7 Joining Tables

Information in Multiple Tables

We spent a lot of time while we were designing our tables removing redundant or duplicate data from our database. Instead of inserting multiple instances of text into a table, we put the text once into another table, and then inserted the primary key pointing to that data into the table that would have contained the duplicates. This leads to very efficiently stored data, but means that we have to join the tables to get the data back from them in a readable form.

For instance, Figure 7.1 shows all the data that is stored in our log table. Notice that the second and third columns contain foreign keys from the cookie and webpage tables. It may be easy to remember that webpageID one is the Home page but what about the other IDs? Also what is so special about cookieID two? It seems that they are the only person that has looked at the website, so did they enter their name in the visitor book as well? We can answer these sort of questions by joining tables together.

There are two basic types of joins in SQL:

- the inner join, and
- the outer join.

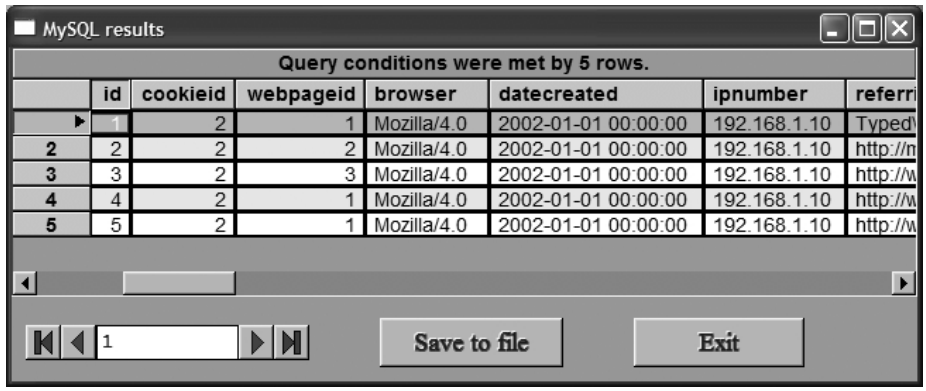

Figure 7.1 The Log table contents.

Cross Joins

The simplest way of joining two or more tables together is by specifying more than one table after the FROM keyword as follows:

```
SELECT    columns
FROM      table1, table2, etc
```

However, that seldom produces the effect that you might expect. Try joining the *Log* table to the *webpage* table as follows:

```
SELECT    *
FROM      log, webpage
```

Figure 7.2 shows the results of this join, called the cross join.

Can you see what this join has done? It is probably easier if you look at this on the screen as it is difficult to show all of the rows in the figure. Look at how many rows the query has returned. By selecting two tables, containing five and seven rows respectively, the query has generated 35 rows as a result. This is because the cross join links every row in the *Log* table with every row in the *webpage* table ($5 \times 7 = 35$). If you scroll along the results you will see that the contents of the rows of both tables are represented in each row of these results.

Now scroll across and look at all the columns that the query has returned. You will see that each record has returned a row that has every column of every table specified in it. You

