# 3

# Extracting Information

So far you've learned how to set up a database and how to insert data into it, so now you can learn how to extract data from your database. Arguably, SQL's most powerful feature is its ability to extract data, and extracting data can be as simple or complex as you require. You can simply extract data in the same form in which it was entered into the database, or you could query the database and obtain answers to questions that are not obvious from the basic data. In your example database, you can use SQL to find out which meeting locations are most popular, or you could find out which meeting locations are most popular for which film category. It might turn out that Windy Village has a particularly large number of sci-fi fans. If the film club decides to show a film at that location, you would be aware that a sci-fi film is likely to be popular. The ability to ask the database questions and get answers via SQL queries makes SQL so popular and useful.

The key to getting data out is the `SELECT` statement, which in its basic form is very simple and easy to use. However, as you go through this chapter and then the advanced chapters, you see lots of extra options that make the `SELECT` statement very powerful. Before getting too complicated, however, you need to familiarize yourself with the `SELECT` statement.

## The SELECT Statement

At its simplest, the `SELECT` requires you to tell it which columns and from what table you want to obtain data. The basic syntax is as follows:

```
SELECT column1, column2,.....columnx FROM table_name
```

Using the basic syntax, the SQL required to select the MemberId and FirstName columns from all records in the MemberDetails table is

```
SELECT MemberId, FirstName FROM MemberDetails;
```

The order in which you list the columns in the `SELECT` statement determines the order in which they are returned in the results. The preceding query returns this order:

| MemberId | FirstName |
|---|---|
| 1 | Katie |
| 3 | Sandra |
| 4 | Steve |
| 5 | John |
| 6 | Jenny |
| 7 | John |
| 8 | Jack |
| 9 | Seymour |

The order of the results in the example database table usually reflects on the order in which the records were first entered into the database, but the order is not guaranteed, so don't worry if your order looks different from what appears in the preceding table. Chapter 4 shows you how to create a columns index on columns to determine the order of results. Later in this chapter, you learn how to use the `ORDER BY` clause to specify the order in which records are returned.

Specifying which columns you want returned is fine, but sometimes you may want all of the columns, which is a lot of typing if your table has many fields. The good news is that SQL provides a shorthand way of selecting all the columns without having to type out all their names. Instead of typing the column names, just type an asterisk:

```
SELECT * FROM Location;
```

The preceding code fragment is the same as writing the following:

```
SELECT LocationId, Street, City, State FROM Location;
```

Both lines of code return the following results:

| LocationId | Street | City | State |
|---|---|---|---|
| 1 | Main Street | Orange Town | New State |
| 2 | Winding Road | Windy Village | Golden State |
| 3 | Tiny Terrace | Big City | Mega State |

However, use shorthand only when you need all the columns, otherwise you make the database system provide information you don't need, which wastes CPU power and memory. Memory and CPU power may not matter on a small database, but they make a huge difference on a large database being accessed by many people.

## Returning Only Distinct Rows

If you want to know all the unique values in a record, how would you go about retrieving them? The answer is by using the `DISTINCT` keyword. The DISTINCT keyword is added to the `SELECT` statement's column listing, directly after the `SELECT` keyword. For example, if someone asks you which cities members come from, you could try a query similar to the following:

```
SELECT City FROM MemberDetails;
```

Executing the query gives you the following results:

| City |
| --- |
| Townsville |
| Orange Town |
| New Town |
| Orange Town |
| Orange Town |
| Big City |
| Windy Village |

As you can see, Orange Town is listed three times because there are three members from that city. But if you simply want a list of the unique places that members live in, you could add the `DISTINCT` keyword:

```
SELECT DISTINCT City FROM MemberDetails;
```

Executing the modified query gives these results:

| City |
| --- |
| Big City |
| New Town |
| Orange Town |
| Townsville |
| Windy Village |

This time, Orange Town is mentioned just once. The `DISTINCT` keyword works on all columns in combination; all the columns listed in the `SELECT` statement must be unique. If you change the previous query to include MemberId, which is unique for every single row, and rerun the query, you end up with all the rows:

```
SELECT DISTINCT City, MemberId FROM MemberDetails;
```

The results are as follows:

| City | MemberId |
|---|---|
| Big City | 8 |
| New Town | 4 |
| Orange Town | 5 |
| Orange Town | 6 |
| Orange Town | 7 |
| Townsville | 1 |
| Windy Village | 9 |

Orange Town appears three times because MemberId is unique on each row. In fact, using the `DISTINCT` keyword where one of the columns is always unique is pointless.

## Using Aliases

Just as James Bond is also known by the alias 007, you can give column names an alias in the results. An *alias* is simply a secondary or symbolic name for a collection of data. If, for example, instead of LastName you want your results to return an alias called `Surname`, you would write the following query:

```
SELECT LastName AS Surname FROM MemberDetails;
```

Specifying `Surname` with the `AS` keyword tells the database system that you want the results to be known as the alias `Surname`. Using an alias doesn't change the results returned in any way, nor does it rename the LastName in the MemberDetails tables. It affects only the name of the column in the results set. Using aliases may not seem that useful right now, but later on in the book you use aliases as a shorthand way of referring to table names, among other uses.

So far all the data from tables has been returned, but what if you just want specific data — for example, details of members older than 60 years old? In such a case, you need to use a `WHERE` clause, which is the topic of the next section.

## Filtering Results with the WHERE Clause

Although you may occasionally have to select all the records in a table, it's much more common to filter results so that you get only the information you want. For example, you could filter query results to find out all the names of members living in New State. In Chapter 2 you saw how you can use the `WHERE` clause to update or delete specific records. You can also use the `WHERE` clause with `SELECT` statements to filter results so that you get back only the data you want.

The good news is that everything you learned in Chapter 2 about the `WHERE` clause can be applied to `WHERE` clauses used with `SELECT` statements. This chapter delves even deeper into `WHERE` clauses and looks at some of the more sophisticated stuff not covered in Chapter 2.

To recap briefly, the `WHERE` clause allows you to set one or more conditions that must be satisfied by each record before it can form part of the results. So if you were asked for a list of members who live in Big City, you would need to specify that the column City must be equal to `Big City`. The SQL for such a request appears below:

```
SELECT FirstName + ' ' + LastName AS [Full Name]
FROM MemberDetails
WHERE City = 'Big City';
```

The query provides the following results:

| Full Name |
| --- |
| John Jackson |
| Jack Johnson |

You can also use the operators you saw in Chapter 2 to find out the names of all the films released before 1977:

```
SELECT FilmName
FROM Films
WHERE YearReleased < 1977
```

This query gives these results:

| FilmName |
| --- |
| On Golden Puddle |
| Planet of the Japes |
| The Maltese Poodle |
| Soylent Yellow |

However, if you want to find out which films were released in or before 1977, then change the "less than" operator (<) to a "less than or equal to" (<=) operator:

```
SELECT FilmName
FROM Films
WHERE YearReleased <= 1977
```

The results to this query also include *The Lion, the Witch, and the Chest of Drawers*, a film released in 1977:

| FilmName |
|---|
| On Golden Puddle |
| The Lion, the Witch, and the Chest of Drawers |
| Planet of the Japes |
| The Maltese Poodle |
| Soylent Yellow |

*Before moving on, you should note that while MS Access is happy inserting dates delimited (enclosed by single quotes), when it comes to* `SELECT` *statements and* `WHERE` *clauses, Access requires any date literals to be enclosed by the hash (#) symbol. For example, consider the following code:*

```
WHERE DateOfBirth < #2005-12-23#
```

SQL queries are all about finding answers to questions. The following Try It Out provides a few questions and shows you how SQL can provide answers.

## Try It Out — Querying Your Database

Five new members have joined the club, so their details need to be added to the database. The following steps detail how you would add the new members to the Film Club database:

1. Enter the SQL code into your database or download the code from `www.wrox.com` and then execute it. Included are the new members' information and `INSERT` statements to record their favorite category of films. Note that if you're using Oracle, the date formats must be changed from their current format of year-month-day to the day-month-year format.

```
INSERT INTO
MemberDetails (MemberId,
FirstName,LastName,DateOfBirth,Street,City,State,ZipCode,Email,DateOfJoining)
VALUES
(
  10, 'Susie','Simons','1937-1-20','Main Road','Townsville',
  'Mega State','123456','susie@mailme.com','2005-08-20'
);

INSERT INTO
FavCategory (CategoryId, MemberId)
VALUES ( 1, 10 );

INSERT INTO
FavCategory (CategoryId, MemberId)
VALUES ( 3, 10 );

INSERT INTO
FavCategory (CategoryId, MemberId)
VALUES ( 6, 10 );

INSERT INTO
```

```
 MemberDetails (MemberId,
FirstName,LastName,DateOfBirth,Street,City,State,ZipCode,Email,DateOfJoining)
VALUES
(
  11, 'Jamie','Hills','1992-07-17','Newish Lane','Orange Town',
  'New State','88776','jamie@the_hills.com','2005-08-22'
);

INSERT INTO
FavCategory (CategoryId, MemberId)
VALUES ( 4, 11 );

INSERT INTO
FavCategory (CategoryId, MemberId)
VALUES ( 3, 11 );

INSERT INTO
FavCategory (CategoryId, MemberId)
VALUES ( 5, 11 );

INSERT INTO
MemberDetails (MemberId,
FirstName,LastName,DateOfBirth,Street,City,State,ZipCode,Email,DateOfJoining)
VALUES
(
  12, 'Stuart','Dales','1956-08-07','Long Lane','Windy Village',
  'Golden State','65422','sdales@mymail.org','2005-08-27'
);

INSERT INTO
FavCategory (CategoryId, MemberId)
VALUES ( 1, 12 );

INSERT INTO
FavCategory (CategoryId, MemberId)
VALUES ( 4, 12 );

INSERT INTO
FavCategory (CategoryId, MemberId)
VALUES ( 6, 12 );

INSERT INTO
MemberDetails (MemberId,
FirstName,LastName,DateOfBirth,Street,City,State,ZipCode,Email,DateOfJoining)
VALUES
(
  13, 'William','Doors','1994-05-28','Winding Road','Big City',
  'Mega State','34512','knockon@thedoors.com','2005-08-29'
);

INSERT INTO
FavCategory (CategoryId, MemberId)
VALUES ( 3, 13 );

INSERT INTO
```

```
FavCategory (CategoryId, MemberId)
VALUES ( 5, 13 );

INSERT INTO
MemberDetails (MemberId,
FirstName,LastName,DateOfBirth,Street,City,State,ZipCode,Email,DateOfJoining)
VALUES
(
  14, 'Doris','Night','1997-05-28','White Cliff Street','Dover',
  'Golden State','68122','dnight@whitecliffs.net','2005-09-02'
);

INSERT INTO
FavCategory (CategoryId, MemberId)
VALUES ( 2, 14 );

INSERT INTO
FavCategory (CategoryId, MemberId)
VALUES ( 6, 14 );
```

2. Query the database to find the names and addresses of members who joined in January 2005 using this SQL:

```
SELECT FirstName, LastName, Street, City, State, ZipCode
FROM MemberDetails
WHERE DateOfJoining >= '2005-01-01' AND DateOfJoining <= '2005-01-31';
```

   Remember to change the date format for Oracle. Also, for MS Access, you need to enclose date literals inside the hash sign (#) rather than single quotes. Use the following statement for Access:

```
SELECT FirstName, LastName, Street, City, State, ZipCode
FROM MemberDetails
WHERE DateOfJoining >= #2005-01-01# AND DateOfJoining <= #2005-01-31#;
```

3. Query the database to find the names of members over 16 years of age who live in New State using this SQL:

```
SELECT FirstName, LastName
FROM MemberDetails
WHERE DateOfBirth <= '1989-05-01' AND State = 'New State';
```

   Again, remember to change the date format for Oracle and the single quotes around the date value to the hash (#) symbol.

## How It Works

In the first step, you inserted new data into the database using a series of `INSERT INTO` statements. You added five new members, plus details of their favorite film categories. Next came the SQL `SELECT` statements required to answer each of the three questions.

Begin your `SELECT` statement by choosing which columns and from what table it obtains its data. The MemberDetails table contains information on members' names, addresses, and joining dates. Within the

MemberDetails table, the columns FirstName, LastName, Street, City, State, and ZipCode contain all the information you need. So the `SELECT` statement begins with the following:

```
SELECT FirstName, LastName, Street, City, State, ZipCode
FROM MemberDetails
```

As it stands, that statement returns all the records, but now add a `WHERE` clause to select only those records where the joining date falls between January 1, 2005, and January 31, 2005. You need to check the DateOfJoining column to see if the value is greater than January 1 but less than or equal to January 31:

```
WHERE DateOfJoining >= '2005-01-01' And DateOfJoining <= '2005-Jan-31'
```

Adding the `WHERE` clause to the `SELECT` statement generates the following results:

| FirstName | LastName | Street | City | State | Zip |
|---|---|---|---|---|---|
| John | Jones | Newish Lane | Orange Town | New State | 88776 |
| Jenny | Jones | Newish Lane | Orange Town | New State | 88776 |

The next query asks for the names of all members older than 16 years of age who live in New State. This time the data you want is first name and last name, again from the MemberDetails table:

```
SELECT FirstName, LastName
FROM MemberDetails
```

Next comes the `WHERE` clause, which must specify that the member's age is greater than 16 and that he or she lives in New State. Assuming that today's date is May 31, 2005, anyone born after that date is less than 16 years old. So far, the `WHERE` clause looks like this:

```
WHERE DateOfBirth <= '1989-05- 31'
```

However, for the record to be included in the results, its State column must equal `New State`. So the final query looks like this:

```
SELECT FirstName, LastName
FROM MemberDetails
WHERE DateOfBirth <= '1989-05- 31' AND State = 'New State';
```

Both DateOfBirth and State conditions must be true; hence the use of the `AND` operator. The results of the query are as follows:

| FirstName | LastName |
|---|---|
| John | Jackson |
| Steve | Gee |
| John | Jones |
| Jenny | Jones |

# Logical Operators and Operator Precedence

With the AND and OR operators out of the way, this section introduces a few more useful operators. This section also examines *operator precedence*, or the rules that determine the order in which operators are evaluated when there is more than one operator in a condition.

The effects of incorrect operator precedence are numerous, so you need to be familiar with operator precedence before you can manipulate SQL to its full advantage.

