# Selecting Data from Different Tables

In Chapter 3, you learned how to use inner joins to create results sets formed from more than one table. In this chapter, to use the words of Chef Elzar from Futurama, you take it up a notch and look at how you can deal with more complex problems using more sophisticated inner joins, and you also something you've not yet encountered: outer joins. Ultimately, this chapter is all about finding a way to get the data you need to answer a question, and you can expect to see lots of practical examples in this chapter.

## Joins Revisited

In Chapter 3, you learned about inner joins and how they enable you to retrieve related data from different tables and display that data in one results set.

Remember, an inner join consists of the `INNER JOIN` keyword, which specifies which two tables are to be joined. After the `INNER JOIN` keyword, you specify an `ON` clause, which identifies the condition that must be true for a row from each table to be included in the final joined results. The basic syntax of an inner join is as follows:

```
name_of_table_on_left INNER JOIN name_of_table_on_right
ON condition
```

For example, the Attendance table doesn't contain the name and address of each person attending a meeting; rather, it contains only the MemberId. However, the MemberDetails table contains matching MemberIds as well as members' full details. So if you want to produce results that contain a meeting date and the name of the member attending, you'd have to create a `SELECT` statement (similar to the following) that produces a results set joining the two tables together:

```
SELECT MeetingDate, FirstName, LastName
FROM Attendance INNER JOIN MemberDetails
ON Attendance.MemberId = MemberDetails.MemberId;
```

The INNER JOIN links the Attendance and MemberDetails tables together in the results. Remember that the link is only for the results produced here; it doesn't change the database structure in any way. The glue that binds them, as it were, is the MemberId columns contained in each table. Strictly speaking, the column names don't have to be the same; it's just the way the database has been designed. In the Film Club database, they are linked so that a MemberId in the Attendance table must have a matching value in the MemberId column in the MemberDetails table.

Executing the statement provides the results you want:

| MeetingDate | FirstName | LastName |
|---|---|---|
| 2004-01-01 | Katie | Smith |
| 2004-01-01 | John | Jones |
| 2004-01-01 | Jenny | Jones |
| 2004-03-01 | Katie | Smith |
| 2004-03-01 | Steve | Gee |

This is an inner join, which requires each result in one table to match a result in the other table. For example, in the preceding table, Katie Smith's record has been matched by a record in the Attendance table, based on Katie's MemberId having an equal MemberId in the Attendance table, a condition that was specified in the ON clause. Shortly, this chapter examines outer joins, which don't require a matching record in every table involved in the join. However, before moving on to outer joins, you need to take a more in-depth look at inner joins.

## Inner Joins: An In-Depth Look

Although Chapter 3 covered inner joins, it detailed only fairly simple ones. In fact, the fairly simple ones are also the most common. Occasions arise, though, when you can't obtain the necessary results with a basic inner join. This section refreshes your memory about inner joins but also covers more complex inner joins such as cross joins and self-joins. The section begins by looking at equijoins and non-equijoins.

### Equijoins and Non-equijoins

An equijoin is simply the correct terminology for an inner join where the ON clause's condition contains an equals (=) operator. Although you haven't encountered the terminology *equijoin* before, Chapter 3 demonstrated them, and almost all of the joins in the book so far have been equijoins; they are far and away the most common and useful of all joins. An equijoin is simply an inner join where the join clause's condition specifies that a field in one table must equal a field in the other table. Consider this basic syntax:

```
one_table INNER JOIN another_table
ON one_table.some_field = another_table.another_field
```

A non-equijoin is, as you've probably already guessed, a join where the condition is not equal, which means that the condition uses operators such as less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=). In theory, you can use most operators, even ones such as LIKE, which would be a bit unusual but perfectly legal. The syntax of a non-equijoin is identical to that of the equijoin, which you've been using throughout the book.

You could use a non-equijoin if, for example, you want a list of films released on or after the year of birth of each member living in Golden State. The SQL required for this example is as follows (note that the example won't work on Oracle, as Oracle doesn't support the `YEAR()` function):

```
SELECT FilmName, FirstName, LastName, State, YearReleased, YEAR(DateOfBirth) AS
YearOfBirth
FROM MemberDetails INNER JOIN Films
ON Films.YearReleased >= YEAR(MemberDetails.DateOfBirth)
WHERE State = 'Golden State'
ORDER BY LastName, FirstName;
```

The `ON` statement joins the two tables by the YearReleased column of the Films table and the result of the `YEAR()` function when passed to the DateOfBirth field in the MemberDetails table. (To refresh your memory about the `YEAR()` function, see Chapter 5.) The statement extracts the year as part of a date from a date expression. Non-equijoins often produce a lot of results — in this case, 31 records:

| FilmName | FirstName | LastName | State | YearReleased | YearOfBirth |
|---|---|---|---|---|---|
| The Life of Bob | Seymour | Botts | Golden State | 1984 | 1956 |
| The Dirty Half Dozen | Seymour | Botts | Golden State | 1987 | 1956 |
| The Good, the Bad, and the Facially Challenged | Seymour | Botts | Golden State | 1989 | 1956 |
| The Lion, the Witch, and the Chest of Drawers | Seymour | Botts | Golden State | 1977 | 1956 |
| Soylent Yellow | Seymour | Botts | Golden State | 1967 | 1956 |
| Planet of the Japes | Seymour | Botts | Golden State | 1967 | 1956 |
| Raging Bullocks | Seymour | Botts | Golden State | 1980 | 1956 |
| Sense and Insensitivity | Seymour | Botts | Golden State | 2001 | 1956 |
| Gone with the Window Cleaner | Seymour | Botts | Golden State | 1988 | 1956 |
| The Wide-Brimmed Hat | Seymour | Botts | Golden State | 2005 | 1956 |
| On Golden Puddle | Seymour | Botts | Golden State | 1967 | 1956 |
| 15th Late Afternoon | Seymour | Botts | Golden State | 1989 | 1956 |
| Nightmare on Oak Street, Part 23 | Seymour | Botts | Golden State | 1997 | 1956 |
| One Flew over the Crow's Nest | Seymour | Botts | Golden State | 1975 | 1956 |
| 15th Late Afternoon | Stuart | Dales | Golden State | 1989 | 1956 |
| Nightmare on Oak Street, Part 23 | Stuart | Dales | Golden State | 1997 | 1956 |