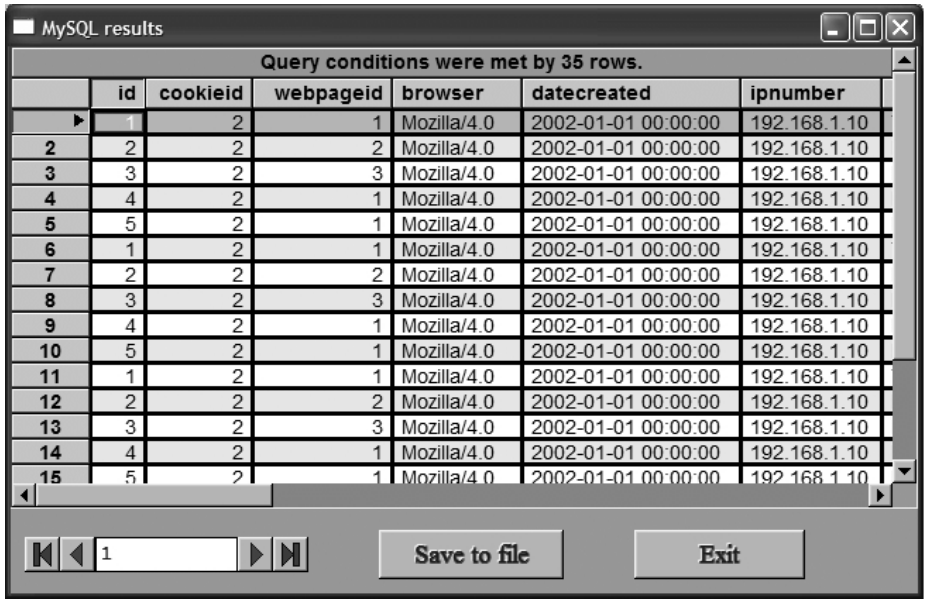

Figure 7.2 The cross join.

might think that you can restrict this by specifying only certain column names in the select statement. This works to an extent, as you can see if you run the following query:

```
SELECT    cookieid
FROM      log,webpage
```

Even though the results now contain only one column, the query has still returned 35 rows, as can be seen in Figure 7.3.

Be careful when using cross joins; they are of little use and can cause incredible performance hits on your database system if you join big tables together. Try cross joining your *webpage*, *log*, *cookies* and *visitorbook* tables together and you will see how quickly this can get out of hand. Joining a few little tables produces a huge result set.

Inner Joins

The cross join as described above is an inner join. Simply put, inner joins require a match between both tables that are being joined. If no match is specified then all rows are matched such as in the cross join.

Equi-join

An inner join that is more useful than the cross join is the equi-join. In an equi-join you specify something to match between the tables being joined, which results in a much more controlled join. The basic format of such a join is as follows:

```
SELECT    columns
FROM      table1, table2, etc
WHERE     condition
```

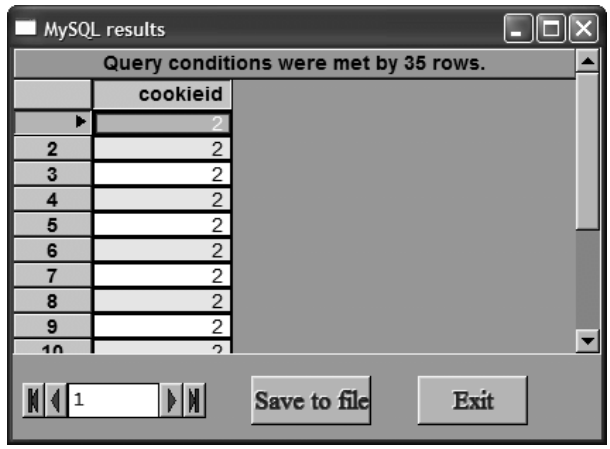

Figure 7.3 A cross join restricted to a single column.

In this case the condition will check for a matching piece of data in a column that appears in the tables to be joined. Most likely this will be matching a foreign key in one table with a primary key in another.

The last paragraph may seem a bit confusing, but as ever with SQL its much easier to demonstrate by example, so we will turn the previous cross join into an equi-join by adding a WHERE clause as follows:

```
SELECT    *
FROM      log, webpage
WHERE     webpage.id = log.webpageid
```

Figure 7.4 shows the results of the above query. Notice how the WHERE clause restricted the number of rows of the query to 5, and got all and more of the data that we want on each row.

Notice that in the WHERE clause we specified the table name as well as the column name for the match. It is not always necessary to do this but if you get into the habit of doing so it can save you a lot of time debugging later. For instance, in our example, if we had just specified the ID column without the table name, we could have been referring to the ID column in either the *webpage* table or the *log* table, as there are two with the same name.

If you cast your mind back to the start of this chapter you will remember we were trying to get more information from the *log* table so that we could see the title of the page that we were looking for as opposed to only its ID. We needed to join the pages together to find this out. We can now tidy up this equi-join by restricting the columns we return as follows:

```
SELECT    webpage.title, log.*
FROM      log, webpage
WHERE     webpage.id = log.webpageid
```

Figure 7.5 shows the results of this equi-join. What the join has done is take every row from the *log* table, which contains the foreign key pointing to the entry in the *webpage* table. It then has included the relevant column from the *webpage* table (*title*) which matches that foreign key.