## Introducing Operator Precedence

The American Declaration of Independence states that all men are created equal, but it fails to mention anything about operators. If it did, it would have to say that all operators are definitely not created equal. A hierarchy of operators determines which operator is evaluated first when a condition has multiple operators. If all the operators have equal precedence, then the conditions are interpreted from left to right. If the operators have different precedence, then the highest ones are evaluated first, then the next highest, and so on. The following table details all the logical operators. Their order in the table mirrors their order of precedence from highest to lowest. Operators contained in the same row have the same order of precedence; for example, OR has the same precedence as ALL.

| Operator |
| --- |
| Brackets ( ) |
| NOT |
| AND |
| ALL, ANY, BETWEEN, IN, LIKE, OR, SOME |

So far, you've experienced only the AND and OR operators. The next section details NOT, BETWEEN, LIKE, and IN operators, and the remaining operators are covered in Chapter 7.

Remembering that the AND operator has precedence over the OR operator, can you guess how the following SQL would be evaluated?

```
SELECT State, DateOfJoining
FROM MemberDetails
WHERE State = 'New State' OR State = 'Golden State'
AND DateOfJoining >= '2005-08-01';
```

If the AND and OR operators were of equal precedence, the preceding code would be evaluated as follows: State is equal to New State OR Golden State, AND the DateOfJoining must be greater than or equal to August 1, 2005.

If such an interpretation were true, then it would give the following results:

| State | DateOfJoining |
|---|---|
| New State | 2005-11-21 |
| New State | 2005-08-22 |
| Golden State | 2005-08-27 |
| Golden State | 2005-09-02 |

However, you won't see the preceding results, because the `AND` operator has a higher precedence than the `OR` operator, which is found in the `WHERE` clause of the SQL statement:

```
WHERE State = 'New State' OR State = 'Golden State'
AND DateOfJoining >= '2005-08-01';
```

The `WHERE` clause is actually evaluated like this: Is the State column equal to Golden State `AND` is the DateOfJoining on or after August 1, 2005, `OR` is the State column equal to New State?

This interpretation gives quite different results:

| State | DateOfJoining |
|---|---|
| New State | 2004-02-22 |
| New State | 2005-01-02 |
| New State | 2005-01-02 |
| New State | 2005-11-21 |
| New State | 2005-08-22 |
| Golden State | 2005-08-27 |
| Golden State | 2005-09-02 |

If you want the database to provide a list of members in New State or Mega State who joined on or after August 1, 2005, then clearly the query results are wrong. To solve this problem, use brackets to increase the precedence of the operators inside them, just as you would in math to differentiate between (1 + 1) _ 2 and 1 + 1 _ 2. Brackets are right at the top of the precedence hierarchy, so they are always evaluated first. If you add brackets to the SQL statement, the condition inside the brackets is evaluated first:

```
SELECT State, DateOfJoining
FROM MemberDetails
WHERE (State = 'New State' OR State = 'Golden State')
AND DateOfJoining >= '2005-08-01';
```

Because brackets are the highest precedence, the SQL statement now reads as follows: State is equal to New State `OR` Golden State, `AND` the DateOfJoining must be greater than or equal to August 1, 2005.

The statement returns these results, which are exactly what you want:

| State | DateOfJoining |
|---|---|
| New State | 2005-11-21 |
| New State | 2005-08-22 |
| Golden State | 2005-08-27 |
| Golden State | 2005-09-02 |

Using brackets is the key to ensuring operator precedence. Additionally, brackets can make the SQL easier to read because they make it clear which conditions are evaluated first, which is quite handy if the conditions are quite complex. Otherwise you have to remember the order of operator precedence.

To illustrate operator precedence, try out a more complex `WHERE` statement where lots of brackets are necessary. What's required this time is a list of all the names, cities, and dates of birth of members who live in either Townsville or Big City and are either older than 60 or younger than 16. Again, assume that today's date is May 31, 2005. Members under 16 years of age must have been born after May 31, 1989, and members over 60 years of age must have been born on or before May 31, 1945.

## Try It Out — Increasing Operator Precedence Using Brackets

1. The film club chairperson wants to know which members live in either Townsville or Big City and were born either before May 31, 1945, or after May 31, 1989. The SQL to answer this question is as follows:

```
SELECT FirstName, LastName, City, DateOfBirth
FROM MemberDetails
WHERE
( City = 'Townsville' OR City = 'Big City' )
AND
(DateOfBirth > '1989-05-31' OR DateOfBirth <= '1945-05-31')
```

   If you're using MS Access, then the date literals in the last line of code must be enclosed in hash characters (#) rather than single quotes, as shown below. All other components of the statement remain the same:

```
(DateOfBirth > #1989-05-31# OR DateOfBirth <= #1945-05-31#)
```

   If you're using Oracle, you may find that you need to change the date format in the last line of code to day-month-year, as shown below. All other components of the statement remain the same:

```
(DateOfBirth > '31 May 1989' OR DateOfBirth <= '31 May 1945')
```

## How It Works

The `SELECT` part of the query is fairly straightforward:

```
SELECT FirstName, LastName, City, DateOfBirth
FROM MemberDetails
```

It simply tells the database system to retrieve the values from the FirstName, LastName, City, and DateOfBirth columns of the MemberDetails table.

The `WHERE` clause is slightly trickier. This time, you're looking for records where the member lives in Townsville or Big City and is younger than 16 or older than 60. There are two main conditions: the city the person lives in and their date of birth. You can start with the city condition:

```
( City = 'Townsville' OR City = 'Big City' )
```

The other condition checks their ages:

```
( 'DateOfBirth' > '1989-05-31' OR DateOfBirth <= '1945-05-31' )
```

Both of the main conditions must be true for the record to be part of the results, so an `AND` operator is required:

```
( City = 'Townsville' OR City = 'Big City' )
AND
( 'DateOfBirth' > '1989-05-31' OR DateOfBirth <= '1945-05-31' )
```

You must enclose the conditions in brackets because the `AND` operator is of higher precedence than the `OR` operator. If you don't include brackets, SQL evaluates the `AND` operator first.

Putting the whole query together results in the final SQL shown below:

```
SELECT FirstName, LastName, City, DateOfBirth
FROM MemberDetails
WHERE
( City = 'Townsville' OR City = 'Big City' )
AND
(DateOfBirth > '1989-05-31' OR DateOfBirth <= '1945-05-31')
```

Executing the query provides the following results:

| FirstName | LastName | City | DateOfBirth |
|---|---|---|---|
| Susie | Simons | Townsville | Jan 20 1937 |
| William | Doors | Big City | May 28 1994 |

The next section examines the `NOT`, `BETWEEN`, `LIKE`, and `IN` logical operators in turn. The remaining operators are covered later in Chapter 7.

## Using Logical Operators

Now that you know how to use the `AND` and `OR` logical operators, you can learn how to use a few new ones, starting with the `NOT` operator.

## NOT Operator

Examples thus far have been filtered based on true conditions. The NOT operator, however, selects a record if the condition is false. The following SQL selects records where the State field is not equal to Golden State.

```
SELECT FirstName
FROM MemberDetails
WHERE NOT State = 'Golden State';
```

The preceding example is the same as this example:

```
SELECT FirstName
FROM MemberDetails
WHERE State <> 'Golden State';
```

The only difference is the use of the "not equal to" operator (<>) instead of the NOT operator. In this situation, the "not equal to" operator (<>) reads easier than the NOT operator.

You can also use the NOT operator with brackets:

```
SELECT City
FROM MemberDetails
WHERE NOT (City = 'Townsville' OR City = 'Orange Town' OR City = 'New Town');
```

The preceding SQL selects all records where the conditions inside the brackets are not true. In this case, the condition inside the brackets is that the City is equal to Townsville or it is equal to Orange Town or it is equal to New Town. Using the NOT operator is the same as saying "is not true," which is the same as saying "is false." So, you could rephrase the explanation to say that the query is looking for values that are false, that do not equal Townsville, Orange Town, or New Town. The results returned are shown here:

| City |
| --- |
| Big City |
| Windy Village |
| Windy Village |
| Big City |
| Big City |

As you see shortly, you can use the NOT operator in combination with other operators such as BETWEEN, ANY, SOME, AND, OR, or LIKE.

## BETWEEN Operator

The BETWEEN operator allows you to specify a range, where the range is between one value and another. Until now, when you needed to check for a value within a certain range, you used the "greater than or equal to" operator (>=) or the "less than or equal to" (<=) operator.

The BETWEEN operator functions exactly the same way, except it's shorter — it saves on typing and also makes the SQL more readable. The following SQL uses the BETWEEN operator to select films with a rating between 3 and 5:

```
SELECT FilmName, Rating
FROM Films
WHERE Rating BETWEEN 3 AND 5
```

If you use the BETWEEN operator, you see that it provides exactly the same results as the "greater than or equal to" (>=) and "less than or equal to" (<=) operators do. It is extremely important to remember that the BETWEEN operator is inclusive, meaning that in the preceding code, 3 and 5 are also included in the range.

You can use BETWEEN with data types other than numbers, such as text and dates. You can also use the BETWEEN operator in conjunction with the NOT operator, in which case SQL selects a value that is not in the range specified, as you see in the following Try It Out.

## Try It Out — Using the NOT and BETWEEN Operators

1. The film club has added more films to its database. The SQL to add the films is listed below and must be executed against the database.

```
INSERT INTO Films
  (FilmId, FilmName, YearReleased, PlotSummary, AvailableOnDVD, Rating, CategoryId)
VALUES
  (9, 'One Flew Over the Crow''s Nest',1975
  'Life and times of a scary crow.', 'Y',2,3);

INSERT INTO Films
  (FilmId, FilmName, YearReleased, PlotSummary, AvailableOnDVD, Rating, CategoryId)
VALUES
  (10, 'Raging Bullocks',1980,
  'A pair of bulls get cross with each other.', 'N',4,1);

INSERT INTO Films
  (FilmId, FilmName, YearReleased, PlotSummary, AvailableOnDVD, Rating, CategoryId)
VALUES
  (11, 'The Life Of Bob',1984,
  'A 7 hour drama about Bob''s life. What fun!', 'Y',1,1

INSERT INTO Films
  (FilmId, FilmName, YearReleased, PlotSummary, AvailableOnDVD, Rating, CategoryId)
VALUES
  (12, 'Gone With the Window Cleaner',1988,
  'Historical documentary on window cleaners. Thrilling', 'Y',3,6);

INSERT INTO Films
  (FilmId, FilmName, YearReleased, PlotSummary, AvailableOnDVD, Rating, CategoryId)
VALUES
  (12, 'The Good, the Bad, and the Facially Challenged',1989,
  'Joe seeks plastic surgery in this spaghetti Western.', 'Y',5,6);
```

Remember to commit the data if you're using Oracle.

2. Using the following SQL, query the database to produce a list of all the films released in the 1980s that have a rating between 2 and 4 and are available on DVD. The answer should detail the film name, date of release, rating, and availability on DVD.

```
SELECT FilmName, YearReleased, Rating, AvailableOnDVD
FROM Films
WHERE ( YearReleased BETWEEN 1980 AND 1989 )
AND
( Rating BETWEEN 2 AND 4 )
AND
( AvailableOnDVD = 'Y' );
```

3. Query the database to produce a list of films of any decade except the 1960s whose names are between *P* and *T*. The answer should detail the film names. Use the following SQL to execute the query:

```
SELECT FilmName
FROM Films
WHERE ( YearReleased NOT BETWEEN 1960 AND 1969 )
AND
( FilmName BETWEEN 'P' AND 'T' );
```

## How It Works

You begin by inserting five new films into the Films table using `INSERT INTO` statements. Remember to commit the data if you're using Oracle using the `COMMIT` command, unless you've already set the auto-commit to on.

The `SELECT` clause of the first query is fairly straightforward. It simply asks the database to select FilmName, YearReleased, Rating, and AvailableOnDVD from the Films database.

```
SELECT FilmName, YearReleased, Rating, AvailableOnDVD
FROM Films
```

Next comes the query's `WHERE` clause. First, it requires that the film be from the 1980s, and therefore it requires a range between 1980 and 1989. The `BETWEEN` operator is ideal for such a requirement:

```
 WHERE YearReleased BETWEEN 1980 AND 1989
```

The next requirement is that the film must have a rating between 2 and 4. Again, a `BETWEEN` operator is the obvious choice. Because both conditions (YearReleased and Rating) must be true, an `AND` operator is required to link the two:

```
WHERE ( YearReleased BETWEEN 1980 AND 1989 )
AND
( Rating BETWEEN 2 AND 4 )
```

Notice that each condition appears inside brackets. Using brackets is not strictly necessary, but it does make things more readable: Brackets make it clear that an `AND` operator joins two unrelated conditions.

The final condition required is a check to see if the film is available on DVD. Again, the compulsory condition is matched. If you want the right answer to the question, an `AND` statement must be used. This condition is fairly simple, just a check if the AvailableOnDVD column contains a single character `Y`:

```
WHERE ( YearReleased BETWEEN 1980 AND 1989 )
AND
( Rating BETWEEN 2 AND 4 )
AND
AvailableOnDVD = 'Y'
```

Putting it all together, you have the following SQL:

```
SELECT FilmName, YearReleased, Rating, AvailableOnDVD
FROM Films
WHERE ( YearReleased BETWEEN 1980 AND 1989 )
AND
( Rating BETWEEN 2 AND 4 )
AND
( AvailableOnDVD = 'Y' )
```

Execute the SQL and the result is just one record:

| FilmName | YearReleased | Rating | AvailableOnDVD |
|---|---|---|---|
| Gone with the Window Cleaner | 1988 | 3 | Y |

The `SELECT` clause of the second query is also quite straightforward. It simply asks the database system to retrieve records from the FilmName column of the Films table:

```
SELECT FilmName FROM Films
```

The query's first condition requires that the film must not have been released in the 1960s, or to rephrase it, the film's year of release must not be between 1960 and 1969. This condition requires a `BETWEEN` operator:

```
WHERE ( YearReleased NOT BETWEEN 1960 AND 1969 )
```

The next condition requires that film's name must begin with a letter in the range of *P* to *T*:

```
WHERE ( YearReleased NOT BETWEEN 1960 AND 1969 )
AND
( FilmName BETWEEN 'P' AND 'T' )
```

Putting it all together provides the following SQL:

```
SELECT FilmName
FROM Films
WHERE ( YearReleased NOT BETWEEN 1960 AND 1969 )
AND
( FilmName BETWEEN 'P' AND 'T' )
```

When you execute the query, you get the following results:

| FilmName |
|---|
| Sense and Insensitivity |
| Raging Bullocks |

Continuing the look at logical operators, the next section takes you through the `LIKE` operator.

