   ```

   A result set can, at any time, change its default fetch size by setting a new fetch size with the `ResultSet` version of the method `setFetchSize`. Continuing from the previous code fragment, the following line of code changes the fetch size of rs to 50:

   ```
   rs.setFetchSize(50);
   ```

   Normally the most efficient fetch size is already the default for the driver. The method `setFetchSize` simply allows a programmer to experiment to see if a certain fetch size is more efficient than the default for a particular application.

2. The direction in which rows will be processed

   The interface `ResultSet` defines the following three constants for specifying the direction in which to process rows: `FETCH_FORWARD`, `FETCH_REVERSE`, and `FETCH_UNKNOWN`. As with the fetch size, there are two methods for setting the fetch direction, one in the interface `Statement`, and the other in the interface `ResultSet`. The statement that creates the result set determines the default fetch direction by using the `Statement` method `setFetchDirection`. The following code fragment sets the fetch direction for the `ResultSet` object rs so that it will process rows from the bottom up. Until the fetch direction is changed, any result set created by the `Statement` object stmt will automatically have a fetch direction of backward.

   ```
   Statement stmt = con.createStatement();
   stmt.setFetchDirection(FETCH_REVERSE);
   ResultSet rs = stmt.executeQuery(SELECT * FROM EMPLOYEES);
   ```

   A result set can, at any time, change its default fetch direction by setting a new fetch direction with the `ResultSet` method `setFetchDirection`. Continuing from the previous code fragment, the following line of code changes the fetch direction of rs to forward.

   ```
   rs.setFetchDirection(FETCH_FORWARD);
   ```

   The `ResultSet` object rs will hint that the driver process rows in a forward direction. This hint will be in effect until the method `ResultSet.setFetchDirection` is again called on rs to change the suggested fetch direction.

   As with the fetch size, drivers are commonly optimized to use the most efficient fetch direction, and changing the default may actually work against this optimization. The method `setFetchDirection` simply allows a programmer to try to fine tune an application for even better performance.

5.1.10 Creating Different Types of Result Sets

A result set is created by executing a query, and the type of result set depends on the arguments that are supplied to the `Connection` method `createStatement` (or `prepareStatement` or `prepareCall`). The following code fragment, which uses only JDBC 1.0 API, supplies no arguments to the method `createStatement` and thus creates a default `ResultSet` object, one that is forward-only and uses read-only concurrency.

```
Connection con = DriverManager.getConnection(
                "jdbc:my_subprotocol:my_subname");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(
        "SELECT EMP_NO, SALARY FROM EMPLOYEES");
```

The variable rs represents a `ResultSet` object that contains the values for the columns `EMP_NO` and `SALARY` from every row in the table `EMPLOYEES`. This result set is not scrollable, so only the method `next` can be used to move the cursor from the top down through the rows of the result set. The `ResultSet` object rs cannot be updated, and since no performance hints were given, the driver is free to do whatever it thinks will produce the best performance. The transaction isolation level was likewise not set, so rs will use the default transaction isolation level of the underlying database. (See "Transaction Isolation Levels" on page 22 for an explanation of transaction isolation levels.)

The next example uses the new JDBC 2.0 core API to create a scrollable result set that is sensitive to updates (by specifying `ResultSet.TYPE_SCROLL_SENSITIVE`) and that is updatable (by specifying `ResultSet.CONCUR_UPDATABLE`).

```
Connection con = DriverManager.getConnection(
                "jdbc:my_subprotocol:my_subname");

Statement stmt = con.createStatement(
                ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_UPDATABLE);
stmt.setFetchSize(25);

ResultSet rs2 = stmt.executeQuery(
                "SELECT EMP_NO, SALARY FROM EMPLOYEES");
```

The variable rs2 contains the same values as rs, from the previous example, but unlike rs, it is scrollable, updatable, and sensitive to changes in the underlying table's data. It also hints that the driver should fetch 25 rows from the database each time new rows are needed. Each time the `Statement` object stmt is executed, it will create a result set that is scrollable, is updatable, is sensitive to changes in its data, and has a fetch size of 25. The result set may change its fetch size, but it cannot change its type or concurrency.

As stated previously, there is a cost to making a result set scrollable or updatable, so it is good practice to create result sets with these features only when they are needed.

5.1.11 Using a Prepared Statement to Create Result Sets

Because `PreparedStatement` and `CallableStatement` objects inherit the methods defined in the `Statement` interface, they, too, can create different types of `ResultSet` objects.

The following code fragment creates a result set using a `PreparedStatement` object instead of a `Statement` object. The result set has the same attributes as in the previous example, except that a transaction isolation level is set for the connection.