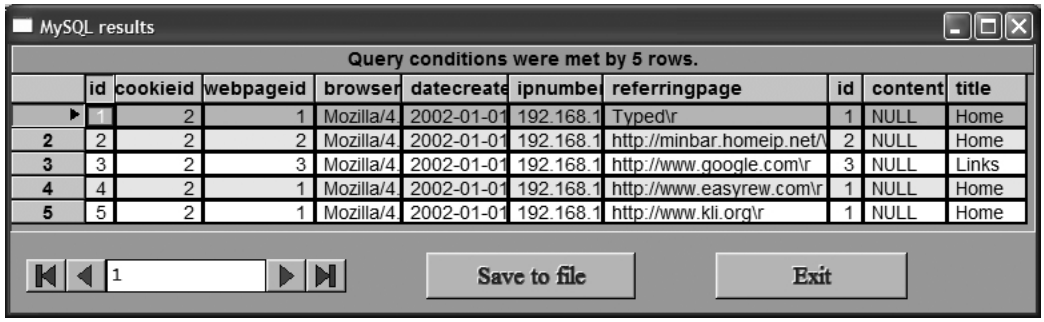

Figure 7.4 A basic equi-join.

If you look at the query again after the SELECT keyword, you will see that we have specified what we want from each table. The entry,

```
log.*
```

is selecting everything from the *log* table, whereas,

```
webpage.title
```

will just select the single column *title* from the *webpage* table. On a previous query we selected all columns from all tables using the asterisk. You can see that using the asterisk to select everything would be the same as using,

```
log.*, webpage.*
```

We will do one more thing to make our results tidier. If you look at Figure 7.5 again you will see that by selecting everything from the *log* table we also select the foreign key *webpageID*. As we already have the title of the webpage which is more useful, we do not need to return that foreign key as well. Unfortunately, this means that we will have to specify all of the other columns in the log table instead of just using the asterisk. We will shuffle the order of the columns slightly and use an alias as well so as to make the output look more like the *log* table. Our final script will now read as follows:

```
SELECT    log.id as logid,
          log.cookieid,
          webpage.title AS pagetitle,
          log.browser,
          log.datecreated,
          log.ipnumber,
          log.referringpage
FROM      log, webpage
WHERE     webpage.id = log.webpageid
```

Figure 7.6 shows the results of our completed query. We've given the *log*'s *id* column the alias *logid* in the output to save it getting confused with any other *id* columns elsewhere. We

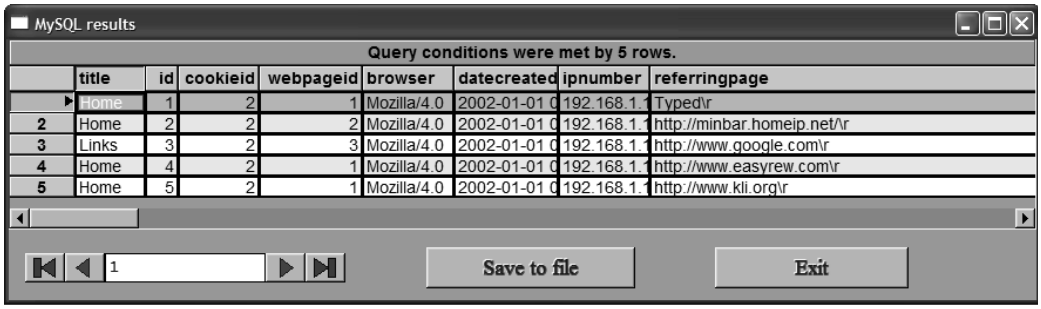

Figure 7.5 An equi-join with selected columns.

