2

Entering Information

The last chapter examined creating a database and adding tables, so now you're ready to start adding data. Most RDBMSs provide management tools that allow you to view tables and the records they hold, as well as allowing you to add, modify, and delete the data. These tools are very convenient when you have small amounts of data or when you're just testing the database. However, you don't generally enter data using the management tools. Much more common is some sort of program or Web page that acts as a pleasant front end into which the user enters and views data. This chapter's focus is on how to use SQL statements to insert, update, or delete data contained in a database.

This chapter covers the three SQL statements that deal with altering data. The first is the `INSERT INTO` statement, which inserts new data. The `UPDATE` statement updates existing data in the database. Finally, this chapter covers the `DELETE` statement, which (surprise, surprise) deletes records.

Inserting New Data

The `INSERT INTO` statement makes inserting new data into the database very easy. All you need to do is specify into which table you want to insert data, into which columns to insert data, and finally what data to insert. The basic syntax is as follows:

```
INSERT INTO table_name (column_names) VALUES (data_values)
```

This line of code adds a record to the Category table:

```
INSERT INTO Category (CategoryId, Category) VALUES (1, 'Thriller');
```

You can see that inserting data is simply a matter of listing each column name (separated by a comma) in the brackets after the table name. In the brackets after the `VALUES` statement, simply list each item of data to go into the matching column and separate each with a comma. Character and date data must be wrapped up inside single quotes. Delimiters are unnecessary around numerical data; simply insert them as is. If you load your RDBMS's SQL editor, connect to your Film Club database, and then enter and execute the statement, the following record is added to the Category table:

CategoryId	Category
1	Thriller

To check whether it worked, either use your RDBMS's management tools to view table data or use the following SQL statement:

```
SELECT * FROM Category
```

This statement displays all the records in the Category table. Chapter 3 covers the full details of the `SELECT` statement. For now, just use it to ensure that the `INSERT INTO` statement worked.

Once you make sure the first `INSERT` statement works, you can insert more values into your Category table:

```
INSERT INTO Category (CategoryId, Category) VALUES (2, 'Romance');
INSERT INTO Category (CategoryId, Category) VALUES (3, 'Horror');
INSERT INTO Category (CategoryId, Category) VALUES (4, 'War');
INSERT INTO Category (CategoryId, Category) VALUES (5, 'Sci-fi');
```

Now your Category table should contain the following values:

CategoryId	Category
1	Thriller
2	Romance
3	Horror
4	War
5	Sci-fi

Check whether yours does by using the following `SELECT` statement:

```
SELECT * FROM Category
```

You can specify the column names in any order you like, so you could also write the above SQL as follows:

```
INSERT INTO Category (Category, CategoryId) VALUES ('Historical', 6);
```

Regardless of category order, SQL performs exactly the same way, as long as you match column names in the first set of brackets with the correct data in the second set.

If you want to, you can leave off the column names altogether. You can write the code:

```
INSERT INTO Category (CategoryId, Category) VALUES (6, 'Historical')
```

like this:

```
INSERT INTO Category VALUES (6, 'Historical')
```

The RDBMS interprets it as meaning the following:

```
INSERT INTO Category (CategoryId, Category) VALUES (6, 'Historical')
```

The RDBMS decides the columns' order based on their order when you defined the table. Remember that you defined the Category table with CategoryId first and Category second, like this:

```
CREATE TABLE Category
(
    CategoryId integer,
    Category varchar(100)
)
```

You defined your MemberDetails table like this:

```
CREATE TABLE MemberDetails
(
    MemberId integer,
    FirstName nvarchar(50),
    LastName nvarchar(50),
    DateOfBirth date,
    Street varchar(100),
    City varchar(75),
    State varchar(75),
    ZipCode varchar(12),
    Email varchar(200),
    DateOfJoining date

)
```

Based on this information, the column order is MemberId, FirstName, LastName, DateOfBirth, Street, City, State, ZipCode, Email, DateOfJoining.

Writing the following `INSERT INTO` statement:

```
INSERT INTO MemberDetails
VALUES
 (
  1,
  'Katie',
  'Smith',
  '1977-01-09',
  'Main Road',
  'Townsville',
  'Stateside',
  '123456',
  'katie@mail.com',
  '2004-02-23'
 );
```

makes the MemberDetails table look like this:

Member Id	First Name	Last Name	Date OfBirth	Street	City	State	Zip Code	Email	Date OfJoining
1	Katie	Smith	1977-01-09	Main Road	Townsville	Stateside	123456	Katie@mail.com	2004-02-23

Notice that dates in the preceding table are specified in the form year-month-day, so February 23, 2004, is entered as 2004-02-23. The exact format acceptable to each database system varies not only with the database system but also with the way the database was first created and installed, as well as the date/time format specified by the computer itself. However, the format year-month-day works in most circumstances. However, the year-month-day format doesn't work on the default installation of Oracle. For Oracle, use the format day-month_name-year. So you would write January 9, 1977, as 9 January 1977 and February 23, 2004, as 23 Feb 2004.