## LIKE Operator

The `LIKE` operator allows you to use *wildcard characters* when searching a character field. A wildcard character is one that doesn't match a specific character but instead matches any one character or any of one or more characters. One example of its use would be finding out details of all members in the film club whose surname begins with *J*.

The following table details the two available wildcard characters.

| Wildcard | Description |
|---|---|
| `%` | Matches one or more characters. Note that MS Access uses the asterisk (`*`) wildcard character instead of the percent sign (`%`) wildcard character. |
| _ | Matches one character. Note that MS Access uses a question mark (`?`) instead of the underscore (_) to match any one character. |

The SQL to match all names beginning with a *J* is as follows:

```
SELECT LastName FROM MemberDetails
WHERE LastName LIKE 'J%';
```

Remember, if you're using MS Access you need to change the percent sign (`%`) to an asterisk (`*`):

```
SELECT LastName FROM MemberDetails
WHERE LastName LIKE 'J*';
```

The preceding code fragment produces these results:

| LastName |
|---|
| Jackson |
| Jones |
| Jones |
| Johnson |

*In some database systems, the* `LIKE` *operator is case-sensitive; in others it is not. Oracle, for example, is case-sensitive, so* `LIKE 'J%'` *matches only an uppercase J followed by one or more characters. In SQL Server,* `LIKE 'J%'` *matches an uppercase or lowercase J followed by one or more characters.*

You can use as many or as few wildcard characters as you wish, and you can mix percent signs and underscores (if required) when searching your database as well, as shown in the following code:

```
SELECT LastName FROM MemberDetails
WHERE LastName LIKE 'D___s';
```

The preceding `SELECT` statement matches any last name that starts with a *D*, ends with an *s*, and has any three characters in between. The results from the example database are as follows:

| LastName |
|---|
| Dales |
| Doors |

*Remember, on some database systems, the* `LIKE` *operator is case-sensitive, so* `LIKE D___s` *matches only strings starting with a capital D. On other systems, it matches uppercase and lowercase Ds. Oracle and DB2 are case-sensitive; MS SQL Server, MySQL, and MS Access are not. Also remember that on MS Access you need to use a question mark instead of the underscore.*

You can also use the `NOT` operator in concert with the `LIKE` operator, which produces a match when the character and wildcard combination is not found. For example, the condition in the following `WHERE` clause is true if the LastName column doesn't start with a *J* followed by one or more characters:

```
SELECT LastName FROM MemberDetails
WHERE LastName NOT LIKE 'J%';
```

Executing the `WHERE` clause provides reverse results of what you saw in the earlier example:

| LastName |
|---|
| Smith |
| Simons |
| Gee |
| Botts |
| Hills |
| Dales |
| Doors |
| Night |

Now that you're acquainted with `LIKE` and `NOT LIKE`, you can use them to query your Film Club database to find specific information about your members. In the following Try It Out, see if you can answer two questions.

## Try It Out Querying a Database with LIKE and NOT LIKE

1. Using the following SQL, query the database to draw up a list of all members' names and zip codes where their zip code starts with 65.

```
SELECT FirstName, LastName, ZipCode
FROM MemberDetails
WHERE ZipCode LIKE '65%'
```

2. Use the following SQL to query the database to find out which people don't live on a street called Road or Street.

```
SELECT FirstName, LastName, Street
FROM MemberDetails
WHERE Street NOT LIKE '% Road' AND Street NOT LIKE '% Street'
```

## How It Works

Specify that the database provide results from the FirstName, LastName, and ZipCode fields from the MemberDetails database:

```
SELECT FirstName, LastName, ZipCode
FROM MemberDetails
```

The `LIKE` clause is ideal to filter records that have a ZipCode column starting with 65 and ending with any numbers:

```
WHERE ZipCode LIKE '65%'
```

The `65` matches the numbers 6 and 5, and the percent sign (`%`) is the wildcard that matches one or more characters after the 65.

Putting all the elements together provides this SQL:

```
SELECT FirstName, LastName, ZipCode
FROM MemberDetails
WHERE ZipCode LIKE '65%';
```

When you execute the SQL, the results are as follows:

| FirstName | LastName | ZipCode |
|---|---|---|
| Seymour | Botts | 65422 |
| Stuart | Dales | 65422 |

The second query is a negative, in that it requires that the Street field should not end in either Road or Street. Using the `NOT` and `LIKE` operators in conjunction with one another is the obvious choice. Another obvious choice is to use the `AND` operator, which tells the database to search the Street field for streets that don't end in Street and don't end in Road.

```
SELECT FirstName, LastName, Street
FROM MemberDetails
WHERE Street NOT LIKE '% Road' AND Street NOT LIKE '% Street';
```

Executing the SQL provides the results you want:

| FirstName | LastName | Street |
|---|---|---|
| John | Jackson | Long Lane |
| John | Jones | Newish Lane |
| Jenny | Jones | Newish Lane |
| Seymour | Botts | Long Lane |
| Jamie | Hills | Newish Lane |
| Stuart | Dales | Long Lane |

## IN Operator

So far you've used the `OR` operator to check whether a column contains one of two or more values. For example, if you want to check whether a member lives in Townsville, Windy Village, Dover, or Big City, you'd write the following:

```
SELECT City
FROM MemberDetails
WHERE
City = 'Townsville'
OR
City = 'Windy Village'
OR
City = 'Dover'
OR
City = 'Big City';
```

That query is a bit long-winded, and that's where the `IN` operator helps: it functions exactly like the `OR` operator but requires much less typing!

Using the `IN` operator, you can rewrite the preceding SQL like this:

```
SELECT City
FROM MemberDetails
WHERE
City IN ('Townsville', 'Windy Village', 'Dover', 'Big City');
```

It's as simple as that! The `IN` operator checks the database to see if the specified column matches one or more of the values listed inside the brackets. You can use the `IN` operator with any data type, not just text as shown above. The preceding SQL produces the following results:

| City |
|---|
| Townsville |
| Townsville |
| Big City |
| Windy Village |
| Windy Village |
| Big City |
| Dover |

Chapter 8 shows you how to use a SQL `SELECT` statement instead of a list of literal values. In the following Try It Out, however, you stick with the `IN` operator and literal values.

## Try It Out — Using the IN Operator

1. Using the following SQL, query the Film Club database to see which films were released in 1967, 1977, or 1987 and have a rating of either 4 or 5. Include FilmName, YearReleased, and Rating in the search.

```
SELECT FilmName, YearReleased, Rating
FROM Films
WHERE
YearReleased IN (1967, 1977, 1987)
AND
Rating IN (4,5);
```

2. Execute the SQL.

## How It Works

The `SELECT` statement simply specifies the database and the fields to search:

```
SELECT FilmName, YearReleased, Rating
FROM Films
```

The `WHERE` clause employs the `IN` operator to search for a film's year of release:

```
YearReleased IN (1967, 1977, 1987)
```

Use the `IN` operator again to search for films with a rating of either 4 or 5:

```
Rating IN (4,5)
```

Because both conditions (YearReleased and Rating) must be true, an AND statement is required to link them:

```
YearReleased IN (1967, 1977, 1987)
AND
Rating IN (4,5)
```

Putting the whole statement together gives you the following SQL:

```
SELECT FilmName, YearReleased, Rating
FROM Films
WHERE
YearReleased IN (1967, 1977, 1987)
AND
Rating IN (4,5);
```

Executing the SQL statement provides the following results:

| FilmName | YearReleased | Rating |
|---|---|---|
| On Golden Puddle | 1967 | 4 |
| Planet of the Japes | 1967 | 5 |
| Soylent Yellow | 1967 | 5 |

That completes the look at logical operators for this chapter. Chapter 8 discusses logical operators in greater depth. The next section focuses on how to order your query's results.

# Ordering Results with ORDER BY

So far, query results have come in whatever order the database decides, which is usually based on the order in which the data was entered, unless the database is designed otherwise (as you see in later chapters). However, listing query results in a certain order (a list of names in alphabetical order or a list of years in numerical order) often comes in handy. SQL allows you to specify the order of results with the ORDER BY clause.

The ORDER BY clause goes right at the end of the SELECT statement. It allows you to specify the column or columns that determine the order of the results and whether the order is ascending (smallest to largest) or descending (largest to smallest). For example, the following SQL statement displays a list of film years, ordered from earliest to latest:

```
SELECT YearReleased
FROM Films
ORDER BY YearReleased;
```

By default, ORDER BY sorts into ascending order, which is why the results of the preceding SQL sort from lowest to highest number:

| YearReleased |
|---|
| 1947 |
| 1967 |
| 1967 |
| 1967 |
| 1975 |
| 1977 |
| 1980 |
| 1984 |
| 1987 |
| 1988 |
| 1989 |
| 1989 |
| 1997 |
| 2001 |
| 2005 |

If you require descending order, however, you must add `DESC` after the list of columns in the `ORDER BY` clause:

```
SELECT YearReleased
FROM Films
ORDER BY YearReleased DESC;
```

If you execute the preceding `SELECT` statement, the results are displayed from highest to lowest number, as shown in the following table:

| YearReleased |
|---|
| 2005 |
| 2001 |
| 1997 |
| 1989 |
| 1989 |
| 1988 |
| 1987 |

| YearReleased |
|---|
| 1984 |
| 1980 |
| 1977 |
| 1975 |
| 1967 |
| 1967 |
| 1967 |
| 1947 |

Because ascending order is the default for `ORDER BY`, specifying ascending order is not necessary in the SQL, but for completeness, adding `ASC` after the `ORDER BY` clause ensures that results display in ascending order:

```
SELECT YearReleased
FROM Films
ORDER BY YearReleased ASC;
```

The column used to order the results, however, doesn't have to form part of the results. For example, in the following SQL, the `SELECT` statement returns the FilmName and Rating, but the YearReleased column determines order:

```
SELECT FilmName, Rating
FROM Films
ORDER BY YearReleased;
```

The preceding SQL produces the following results:

| FilmName | Rating |
|---|---|
| The Maltese Poodle | 1 |
| On Golden Puddle | 4 |
| Soylent Yellow | 5 |
| Planet of the Japes | 5 |
| One Flew over the Crow's Nest | 2 |
| The Lion, the Witch, and the Chest of Drawers | 1 |
| Raging Bullocks | 4 |
| The Life of Bob | 1 |
| The Dirty Half Dozen | 2 |

*Table continued on following page*

| FilmName | Rating |
|---|---|
| Gone with the Window Cleaner | 3 |
| The Good, the Bad, and the Facially Challenged | 5 |
| 15th Late Afternoon | 5 |
| Nightmare on Oak Street, Part 23 | 2 |
| Sense and Insensitivity | 3 |
| The Wide Brimmed Hat | 1 |

So far, you've sorted results with just one column, but you can use more than one column to sort results. To sort by more than one column, simply list each column by which to sort the results and separate each column with a comma, just as in the column list for a `SELECT` statement. The order in which the columns are listed determines the order of priority in sorting. For example, the SQL to obtain a list of film names, ratings, and years of release and to order them by rating, year of release, and name is as follows:

```
SELECT FilmName, Rating, YearReleased
FROM Films
ORDER BY Rating, YearReleased, FilmName;
```

The preceding SQL produces the following results set:

| FilmName | Rating | YearReleased |
|---|---|---|
| The Maltese Poodle | 1 | 1947 |
| The Lion, the Witch, and the Chest of Drawers | 1 | 1977 |
| The Life of Bob | 1 | 1984 |
| The Wide Brimmed Hat | 1 | 2005 |
| One Flew over the Crow's Nest | 2 | 1975 |
| The Dirty Half Dozen | 2 | 1987 |
| Nightmare on Oak Street, Part 23 | 2 | 1997 |
| Gone with the Window Cleaner | 3 | 1988 |
| Sense and Insensitivity | 3 | 2001 |
| On Golden Puddle | 4 | 1967 |
| Raging Bullocks | 4 | 1980 |
| Planet of the Japes | 5 | 1967 |
| Soylent Yellow | 5 | 1967 |
| 15th Late Afternoon | 5 | 1989 |
| The Good, the Bad, and the Facially Challenged | 5 | 1989 |

First, the database system orders the results by the first column specified in the `ORDER BY` clause, in this case the Rating column. You can see that the Rating column is in complete order from highest to lowest. If multiple records in the Rating column contain two or more identical values, the database system looks at the next column specified in the `ORDER BY` clause (YearReleased, in this case) and orders results by that column.

Look at the results where the rating is 1. You can also see that the YearReleased column is in ascending order for that rating. The same is true for the other ratings.

Finally, if the values for the first and second columns specified in the `ORDER BY` clause have the same value as one or more other records, the final column specified in the clause determines order. In the preceding example, the FilmName column is the final column specified in the clause. If you look at the results with a rating of 5, you notice that two of the films were released in 1967 and two were released in 1989. You also see that the two 1967 films appear in alphabetical order based on the FilmName column, which is specified as the final sort column in the event that the other two columns are of the same value. The same is true of the 1989 films, which again are ordered by FilmName.

You can use `WHERE` clauses in conjunction with the `ORDER BY` clause without any problem. You must ensure that the `ORDER BY` clause goes after the `WHERE` clause. The following example produces a list of films released in the 1960s and orders the results by FilmName:

```
SELECT FilmName, Rating, YearReleased
FROM Films
WHERE YearReleased BETWEEN 1960 AND 1969
ORDER BY FilmName;
```

The preceding SQL produces the results set shown here:

| FilmName | Rating | YearReleased |
|---|---|---|
| On Golden Puddle | 4 | 1967 |
| Planet of the Japes | 5 | 1967 |
| Soylent Yellow | 5 | 1967 |

Notice that the FilmName column appears in ascending alphabetical order. Curiously, the rating column is in order as well, but this is just a fluke, a coincidence. When you don't specify order, you can't rely on results being in any particular order, unless you set up the database to produce specific results. Chapter 3 explores such database setup in greater depth. In the meantime, use the following Try It Out to generate a list of all film club members sorted in alphabetical order by last name, then by date of birth, and finally by first name.