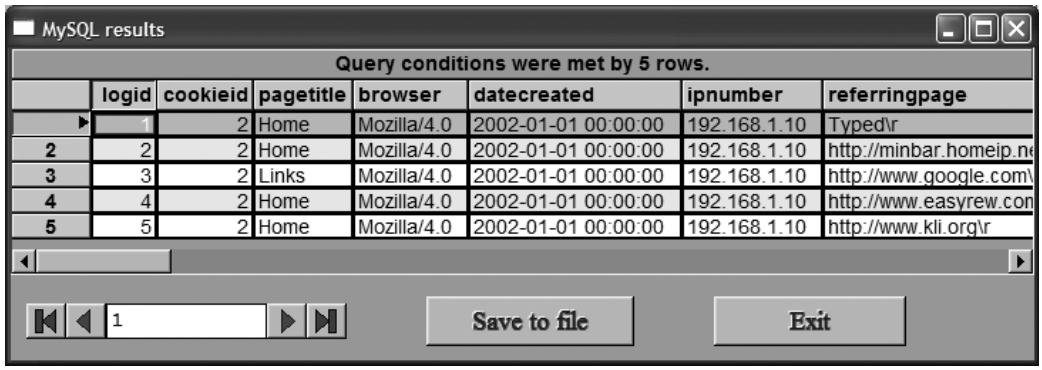

Figure 7.6 The completed equi-join query.

have done the same with the *webpage*'s *title* column so that what we are referring to becomes clearer.

You will also notice that we do not have to return a column that we use in the match conditions. In our example, we used *webpage.id* in the match query but filtered that column out of the actual results by not specifying it in our output column list after the SELECT keyword.

Equi-joins on More Tables

To demonstrate further, you can use an equi-join to inner join more than one table. We will now join the results we obtained above with the cookie table, so that we can get the date that the person viewing the page first looked at our site. This will show if they are visiting for the first time or have come back to look again. To do this we will take the above query and add another column match clause to the WHERE statement, and rename a few columns for clarity:

```
SELECT    log.id as logid,
          webpage.title AS pagetitle,
          log.browser,
          log.datecreated AS logdate,
          log.ipnumber,
          log.referringpage,
          cookies.datecreated AS cookiecreated
FROM      log, webpage, cookies
WHERE     webpage.id = log.webpageid
          AND cookies.cookieid=log.cookieid
```

Figure 7.7 shows this query working. By comparing the datestamp of the *log* column, alias *logdate*, with the cookie creation date, alias *cookiecreated*, we can now see if the viewer is looking at the site for the first time or returning.

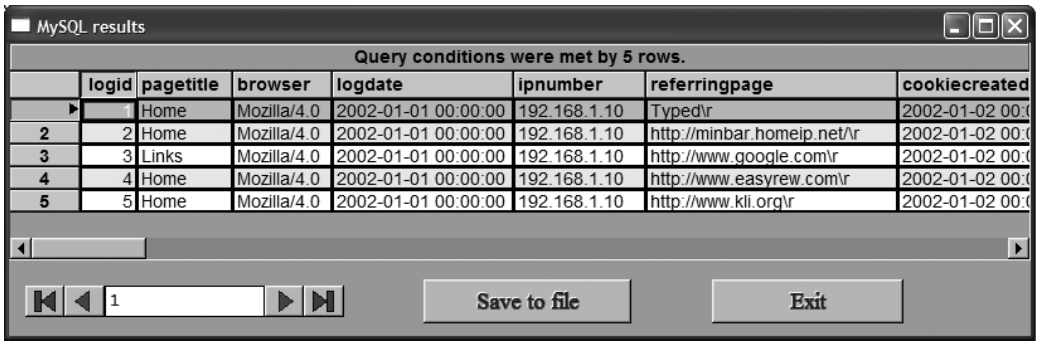

Figure 7.7 An equi-join query with three rows.