*Table continued on following page*

| FilmName | FirstName | LastName | State | YearReleased | YearOfBirth |
|---|---|---|---|---|---|
| One Flew over the Crow's Nest | Stuart | Dales | Golden State | 1975 | 1956 |
| The Life of Bob | Stuart | Dales | Golden State | 1984 | 1956 |
| The Dirty Half Dozen | Stuart | Dales | Golden State | 1987 | 1956 |
| The Good, the Bad, and the Facially Challenged | Stuart | Dales | Golden State | 1989 | 1956 |
| The Lion, the Witch, and the Chest of Drawers | Stuart | Dales | Golden State | 1977 | 1956 |
| Soylent Yellow | Stuart | Dales | Golden State | 1967 | 1956 |
| Planet of the Japes | Stuart | Dales | Golden State | 1967 | 1956 |
| Raging Bullocks | Stuart | Dales | Golden State | 1980 | 1956 |
| Sense and Insensitivity | Stuart | Dales | Golden State | 2001 | 1956 |
| Gone with the Window Cleaner | Stuart | Dales | Golden State | 1988 | 1956 |
| The Wide-Brimmed Hat | Stuart | Dales | Golden State | 2005 | 1956 |
| On Golden Puddle | Stuart | Dales | Golden State | 1967 | 1956 |
| Sense and Insensitivity | Doris | Night | Golden State | 2001 | 1997 |
| The Wide-Brimmed Hat | Doris | Night | Golden State | 2005 | 1997 |
| Nightmare on Oak Street, Part 23 | Doris | Night | Golden State | 1997 | 1997 |

Originally, this example didn't include the `WHERE` clause, but that generated a whopping 130 results! The Film Club database actually has very few records, but if your database contains a large amount of records, be aware that you could end up with unmanageable numbers of results unless you use a `WHERE` clause in your non-equijoins.

## Multiple Joins and Multiple Conditions

To reiterate a point from Chapter 3, you can have more than one join in a query, which is essential when you need to join more than two tables at a time. For example, if you want a list of members' names and the names of their favorite film categories, you need to join the MemberDetails, Category, and FavCategory tables with the following query:

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

Note that MS Access insists that you place brackets around joins where there are multiple joins, so you need to rewrite the preceding SQL as follows:

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

Notice the inner join on the MemberDetails and FavCategory tables. The results of that inner join are then joined with the Category table. This query produces the following results:

| FirstName | LastName | Category |
|---|---|---|
| Stuart | Dales | Thriller |
| Stuart | Dales | War |
| Stuart | Dales | Historical |
| Stuart | Dales | Comedy |
| William | Doors | Sci-fi |
| William | Doors | Horror |
| Steve | Gee | Sci-fi |
| Steve | Gee | Comedy |
| Jamie | Hills | War |
| Jamie | Hills | Sci-fi |
| Jamie | Hills | Horror |
| Jenny | Jones | War |
| Jenny | Jones | Comedy |
| John | Jones | Thriller |
| Doris | Night | Romance |
| Doris | Night | Historical |
| Susie | Simons | Historical |
| Susie | Simons | Horror |
| Susie | Simons | Thriller |
| Katie | Smith | Romance |
| Katie | Smith | War |

As mentioned, that works just fine for MS SQL Server 2000, IBM DB2, MySQL, and Oracle, but MS Access can't handle it. In order for this query to work in MS Access, you have two options. You saw the first in Chapter 3: using brackets to isolate the inner joins. Consider the following statement:

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

This is called nesting a query. The results are the same and it seems to keep MS Access happy!

The second option is to rewrite the query in a different syntax, with the explicit `INNER JOIN` statements removed, and instead join the tables with a `WHERE` clause, as shown in the following statement:

```
SELECT FirstName, LastName, Category.Category
FROM Category, MemberDetails, FavCategory
WHERE  MemberDetails.MemberId = FavCategory.MemberId
AND FavCategory.CategoryId = Category.CategoryId
ORDER BY LastName, FirstName;
```

Executing this statement produces the same results; they're simply written differently (based on pre-ANSI 92 syntax). It's preferable, however, to use the `INNER JOIN` keywords, because they make explicit what's happening and make the SQL easier to read and understand. As soon as you see the `INNER JOIN` keywords, you know that more than one table is involved in the query and that the tables are joined in some way.

Although there is no strict limit to the number of inner joins allowed in a query, you'll find that, depending on the circumstances, more than a dozen or so start to significantly slow down data retrieval.

Before moving on to cross joins, it's worth mentioning that the `ON` statement is not limited to just one condition; it can contain two or more. For example, the following SQL has two conditions in the `ON` clause, which join on the State and City fields in the Location and MemberDetails tables:

```
SELECT MemberDetails.City, MemberDetails.State
FROM Location INNER JOIN MemberDetails
ON Location.State = MemberDetails.State AND Location.City = MemberDetails.City;
```

The query provides the following results:

| City | State |
|---|---|
| Orange Town | New State |
| Orange Town | New State |
| Orange Town | New State |
| Big City | Mega State |
| Windy Village | Golden State |

| City | State |
|---|---|
| Orange Town | New State |
| Windy Village | Golden State |
| Big City | Mega State |

It may be easier to think of the `ON` clause as a `WHERE` clause that joins tables. The preceding SQL uses an `AND` operator to join the two conditions, but you can use any of the logical operators.