```
Connection con = DriverManager.getConnection(
                        "jdbc:my_subprotocol:my_subname");
```

```
con.setTransactionIsolation(TRANSACTION_READ_COMMITTED);

PreparedStatement pstmt = con.prepareStatement(
                        "SELECT EMP_NO, SALARY FROM EMPLOYEES WHERE EMP_NO = ?",
                        ResultSet.TYPE_SCROLL_SENSITIVE,
                        ResultSet.CONCUR_UPDATABLE);
pstmt.setFetchSize(25);
pstmt.setString(1, "1000010");

ResultSet rs3 = pstmt.executeQuery();
```

The variable rs3 contains the values from the columns `EMP_NO` and `SALARY` for the row where the value for `EMP_NO` is `1000010`. The `ResultSet` object rs3 is like rs2 in that it is scrollable, is updatable, is sensitive to changes in its data, and hints that the driver should fetch 25 rows at a time from the database. It is different in that its connection specifies that dirty reads (reading values before they are committed) will be prevented. Because no transaction isolation level was set for rs2, it will by default have the isolation level of the underlying database.

5.1.12 Requesting Features That Are Not Supported

With the addition of new functionality in the JDBC 2.0 API, it is possible for an application to request features that a DBMS or driver do not support. If the driver does not support scrollable result sets, for example, it may return a forward-only result set. Also, some queries will return a result set that cannot be updated, so requesting an updatable result set would have no effect for those queries. A general rule is that a query should include the primary key as one of the columns it selects, and it should reference only one table.

New methods in the JDBC 2.0 API let an application discover which result set features a driver supports. If there is any doubt about whether a feature is supported, it is advisable to call these methods before requesting the feature. The following `DatabaseMetaData` methods indicate whether a driver supports a given result set type or a given result set concurrency:

- `DatabaseMetaData.supportsResultSetType` - returns a `boolean` indicating whether the driver supports the given result set type
- `DatabaseMetaData.supportsResultSetConcurrency` - returns a `boolean` indicating whether the driver supports the given concurrency type in combination with the given result set type

The following `ResultSet` methods return the result set type and result set concurrency for the particular result set on which the method is called:

- `ResultSet.getType` - returns the type of this result set
- `ResultSet.getConcurrency` - returns the concurrency mode of this result set

If an application specifies a scrollable result set and the driver does not support scrolling, the driver will issue a warning on the `Connection` object that produced the statement and return a result set that is forward-only. Even if the driver supports scrollable result sets, it is possible for an application to request a scrollable type that the driver does not support. In such a case, the driver will issue an `SQLWarning` on the `Connection` object that produced the statement and return a scrollable result set of a type that it does support, even if it differs from the exact type requested. For example, if an application requests a `TYPE_SCROLL_SENSITIVE` result set and the driver does not support that type, it could return a `TYPE_SCROLL_INSENSITIVE` result set if it supports that type. The driver would also alert the application that it did not return the exact type requested by issuing an `SQLWarning` on the `Connection` object that produced the statement requesting the unsupported result set type.

Similarly, if an application specifies an updatable result set, a driver that does not support updatable result sets will issue an `SQLWarning` on the `Connection` object that produced the statement and return a read-only result set. If the application requests both an unsupported result set type and an unsupported concurrency type, the driver should choose the result set type first.

In some situations, a driver may need to choose an alternate result set type or concurrency type at statement execution time. For example, a `SELECT` statement that contains a join over multiple tables might produce a result set that is not updatable. In such a situation, the driver will issue an `SQLWarning` on the `Statement`, `PreparedStatement`, or `CallableStatement` object that tried to create the result set instead of issuing it on the `Connection` object. The driver will then choose an appropriate result set type and/or concurrency type according to the guidelines in the preceding two paragraphs.

5.1.13 Using updateXXX Methods

A `ResultSet` object may be updated (have its rows modified, inserted, or deleted) programmatically if its concurrency type is `CONCUR_UPDATABLE`. The JDBC 2.0 API adds `updateXXX` methods and various other methods to the `ResultSet` interface so that rows can be programmatically updated in both the `ResultSet` object and the database.

The new `updateXXX` methods make it possible to update values in a result set without using SQL commands. There is an `updateXXX` method for each data type, and as with the `getXXX` and `setXXX` methods, the `XXX` is a data type in the Java programming language. As with the `setXXX` methods, the driver converts this data type to an SQL data type before sending it to the database. So for example, the method `updateBoolean` sends a JDBC `BIT` value to the database, and the method `updateCharacterStream` sends a JDBC `LONGVARCHAR` value to the database.

The `updateXXX` methods take two parameters, the first to indicate which column is to be updated, and the second to give the value to assign to the specified column. As is true with the `getXXX` methods, the column can be specified by giving either its name or its index. If an application retrieved a value from a result set by using the column name, it will generally use the column name when it wants to update that value. Similarly, if the `getXXX` method was given a column index to retrieve a value, the corresponding `updateXXX` method will generally use the column index to update that value.

Note that the column index used with `ResultSet` methods refers to the column number in the result set, not the column number in the database table, which might well be different. (The column numbers will be the same only in the case where all of a table's columns are selected.) In both result set tables and database tables, the index for the first column is `1`, the index for the second column is `2`, and so on.

In the following code fragment, the value in the third column of the `ResultSet` object rs is retrieved using the method `getInt`, and the method `updateInt` is used to update that column value with an `int` value of `88`:

```
int n = rs.getInt(3); // n contains the value in column 3 of rs
. . .
rs.updateInt(3, 88); // the value in column 3 of rs is set to 88
int n = rs.getInt(3); // n = 88
```

If the third column is named `SCORES`, the following lines of code will also update the third column of the `ResultSet` object rs by assigning it the `int` value `88`:

```
int n = rs.getInt("SCORES");
. . .
rs.updateInt("SCORES", 88);
```

The `updateXXX` methods update a value in the current row of the result set, but they do not update the value in the underlying database table. It is the method `updateRow` that updates the database. It is very important that the `updateRow` method be called while the cursor is still on the current row (the row to be updated). In fact, if an application moves the cursor before it calls `updateRow`, the driver must discard the update, and neither the result set nor the database will be updated.

An application may explicitly cancel the updates to a row by calling the method `cancelRowUpdates`. To take effect, it must be called after the method `updateXXX` is called and before the method `updateRow` is called. If `cancelRowUpdates` is called at any other time, it has no effect.

The following example demonstrates updating the second and third columns in the fourth row of the `ResultSet` object rs. Since updates affect the current row, the cursor is first moved to the row to be updated, which in this case is the fourth row. Next the method `updateString is` called to change the value in the second column of rs to `321 Kasten`. The method `updateFloat` changes the value in the third column of rs to `10101.0`. Finally, the method `updateRow` is called to update the row in the database that contains the two modified column values.

```
rs.absolute(4);
rs.updateString(2, "321 Kasten");
rs.updateFloat(3, 10101.0f);
rs.updateRow();
```

If the second column is named `ADDRESS` and the third column is named `AMOUNT`, the following code will have exactly the same effect as the previous example.