The advantages of not naming the columns in your `INSERT` statement are that it saves typing and makes for shorter SQL. The disadvantage is that it's not as easy to see what data goes into which columns. For example, in the following `INSERT` statement, it's easy to match up column names to data being inserted without having to use the management tool to remind yourself of the column names.

```
INSERT INTO MemberDetails
(
  MemberId,
  FirstName,
  LastName,
  DateOfBirth,
  Street,
  City,
  State,
  ZipCode,
  Email,
  DateOfJoining
)
VALUES
 (
  2,
  'Bob',
  'Robson',
  '1987-01-09',
  'Little Street',
  'Big City',
  'Small State',
  '34565',
  'rob@mail.com',
  '2004-03-13'
 );
```

Of course, while it's all fresh in your mind it's not that hard to match column name and data. However, how about in six months or a year's time when you're asked to change the database structure or even to identify bugs? Your code is a little more readable if the column names are in there.

That's pretty much all there is to the `INSERT INTO` statement, so now you can finish it off by inserting data into the Film Club database.

Inserting Data into the Case Study Database

You've already inserted six records into the Category table and two records into the MemberDetails table. For the rest of the data to be inserted, you can turn to Appendix C to add the remaining data. Note that Appendix C and the book's downloadable files include for completeness the six records you already added to the Category table and the two MemberDetails records, so if you've already added them, don't add them a second time. Note that some RDBMSs allow you to insert all the data at once; others, such as MS Access, require you to insert one SQL statement at a time.

Now that you've got some data in the database, take a look at how you can alter it.

Updating Data

Not only do you need to add new records, but at some point you also need to change the records. To update records, you use the `UPDATE` statement. Specifying which records to change is the main difference between inserting new data and updating existing data. You specify which records to update with the `WHERE` clause, which allows you to specify that the only records to update are those where a certain condition is true. For example, say film club member Steve Gee has changed his address. His MemberId is `4`, so you could tell the database to update the address where the MemberId is `4`.

You don't, however, have to update all the details in a record. As in the example of Steve's change of address, only his address changes, not his name, date of birth, and so on. The `UPDATE` statement allows you to set which fields to update and with what data to update the field. The basic syntax of the `UPDATE` statement is as follows:

```
UPDATE table_name
SET column_name = value
WHERE condition
```

Start with a simple example, where film club member Steve Gee's new address is the following:

45 Upper Road
New Town
New State
99112

Keep in mind that Steve's MemberId is `4`. The SQL needed to make the changes is shown below:

```
UPDATE MemberDetails
SET
Street = '45 Upper Road',
City = 'New Town',
State = 'New State',
ZipCode = '99112'
WHERE MemberId = 4;
```

Now Steve's record looks like this:

Member Id	First Name	Last Name	Date OfBirth	Street	City	State	Zip Code	Email	Date Of Joining
4	Steve	Gee	Oct 5, 1967	45 Upper Road	New Town	New State	99112	steve@gee.com	Feb 22 2004

Looking at the code, first you specified which table to update with the following `UPDATE` statement:

```
UPDATE MemberDetails
```

In the `SET` clause, you specified each column name and the new value each should hold:

```
Street = '45 Upper Road',
City = 'New Town',
State = 'New State',
ZipCode = '99112'
```

As you can see in the example, with the `UPDATE` statement, you can use the `SET` clause to specify one or more columns to be updated; you just need to separate each column and name/value pair with a comma. Finally, you have the `WHERE` clause:

```
WHERE MemberId = 4
```

This clause says to update all records where the MemberId column has a value equal to `4`. Because the MemberId is the unique primary key value in the MemberDetails table, this means that only one record is changed. However, if there is more than one match, then every matching record's column values are updated. For example, say the people of Big Apple City get bored with their city's name and want to change it to Orange Town. To make this update in your database, you'd write the following SQL:

```
UPDATE MemberDetails
SET City = 'Orange Town'
WHERE City = 'Big Apple City';
```

Execute this SQL in your RDBMS's SQL tool, and the two records in the MemberDetails table that have a City column with the value `Big Apple City` are changed to `Orange Town`. However, it would be easy to forget that the Location table also contains City names, one of which is `Big Apple City`. You also need to update the Location table with this query:

```
UPDATE Location SET City = 'Orange Town' WHERE City = 'Big Apple City';
```

The ability to update more than one record at once is very powerful, but watch out! If you get your `WHERE` clause wrong, you could end up corrupting your database's data. And if you leave out the `WHERE` clause altogether, all records are updated in the specified table.