## Cross Joins

A cross join is the most basic join, as there's no `ON` clause to join the tables. Instead, all the rows from all the tables listed in the join are included in the results set. Only the usual filtering clauses associated with `SELECT` statements, such as a `WHERE` clause, limit the results set. You can define a cross join in two ways. The first way of creating a cross join uses the `CROSS JOIN` statement, as shown in the following code, where the Category and Location tables are cross-joined:

```
SELECT Category, Street
FROM Category CROSS JOIN Location
ORDER BY Street;
```

The preceding syntax is supported by Oracle, MySQL, and SQL Server, but not by IBM's DB2 or MS Access. An alternative syntax for a cross join is simply to list all the tables to be cross-joined in the `FROM` clause, as shown here:

```
SELECT Category, Street
FROM Category, Location
ORDER BY Street;
```

Virtually all databases support the second way of cross joining, so it's likely the best way to perform a cross join. Both of the preceding queries do the same thing and produce the same results, which are shown here:

| Category | Street |
|---|---|
| Thriller | Main Street |
| Romance | Main Street |
| Horror | Main Street |
| War | Main Street |
| Sci-fi | Main Street |
| Historical | Main Street |
| Comedy | Main Street |
| Thriller | Tiny Terrace |

*Table continued on following page*

| Category | Street |
|---|---|
| Romance | Tiny Terrace |
| Horror | Tiny Terrace |
| War | Tiny Terrace |
| Sci-fi | Tiny Terrace |
| Historical | Tiny Terrace |
| Comedy | Tiny Terrace |
| Thriller | Winding Road |
| Romance | Winding Road |
| Horror | Winding Road |
| War | Winding Road |
| Sci-fi | Winding Road |
| Historical | Winding Road |
| Comedy | Winding Road |

As you can see, the queries produce a lot of results given that the Location table has only three records and the Category table has only seven records. Why so many results? The results set is the *Cartesian* product of the two tables, which in plain English means all the rows from one table combined with all the rows from the other. This means that every single combination of the two tables appears in the results set. If you look at the results in the preceding table, you'll see `Main Street` combined with `Thriller`, `Main Street` combined with `Romance` — indeed, `Main Street` combined with every record in the Category table. The number of records in a results set will always be the number of records in one table multiplied by the number of records in the other table, In this case, three records in the Location table multiplied by seven records in the Category table equals 21 total records.

Generally speaking, cross joins are not that useful. Inner joins are much more common when joining two or more tables.

## Self-Joins

A *self-join* occurs when a table is joined to itself rather than to another table. Okay, the idea of joining a table to itself might sound a bit crazy, but there are uses for it. The main use is for finding matching pairs of records in a database. However, self-joins are also very useful in conjunction with subqueries, which are covered later in Chapter 8.

How to use a self-join becomes clearer shortly when you see an example. First, though, is the syntax necessary for a self-join. As a self-join is like any other join, the syntax is identical but with just one difference: When joining a table to itself, you must give the table an alias. (Aliases are discussed in Chapter 3.) To recap, to give a table or column an alias, you simply put the keyword `AS` after the table or column name and specify what you want the table to be known as. It doesn't change the table name in the database, only its name when the database system is generating the results set. For example, in the following SQL, the Films table receives the alias `FM1`:

```
SELECT FM1.FilmName
FROM Films AS FM1;
```

Self-joins require an alias because the same table is being used twice, so you need some way to distinguish between the same table that is included twice. An alias is also sometimes referred to as the *correlation name*. Note that while Oracle supports aliases, it does not use the `AS` keyword. Instead, simply put the alias after the table name but without the `AS` keyword, as shown here:

```
SELECT FM1.FilmName
FROM Films FM1;
```

To demonstrate how a self-join works, imagine that the film club's chairperson has asked you to provide a list of all members who are living in the same house as another member. For the purposes of this example, you can assume that the house is the same if the street and zip code are the same. Strictly speaking, you need the street address, but in the example database this is not recorded.

Is there any way you could do this example without resorting to a self-join? In short, the answer is no. In order to find out if the Street and ZipCode fields in one record are the same, you need to compare them. If you try this without a self-join, then all you're doing is comparing the Street and ZipCode fields for the same record, and obviously they'll be the same, as you can see from the following SQL:

```
SELECT FirstName, LastName
FROM MemberDetails
WHERE Street = Street AND ZipCode = ZipCode;
```

For example, taking just one row, if Street is `Main Road` and ZipCode is `123456`, then the preceding SQL reads as follows:

```
SELECT FirstName, LastName
FROM MemberDetails
WHERE 'Main Road' = 'Main Road' AND '123456' = '123456';
```

This statement always evaluated to `true`, as the same column from the same row always equals itself! So you just end up with all the records in the table, just as if there were no `WHERE` clause.

If you need to compare the same fields but different records, you need a self-join. The following SQL is a first attempt at doing a self-join, but it doesn't produce the results you need. Remember to remove the `AS` keywords if you're using Oracle:

```
SELECT MD1.FirstName, MD1.LastName, MD2.FirstName, MD2.LastName,
MD1.ZipCode,MD2.ZipCode, MD1.Street,MD2.Street
FROM MemberDetails AS MD1 INNER JOIN MemberDetails AS MD2
ON MD1.Street = MD2.Street AND MD1.ZipCode = MD2.ZipCode;
```