```
rs.absolute(4);
rs.updateString("ADDRESS", "321 Kasten");
rs.updateFloat("AMOUNT", 10101.0f);
rs.updateRow();
```

In addition to making updates programmatically, the JDBC 2.0 core API provides the ability to send batch updates. The batch update facility operates through a `Statement` object, which is explained in the section "Sending Batch Updates" on page 39.

5.1.14 Deleting a Row

The JDBC 2.0 API provides the method `deleteRow` so that a row in a `ResultSet` object can be deleted using only methods in the Java programming language. This method deletes the current row, so before calling `deleteRow`, an application must position the cursor on the row it wants to delete. Unlike the `updateXXX` methods, which affect only a row in the result set, this method affects both the current row in the result set and the underlying row in the database. The following two lines of code remove the first row of the `ResultSet` object rs and also delete the underlying row from the database (which may or may not be the first row of the database table).

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

5.1.15 Inserting Rows

New rows may be inserted into a result set table and into the underlying database table using new methods in the JDBC 2.0 core API. To make this possible, the API defines the concept of an *insert row*. This is a special row, associated with the result set but not part of it, that serves as a staging area for building the row that is to be inserted. To access the insert row, an application calls the `ResultSet` method `moveToInsertRow`, which positions the cursor on the insert row. Then it calls the appropriate `updateXXX` methods to add column values to the insert row. When all of the columns of the row to be inserted have been set, the application calls the method `insertRow`. This method adds the insert row to both the result set and the underlying database simultaneously. Finally, the application needs to position the cursor on a row back in the result set.

The following code fragment demonstrates these steps for inserting a row from an application written in the Java programming language.

```
rs.moveToInsertRow();
rs.updateObject(1, myArray);
rs.updateInt(2, 3857);
rs.updateString(3, "Mysteries");
rs.insertRow();
rs.first();
```

Several details deserve attention. First, it is possible to retrieve values from the insert row using the `ResultSet.getXXX` methods. Until a value has been assigned to the insert row with an `updateXXX` method, however, its contents are undefined. Therefore, if a `getXXX` method is called after the `moveToInsertRow` method has been called but before an `updateXXX` method has been called, the value it returns will be undefined.

Second, calling an `updateXXX` method on the insert row is different from calling it on a row in `the ResultSet` object. When the cursor is on a row in a result set, a call to an `updateXXX` method changes a value in the result set. When the cursor is on the insert row, a call to an `updateXXX` method updates a value in the insert row but does nothing to the result set. In both cases, though, the `updateXXX` method has no effect on the underlying database.

Third, calling the method `insertRow`, which adds the insert row to both the result set and database, may throw an `SQLException` if the number of columns in the insert row does not match the number of columns in the database table. For example, if a column is not given a value by calling an `updateXXX` method, an `SQLException` will be thrown unless that column allows null values. Also, if the result set is missing a column, that, too, will cause an `SQLException` to be thrown unless the column allows null values.

Fourth, a result set keeps track of where its cursor was positioned when the cursor moved to the insert row. As a result, a call to the method `ResultSet.moveToCurrentRow` will return the cursor to the row that was the current row immediately before the method `moveToInsertRow` was called. The other cursor movement methods also work from the insert row, including those that use positioning relative to the current row.

5.1.16 Positioned Updates

Before the JDBC 2.0 API made programmatic updates available in the Java programming language, the only way to change a row that had been fetched with a result set was to use what is called a *positioned update*. A positioned update is done with SQL commands and requires a named cursor to indicate the result set row in which updates are to be made.

The `Statement` interface provides the method `setCursorName`, which allows an application to specify a cursor name for the cursor associated with the next result set produced by a statement. This name can then be used in SQL positioned update or delete statements to identify the current row in the `ResultSet` object generated by the statement. In order to enable a positioned update or delete on a result set, the query that produces it must have the following form:

```
SELECT . . . FROM . . . WHERE . . . FOR UPDATE . . .
```

Including the words "`FOR UPDATE`" ensures that the cursor has the proper isolation level to support an update.

After the method `executeQuery` has been called on the statement, the cursor name for the resulting `ResultSet` object can be obtained by calling the `ResultSet` method `getCursorName`. If a DBMS allows positioned updates or positioned deletes, the name of the cursor can be supplied as a parameter to the SQL command for updates or deletes. A `Statement` object other than the one that created the `ResultSet` object must be used for the positioned update. The following code fragment, in which stmt and stmt2 are two different `Statement` objects, demonstrates the form for naming a cursor and then using it in an SQL update statement:

```
stmt.setCursorName("x");
ResultSet rs = stmt.executeQuery(
                "SELECT . . . FROM . . . WHERE . . . FOR UPDATE . . .")
String cursorName = rs.getCursorName;
int updateCount = stmt2.executeUpdate(
                "UPDATE . . . WHERE CURRENT OF " + cursorName);
```

Note that just because the method `getCursorName` has been invoked on a `ResultSet` object does not necessarily mean that it can be updated using the `ResultSet.updateXXX` methods available in the JDBC 2.0 core API. In order to update a `ResultSet` object using the `updateXXX` methods, the `executeQuery` statement that produces the result set must include the specification `CONCUR_UPDATABLE`. Positioned updates, however, are possible for a result set created without this specification if all the proper steps are taken: (1) a cursor is named, (2) the SQL query that produces the result set is of the form SELECT . . . FROM . . . WHERE . . . FOR UPDATE . . ., and (3) the SQL update statement is of the form `UPDATE . . . WHERE CURRENT OF <cursorName>`.

Not all DBMSs support positioned updates. To verify that a DBMS supports positioned updates, an application can call the `DatabaseMetaData` methods `supportsPositionedDelete` and `supportsPositionedUpdate` to discover whether a particular connection supports these operations. When they are supported, the DBMS/driver must ensure that rows selected are properly locked so that positioned updates do not result in update anomalies or other concurrency problems.

5.1.17 Queries That Produce Updatable Result Sets

Some queries will produce result sets that cannot be updated no matter what the result set type. For example, a query that does not select the primary key column might generate a result set that cannot be updated. Because of differences in database implementations, the JDBC 2.0 core API does not specify an exact set of SQL queries that must yield updatable result sets. Instead it defines a set of criteria that should generally produce updatable result sets for JDBC Compliant drivers that support updatability. If queries adhere to the following guidelines, a developer can generally expect that they will produce updatable result sets:

1. The query references only a single table in the database
2. The query does not contain a join operation or a `GROUP BY` clause
3. The query selects the primary key of the table it references

   If inserts are to be performed on the result set, an SQL query should satisfy conditions one through three plus the following three additional conditions:

4. The user has read/write database privileges on the table
5. The query selects all of the nonnullable columns in the underlying table
6. The query selects all columns that do not have a default value

The fourth and fifth conditions are necessary because a row to be inserted into a table must have a value for each column in the table unless the column accepts null values or default values. If the result set on which insertion operations are to be performed does not contain every column that requires a value, the insertion will fail.

Result sets created by means other than the execution of a query, such as those returned by several methods in the `DatabaseMetaData` interface, are not scrollable or updatable, nor are they required to be.

5.1.18 Using Streams for Very Large Row Values

Two new interfaces in the JDBC 2.0 core API, `Blob` and `Clob`, are the mapping of the SQL3 data types `BLOB` (Binary Large Object) and `CLOB` (Character LargeObject) in the Java programming language. With the availability of these data types, databases will undoubtedly start using them to store very large binary or character objects. If this is the case, the `ResultSet` methods `getBlob` and `getClob` should be used to retrieve them.

Using only the JDBC 1.0 API, a `ResultSet` object still makes it possible to retrieve arbitrarily large `LONGVARBINARY` or `LONGVARCHAR` data. The methods `getBytes` and `getString` return data as one large chunk (up to the limits imposed by the return value of `Statement.getMaxFieldSize`). It is possible to retrieve this large chunk of data in smaller, fixed-size chunks. This is done by having the `ResultSet` class return `java.io.Input` streams from which data can be read in chunks. Note that these streams must be accessed immediately because they will be closed automatically when the next `getXXX` method is called on the `ResultSet` object. (This behavior is not a limitation of the JDBC API but rather a constraint on large blob access imposed by the underlying implementations in some database -systems.)

The JDBC 1.0 API has three separate methods for getting streams, each with a different return value:

- `getBinaryStream` - returns a stream that simply provides the raw bytes from the database without any conversion
- `getAsciiStream` - returns a stream that provides one-byte ASCII characters. This method can be more efficient for a DBMS that stores characters in ASCII format.
- `getUnicodeStream` - returns a stream that provides two-byte Unicode characters. This method, though still available, has been deprecated in favor of the new method `getCharacterStream`. (See below.)

The following method for retrieving streams of both ASCII and Unicode characters is new in the JDBC 2.0 core API:

- `getCharacterStream` - returns a `java.io.Reader` object that provides Unicode characters. No matter how a DBMS stores characters, the driver will return them as a stream of Unicode characters.

Note that the stream returned by `getAsciiStream` returns a stream of bytes in which each byte is an ASCII character. This differs from `getCharacterStream`, which returns a stream of two-byte Unicode characters. The method `getCharacterStream` can be used for both ASCII and Unicode characters because the driver will convert ASCII characters to Unicode before it returns a `Reader` object. If you must use `getUnicodeStream` because your DBMS and driver do not support the JDBC 2.0 API, note also that JDBC Unicode streams return big-endian data; that is, they expect data with the high byte first and the low byte second. This conforms to the standard endian defined by the Java programming language, which is important if a program is to be portable. Refer to *The Java*tm *Virtual Machine Specification*, by Tim Lindholm and Frank Yellin, for more detailed information about big-endian order.

The following code fragment demonstrates how to use the `getAsciiStream` method.