## Try It Out — Generating Ordered Results

1. Query the database to produce a list of all members ordered alphabetically by last name, ordered next by date of birth, and finally by first name. Use the following SQL:

```
SELECT LastName, FirstName, DateOfBirth
FROM MemberDetails
ORDER BY LastName, DateOfBirth, FirstName;
```

2. Add a `WHERE` clause to the preceding SQL to filter the list so that it includes only those members who joined in 2005.

```
SELECT LastName, FirstName, DateOfBirth
FROM MemberDetails
WHERE DateOfJoining BETWEEN '2005-01-01' AND '2005-12-31'
ORDER BY LastName, DateOfBirth, FirstName;
```

3. Finally, query the database with the following SQL to produce a list of members' names and dates of birth sorted in descending order by date of birth.

```
SELECT LastName, FirstName, DateOfBirth
FROM MemberDetails
ORDER BY DateOfBirth DESC;
```

## How It Works

Write the `SELECT` statement to select the LastName, FirstName, and DateOfBirth fields from the MemberDetails table:

```
SELECT LastName, FirstName, DateOfBirth
FROM MemberDetails ;
```

To return the results in the order you want, an `ORDER BY` clause is required. Because the `ORDER BY` clause determines priority of ordering, be sure to list the columns in the order LastName, DateOfBirth, FirstName:

```
SELECT LastName, FirstName, DateOfBirth
FROM MemberDetails
ORDER BY LastName, DateOfBirth, FirstName;
```

When you execute the preceding SQL, you get the results you want:

| LastName | FirstName | DateOfBirth |
|---|---|---|
| Botts | Seymour | 1956-10-21 |
| Dales | Stuart | 1956-08-07 |
| Doors | William | 1994-05-28 |
| Gee | Steve | 1967-10-05 |
| Hills | Jamie | 1992-07-17 |
| Jackson | John | 1974-05-27 |
| Johnson | Jack | 1945-06-09 |
| Jones | John | 1952-10-05 |
| Jones | Jenny | 1953-08-25 |

| LastName | FirstName | DateOfBirth |
|---|---|---|
| Night | Doris | 1997-05-28 |
| Simons | Susie | 1937-01-20 |
| Smith | Katie | 1977-01-09 |

To filter the results to include only members who joined in 2005, simply add a `WHERE` clause before the `ORDER BY` clause. Using a `BETWEEN` clause is the easiest way to include only dates in 2005. Currently the database contains only dates in 2004 and 2005, so it might be tempting to filter all dates with a `>= 2005` operator, but if you add data later that includes 2006 dates and beyond, the SQL would no longer return valid results. With the `WHERE` clause added, the SQL is now as follows:

```
SELECT LastName, FirstName, DateOfBirth
FROM MemberDetails
WHERE DateOfJoining BETWEEN '2005-01-01' AND '2005-12-31'
ORDER BY LastName, DateOfBirth, FirstName;
```

Remember that Oracle's date format is day-month-year. Also remember that MS Access needs the hash sign (`#`) rather than single quotes around the dates.

When executed, the preceding SQL provides these results:

| LastName | FirstName | DateOfBirth |
|---|---|---|
| Botts | Seymour | 1956-10-21 |
| Dales | Stuart | 1956-08-07 |
| Doors | William | 1994-05-28 |
| Hills | Jamie | 1992-07-17 |
| Jackson | John | 1974-05-27 |
| Johnson | Jack | 1945-06-09 |
| Jones | John | 1952-10-05 |
| Jones | Jenny | 1953-08-25 |
| Night | Doris | 1997-05-28 |
| Simons | Susie | 1937-01-20 |

Previously, results have appeared in ascending order, which is the `ORDER BY` clause's default order. This time, however, you want results to appear in descending order, so you must add `DESC` at the end of the `ORDER BY` clause's list of columns:

```
SELECT LastName, FirstName, DateOfBirth
FROM MemberDetails
ORDER BY DateOfBirth DESC;
```

Executing this SQL provides the following results:

| LastName | FirstName | DateOfBirth |
|---|---|---|
| Night | Doris | 1997-05-28 |
| Doors | William | 1994-05-28 |
| Hills | Jamie | 1992-07-17 |
| Smith | Katie | 1977-01-09 |
| Jackson | John | 1974-05-27 |
| Gee | Steve | 1967-10-05 |
| Botts | Seymour | 1956-10-21 |
| Dales | Stuart | 1956-08-07 |
| Jones | Jenny | 1953-08-25 |
| Jones | John | 1952-10-05 |
| Johnson | Jack | 1945-06-09 |
| Simons | Susie | 1937-01-20 |

The next section looks at how columns in the results set can be joined together, for example, returning FirstName and LastName columns as one column called FullName.

# *Joining Columns — Concatenation*

Not only does SQL allow you to query various columns, but it also allows you to combine one or more columns and give the resulting column an alias. Note that using an alias has no effect on the table itself; you're not really creating a new column in the table, only one for the results set. When you join columns together, you *concatenate* them. For example, if you have the data ABC and concatenate it to DEF, you get ABCDEF.

This chapter only attempts to concatenate text literals or columns that have the `char` or `varchar` data type. Joining text with a number can cause errors. Chapter 5 shows you how to convert data types and use this to convert numbers to text and then concatenate.

So how do you go about concatenating text? Unfortunately, concatenating text varies depending on the database system you're using. Because there are significant differences among the five database systems covered in this chapter, each is taken in turn, starting with SQL Server and MS Access. You simply need to read the section relevant to the database system you're using — feel free to skip over the others.

## MS SQL Server and MS Access

Both SQL Server and Access concatenate columns and data in the same manner — with the concatenation operator, which on these systems is the plus (+) sign. So, in order to return the full name, that is first and last name, of members, you'd write the following:

```
SELECT FirstName + ' ' + LastName AS FullName FROM MemberDetails;
```

The preceding SQL returns the following results:

| FullName |
|---|
| Katie Smith |
| Susie Simons |
| John Jackson |
| Steve Gee |
| John Jones |
| Jenny Jones |
| Jack Johnson |
| Seymour Botts |
| Jamie Hills |
| Stuart Dales |
| William Doors |
| Doris Night |

Notice that not only can you join columns, but you can also join text. In the preceding example, you added a space between the first name and last name to make it more readable; otherwise, results would have been KatieSmith, SandraTell, and so on. When you join columns, you can add whatever text you wish. Consider the following query:

```
SELECT 'First name is ' + FirstName + ', last name is ' + LastName FullName
FROM MemberDetails;
```

The preceding code produces these results:

| FullName |
|---|
| First name is Katie |
| First name is Susie |
| First name is John |
| First name is Steve |
| First name is John |
| First name is Jenny |
| First name is Jack |
| First name is Seymour |
| First name is Jamie |

*Table continued on following page*

| FullName |
|---|
| First name is Stuart |
| First name is William |
| First name is Doris |

Likewise, you can assign more than one alias in the `SELECT` statement, as in the following code:

```
SELECT LastName AS Surname, FirstName AS ChristianName
FROM MemberDetails;
```

The results of the preceding `SELECT` statement are as follows:

| Surname | ChristianName |
|---|---|
| Smith | Katie |
| Simons | Susie |
| Jackson | John |
| Gee | Steve |
| Jones | John |
| Jones | Jenny |
| Johnson | Jack |
| Botts | Seymour |
| Hills | Jamie |
| Dales | Stuart |
| Doors | William |
| Night | Doris |

Finally, if you want to use an alias that contains spaces or any other characters normally not permitted for column names or aliases, then you must enclose the alias name inside square brackets, as shown in the following statement:

```
SELECT LastName AS Surname, FirstName AS [Christian Name]
FROM MemberDetails;
```

Using this SQL, the alias `Christian Name` has a space, so it's enclosed inside the square brackets. Square brackets allow you to use names for columns or aliases that contain characters not normally considered legal. For example, you would receive an error if you tried to use the alias `One***Two`, as in the following query:

```
SELECT DateOfBirth AS One***Two FROM MemberDetails;
```

If you put square brackets around it (as shown below), the database system is happy:

```
SELECT DateOfBirth AS [One***Two] FROM MemberDetails;
```

That covers concatenation in MS Access and SQL Server; now it's time to look at Oracle and DB2's way of concatenating.

## Oracle and IBM DB2

There are two ways of concatenating text data or text-based columns in Oracle and DB2. The first is to use the concatenation operator, which in these systems is two vertical pipe (||) characters (see the following statement). The second is to use the `CONCAT()` function, which is covered later in this section.

```
SELECT FirstName || ' ' || LastName AS FullName FROM MemberDetails;
```

The preceding SQL returns the following results:

| FullName |
|---|
| Katie Smith |
| Susie Simons |
| John Jackson |
| Steve Gee |
| John Jones |
| Jenny Jones |
| Jack Johnson |
| Seymour Botts |
| Jamie Hills |
| Stuart Dales |
| William Doors |
| Doris Night |

As with SQL Server, not only can you join columns, but you can also join text. In the preceding example, you added a space between the first name and last name to improve readability; otherwise, results would have been KatieSmith, SandraTell, and so on. When you join columns, you can add whatever text you wish, as shown in the following statement:

```
SELECT 'First name is ' || FirstName || ', last name is ' || LastName FullName
FROM MemberDetails;
```

The preceding code produces these results:

| FullName |
|---|
| First name is Katie, last name is Smith |
| First name is Susie, last name is Simons |
| First name is John, last name is Jackson |
| First name is Steve, last name is Gee |
| First name is John, last name is Jones |
| First name is Jenny, last name is Jones |
| First name is Jack, last name is Johnson |
| First name is Seymour, last name is Botts |
| First name is Jamie, last name is Hills |
| First name is Stuart, last name is Dales |
| First name is William, last name is Doors |
| First name is Doris, last name is Night |

Likewise, you can assign more than one alias in the `SELECT` statement. For example, the following statement uses the aliases `Surname` and `ChristianName`:

```
SELECT LastName AS Surname, FirstName AS ChristianName
FROM MemberDetails;
```

The results of the preceding `SELECT` statement are as follows:

| Surname | ChristianName |
|---|---|
| Smith | Katie |
| Simons | Susie |
| Jackson | John |
| Gee | Steve |
| Jones | John |
| Jones | Jenny |
| Johnson | Jack |
| Botts | Seymour |
| Hills | Jamie |
| Dales | Stuart |
| Doors | William |
| Night | Doris |

Finally, if you want to use an alias that contains spaces or any other characters normally not permitted for column names or aliases, then you must enclose the alias name inside square brackets, as shown below:

```
SELECT LastName AS Surname, FirstName AS [Christian Name]
FROM MemberDetails;
```

Using this SQL, the alias `Christian Name` has a space, so it's enclosed inside the square brackets. Square brackets allow you to use names for columns or aliases that contain characters not normally considered legal. For example, you would receive an error if you tried to use the alias `One***Two`, as shown in the following statement:

```
SELECT DateOfBirth AS One***Two FROM MemberDetails;
```

But if you put square brackets around the alias, the database system is happy:

```
SELECT DateOfBirth AS [One***Two] FROM MemberDetails;
```

Both Oracle and DB2 support a second way of concatenating: the `CONCAT()` function. You pass the two things you want to join, either columns or literal strings, as arguments to the function, and the function returns them joined. For example, to join FirstName and LastName columns, you would write the following query:

```
SELECT CONCAT(FirstName, LastName) FROM MemberDetails;
```

Executing this query produces the following results:

```
KatieSmith
SusieSimons
JohnJackson
SteveGee
JohnJones
JennyJones
JackJohnson
SeymourBotts
JamieHills
StuartDales
WilliamDoors
DorisNight
```

Although `CONCAT()` does the same thing in Oracle and DB2, there are plenty of subtle differences. One difference is that Oracle does its best to convert values into text. Although the following statement works on Oracle, it produces an error in DB2:

```
SELECT CONCAT(DateOfBirth, LastName) FROM MemberDetails;
```

Even though DateOfBirth is a date column and not a `character` data type, if you try to execute the same query in DB2, you get an error. For most of this book, you'll find it easier to use the double vertical pipe (||) to concatenate data.

That covers concatenation in Oracle and DB2. Now it's time to look at MySQL's way of concatenating.

## MySQL

MySQL concatenates using one of two functions. The first is the `CONCAT()` function, which works in a way similar to `CONCAT()` used with Oracle and DB2. However, unlike under those database systems, it can take two or more arguments. So, if you want to join three columns, you would write a query similar to the following:

```
SELECT CONCAT(MemberId,FirstName,LastName) FROM MemberDetails;
```

Executing the query gives these results:

```
1KatieSmith
4SteveGee
5JohnJones
6JennyJones
7JohnJackson
8JackJohnson
9SeymourBotts
10SusieSimons
11JamieHills
12StuartDales
13WilliamDoors
14DorisNight
```

Notice that a numeric data type column is concatenated. MySQL's `CONCAT()` function will, if possible, convert numeric data types to string values before concatenating.

As well as columns, `CONCAT()` can also join string literals. Consider the following code:

```
SELECT CONCAT('The member is called ',FirstName,' ',LastName) AS 'Member Name' FROM
MemberDetails;
```

Executing the preceding query creates the following results:

| Member Name |
| --- |
| The member is called Katie Smith |
| The member is called Steve Gee |
| The member is called John Jones |
| The member is called Jenny Jones |
| The member is called John Jackson |
| The member is called Jack Johnson |
| The member is called Seymour Botts |
| The member is called Susie Simons |
| The member is called Jamie Hills |
| The member is called Stuart Dales |
| The member is called William Doors |
| The member is called Doris Night |

Notice that the results returned by `CONCAT()` are given an alias of `Member Name`. Notice also that the alias is enclosed in single quotes because there are spaces in the alias. The same is true if you want to use characters such as punctuation in the alias.

In the preceding example, you can see that spaces are added to ensure that the sentence reads correctly. A space should appear between a member's first name and a member's last name: "The member is called Katie Smith," not "The member is called KatieSmith."

The second concatenation option provided by MySQL is the `CONCAT_WS()` function, which adds a separator between each of the columns or literals to be concatenated. If you want a single space between each column, you could write a query similar to the following:

```
SELECT CONCAT_WS(' ', 'The member is called',FirstName,LastName) AS 'Member Name'
FROM MemberDetails;
```

Executing the query with the `CONCAT_WS()` function provides exactly the same results as the previous example:

| Member Name |
| --- |
| The member is called Katie Smith |
| The member is called Steve Gee |
| The member is called John Jones |
| The member is called Jenny Jones |
| The member is called John Jackson |

*Table continued on following page*

| Member Name |
| --- |
| The member is called Jack Johnson |
| The member is called Seymour Botts |
| The member is called Susie Simons |
| The member is called Jamie Hills |
| The member is called Stuart Dales |
| The member is called William Doors |
| The member is called Doris Night |

That completes the look at concatenation. The next section shows you how to select data from more than one table at a time.

# Selecting Data from More Than One Table

Using the SQL you've learned so far, you can extract data from only one table in the database, which is quite limiting because often answers require data from more than one table. The developers of SQL realized this limitation and implemented a way of joining data from more than one table into one results set. Using the word *joining* is no accident: in SQL the `JOIN` keyword joins one or more tables together in a results set. Chapter 8 examines all the different types of joins, but this chapter covers the most commonly used (and also the easiest to use) join: the inner join.