There's nothing wrong with the syntax; it's a perfectly valid self-join. In the inner join, you have joined the MemberDetails table (giving it an alias of `MD1`) with the MemberDetails table, giving the second occurrence of MemberDetails the alias `MD2`. You can choose any alias name so long as it contains valid characters. The `MD1` and `MD2` tables are joined on the Street and ZipCode fields. However, while it's a valid query, you can see that it has gone wrong somewhere when you execute it to get the results:

| MD1.First Name | MD1.Last Name | MD2.First Name | MD2.Last Name | MD1.Zip Code | MD2.Zip Code | MD1. Street | MD2. Street |
|---|---|---|---|---|---|---|---|
| Katie | Smith | Katie | Smith | 123456 | 123456 | Main Road | Main Road |
| Katie | Smith | Susie | Simons | 123456 | 123456 | Main Road | Main Road |
| Steve | Gee | Steve | Gee | 99112 | 99112 | 45 Upper Road | 45 Upper Road |
| John | Jones | John | Jones | 88776 | 88776 | Newish Lane | Newish Lane |
| John | Jones | Jenny | Jones | 88776 | 88776 | Newish Lane | Newish Lane |
| John | Jones | Jamie | Hills | 88776 | 88776 | Newish Lane | Newish Lane |
| Jenny | Jones | John | Jones | 88776 | 88776 | Newish Lane | Newish Lane |
| Jenny | Jones | Jenny | Jones | 88776 | 88776 | Newish Lane | Newish Lane |
| Jenny | Jones | Jamie | Hills | 88776 | 88776 | Newish Lane | Newish Lane |
| John | Jackson | John | Jackson | 88992 | 88992 | Long Lane | Long Lane |
| Jack | Johnson | Jack | Johnson | 34566 | 34566 | Main Street | Main Street |
| Seymour | Botts | Seymour | Botts | 65422 | 65422 | Long Lane | Long Lane |
| Seymour | Botts | Stuart | Dales | 65422 | 65422 | Long Lane | Long Lane |
| Susie | Simons | Katie | Smith | 123456 | 123456 | Main Road | Main Road |
| Susie | Simons | Susie | Simons | 123456 | 123456 | Main Road | Main Road |
| Jamie | Hills | John | Jones | 88776 | 88776 | Newish Lane | Newish Lane |
| Jamie | Hills | Jenny | Jones | 88776 | 88776 | Newish Lane | Newish Lane |
| Jamie | Hills | Jamie | Hills | 88776 | 88776 | Newish Lane | Newish Lane |
| Stuart | Dales | Seymour | Botts | 65422 | 65422 | Long Lane | Long Lane |
| Stuart | Dales | Stuart | Dales | 65422 | 65422 | Long Lane | Long Lane |
| William | Doors | William | Doors | 34512 | 34512 | Winding Road | Winding Road |
| Doris | Night | Doris | Night | 68122 | 68122 | White Cliff Street | White Cliff Street |

Just looking at the first line, you can see that Katie Smith lives with Katie Smith! If you look at the ZipCode and Street fields, you can see why this record appears. The ON clause specifies that the Street and ZipCode fields should be the same and, of course, they are for the same person. What you're looking for is the same Street and ZipCode for two different people, so you need to add this to your query:

```
SELECT MD1.FirstName, MD1.LastName, MD2.FirstName, MD2.LastName,
MD1.ZipCode,MD2.ZipCode, MD1.Street,MD2.Street
FROM MemberDetails AS MD1 INNER JOIN MemberDetails AS MD2
ON  MD1.Street = MD2.Street AND
       MD1.ZipCode = MD2.ZipCode AND
       MD1.MemberId <> MD2.MemberId;
```

This statement adds the necessary condition to the ON clause, though you could add a WHERE clause and add it there; it's the same thing, just a different syntax. The new condition at the end of the ON clause checks to see that MD1 table's MemberId column doesn't equal MD2 table's MemberId column; you want to make sure you're getting a different member and not including the same member twice. Now when you execute the query, you almost get the results you want:

| MD1.First Name | MD1.Last Name | MD2.First Name | MD2.Last Name | MD1.Zip Code | MD2.Zip Code | MD1. Street | MD2. Street |
|---|---|---|---|---|---|---|---|
| Katie | Smith | Susie | Simons | 123456 | 123456 | Main Road | Main Road |
| John | Jones | Jenny | Jones | 88776 | 88776 | Newish Lane | Newish Lane |
| John | Jones | Jamie | Hills | 88776 | 88776 | Newish Lane | Newish Lane |
| Jenny | Jones | John | Jones | 88776 | 88776 | Newish Lane | Newish Lane |
| Jenny | Jones | Jamie | Hills | 88776 | 88776 | Newish Lane | Newish Lane |
| Seymour | Botts | Stuart | Dales | 65422 | 65422 | Long Lane | Long Lane |
| Susie | Simons | Katie | Smith | 123456 | 123456 | Main Road | Main Road |
| Jamie | Hills | John | Jones | 88776 | 88776 | Newish Lane | Newish Lane |
| Jamie | Hills | Jenny | Jones | 88776 | 88776 | Newish Lane | Newish Lane |
| Stuart | Dales | Seymour | Botts | 65422 | 65422 | Long Lane | Long Lane |

But if you look closely, you'll notice that there are still doubled-up rows. For example, the top row and the second from the bottom row contain the same data. First, Katie Smith's record from MD1 is matched to Susie Simons's in MD2, and then Susie Simons's record in MD1 is matched to Katie Smith's in MD2. You want only the one result, so you need to determine how to prevent both results from appearing in the results set. The easiest way to prevent both records from appearing is to include only the row where the MemberId field in MD1 is less than the MemberId field in MD2. Remember that the MemberId field is unique, so they can never be the same value, which means that one must be smaller than the other:

```
SELECT MD1.MemberId, MD1.FirstName, MD1.LastName, MD2.FirstName, MD2.LastName,
MD1.ZipCode,MD2.ZipCode, MD1.Street,MD2.Street
FROM MemberDetails AS MD1 INNER JOIN MemberDetails AS MD2
ON MD1.Street = MD2.Street AND
   MD1.ZipCode = MD2.ZipCode AND
   MD1.MemberId < MD2.MemberId;
```

The only change made from the previous version of the query is the last part of the `ON` clause. It had been the following:

```
MD1.MemberId <> MD2.MemberId
```

This `ON` clause applies only where the MemberId fields in both tables must be different.A row is included from `MD1` or `MD2` as long as the MemberId fields are different. It led to doubling up, though, as you saw.