Restricting Equi-joins

However, our log table still only has a few rows. In this situation it is easy to look through the result set. What if we had hundreds of rows returned by this query? If this was the case we can just further filter the rows by adding another condition to the end of the WHERE clause:

```
SELECT    log.id AS logid,
          webpage.title AS pagetitle,
          log.browser,
          log.datecreated AS logdate,
          log.ipnumber,
          log.referringpage,
          cookies.datecreated AS cookiecreated
FROM      log, webpage, cookies
WHERE     webpage.id = log.webpageid
          AND cookies.cookieid=log.cookieid
          AND log.webpageid=1
```

Figure 7.8 shows this query running. We have restricted the whole output by looking for the ID of 1 (the main Home page) in the results. In our example this has reduced five rows to only three rows, but we could continue adding clauses to reduce the number further. For instance, we could also restrict by browser or referring page to see the characteristics of people viewing our websites from different links.

INNER JOIN – Another Format

Look again at the end of the last query:

```
WHERE     webpage.id = log.webpageid
            AND cookies.cookieid=log.cookieid
            AND log.webpageid=1
```

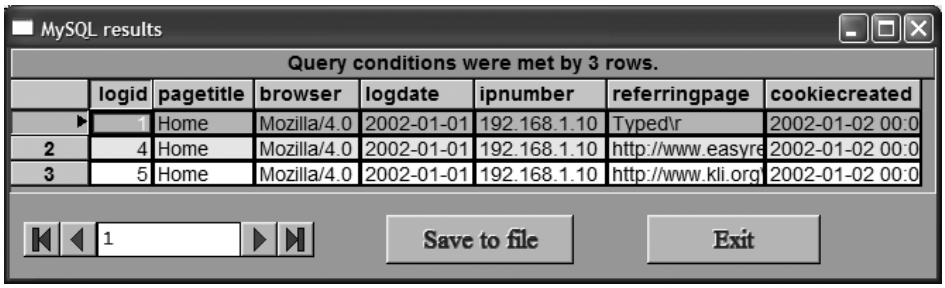

Figure 7.8 An equi-join with restricted rows returned.

You will notice that really there are two different meanings to the three different clauses used. The first two,

```
webpage.id = log.webpageid
AND cookies.cookieid=log.cookieid
```

are used by the two joins to match primary and foreign keys in different tables, whereas the last,

```
log.webpageid=1
```

is just a standard restriction that would work on a normal SELECT statement that did not contain a join. Sometimes this can get confusing so it is useful to know of another format of the inner join that works as follows:

```
SELECT    log.*, webpage.title
FROM      webpage
          INNER JOIN  log ON
                      webpage.id = log.webpageid
```

This query will produce the same results as displayed in Figure 7.3. If we wanted to restrict the rows that we were getting back from the query, in this instance we should not append another clause on the end of the ON condition, we would need to add a WHERE clause as follows:

```
SELECT    log.*, webpage.title
FROM      webpage
          INNER JOIN  log ON
                      webpage.id = log.webpageid
WHERE     log.webpageid = 1
```

Using this format clearly sets a distinction between a restriction WHERE condition and a JOIN condition and so can lead to queries that are easier to read.

On some database systems it is advisable to use the INNER JOIN format rather than the one shown previously, as the DBMS has special code for performing the INNER JOIN that

it only uses if in this format. This may result in your join queries running faster. If nothing else, it save a lot of confusing WHERE clauses!

Outer Joins

The inner join allows you to join two tables which have matching data in certain rows. In an outer join, the whole of one table is returned, along with the matching rows in another table. The first table is returned regardless of whether anything matches with it in the second table. If that sounds confusing to you then don't worry too much. As usual a few examples will make it clearer.

LEFT JOIN

The LEFT JOIN is an outer join that uses the format that we just introduced in the section on INNER JOIN as follows:

```
SELECT    columns
FROM      firsttable
          LEFT JOIN   secondtable ON
                      firsttable.column = secondtable.column
```

To demonstrate the left join, we have to examine the results from the following query:

```
SELECT    webpage.title, log.datecreated
FROM      log, webpage
WHERE     webpage.id = log.webpageid
```

You should now recognize the above as an equi-join. Figure 7.9 shows the results.