```
java.sql.Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT x FROM Table2");
// Now retrieve the column 1 results in 4 K chunks:
byte [] buff = new byte[4096];
while (rs.next()) {
        java.io.InputStream fin = rs.getAsciiStream(1);
        for (;;) {
                int size = fin.read(buff);
                if (size == -1) { // at end of stream
                                        break;
                }


                // Send the newly-filled buffer to some ASCII output stream
                output.write(buff, 0, size);
        }
}
```

5.1.19 NULL Result Values

To determine if a given result value is JDBC `NULL`, one must first read the column and then use the method `ResultSet.wasNull`. This is true because a JDBC `NULL` retrieved by one of the `ResultSet.getXXX` methods may be converted to either `null`, `0`, or `false`, depending on the type of the value.

The following list shows which values are returned by the various `getXXX` methods when they have retrieved a JDBC `NULL`.

- `null`-for those `getXXX` methods that return objects in the Java programming language (`getString`, `getBigDecimal`, `getBytes`, `getDate`, `getTime`, `getTime-stamp`, `getAsciiStream`, `getCharacterStream`, `getUnicodeStream`, `getBinary-Stream`, `getObject`, `getArray`, `getBlob`, `getClob`, and `getRef`)
- `0` (zero)-for `getByte`, `getShort`, `getInt`, `getLong`, `getFloat`, and `getDouble`
- `false`-for `getBoolean`

For example, if the method `getInt` returns `0` from a column that allows `null` values, an application cannot know for sure whether the value in the database was `0` or `NULL` until it calls the method `wasNull`, as shown in the following code fragment, where rs is a `ResultSet` object.

```
int n = rs.getInt(3);
boolean b = rs.wasNull();
```

If b is `true`, the value stored in the third column of the current row of rs is JDBC `NULL`. The method `wasNull` checks only the last value retrieved, so to determine whether n was `NULL`, `wasNull` had to be called before another `getXXX` method was invoked.

5.1.20 Closing a ResultSet Object

Normally, nothing needs to be done to close a `ResultSet` object; it is automatically closed by the `Statement` object that generated it when that `Statement` object is closed, is re-executed, or is used to retrieve the next result from a sequence of multiple results. The method `close` is provided so that a `ResultSet` object can be closed explicitly, thereby immediately releasing the resources held by the `ResultSet` object. This could be necessary when several statements are being used and the automatic close does not occur soon enough to prevent database resource conflicts.

5.1.21 JDBC Compliance

Drivers that are JDBC Compliant should normally support scrollable result sets, but they are not required to do so. The intent is for JDBC drivers to implement scrollable result sets using the support provided by the underlying database systems. If the DBMS does not provide support for scrollability, then the driver may omit this feature.

Making scrollability optional is not meant to encourage omitting it. It is simply meant to minimize the complexity of implementing JDBC drivers for data sources that do not support scrollability. Indeed, the recommended alternative is for a driver to implement scrollability as a layer on top of the DBMS. One way to do this is to implement a result set as a rowset. The `RowSet` interface, which provides methods for doing this, is part of the JDBC Standard Extension API.

[Top] [Prev] [Next]