However, the `ON` clause has been changed to the following:

```
MD1.MemberId < MD2.MemberId
```

Now a row from `MD1` table appears in the results only if MemberId is less than MemberId from a row in `MD2` table. This ensures that a member only ever appears once in the final results, and finally you get the results you want and only the results you want:

| MD1.First Name | MD1.Last Name | MD2.First Name | MD2.Last Name | MD1.Zip Code | MD2.Zip Code | MD1. Street | MD2. Street |
|---|---|---|---|---|---|---|---|
| 1 | Katie | Smith | Susie | Simons | 123456 | 123456 | Main Road |
| 5 | John | Jones | Jenny | Jones | 88776 | 88776 | Newish Lane |
| 5 | John | Jones | Jamie | Hills | 88776 | 88776 | Newish Lane |
| 6 | Jenny | Jones | Jamie | Hills | 88776 | 88776 | Newish Lane |
| 9 | Seymour | Botts | Stuart | Dales | 65422 | 65422 | Long Lane |

If you didn't have that unique ID column, you would have needed to do the same comparison after choosing another unique column or a set of columns that in combination are unique.

As you can see from this section, inner joins can be complicated, but they can also aid you in retrieving exactly the results you want to retrieve. The next section deals with creating outer joins, which are another useful SQL tool.

## Outer Joins

So far, this book has discussed only inner joins. As you certainly recall, an inner join requires that both tables involved in the join must include matching records; indeed, the `ON` clause must evaluate to `true`.

An outer join, however, doesn't require a match on both sides. You can specify which table always returns results regardless of the conditions in the `ON` clause, although the results are still subject to the filtering effects of any WHERE clause. There are three types of outer join: right outer join, left outer join, and full outer join. The syntax for outer joins is identical to that for inner joins. The only elements that change are the OUTER JOIN keyword and the difference in results produced.

*A word of warning: When using outer joins, MS Access requires that each column name be prefixed with the name of the table. For example, you would write* MemberDetails.Street *and not just* Street*. Other database systems require the table name only if two or more tables have a column or columns with the same name, as in the preceding example where* Street *is a field in both Location and MemberDetails. While this is true of* GROUP BY *and* WHERE *clauses, it's not strictly necessary in the column selection part of the* SELECT *statement unless it's not clear which table the column is from, such as when two or more tables have columns with the same name.*

The discussion begins with the left outer join.

## Left Outer Join

In a *left outer join*, all the records from the table named on the left of the OUTER JOIN statement are returned, regardless of whether there is a matching record in the table on the right of the OUTER JOIN statement. The syntax for the left outer join is as follows:

```
SELECT column_list
FROM left_table LEFT OUTER JOIN right_table
ON condition
```

The following SQL is a left outer join that returns records from the Location and MemberDetails tables:

```
SELECT Location.Street, MemberDetails.Street
FROM Location LEFT OUTER JOIN MemberDetails
ON Location.Street = MemberDetails.Street;
```

Executing this query returns three results:

| Location.Street | MemberDetails.Street |
|---|---|
| Main Street | NULL |
| Winding Road | Winding Road |
| Tiny Terrace | NULL |

There are only three records in the Location table (the table on the left), and all of them are included in the results even when there's no match in the condition in the ON clause. When there's no match in the table on the right of the outer join, NULL is returned, though in your database system you might just see a blank space; either way, the value returned is NULL.

In another example, imagine that the film club chairperson asks for a list of all the film categories and all the films, if any, under each category. Because she wants all the film categories regardless of whether there are matching films, you need to use an outer join. Since this is the section on left outer joins, why don't you try a left outer join! Consider the following statement:

```
SELECT Category, FilmName
FROM Category LEFT OUTER JOIN Films
ON Category.CategoryId = Films.CategoryId;
```

It's a left outer join, so all records in the table on the left of the join statement are included in the results, even if the condition in the ON statement is false:

| Category | FilmName |
|---|---|
| Thriller | The Maltese Poodle |
| Thriller | Raging Bullocks |
| Thriller | The Life of Bob |
| Romance | On Golden Puddle |
| Horror | The Lion, the Witch, and the Chest of Drawers |
| Horror | Nightmare on Oak Street, Part 23 |
| Horror | One Flew over the Crow's Nest |
| War | The Dirty Half Dozen |
| War | Planet of the Japes |
| Sci-fi | The Wide-Brimmed Hat |
| Sci-fi | Soylent Yellow |
| Historical | Sense and Insensitivity |
| Historical | Gone with the Window Cleaner |
| Historical | The Good, the Bad, and the Facially Challenged |
| Historical | 15th Late Afternoon |
| Comedy | NULL |

The results include every row from the Category table, and if there's more than one film of that category, then the category name is included for each film. The Comedy category includes no matching films, so NULL is returned for the FilmName, though it might simply be displayed as an empty cell in your database system.