What we are trying to display is a list of all of the webpages that we have on our site and a relevant log entry for each page. However, if we look at Figure 7.9 we can see that we have

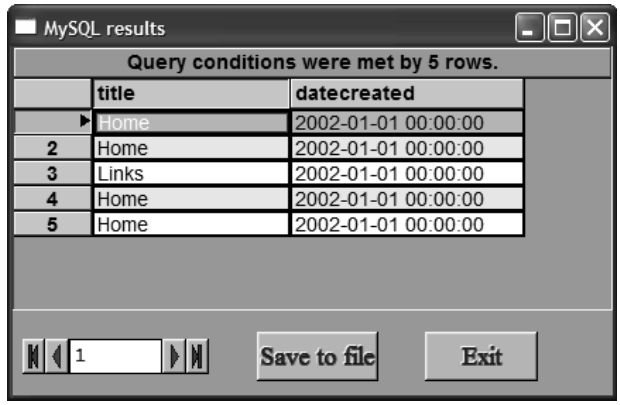

Figure 7.9 An equi-join fails to show all web pages.

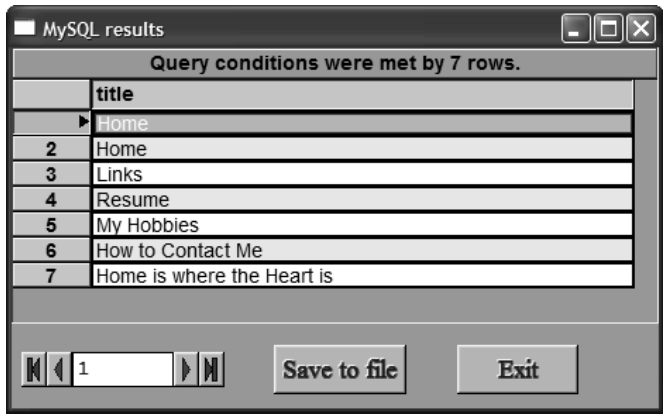

Figure 7.10 All of the entries in the webpages table.

the Home page listed four times and the Links page once. We are only displaying the page title for rows in our log table: the rows that match. Figure 7.10 will remind you of all of the entries in the log table.

So we need to convert this equi-join into a left join to get the desired results. Execute the following query:

```
SELECT    webpage.title, log.datecreated
FROM      webpage
          LEFT JOIN   log ON
                      webpage.id = log.webpageid
```

Look at the results in Figure 7.11. You will see that we still have all of the rows that appeared in Figure 7.9, with the addition of the extra rows in the *webpage* table. As there is no corresponding entry in the last four rows, the MySQL server has returned a NULL for the value in the *datecreated* column.

If we look at the query again, we can see why we call it a left join:

```
SELECT webpage.title, log.datecreated FROM webpage LEFT JOIN log ON ….
```

This has been formatted in a different way so that you can see that the *webpage* table is on the left of the *log* table as it is written. All of the rows in the left table – the *webpage* table – will be returned with the matching rows in the rightmost table. This is a good way to remember which column does which in a left join. The leftmost column in the query will return all its rows irrespective of matches in the right column.

Let us now swap the two columns around in the query, so that we make the log table the leftmost:

```
SELECT    webpage.title, log.datecreated
FROM      log
          LEFT JOIN   webpage ON
                      webpage.id = log.webpageid
```

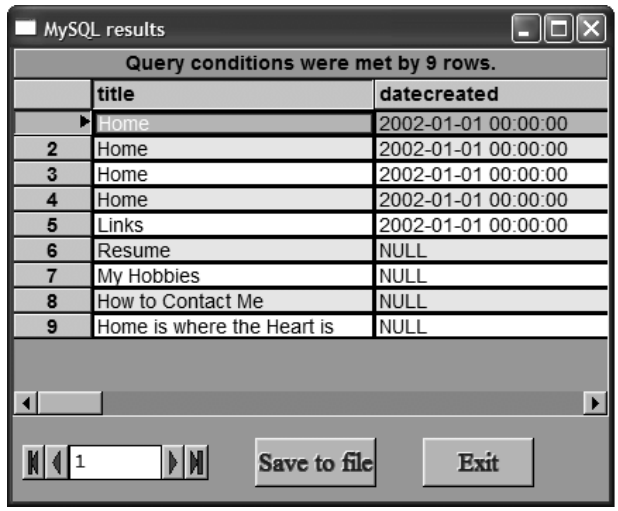

Figure 7.11 A left join.