To see why joins are necessary and useful, begin with a problem. Say that you want a list of all the film names, years of release, and ratings for the `Historical` film category. Assume that you know the category name but don't know what the CategoryId value is for `Historical`.

If SQL didn't support joins, your first task would be to look in the Category table for the CategoryId for the category with a value `Historical`:

```
SELECT CategoryId
FROM Category
WHERE Category = 'Historical';
```

The preceding SQL returns just one result: `6`. Now you know that the CategoryId for `Historical` is `6`, and that can be used with the CategoryId column in the Films table to get a list of films in the `Historical` category:

```
SELECT FilmName, YearReleased, Rating
FROM Films
WHERE CategoryId = 6;
```

Running the preceding SQL returns the following results:

| FilmName | YearReleased | Rating |
|---|---|---|
| Sense and Insensitivity | 2001 | 3 |
| 15th Late Afternoon | 1989 | 5 |
| Gone with the Window Cleaner | 1988 | 3 |
| The Good, the Bad, and the Facially Challenged | 1989 | 5 |

You might argue that if your database has only six categories, looking up each category is not that hard or time-consuming. It's a different story altogether, though, if your database contains 50 or 100 categories. Also, while computers might be more at home with numbers, most humans prefer names. For example, imagine that you create a film club Web site that contains a page allowing users to choose a category and then display all the films for that category. It's unlikely that the Web site user would want to choose categories based on category IDs. What's more likely is that users would choose by name and allow the database to work out the ID and display the results.

That said, how can you use a join to obtain a list of films in the `Historical` category?

First, you need to determine which table contains category names and allows you to look up the CategoryId. From the previous example, it's clearly the Category table that provides this information. Second, you need to determine which table or tables provide the results you want. Again, based on the preceding example, you know that the Films table contains the data needed. The task now is to join the two tables together. To join the tables, you need to find a link between the two tables. No prizes for guessing that the CategoryId field, which is present in both tables, links the two!

The type of join to use in this case is an *inner join*. An inner join combines two tables and links, or joins, them based on columns within the tables. The inner join allows you to specify which columns form the join and under what conditions. For example, you could specify a condition that says the MemberId column in the MemberDetails table matches a value from the MemberId column in the FavCategory table. Then only records where there is a matching MemberId in both tables are included in the results. To create an inner join, you must specify the two tables to be joined and the column or columns on which the join is based. The syntax looks like this:

```
table1 INNER JOIN  table2 ON column_from_table1 = column_from_table2
```

Applying the syntax to the problem at hand yields the following code*:*

```
SELECT FilmName, YearReleased, Rating
FROM Films INNER JOIN Category
ON Films.CategoryId = Category.CategoryId
WHERE Category.CategoryId = 6;
```

At the top is the `SELECT` statement's list of the columns required to form the results. On the following line are the tables from which the results are drawn. Specify them as normal, except this time the `INNER JOIN` keyword specifies that the two tables should be joined. The `ON` keyword that follows specifies what joins the tables; in this case, the CategoryId field joins them. Note that you must specify the table names in front of the CategoryId, otherwise the database doesn't know if you mean the CategoryId in the Films table or the CategoryId in the Category table. In fact, the two columns don't actually need to have the same name, depending on the database's design. Designing a database that way doesn't make sense, though; when someone looks at the database design, they should be able to see that CategoryId in each table relates to the same set of values.

The preceding SQL produces the same results as before:

| FilmName | YearReleased | Rating |
|---|---|---|
| Sense and Insensitivity | 2001 | 3 |
| 15th Late Afternoon | 1989 | 5 |
| Gone with the Window Cleaner | 1988 | 3 |
| The Good, the Bad, and the Facially Challenged | 1989 | 5 |

The `INNER` part of `INNER JOIN` is actually optional in most database systems: `INNER JOIN` is the default join because it's the most common. That said, you can write the SQL as follows:

```
SELECT FilmName, YearReleased, Rating
FROM Films JOIN Category
ON Films.CategoryId = Category.CategoryId
WHERE Category.CategoryId = 6;
```

Using `INNER JOIN` or simply `JOIN` to create an inner join between tables is not, in fact, the only way to join tables. You may prefer it, though, because `INNER JOIN` and `JOIN` make explicit which tables are being joined, and that in turn makes the SQL easier to read and understand. The alternative way of creating an inner join is simply to specify the link in the `WHERE` clause. Rewriting the preceding SQL by specifying the link looks like this:

```
SELECT FilmName, YearReleased, Rating
FROM Films, Category
WHERE Films.CategoryId = Category.CategoryId AND
Category.CategoryId = 6;
```

The `WHERE` clause specifies that Films.CategoryId should equal Category.CategoryId, which creates the join.

So far, you've used the equals operator (=) to join tables, which is termed *equijoin*. Equijoin is the most common join type, but using any of the other operators is fine.

As mentioned earlier, you're not limited to joining just two tables together in one `SELECT` statement; within reason, it's possible to join as many tables as you like. You are sure to encounter a problem that requires joining multiple tables. For example, say you want to produce a list of each film club member's name and all the films they enjoy based on their favorite film category. In the results, you want to display the members' first and last names, the name of each film, each film's year of release, and finally the category in which each film belongs.

Now this might seem like a fairly simple problem, but it actually involves the most complex SQL so far in the book. When illustrated step-by-step, however, it's not so bad at all.

Having a diagram of the database tables and their fields often helps when it comes to the more tricky SQL queries. Shown in Figure 3-1 is a diagram of the Film Club database.

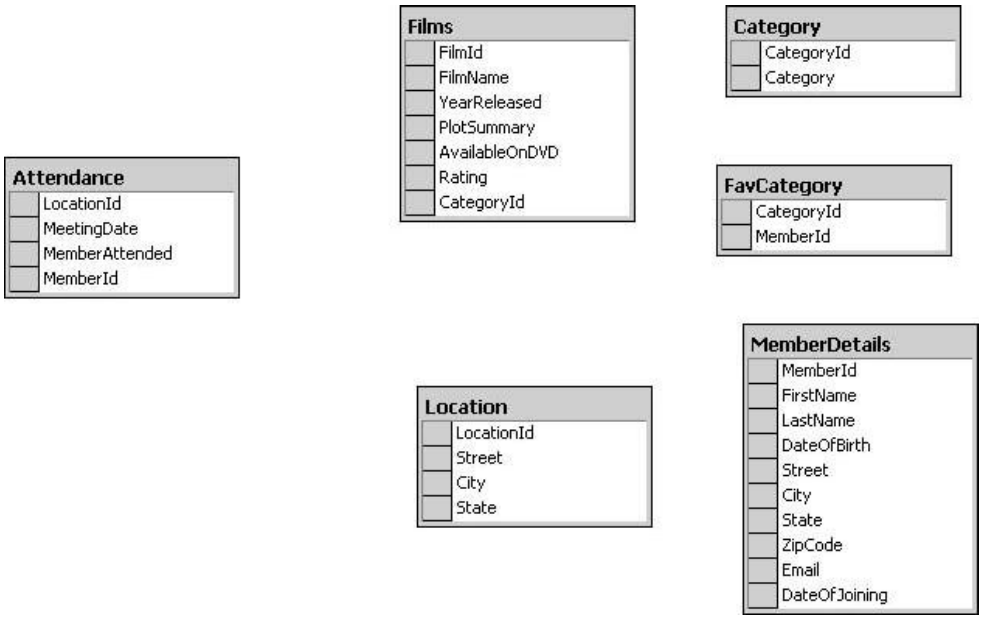


Figure 3-1

The first task when tackling tricky SQL problems is to work out what information is required and which tables contain that information. The preceding scenario specifies that the results must contain the following information:

- Members' names
- All the films that are in the members' favorite categories
- Film names
- Films' year of release
- Category to which each film belongs

You can obtain the members' first and last names from the MemberDetails table. Details of the film names come from the Films table. The category each film belongs to is slightly trickier. Although the Films table contains a CategoryId for each record of a film, this is just a number — the results need to display the category's name. Looking at the table diagram, you can see that the Category table contains the category name, so the Category table provides the category name. But you need to link the Category table to the Films table. The CategoryId field is in both tables, so the CategoryId field provides the link between the two. Remember that when you set up the database, the CategoryId field was a primary key field in the Category table and a foreign key field in the Films table; this field provides the link between the two data sets they hold. Finally, the results ask for each film in a member's favorite category. The FavCategory table contains this information, but again it just contains MemberId and CategoryId, so in order to get the information in a human-friendly format, you need to link these two tables to the MemberDetails and Category tables.

Below is a list summing up the tables you need to use to get the results you want:

- MemberDetails
- Films

- Category
- FavCategory

Now you need to work out how to link them all up. The results you're after comprise a list of films in each member's favorite category, and the FavCategory table is central to the results, so the first step is to link that table. You don't need to use a `WHERE` clause in this SQL because you want all the results. Begin with a simple `SELECT` statement to return all the results from the FavCategory table:

```
SELECT FavCategory.CategoryId, FavCategory.MemberId
FROM FavCategory;
```

That's simple enough and returns the following results:

| CategoryId | MemberId |
|---|---|
| 1 | 3 |
| 1 | 5 |
| 1 | 10 |
| 2 | 1 |
| 2 | 3 |
| 3 | 3 |
| 4 | 6 |
| 4 | 1 |
| 3 | 10 |
| 5 | 3 |
| 5 | 4 |
| 6 | 10 |
| 4 | 11 |
| 3 | 11 |
| 5 | 11 |
| 1 | 12 |
| 4 | 12 |
| 6 | 12 |
| 3 | 13 |
| 5 | 13 |
| 2 | 14 |
| 6 | 14 |
| 1 | 3 |

That's all well and good, but so far your results are only numbers; you're after the category's name. To get the category's name, you need to link to the Category table via the CategoryId column, which links both tables. To link them, use an `INNER JOIN`:

```
SELECT Category.Category, FavCategory.MemberId
FROM FavCategory INNER JOIN Category
ON FavCategory.CategoryId = Category.CategoryId;
```

The SQL produces the following results:

| Category | MemberId |
|---|---|
| Thriller | 3 |
| Thriller | 5 |
| Thriller | 10 |
| Thriller | 3 |
| Romance | 1 |
| Romance | 3 |
| Horror | 3 |
| War | 6 |
| War | 1 |
| Horror | 10 |
| Sci-fi | 3 |
| Sci-fi | 4 |
| Historical | 10 |
| War | 11 |
| Horror | 11 |
| Sci-fi | 11 |
| Thriller | 12 |
| War | 12 |
| Historical | 12 |
| Horror | 13 |
| Sci-fi | 13 |
| Romance | 14 |
| Historical | 14 |

You're one step further, but now you need to get rid of the MemberId and replace it with the members' first and last names. To do this, you need to link to the MemberDetails table and get the data from there, which involves a second `INNER JOIN`:

```
SELECT Category.Category, MemberDetails.FirstName, MemberDetails.LastName
FROM FavCategory INNER JOIN Category
ON FavCategory.CategoryId = Category.CategoryId
INNER JOIN MemberDetails
ON FavCategory.MemberId = MemberDetails.MemberId
ORDER BY MemberDetails.LastName, MemberDetails.FirstName;
```

The preceding code includes a second `INNER JOIN` statement after the first one. The `ON` statement links to the first `INNER JOIN` by linking the FavCategory and MemberDetails tables. The `ORDER BY` statement orders the results by last name and then first name to identify which member likes which categories. Note that if you're using MS Access, you must change the SQL slightly. Access is happy with just one join but insists that you put brackets around each additional join. So the preceding code needs to be rewritten as follows:

```
SELECT Category.Category, MemberDetails.FirstName, MemberDetails.LastName
FROM (FavCategory INNER JOIN Category
ON FavCategory.CategoryId = Category.CategoryId)
INNER JOIN MemberDetails
ON FavCategory.MemberId = MemberDetails.MemberId
ORDER BY MemberDetails.LastName, MemberDetails.FirstName;
```

Notice the brackets around the first inner join. The second inner join, which joins to the results of the first, doesn't need brackets. Note that this code works on the other database systems but that the brackets aren't required around the additional joins.

The results so far are as follows:

| Category | FirstName | LastName |
|---|---|---|
| Thriller | Stuart | Dales |
| War | Stuart | Dales |
| Historical | Stuart | Dales |
| Horror | William | Doors |
| Sci-fi | William | Doors |
| Sci-fi | Steve | Gee |
| War | Jamie | Hills |
| Horror | Jamie | Hills |
| Sci-fi | Jamie | Hills |

| Category | FirstName | LastName |
|---|---|---|
| War | Jenny | Jones |
| Thriller | John | Jones |
| Romance | Doris | Night |
| Historical | Doris | Night |
| Thriller | Susie | Simons |
| Horror | Susie | Simons |
| Historical | Susie | Simons |
| Romance | Katie | Smith |
| War | Katie | Smith |

What you have at the moment is a results set that details each member's favorite film categories. What you need, though, is a list of all the films under each category for each member. To produce such a list, you need to link to the Films table where all the film data is stored. The field that links the two tables is the CategoryId field, which was a primary key field in the Category table and a foreign key field in the Films table when you designed the database. You need to add the following `INNER JOIN` to the bottom of the current `INNER JOIN` list:

```
INNER JOIN Films
ON Films.CategoryId = Category.CategoryId
```

This is the final link needed. It joins the Films table to the results and allows details such as FilmName and YearReleased to be included in the results:

```
SELECT MemberDetails.FirstName, MemberDetails.LastName, Category.Category,
FilmName, YearReleased
FROM ((FavCategory INNER JOIN Category
ON FavCategory.CategoryId = Category.CategoryId)
INNER JOIN MemberDetails
ON FavCategory.MemberId = MemberDetails.MemberId)
INNER JOIN Films
ON Films.CategoryId = Category.CategoryId
ORDER BY MemberDetails.LastName, MemberDetails.FirstName;
```

Note the brackets around the joins, which ensures that the code works with MS Access. The other database systems don't need the brackets, and they can be left off. Notice also that the results of each join are bracketed. The first join is bracketed on its own, then this is bracketed together with the second join, and finally the third join links to these results and doesn't need brackets