Don't forget that while a left outer join includes all records from the table on the left of the join, it provides just the initial results set. The database system then applies WHERE and GROUP BY clauses. Change the previous query and add a WHERE clause that includes only records from the Films table where the film is available on DVD:

```
SELECT Category, FilmName
FROM Category LEFT OUTER JOIN Films
ON Category.CategoryId = Films.CategoryId
WHERE AvailableOnDVD = 'Y';
```

Now re-execute the query. You get a much smaller results set due to the filtering effect of the `WHERE` clause:

| Category | FilmName |
|---|---|
| Romance | On Golden Puddle |
| Horror | Nightmare on Oak Street, Part 23 |
| Historical | Sense and Insensitivity |
| War | Planet of the Japes |
| Thriller | The Maltese Poodle |
| Horror | One Flew over the Crow's Nest |
| Thriller | The Life of Bob |
| Historical | Gone with the Window Cleaner |
| Historical | The Good, the Bad, and the Facially Challenged |
| Sci-fi | Soylent Yellow |

Notice that there is no Comedy category in this results set. Although the left outer join returns all the records in the Category table, the `WHERE` clause filters these records and returns only the results where the Films table's AvailableOnDVD field equals `Y`. Because no films match the Comedy category, no results are returned.

## Right Outer Join

A right outer join is simply the reverse of a left outer join, in that instead of all the records in the left table being returned regardless of a successful match in the `ON` clause, now records from the table on the right of the join are returned. The following is the basic syntax for a right outer join:

```
SELECT column_list
FROM left_table RIGHT OUTER JOIN right_table
ON condition
```

If you modify the earlier query joining the Location and MemberDetails tables from a left outer join to a right outer join, you can see the effects, where every row in the table on the right — MemberDetails — is included even where there is no matching row in the Location table:

```
SELECT Location.Street, MemberDetails.Street
FROM Location RIGHT OUTER JOIN MemberDetails
ON Location.Street = MemberDetails.Street;
```

Execute the query and you can see that this time every record from the MemberDetails table is returned, regardless of whether the `ON` clause is true and the Location table contains a matching record:

| Location.Street | MemberDetails.Street |
|---|---|
| NULL | Main Road |
| NULL | 45 Upper Road |
| NULL | Newish Lane |
| NULL | Newish Lane |
| NULL | Long Lane |
| NULL | Main Street |
| NULL | Long Lane |
| NULL | Main Road |
| NULL | Newish Lane |
| NULL | Long Lane |
| Winding Road | Winding Road |
| NULL | White Cliff Street |
| NULL | NULL |

Because only one street name matches in the Location table, all the rows except one return `NULL` for `Location.Street`.

Given that you could simply change the table names so that Location is on the right and MemberDetails is on the left, is there really any need or advantage to using a left outer join compared to a right outer join? Not really. Just use whatever makes sense to you at the time — whichever outer join is easiest to read. Performance differences often occur between inner and outer joins, but between left and right outer joins, it's just a matter of personal preference.

The discussion continues with another demonstration of a right outer join, except this time it also uses a second left outer join in the query as well. Imagine that the chairperson wants a list of all the film categories and the members who like films in each category. She wants all the categories listed, even if no member has selected the category as one of their favorites.

Before starting, however, add a new category that no member has listed as his or her favorite, just to demonstrate that the query works and shows all categories even if no member has selected them as a favorite:

```
INSERT INTO Category (CategoryId, Category) VALUES (9,'Film Noir');
```

The Category table contains a list of all the categories, so this is the table from which you want all the rows to be returned, regardless of whether there are matches in any other tables to which you might join it. Clearly, this is going to be the outer join table.

The FavCategory table contains data on who likes which film categories, and each member's details come from the MemberDetails table. You need to outer-join Category and FavCategory together to get the category list and a list of which members like each category. To get details of the members' names, you need to join the results of the first join to the MemberDetails table, but again, you want the results from the previous join even if the MemberDetails table contains no match.

Start simple and create a query that returns all the categories and a list of IDs of members who chose that category as a favorite:

```
SELECT Category.Category, FavCategory.MemberId
FROM FavCategory
RIGHT OUTER JOIN Category ON FavCategory.CategoryId = Category.CategoryId
ORDER BY Category;
```

The Category table is on the right of the join, and as it's the table whose rows you want regardless of a match in FavCategory, the Category table requires a `RIGHT OUTER JOIN`. Executing the query so far produces the following results:

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

*Table continued on following page*

| Category | MemberId |
|---|---|
| War | 11 |
| War | 12 |
| Comedy | 6 |
| Comedy | 12 |
| Film Noir | NULL |

The results include the Film Noir category despite no one having selected it as their favorite, which is why its MemberId value is `NULL`.

Finally, you need to display each member's name and not just his or her ID. To do this, you need to join to the MemberDetails table. Remember, you want the first outer join to produce the previous results, regardless of whether there is a match in the MemberDetails table. Add the new join after the existing join, but make it a left outer join so that the results to the left of the join are preserved even if no matching records are found in the MemberDetails table, which is the table on the right:

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

*Note that this query won't work on MS Access databases unless you add brackets around the results from the first join. In some instances, using brackets can help us humans see the different results sets and how they all link — and it appears Access needs that help, too.*

The results of the query are as follows:

| Category | FirstName | LastName |
|---|---|---|
| Comedy | Steve | Gee |
| Comedy | Jenny | Jones |
| Comedy | Stuart | Dales |
| Film Noir | NULL | NULL |
| Historical | Susie | Simons |
| Historical | Stuart | Dales |
| Historical | Doris | Night |
| Horror | Susie | Simons |
| Horror | Jamie | Hills |

| Category | FirstName | LastName |
|---|---|---|
| Horror | William | Doors |
| Romance | Katie | Smith |
| Romance | Doris | Night |
| Sci-fi | Steve | Gee |
| Sci-fi | Jamie | Hills |
| Sci-fi | William | Doors |
| Thriller | John | Jones |
| Thriller | Susie | Simons |
| Thriller | Stuart | Dales |
| War | Katie | Smith |
| War | Jenny | Jones |
| War | Jamie | Hills |
| War | Stuart | Dales |

You've covered all the joins now apart from one — the full outer join — which is the topic of the next section.