For the most part, the `UPDATE` statement is pretty simple. However, you need to examine the `WHERE` clause in more detail, because it's significantly more complicated.

The WHERE Clause

So far you've seen only a situation in which the database system performs an update if a column is equal to a certain value, but you can use other comparisons as well. The following table details a few of the fundamental comparison operators.

Comparison Operator	Name	Example	Example Matches All Records in Film Table Where Rating Is...
=	Equals	`WHERE Rating = 5`	5
<>	Not equal to	`WHERE Rating <> 1`	2, 3, 4, or 5
>	Greater than	`WHERE Rating > 2`	3, 4, or 5
<	Less than	`WHERE Rating < 4`	1, 2, or 3
>=	Greater than or equal to	`WHERE Rating >= 3`	3, 4, or 5
<=	Less than or equal to	`WHERE Rating <= 2`	1 or 2

The preceding table uses the Film table's Rating column in the Example and Example Matches columns. The comparison operators work with numerical fields, date fields, and text fields. In fact, they work with most data types, though there are exceptions. For example, some RDBMSs support the Boolean data type, which can be one of two values — `true` or `false`. It doesn't make sense to use operators other than the "equals" or "not equal to" operators with that data type.

With text-based data types, the operators >, <, >=, and <= make a comparison that equates to alphabetical order. For example, *a* is less than *b* in the alphabet, so to select all values in the column field where the first letter is lower than *f* in the alphabet, you use the following clause:

```
UPDATE SomeTable SET SomeColumn = 'SomeValue' WHERE Column < 'f';
```

The same principle applies to dates: January 1, 2005, is less than February 1, 2005.

Continuing the film club example, imagine that Sandra Jakes marries Mr. William Tell and that you need to update her surname to Tell.

```
UPDATE MemberDetails SET LastName = 'Tell' WHERE MemberId = 3;
```

This code fragment tells the database to update the MemberDetails table, changing the LastName field to the text value `Tell` for all records where the MemberId is equal to `3`. In this case, because the MemberId field stores a unique number for each record, only one record is changed. Alternatively, you could have used the following SQL to update the record:

```
UPDATE MemberDetails SET LastName = 'Tell' WHERE LastName = 'Jakes';
```

This would work fine with the records currently in the MemberDetails table, because at the moment only one person has the surname Jakes, though there's no guarantee that it will remain the only one. If you have two or more member with the surname Jakes, the SQL would update each record to match the `WHERE` statement.

So far your `WHERE` statement has just checked for one condition, such as `LastName = 3`. However, as you see in the next section, you can check more than one condition.

The Logical Operators AND and OR

The `AND` and `OR` logical operators allow you to test more than one condition in a `WHERE` statement. Their meanings in SQL and their meanings in English are almost identical. The `AND` operator means that both the condition on the left-hand side of the operator and the condition on the right-hand side must be true.

```
WHERE MyColumn = 132 AND MyOtherColumn = 'TEST'
```

The condition on the left-hand side of the operator is as follows:

```
MyColumn = 132
```

The following condition is on the right-hand side of the `AND` operator:

```
MyOtherColumn = 'TEST'
```

For the overall `WHERE` statement to be true, both conditions must also be true. For example, MyColumn must equal `132` and MyOtherColumn must equal `TEST`.

Returning to the example database and the MemberDetails table, the street New Lane in Big Apple City has changed its name to Newish Lane. To update all the appropriate records in the MemberDetails table, you need to check for records where the street name is `New Lane` and the city is `Big Apple` (after all, there could be another New Lane in another city). Use this SQL to update the records:

```
UPDATE MemberDetails SET Street = 'Newish Lane' WHERE Street = 'New Lane'
                                    AND City = 'Orange Town';
```

Notice that the `AND` clause is on its own separate line. That's not necessary; it's done here simply to improve readability. When the SQL is executed, any cities with the name Orange Town and the street name New Lane have their Street column updated and set to the value `Newish Lane`. There are two records in the MemberDetails table that match the criteria, and hence they are updated.

The other logical operator mentioned is the `OR` operator, which means that the condition is true and the record updates where one or both of the expressions are true. For example,

```
WHERE MyColumn = '10' OR MyOtherColumn = 'TEST'
```

The `WHERE` clause is true and the record updates if MyColumn equals `10`, MyOtherColumn equals `TEST`, or both MyColumn equals `10` and MyOtherColumn equals `TEST`.

Now that you're familiarized with the `AND` and `OR` logical operators, you can use them to update records in your database.

Try It Out Using Logical Operators to Update Database Records

A new president has decided that Small State and Stateside states should merge into one new state called Mega State. You need to update the database to reflect these changes. Remember that you store details of states in the Location and the MemberDetails tables, so you need to update both tables.