The final SQL provides the following results set that details all the films in each member's favorite film category:

| FirstName | LastName | Category | FilmName | YearReleased |
|---|---|---|---|---|
| Stuart | Dales | Thriller | The Maltese Poodle | 1947 |
| Stuart | Dales | Thriller | Raging Bullocks | 1980 |
| Stuart | Dales | Thriller | The Life Of Bob | 1984 |
| Stuart | Dales | War | The Dirty Half Dozen | 1987 |
| Stuart | Dales | War | Planet of the Japes | 1967 |
| Stuart | Dales | Historical | Sense and Insensitivity | 2001 |
| Stuart | Dales | Historical | 15th Late Afternoon | 1989 |
| Stuart | Dales | Historical | Gone with the Window Cleaner | 1988 |
| Stuart | Dales | Historical | The Good, the Bad, and the Facially Challenged | 1989 |
| William | Doors | Horror | The Lion, the Witch, and the Chest of Drawers | 1977 |
| William | Doors | Horror | Nightmare on Oak Street, Part 23 | 1997 |
| William | Doors | Horror | One Flew over the Crow's Nest | 1975 |
| William | Doors | Sci-fi | The Wide Brimmed Hat | 2005 |
| William | Doors | Sci-fi | Soylent Yellow | 1967 |
| Steve | Gee | Sci-fi | The Wide Brimmed Hat | 2005 |
| Steve | Gee | Sci-fi | Soylent Yellow | 1967 |
| Jamie | Hills | War | The Dirty Half Dozen | 1987 |
| Jamie | Hills | War | Planet of the Japes | 1967 |
| Jamie | Hills | Horror | The Lion, the Witch, and the Chest of Drawers | 1977 |
| Jamie | Hills | Horror | Nightmare on Oak Street, Part 23 | 1997 |
| Jamie | Hills | Horror | One Flew over the Crow's Nest | 1975 |
| Jamie | Hills | Sci-fi | The Wide Brimmed Hat | 2005 |
| Jamie | Hills | Sci-fi | Soylent Yellow | 1967 |
| Jenny | Jones | War | The Dirty Half Dozen | 1987 |
| Jenny | Jones | War | Planet of the Japes | 1967 |
| John | Jones | Thriller | The Maltese Poodle | 1947 |
| John | Jones | Thriller | Raging Bullocks | 1980 |
| John | Jones | Thriller | The Life of Bob | 1984 |

| FirstName | LastName | Category | FilmName | YearReleased |
|---|---|---|---|---|
| Doris | Night | Romance | On Golden Puddle | 1967 |
| Doris | Night | Historical | Sense and Insensitivity | 2001 |
| Doris | Night | Historical | 15th Late Afternoon | 1989 |
| Doris | Night | Historical | Gone with the Window Cleaner | 1988 |
| Doris | Night | Historical | The Good, the Bad, and the Facially Challenged | 1989 |
| Susie | Simons | Thriller | The Maltese Poodle | 1947 |
| Susie | Simons | Thriller | Raging Bullocks | 1980 |
| Susie | Simons | Thriller | The Life of Bob | 1984 |
| Susie | Simons | Horror | The Lion, the Witch, and the Chest of Drawers | 1977 |
| Susie | Simons | Horror | Nightmare on Oak Street, Part 23 | 1997 |
| Susie | Simons | Horror | One Flew over the Crow's Nest | 1975 |
| Susie | Simons | Historical | Sense and Insensitivity | 2001 |
| Susie | Simons | Historical | 15th Late Afternoon | 1989 |
| Susie | Simons | Historical | Gone with the Window Cleaner | 1988 |
| Susie | Simons | Historical | The Good, the Bad, and the Facially Challenged | 1989 |
| Katie | Smith | Romance | On Golden Puddle | 1967 |
| Katie | Smith | War | The Dirty Half Dozen | 1987 |
| Katie | Smith | War | Planet of the Japes | 1967 |

If you want only one member's list of films based on their favorite film categories, all you need to do is add a `WHERE` clause and specify their MemberId. The following SQL specifies Jamie Hills's ID, which is 11:

```
SELECT MemberDetails.FirstName, MemberDetails.LastName, Category.Category,
FilmName, YearReleased
FROM (( FavCategory INNER JOIN Category
ON FavCategory.CategoryId = Category.CategoryId)
INNER JOIN MemberDetails
ON FavCategory.MemberId = MemberDetails.MemberId)
INNER JOIN Films
ON Films.CategoryId = Category.CategoryId
WHERE MemberDetails.MemberId = 11
ORDER BY MemberDetails.LastName, MemberDetails.FirstName;
```

This time, you achieve more specific results:

| FirstName | LastName | Category | FilmName | YearReleased |
|---|---|---|---|---|
| Jamie | Hills | War | The Dirty Half Dozen | 1987 |
| Jamie | Hills | War | Planet of the Japes | 1967 |
| Jamie | Hills | Horror | The Lion, the Witch, and the Chest of Drawers | 1977 |
| Jamie | Hills | Horror | Nightmare on Oak Street, Part 23 | 1997 |
| Jamie | Hills | Horror | One Flew over the Crow's Nest | 1975 |
| Jamie | Hills | Sci-fi | The Wide Brimmed Hat | 2005 |
| Jamie | Hills | Sci-fi | Soylent Yellow | 1967 |

As you created the query, you probably noticed that each time you ran the query it produced a unique set of results. That happens because each additional `INNER JOIN` linked to the results set created by the previous SQL. Before moving on to the section that explains the set-based nature of SQL, you should gain more familiarity with the use of brackets in MS Access.

## Using Brackets around Inner Joins in MS Access

You can skip over this section if you're not using MS Access. As mentioned previously, MS Access requires brackets around joins when there's more than one join. Each join creates a set of data, which is discussed later. Each set of data needs to be enclosed in brackets, unless there's only one set.

For example, the following statement involves only one join and therefore only one source set of data:

```
SELECT MemberDetails.MemberId
FROM MemberDetails INNER JOIN FavCategory
ON MemberDetails.MemberId = FavCategory.MemberId;
```

However, if you then join that set of data to another table, creating a second set of data, you must enclose the first set of data inside brackets, like so:

```
SELECT MemberDetails.MemberId
FROM (MemberDetails INNER JOIN FavCategory
ON MemberDetails.MemberId = FavCategory.MemberId)
INNER JOIN Category
ON Category.CategoryId = FavCategory.CategoryId;
```

The following excerpt illustrates how the original join is enclosed inside its own brackets:

```
FROM (MemberDetails INNER JOIN FavCategory
ON MemberDetails.MemberId = FavCategory.MemberId)
```

If you take this further and join the current sets of data to yet another table, then the first two joins must be enclosed in brackets. The following is the original join:

```
FROM (MemberDetails INNER JOIN FavCategory
ON MemberDetails.MemberId = FavCategory.MemberId)
```

Then add the join to the Category table. Note that the first join is enclosed in brackets:

```
FROM ((MemberDetails INNER JOIN FavCategory
ON MemberDetails.MemberId = FavCategory.MemberId)
INNER JOIN Category
ON Category.CategoryId = FavCategory.CategoryId)
```

Now you can add the third join:

```
FROM ((MemberDetails INNER JOIN FavCategory
ON MemberDetails.MemberId = FavCategory.MemberId)
INNER JOIN Category
ON Category.CategoryId = FavCategory.CategoryId)
INNER JOIN Films
ON Category.CategoryId = Films.CategoryId;
```

The full SQL is as follows:

```
SELECT MemberDetails.MemberId
FROM ((MemberDetails INNER JOIN FavCategory
ON MemberDetails.MemberId = FavCategory.MemberId)
INNER JOIN Category
ON Category.CategoryId = FavCategory.CategoryId)
INNER JOIN Films
ON Category.CategoryId = Films.CategoryId;
```

If you add a fourth join, you need to enclose the first three joins in brackets:

```
SELECT MemberDetails.MemberId
FROM (((MemberDetails INNER JOIN FavCategory
ON MemberDetails.MemberId = FavCategory.MemberId)
INNER JOIN Category
ON Category.CategoryId = FavCategory.CategoryId)
INNER JOIN Films
ON Category.CategoryId = Films.CategoryId)
INNER JOIN Attendance
ON MemberDetails.MemberId = Attendance.MemberId;
```

And so it would continue if you add a fifth join, a sixth join, and so on.

None of this adding of brackets is necessary for the other database systems, so the extra brackets aren't included in every example; be sure to add them if you're using Access.

## SQL Is Set-Based

You might remember from your high school math days the concept of the set, which is simply a collection, in no particular order, of items of the same type. SQL is set-based, the sets being sets of records. As with sets in math, there is no particular order to SQL data sets unless you specify the order, using the `ORDER BY` clause, for example. With the more straightforward queries (like the ones earlier in the chapter), considering the set-based nature of SQL queries isn't really necessary. However, with trickier queries, especially those involving more than one table, thinking in terms of sets is helpful.

Taking the example from the previous section, examine how the first step looks as a set. The first step's SQL is as follows:

```
SELECT FavCategory.CategoryId, FavCategory.MemberId
FROM FavCategory;
```

Represented as a set diagram, the first step looks like Figure 3-2.

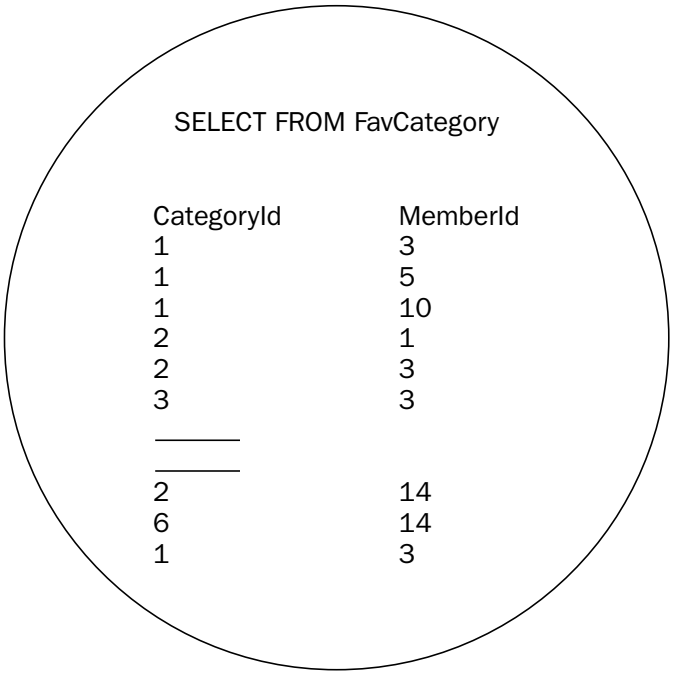


**Figure 3-2**

The set simply contains all the records and columns from the FavCategory table.

The next step joins the Category and FavCategory tables with this SQL:

```
SELECT Category.Category, FavCategory.MemberId
FROM FavCategory INNER JOIN Category
ON FavCategory.CategoryId = Category.CategoryId;
```

Figure 3-3 shows a set diagram of the SQL.

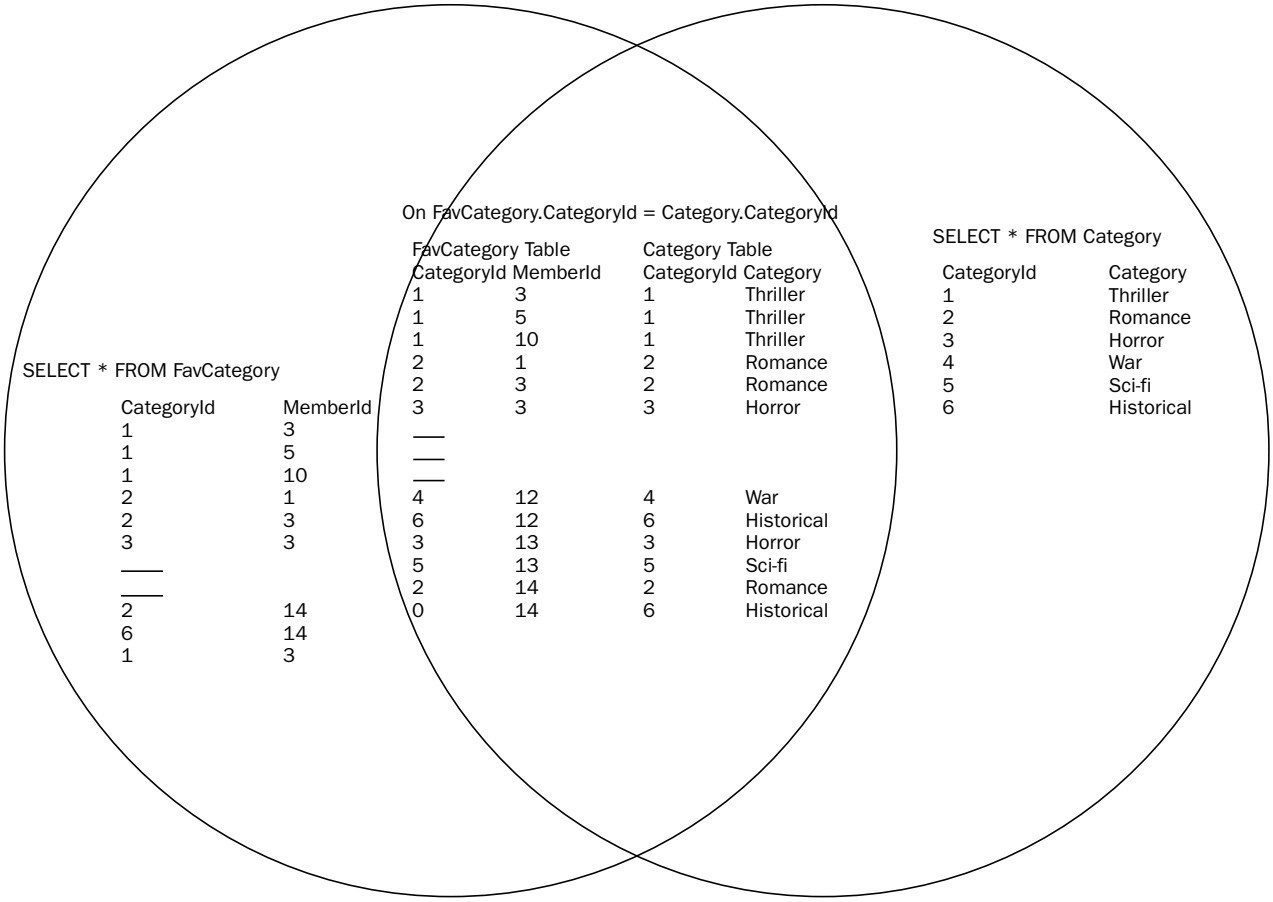


**Figure 3-3**

The circle on the left represents the set of records from the FavCategory table. Because the set isn't filtered (for example, with a `WHERE` clause), it includes all the FavCategory records, though the diagram shows only a handful. The circle on the right is the set of records from the Category table, again unfiltered to include all the records. In the center is the overlap between the two sets, defined by the `ON` clause in the `INNER JOIN`. The results set in the overlap is the final results obtained. It includes all the records from the other two results sets, where there is a matching CategoryId in each table for each record. As it happens, the CategoryId in every record in the FavCategory table finds a matching CategoryId in the CategoryId column, so every record from the FavCategory table is included. The SQL statement selects only two columns, Category and MemberId, to be returned in the results. However, when the database performs the join, all the fields listed in the `ON` statement are also considered.