## Full Outer Join

The final join examined in this chapter is the full outer join. Of all the joins, this is the least well supported by database systems, and indeed, neither MS Access nor MySQL supports it. To be honest, it's not a huge loss, and you rarely need to use a full outer join. A full outer join is essentially a combination of left and right outer joins in that records from the table on the left are included even if there are no matching records on the right, and records from the table on the right are included even if there are no matching records on the left. Aside from the `FULL OUTER JOIN` keywords, the full outer join syntax is identical to that of the other joins:

```
SELECT column_list
FROM left_table FULL OUTER JOIN right_table
ON condition
```

Go back to the original example, which obtains records from the Location and MemberDetails tables, and modify it so that it's a full outer join:

```
SELECT Location.Street, MemberDetails.Street
FROM Location FULL OUTER JOIN MemberDetails
ON Location.Street = MemberDetails.Street;
```

Look at the results that this query gives when executed:

| Location.Street | MemberDetails.Street |
|---|---|
| NULL | Main Road |
| NULL | 45 Upper Road |
| NULL | Newish Lane |
| NULL | Newish Lane |
| NULL | Long Lane |
| NULL | Main Street |
| NULL | Long Lane |
| NULL | Main Road |
| NULL | Newish Lane |
| NULL | Long Lane |
| Winding Road | Winding Road |
| NULL | White Cliff Street |
| NULL | NULL |
| Main Street | NULL |
| Tiny Terrace | NULL |

If you look back at the examples of a left outer join and a right outer join, notice that a full outer join is a combination of the two. These results display all the records from the Location table and all the results from the MemberDetails table. Where there isn't a matching record in the other table, a `NULL` value is returned.

That completes the look at all the joins supported by SQL. The most useful is the inner join, but left outer and right outer joins have their uses as well. The next section looks at how to combine two totally separate results sets using the `UNION` operator. Whereas joins link the different results from each table, a `UNION` operator simply combines them with no particular link.

## Combining Results Sets with the UNION Operator

At times, you might want to combine the results of two quite distinct queries. There may be no link between the results of each query; you just want to display them all in one results set.

You can join the results from two or more `SELECT` queries into one results set by using the `UNION` operator. There are, however, a few ground rules. For starters, each query must produce the same number of columns. For example, the following statement is allowed:

```
SELECT myColumn, myOtherColumn, someColumn FROM MyTable
UNION
SELECT anotherColumn, yetAnotherColumn, MoreColumn FROM MyOtherTable;
```

Both queries have three columns in their results. The `UNION` operator is placed between the two queries to indicate to the database system that you want the results from each query to be presented as one results set.

However, this statement doesn't work because one query returns one column and the other returns three:

```
SELECT myColumn FROM MyTable
UNION
SELECT anotherColumn, yetAnotherColumn, MoreColumn FROM MyOtherTable;
```

Another ground rule is that the columns' data types must be the same, or at least the database system must be able to convert the data types to be the same. The data type for each column was determined when you created the table. If you're not sure what data type a column has, you can find out by looking at the database. For example, the following union works because both `SELECT` queries return columns with matching data types; in this case, FilmId and MemberId are `integer` data types, and FilmName and LastName are both `varchar` data types:

```
SELECT FilmName, FilmId FROM Films
UNION
SELECT LastName, MemberId FROM MemberDetails;
```

| | |
|---|---|
| Smith | 1 |
| The Dirty Half Dozen | 1 |
| On Golden Puddle | 2 |
| The Lion, the Witch, and the Chest of Drawers | 3 |
| Gee | 4 |
| Nightmare on Oak Street, Part 23 | 4 |
| Jones | 5 |
| The Wide-Brimmed Hat | 5 |
| Jones | 6 |
| Sense and Insensitivity | 6 |
| Jackson | 7 |
| Planet of the Japes | 7 |
| Johnson | 8 |

*Table continued on following page*

| | |
|---|---|
| The Maltese Poodle | 8 |
| Botts | 9 |
| One Flew over the Crow's Nest | 9 |
| Raging Bullocks | 10 |
| Simons | 10 |
| Hills | 11 |
| The Life of Bob | 11 |
| Dales | 12 |
| Gone with the Window Cleaner | 12 |
| Doors | 13 |
| The Good, the Bad, and the Facially Challenged | 13 |
| 15th Late Afternoon | 14 |
| Night | 14 |
| Hawthorn | 15 |
| Soylent Yellow | 15 |

The results have returned a combination of the results based on the parameters set forth in the `SELECT` statements joined by the `UNION` operator. The results comprise a list of film names and IDs, as well as member last names and IDs. The order is based on the MemberId and FilmId columns, as these are both primary key columns and therefore are ordered.

Remember, though, that the data types and order of the columns must match. For example, the following query doesn't work:

```
SELECT FilmName, FilmId FROM Films
UNION
SELECT MemberId, LastName FROM MemberDetails;
```

The query doesn't work because the first column in the results set is a `varchar` data type in the first `SELECT` statement and an `integer` data type in the second. You can't tell this just by looking at the query; the data type is set during the design of the database.