1. Enter this SQL to update the two records in the MemberDetails table:

```
UPDATE MemberDetails
SET State = 'Mega State'
WHERE
State = 'Small State'
OR
State = 'Stateside';
```

   This statement updates the two records in the MemberDetails table. The SQL to do the same for the Location table is identical except that the name of the table being updated has to be changed.

2. Enter the following SQL to update the records in the Location table:

```
UPDATE Location
SET State = 'Mega State'
WHERE
State = 'Small State'
OR
State = 'Stateside';
```

How It Works

The only difference between these two statements is the name of the table affected by the `UPDATE` statements; otherwise, they are identical. The first bit of SQL tells the database to update the MemberDetails table where the State column holds a value of either `Small State` or `Stateside`. The second SQL fragment tells the database to update the Location table where the State column holds a value of either `Small State` or `Stateside`. The State field in both tables should now be `Mega State`.

So far you've used only one `AND` or `OR` operator in each `WHERE` clause, but you can include as many, within reason, as you like. You can also mix `AND` and `OR` operators in the same `WHERE` clause.

The `WHERE` clause is vital to selecting data from the database, something covered in great detail in Chapter 3. Everything that applies to the `WHERE` clause in Chapter 3 also applies to the `WHERE` clause of an `UPDATE` statement.

Deleting Data

So far in this chapter, you've learned how to add new data and update existing data, so all that remains to learn is how to delete data. The good news is that deleting data is very easy. You simply specify which table to delete the records from, and if required, you add a `WHERE` clause to specify which records to delete.

If you want to delete all the records from a table, you simply leave out the `WHERE` clause, as shown in the following statement:

```
DELETE FROM MemberDetails;
```

The preceding SQL fragment deletes all the records in the MemberDetails table. Don't execute it or else you'll have to enter the records again. If you want to delete some of the records, you use the `WHERE` clause to specify which ones:

```
DELETE FROM MemberDetails WHERE MemberId = 3;
```

Enter and execute this SQL. This SQL deletes all records from the MemberDetails table where the MemberId column has a value of `3`. Because it holds a unique value, only the details of Sandra Tell are deleted — she is the only person whose MemberId is `3`. Now that Sandra is gone from the membership, you also need to delete her details from the FavCategory and Attendance tables. Use the following statements to do so:

```
DELETE FROM Attendance WHERE MemberId = 3;
DELETE FROM FavCategory WHERE MemberId = 3;
```

Everything that applies to the `WHERE` clause when used with the `UPDATE` statement also applies to the `DELETE` statement.

Summary

For now, that completes your introduction to adding, updating, and deleting data. This chapter covered the following:

- How to add new records to a database using the `INSERT INTO` statement. To add a new record, you must specify which table the record should go into, which fields you assign values to, and finally the values to be assigned.
- Next you looked at how to update data already in the database using the `UPDATE` statement, which specifies which table's records receive updates, as well as the fields and new values assigned to each record. Finally, you learned that the `WHERE` clause specifies which records in the table receive updates. If you don't use a `WHERE` clause, all records in the table receive updates.
- The final statement covered in this chapter was the `DELETE` statement, which allows you to delete records from a table. You can either delete all records or use a `WHERE` clause to specify which records you want to delete.

This chapter only touched on the basics; Chapter 3 shows you how to take data from one table and use it to insert data from one table into a second table. Chapter 3 also examines the `WHERE` clause in greater depth.

Exercises

1. Three new members have joined the film club. Add the following details to the database:

   Member ID: 7
   First Name: John
   Last Name: Jackson
   Date of Birth: May 27, 1974
   Street: Long Lane
   City: Orange Town
   State: New State
   Zip Code: 88992
   Email: jjackson@mailme.net
   Date of Joining: November 21, 2005

   Member ID: 8
   First Name: Jack
   Last Name: Johnson
   Date of Birth: June 9, 1945
   Street: Main Street
   City: Big City
   State: Mega State
   Zip Code: 34566
   Email: jjohnson@me.com
   Date of Joining: June 2, 2005

   Member ID: 9
   First Name: Seymour
   Last Name: Botts
   Date of Birth: October 21, 1956
   Street: Long Lane
   City: Windy Village
   State: Golden State
   Zip Code: 65422
   Email: Seymour@botts.org
   Date of Joining: July 17, 2005

   You need to ensure that the date format matches the format expected by your database system. Remember that Oracle accepts day-month-year format (23 January 2004), whereas the other four databases expect the format year-month-day, such as 2004-01-23.

2. Bob Robson, MemberId 2, took a job in Hollywood and left the club. Delete his details from the database. Remember to delete not only his membership details but also his film category preferences.

3. The government has decided to reorganize the boundaries of Orange Town. All residents living on Long Lane in Orange Town now live in Big City. Update the database to reflect this change.