In order to demonstrate that only records with matching CategoryId's in both tables are included in the joined results set, add another record to the Category table. First, though, here are the results without the new record added:

| Category | MemberId |
|---|---|
| Thriller | 5 |
| Thriller | 10 |
| Romance | 1 |
| War | 6 |
| War | 1 |
| Horror | 10 |
| Sci-fi | 4 |
| Historical | 10 |
| War | 11 |
| Horror | 11 |
| Sci-fi | 11 |
| Thriller | 12 |
| War | 12 |
| Historical | 12 |
| Horror | 13 |
| Sci-fi | 13 |
| Romance | 14 |
| Historical | 14 |

Note that there are 18 rows.

Now execute the following SQL to add a new record to the Category table:

```
INSERT INTO Category (CategoryId, Category)
VALUES (7, 'Comedy');
```

Next, re-execute the query:

```
SELECT Category.Category, FavCategory.MemberId
FROM FavCategory INNER JOIN Category
ON FavCategory.CategoryId = Category.CategoryId;
```

The re-execution provides the following results:

| Category | MemberId |
|---|---|
| Thriller | 5 |
| Thriller | 10 |
| Romance | 1 |
| War | 6 |
| War | 1 |
| Horror | 10 |
| Sci-fi | 4 |
| Historical | 10 |
| War | 11 |
| Horror | 11 |
| Sci-fi | 11 |
| Thriller | 12 |
| War | 12 |
| Historical | 12 |
| Horror | 13 |
| Sci-fi | 13 |
| Romance | 14 |
| Historical | 14 |

Notice the difference? That's right, there is no difference; even though you added an extra record to the Category table, the addition doesn't affect the results because no records exist in the FavCategory results set that match the new CategoryId of 7. Now add a few new favorite categories to the FavCategory table that have a CategoryId of 7:

```
INSERT INTO FavCategory (CategoryId, MemberId)
VALUES (7, 6);

INSERT INTO FavCategory (CategoryId, MemberId)
VALUES (7, 4);


INSERT INTO FavCategory (CategoryId, MemberId)
VALUES (7, 12);
```

Execute the SQL and then rerun the `SELECT` query:

```
SELECT Category.Category, FavCategory.MemberId
FROM FavCategory INNER JOIN Category
ON FavCategory.CategoryId = Category.CategoryId;
```

You should see the following results:

| Category | MemberId |
|---|---|
| Thriller | 5 |
| Thriller | 10 |
| Thriller | 12 |
| Romance | 1 |
| Romance | 14 |
| Horror | 10 |
| Horror | 11 |
| Horror | 13 |
| War | 6 |
| War | 1 |
| War | 11 |
| War | 12 |
| Sci-fi | 4 |
| Sci-fi | 11 |
| Sci-fi | 13 |
| Historical | 10 |
| Historical | 12 |
| Historical | 14 |
| Comedy | 6 |
| Comedy | 4 |
| Comedy | 12 |

Because three new records appear in the FavCategory table with a matching record in the Category table, the results appear in the resulting join.

The next stage joins the MemberDetails table to the current results set:

```
SELECT Category.Category, MemberDetails.FirstName, MemberDetails.LastName
FROM FavCategory INNER JOIN Category
ON FavCategory.CategoryId = Category.CategoryId
INNER JOIN MemberDetails
ON FavCategory.MemberId = MemberDetails.MemberId
ORDER BY MemberDetails.LastName, MemberDetails.FirstName;
```

Remember to add brackets around the first join if you're using MS Access:

```
SELECT Category.Category, MemberDetails.FirstName, MemberDetails.LastName
FROM (FavCategory INNER JOIN Category
ON FavCategory.CategoryId = Category.CategoryId)
INNER JOIN MemberDetails
ON FavCategory.MemberId = MemberDetails.MemberId
ORDER BY MemberDetails.LastName, MemberDetails.FirstName;
```

Figure 3-4 shows the resulting diagram.

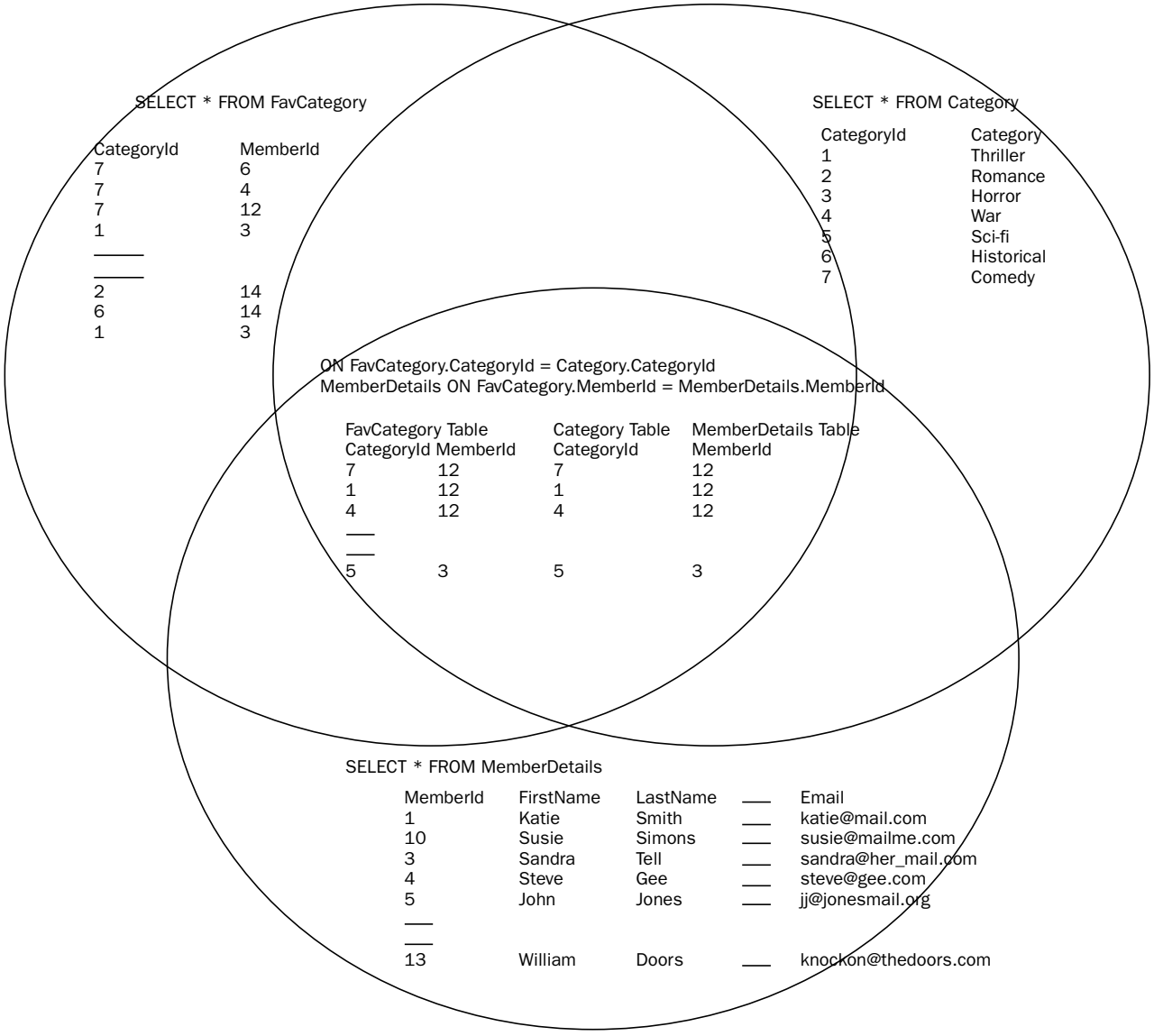


**Figure 3-4**

The overlapping portions of the three data sets form the final results set. The `ON` statements define the overlap area, which is summarized as follows:

- Every record in the FavCategory table must have a matching record in the Category table with the same value in the CategoryId field.
- Every record in the Category table must have a matching record in the FavCategory table with the same value in the CategoryId field.
- Every record in the FavCategory table must have a matching record in the MemberDetails table with the same value in the MemberId field.
- Every record in the MemberDetails table must have a matching record in the Category table with the same value in the MemberId field.

Documenting the remaining steps in the SQL in diagram form would consume precious pages, so hopefully SQL's set-based nature is clear.

By now, you should be familiar enough with sets and inner joins to try out a few.

## Try It Out — Using Inner Joins to Form Sets

1. Using the following SQL, generate a list for the film club chairperson of all the members who don't live in a city in which the club holds meetings. Don't forget to create a set diagram.

```
SELECT MemberDetails.FirstName, MemberDetails.LastName,
       MemberDetails.City, MemberDetails.State
FROM MemberDetails INNER JOIN Location
ON (MemberDetails.City <> Location.City AND MemberDetails.State = Location.State)
OR (MemberDetails.City = Location.City AND MemberDetails.State <> Location.State)
ORDER BY MemberDetails.LastName;
```

2. The club's chairperson also wants a list of all the members who have attended meetings, the date of attendance, and where the meeting was held. To create the list, use the following SQL:

```
SELECT
  MemberDetails.MemberId,
  MemberDetails.FirstName,
  MemberDetails.LastName,
  Attendance.MeetingDate,
  Location.City
FROM
(MemberDetails INNER JOIN Attendance
ON MemberDetails.MemberId = Attendance.MemberId)
INNER JOIN Location ON Location.LocationId = Attendance.LocationId
WHERE Attendance.MemberAttended = 'Y'
ORDER BY MeetingDate, Location.City, LastName, FirstName;
```

## How It Works

First, you need to create a list of all members not living in a city in which club meetings are held. The data for meeting locations resides in the Location table. This information provides one results set. Not

surprisingly, the MemberDetails table holds member details, the second results set. Now it's necessary to combine the two sets to get the result you want: a list of all members whose city is not listed in the Location table. More than one city may have the same name, so assume that such cities in the same state are the same. Therefore, you joined the MemberDetails table to the Location table using an `INNER JOIN` based on the City and State columns:

```
FROM MemberDetails INNER JOIN Location
ON (MemberDetails.City <> Location.City AND MemberDetails.State = Location.State)
OR (MemberDetails.City = Location.City AND MemberDetails.State <> Location.State)
ORDER BY MemberDetails.LastName;
```

The `ON` clause is the key to this query, so you need results where the cities don't match but the states do (same state but different city name). You also want to include records where the city name is the same but the state name is different (same city name but in a different state). It seems obvious when written down, but oftentimes looking at the two results sets and making the comparison in your mind first is helpful. The key fields in both tables are City and State, so you need to make your comparison with those fields. Figure 3-5 shows the set diagram.

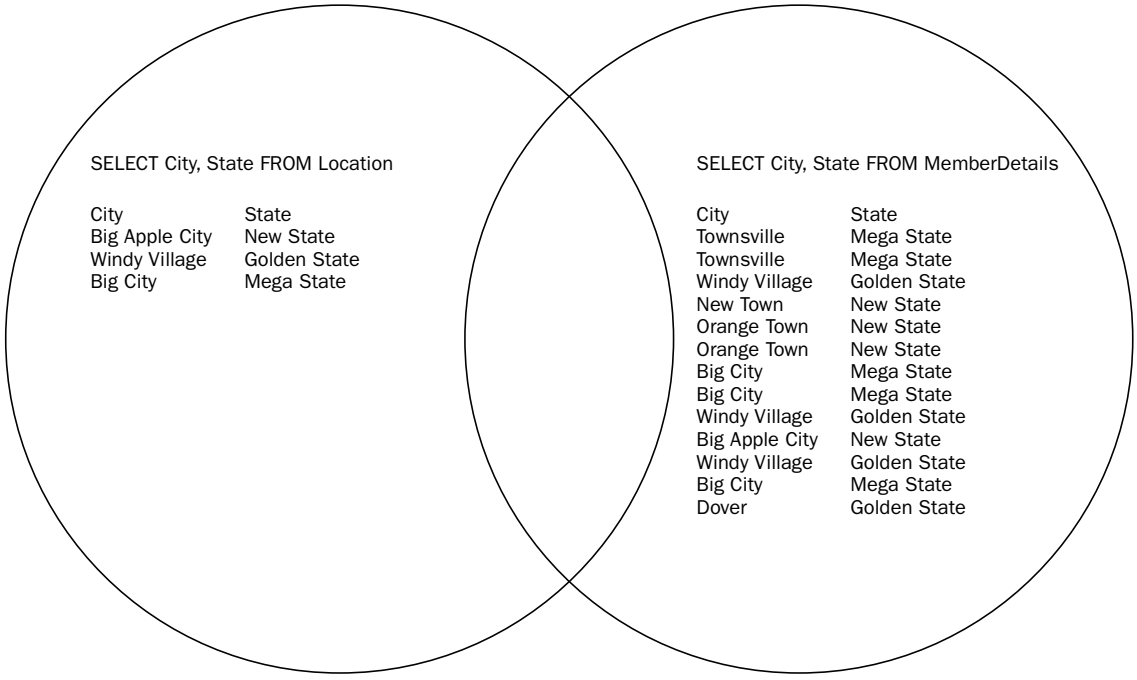


**Figure 3-5**

Check if the Townsville record at the top of the MemberDetails table appears in the City field in any of the records in the Location set. Townsville doesn't appear, so you know that Townsville is not a valid meeting location, and therefore it should form part of the results. You also need to take into account data that isn't there but should be. For example, say that a city called Big City in Sunny State appears in the Location table. In this case, simply comparing the City columns in each table means that Big City, Sunny State, would not be included in the results even though clearly Big City, Mega State, and Big City, Sunny State, are totally different cities.

More than one condition exists in your ON clause. ON clauses are very similar to a SELECT statement's WHERE clause in that you can use the same operators and OR and AND logical operators are allowed. The ON clause in the preceding SQL contains two conditions linked with an OR operator. The first condition states

```
MemberDetails.City <> Location.City AND MemberDetails.State = Location.State
```

It matches a record where the City columns in MemberDetails and Location are not the same. Remember, you want a list of members in cities that don't match in the Location table. The states must be the same, and only the City column must differ. This works so long as each state has cities with unique names!