Try the following query:

```
SELECT FilmId FROM Films
UNION
SELECT MemberId FROM MemberDetails;
```

The preceding query is identical to the first example, except that it returns only the MemberId and FilmId columns in the results set. So the results, shown in the following table, should include all the MemberIds and FilmIds. However, when executed, the results are as follows:

| |
|---|
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |
| 10 |
| 11 |
| 12 |
| 13 |
| 14 |
| 15 |

But how can that be? When you ran the query earlier it returned 29 rows, and many of the FilmId and MemberId values were the same, and yet there's no duplication of MemberId and FilmId in this results set. These results are different from the previous results because, by default, the `UNION` operator merges the two queries but includes only rows that are unique. If you want all rows returned in the results set, regardless of whether they are unique, you need to use the `ALL` statement, as illustrated in the following SQL:

```
SELECT FilmId FROM Films
UNION ALL
SELECT MemberId FROM MemberDetails;
```

When you execute this query, you get all the rows even if some are identical:

| |
|---|
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |

*Table continued on following page*

| |
|---|
| 9 |
| 10 |
| 11 |
| 12 |
| 13 |
| 14 |
| 15 |
| 1 |
| 10 |
| 14 |
| 6 |
| 5 |
| 8 |
| 7 |
| 11 |
| 15 |
| 4 |
| 13 |
| 12 |
| 1 |

You can also order the results by adding an `ORDER BY` clause. However, only one `ORDER BY` clause is allowed, and it must be after the last `SELECT` statement. In addition, you can use only column names from the first `SELECT` statement in your `ORDER BY` clause, although the clause orders all the rows in all the `SELECT` statements. The following query obtains the union of three queries and orders them by the FilmName column. Note that this query won't work in Oracle because Oracle doesn't support the `YEAR()` function and has a different `ORDER BY` syntax:

```
SELECT FilmName, YearReleased FROM Films
UNION ALL
SELECT LastName, YEAR(DateOfBirth) FROM MemberDetails
UNION ALL
SELECT City, NULL FROM Location
ORDER BY FilmName;
```

There are a few things to note about this query. First, even though the YearReleased and DateOfBirth columns have a different data type, the value returned by the `YEAR()` function is an `integer` data type, so this will match the YearReleased column, which is also an `integer` data type. Remove the `YEAR()`

function and the query still works, except you might find that YearReleased is converted from an `integer` into a `date` data type with some quite odd results. The `YEAR()` function is covered in Chapter 5, as is a discussion of data type conversion. Although normally `CAST()` can be used to convert data types, it's not necessary here because `YEAR()` returns the correct data type for the results.

Also notice that the `ORDER BY` clause comes at the end of the `UNION` statements and refers to the first `SELECT` query. Again, this doesn't work on IBM's DB2, which allows the `ORDER BY` statement to come last only if the name of the column being ordered appears in all the queries. Likewise, the `ORDER BY` syntax when using `UNION` is different in Oracle. Rather than specifying the name of the column to be ordered, you specify the position in which the column appears in the `SELECT` list. If you want to order by FilmName, which is the first column, the `ORDER BY` clause would look like the following:

```
ORDER BY 1;
```

If you want to order by the second column, change the `1` to a `2`. If you want to order by the third column, then change the `1` to a `3`, and so on.

Also note that the query includes only the City column from the Location table and that `NULL` is substituted for the second column. This works on many database platforms but not IBM DB2. Change the value from `NULL` to an integer value that makes it clear that the value is not a genuine piece of data (`-1`, for example).

Executing the query provides the following results:

| | |
|---|---|
| 15th Late Afternoon | 1989 |
| Big City | NULL |
| Botts | 1956 |
| Dales | 1956 |
| Doors | 1994 |
| Gee | 1967 |
| Gone with the Window Cleaner | 1988 |
| Hawthorn | NULL |
| Hills | 1992 |
| Jackson | 1974 |
| Johnson | 1945 |
| Jones | 1952 |
| Jones | 1953 |
| Night | 1997 |
| Nightmare on Oak Street, Part 23 | 1997 |

*Table continued on following page*

| | |
|---|---|
| On Golden Puddle | 1967 |
| One Flew over the Crow's Nest | 1975 |
| Orange Town | NULL |
| Planet of the Japes | 1967 |
| Raging Bullocks | 1980 |
| Sense and Insensitivity | 2001 |
| Simons | 1937 |
| Smith | 1977 |
| Soylent Yellow | 1967 |
| The Dirty Half Dozen | 1987 |
| The Good, the Bad, and the Facially Challenged | 1989 |
| The Life of Bob | 1984 |
| The Lion, the Witch, and the Chest of Drawers | 1977 |
| The Maltese Poodle | 1947 |
| The Wide-Brimmed Hat | 2005 |
| Windy Village | NULL |

The results are a combination of the three individual `SELECT` statements. The first statement is as follows:

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

It produces a list of film names and years of release. The second `SELECT` statement is shown here:

```
SELECT LastName, YEAR(DateOfBirth) FROM MemberDetails
```

This statement produces a list of members' last names and years of birth. The third and final statement is as follows:

```
SELECT City, NULL FROM Location
```

This particular statement produces a list of city names, and in the second column, it simply produces the value `NULL`.

That completes the look at the `UNION` operator, and you should know how to produce one single set of results based on the results of more than one query. While the `UNION` operator is in some ways similar to joins, the biggest difference is that each query is totally separate; there's no link between them.

## Summary

This chapter advanced your knowledge of joins far beyond the simple inner joins covered in Chapter 3. You can now answer questions that were impossible to answer with a basic inner join. The chapter started off by revisiting inner joins. It looked at how to achieve multiple joins with multiple conditions. Additionally, this chapter covered the difference between equijoins, which use the equals operator, and non-equijoins, which use comparison operators such as greater than or less than.

The chapter then turned to outer joins. Whereas inner joins require the `ON` condition to return `true`, outer joins don't. The chapter covered the following types of outer joins:

- Left outer joins, which return rows from the table on the left of the join, whether the `ON` clause is true or not
- Right outer joins, which return rows from the table on the right, regardless of the `ON` clause
- Full outer joins, which return rows from the tables on both sides, even if the `ON` clause is false

In the final part of the chapter, you learned about how you can use the `UNION` statement to combine the results from two or more `SELECT` queries into just one set of results.

The next chapter introduces the topic of subqueries and shows you how you can include another query inside a query.

## Exercises

1. List pairs of people who share the same favorite film category. The results set must include the category name and the first and last names of the people who like that category.
2. List all the categories of film that no one has chosen as their favorite.