The second part of the ON clause, joined with the OR statement, checks for cities where the name is the same but the state is different:

```
MemberDetails.City = Location.City AND MemberDetails.State <> Location.State
```

The final SQL is as follows:

```
SELECT MemberDetails.FirstName, MemberDetails.LastName,
       MemberDetails.City, MemberDetails.State
FROM MemberDetails INNER JOIN Location
ON (MemberDetails.City <> Location.City AND MemberDetails.State = Location.State)
OR (MemberDetails.City = Location.City AND MemberDetails.State <> Location.State)
ORDER BY MemberDetails.LastName;
```

Executing the final SQL provide the results shown in the following table:

| FirstName | LastName | MemberDetails.City | MemberDetails.State |
|---|---|---|---|
| Steve | Gee | New Town | New State |
| Doris | Night | Dover | Golden State |
| Susie | Simons | Townsville | Mega State |
| Katie | Smith | Townsville | Mega State |

Phew! That's quite a query, but hopefully it helps to underline SQL's set-based nature.

To find out which members attended which meetings and the dates and locations of attendance, you need to decide which tables hold that data. In this case, the MemberDetails, Attendance, and Location tables hold the data you need. Next, consider how to link the tables together. The Attendance table is central to this query because it contains the LocationId field needed to find out location details, as well as the MemberId field needed to find out member information.

Figure 3-6 shows the set diagram for this problem.

Note that the set in the left circle doesn't contain all the records from a particular table; it contains only records from the Attendance table where the MemberAttended column contains a Y. It reminds you that *set* is not simply another word for *table*; sets contain a selection of a table's records.

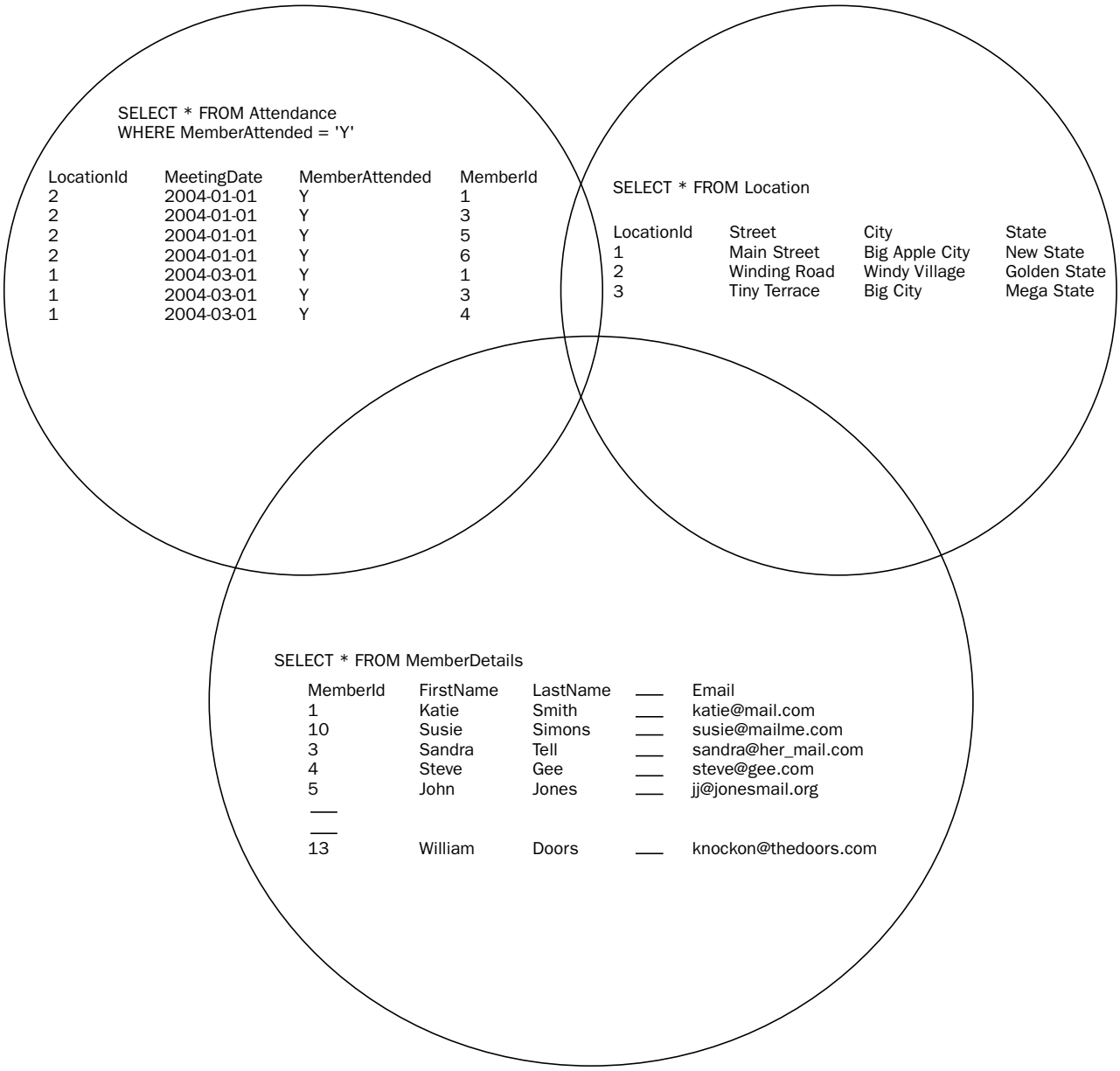


**Figure 3-6**

Start by linking the Location and Attendance tables:

```
SELECT
Attendance.MeetingDate,
Location.City
FROM Attendance
INNER JOIN Location ON Location.LocationId = Attendance.LocationId
```

Next, link the MemberDetails table:

```
SELECT
  MemberDetails.MemberId,
  MemberDetails.FirstName,
  MemberDetails.LastName,
  Attendance.MeetingDate,
  Location.City
FROM
  (MemberDetails INNER JOIN Attendance
  ON MemberDetails.MemberId = Attendance.MemberId)
  INNER JOIN Location ON Location.LocationId = Attendance.LocationId
ORDER BY MeetingDate, Location.City, LastName, FirstName
```

Order the results by meeting date, location, last name, and finally, first name. Remember, however, that you require a list of members who have attended a meeting. Your results set from the Attendance table should include only those records where the MemberAttended field contains a `Y`. So, to finalize your query, add a `WHERE` clause:

```
SELECT
  MemberDetails.MemberId,
  MemberDetails.FirstName,
  MemberDetails.LastName,
  Attendance.MeetingDate,
  Location.City
FROM
(MemberDetails INNER JOIN Attendance
ON MemberDetails.MemberId = Attendance.MemberId)
INNER JOIN Location ON Location.LocationId = Attendance.LocationId
WHERE Attendance.MemberAttended = 'Y'
ORDER BY MeetingDate, Location.City, LastName, FirstName;
```

Executing the final query provides the results shown in the following table:

| MemberId | FirstName | LastName | MeetingDate | City |
|---|---|---|---|---|
| 6 | Jenny | Jones | 2004-01-01 | Windy Village |
| 5 | John | Jones | 2004-01-01 | Windy Village |
| 1 | Katie | Smith | 2004-01-01 | Windy Village |
| 4 | Steve | Gee | 2004-03-01 | Orange Town |
| 1 | Katie | Smith | 2004-03-01 | Orange Town |

One final note before moving on: The vital point to remember is that SQL doesn't compare just one record in one set to just one record in the second set; SQL compares each record in one set to *every* record in the other set, and vice versa. If the `ON` condition is true for a particular record, then the final results set

includes that record. You should always remember that SQL is set-based and compares all records in each set to all the records in the other sets. Reducing set size in `WHERE` clauses is also worthwhile, because smaller sets compare fewer records, which increases the efficiency of queries.

That's it for this section. The next section delves into the unknown!

# Introducing NULL Data

In the sections leading up to this one, you've dealt strictly with known data, but often that's not possible. Given that statement, you might assume that data with no specified value has no value at all. SQL, however, doesn't allow for data to hold no value. Fields with no specified value actually do have a value: `NULL`. `NULL` is not the same thing as nothing; `NULL` represents the unknown. If you were asked how many hairs there are on your head, unless you're bald, you'd have to say that you don't currently know. There is a value, and one day you might know it and be able to store it in a database, but right know it's unknown. This is where `NULL` comes into play. `NULL` represents, and allows you to search for, unknown values.

The following is a SQL statement that inserts a new record into the MemberDetails table. Execute the statement in your own database:

```
INSERT INTO MemberDetails
(MemberId, FirstName, LastName, Email, DateOfJoining)
VALUES (15, 'Catherine','Hawthorn', 'chawthorn@mailme.org', '2005-08-25')
```

The MemberDetails table contains DateOfBirth, Street, City, and State fields, yet the SQL doesn't specify any values for these fields. This is perfectly acceptable, so the question is, what values are contained in the fields where no value is specified? You might suggest that because you specified no values, the values in those fields are no value, or nothing at all. In fact, the database system considers the value not to be nothing but instead to be unknown, or `NULL`.

You might be wondering why you should care about all this.

First of all, `NULL`s can lead to unexpected and overlooked results. For example, you might think that the following SQL would return all the records in the MemberDetails database:

```
SELECT FirstName, LastName, DateOfBirth
FROM MemberDetails
WHERE DateOfBirth <= '1970-01-01' OR DateOfBirth > '1970-01-01';
```

After all, the statement selects all dates of birth on or before January 1, 1970, and also after January 1, 1970, and yet the record just added to the database for Catherine Hawthorn is not there because the date of birth is `NULL` (see the following table). Don't forget to change the date format for Oracle and the single quotes to hash marks (`#`) for Access in the preceding SQL.

| FirstName | LastName | DateOfBirth |
|---|---|---|
| Katie | Smith | 1977-01-09 |
| Steve | Gee | 1967-10-05 |
| John | Jones | 1952-10-05 |
| Jenny | Jones | 1953-08-25 |
| John | Jackson | 1974-05-27 |
| Jack | Johnson | 1945-06-09 |
| Seymour | Botts | 1956-10-21 |
| Susie | Simons | 1937-01-20 |
| Jamie | Hills | 1992-07-17 |
| Stuart | Dales | 1956-08-07 |
| William | Doors | 1994-05-28 |
| Doris | Night | 1997-05-28 |

When the database looks at the records, it says for Catherine's record, "Is `NULL` (unknown) less than January 1, 1970, or is it greater than January 1, 1970?"

Well, the answer is unknown! Records that contain `NULL` values are always excluded from a results set. The same principle applies to any comparison and to inner joins as well. Additionally, most database systems consider unknowns equal when using an `ORDER BY` clause, so all `NULL` values are grouped together.

In order to check for `NULL` values, you must use the `IS NULL` operator. To ensure that a value is not `NULL`, use the `IS NOT NULL` operator, as in the following code:

```
SELECT FirstName, LastName, DateOfBirth
FROM MemberDetails
WHERE DateOfBirth <= '1970-01-01' OR DateOfBirth > '1970-01-01'
OR DateOfBirth IS NULL;
```

The preceding SQL returns all records, shown in the following table:

| FirstName | LastName | DateOfBirth |
|---|---|---|
| Katie | Smith | 1977-01-09 |
| Susie | Simons | 1937-01-20 |
| John | Jackson | 1974-05-27 |
| Steve | Gee | 1967-10-05 |
| John | Jones | 1952-10-05 |

| FirstName | LastName | DateOfBirth |
|---|---|---|
| Jenny | Jones | 1953-08-25 |
| Jack | Johnson | 1945-06-09 |
| Seymour | Botts | 1956-10-21 |
| Jamie | Hills | 1992-07-17 |
| Stuart | Dales | 1956-08-07 |
| William | Doors | 1994-05-28 |
| Doris | Night | 1997-05-28 |
| Catherine | Hawthorn | |

Depending on your database system, it might list DateOfBirth for Catherine as `NULL` or it may just show nothing at all, as in the preceding table.

Generally speaking, you are better off avoiding the `NULL` data type and instead using some default value. For example, if you query for a numerical field, use a number that is never normally part of the results, such as –1 for an age field. For a text field, use an empty string, and so on. Chapter 5 revisits the `NULL` data type when looking at SQL math.

# Summary

This chapter covered a lot of topics, some of which were quite challenging, but they all dealt with how to get answers out of a database. At the end of the day, that's what SQL and databases are all about — getting answers. The key to extracting data with SQL is the `SELECT` query, which allows you to select which columns and from what tables to extract data.

The chapter also discussed the following:

- How to filter results so that you get only the data you require. The `WHERE` clause allows you to specify any number of conditions in order to filter your results to suit your particular query. Only if your specific conditions are met does a record appear in the final results set.
- The logical operators `AND`, `OR`, `NOT`, `BETWEEN`, `IN`, and `LIKE`. Coverage of the `AND` and `OR` operators was a rehash from the previous chapter, but the rest were introduced in this chapter. `NOT` allows you to reverse a condition. The `BETWEEN` operator allows you to specify a range of values and proves a condition true when a column value is within the specified range. When you have a list of potential values, the `IN` operator comes in handy. It proves a condition true when the column has a value that is in the list of given values. Finally, you learned how to use the `LIKE` operator with text. The `LIKE` operator allows the use of wildcard characters.
- After you learned how to get the results set you want, you learned how to use the `ORDER BY` clause, which allows you to list the order of results in ascending or descending order, based on one or more columns.

- The slightly tricky topic of selecting data from more than one table. Up to that point, you could use only one table to create the final results set. However, using the `INNER JOIN` statement allows you to link two or more tables to form a new results set.
- The `NULL` value, which is not the same as no value or zero but in fact signifies an unknown value. The `NULL` value can cause problems when selecting data, however, because when comparing an unknown value to any other value, the result is always unknown and not included in the final results. You must use the `IS NULL` or `IS NOT NULL` operators in order to check for a `NULL` value.

The next chapter returns to database design, this time looking at it in more depth and covering some of the issues not yet covered. This chapter, however, completes the introductory portion of this book; you're now ready to get out there and create your own databases and get your own results! The second half of the book covers more advanced topics, with the aim of developing your SQL skills.

# Exercises

For each of the following exercise questions, write the SQL to list the answers:

1. What is William Doors's (MemberId 13) address?
2. Which members have a surname beginning with the letter *J*?
3. Which members joined before December 31, 2004? Order the results by last name and then by first name.
4. List all the members who attended meetings in Windy Village, Golden State.