This time this query is saying: "Show me every title in the webpage table that matches a row in the log table, and all of the rest of the datecreated entries in the log table". Figure 7.12 shows this query running. It actually gives the same result set as the equi-join that we ran at the start of this chapter, but it has been executed as a left join. As every row in the log table must match a row in the webpage table, to maintain referential integrity, no NULLs will appear.

RIGHT JOIN

As we have a left join, it follows that we will also have a right join. The format of a right join is similar to that of a left join, as below:

```
SELECT    columns
FROM      firsttable
          RIGHT JOIN  secondtable ON
                      firsttable.column = secondtable.column
```

This query is saying: "Show me every column in the first table that matches a column in the second table, and all of the rest of the second table entries". We'll run this with our previous example again:

```
SELECT    webpage.title, log.datecreated
FROM      log
          RIGHT JOIN  webpage ON
                      webpage.id = log.webpageid
```

If you have still got the script on the screen, just change the LEFT keyword to RIGHT and run it again.

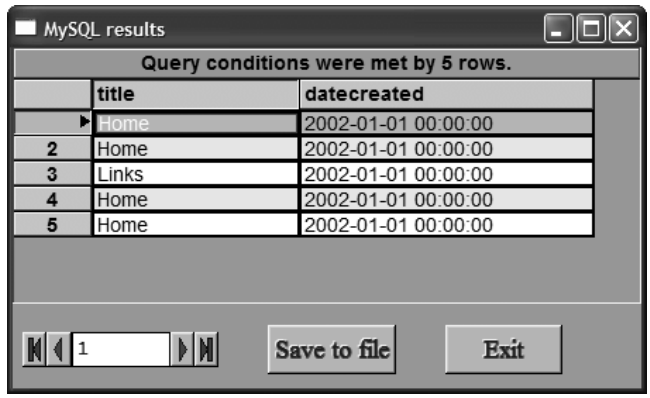

Figure 7.12 Left join swapping the tables' positions.

Again by formatting the query in a line we can see how it works in a clearer way:

```
SELECT webpage.title, log.datecreated FROM log RIGHT JOIN webpage ON ….
```

This time it is the rightmost table, *webpage*, which has all of its rows returned, with only the matching rows in the left table, *log*, in the result set.

If you have just worked through the previous examples, you will notice that running this query produces the same result as the left join shown in Figure 7.11 when the table order is reversed.

This shows an interesting function of the outer join. The following two scripts will produce identical result sets:

```
SELECT    columns
FROM      firsttable
          RIGHT JOIN  secondtable ON
                      firsttable.column = secondtable.column
```

```
SELECT    columns
FROM      secondtable
          LEFT JOIN       firsttable ON
                          firsttable.column = secondtable.column
```

If you understand this, you will realize that there is not actually a need for a system to implement both the left and right join, as you can accomplish both by the re-ordering of table order within the query. MySQL implements both of these joins but some other SQL systems only implement one of them. Though not necessary, the two types of join make query building easier when joining more than two tables with outer joins.

UNION

The UNION keyword allows you to join two result sets together. The result sets must have similar column names. The UNION function is used as follows:

```
SELECT    columns
FROM      tables
WHERE     condition
UNION
SELECT    similarcolumns
FROM      tables
WHERE     condition
```

We can demonstrate this quickly by combining a query that selects all log IDs less than 3 from the log table with another that selects all log IDs that refer to the Home page as follows:

```
SELECT * FROM log WHERE ID < 3
UNION
SELECT * FROM log WHERE webpageID = 1
```

Figure 7.13 shows the results of that query. MySQL has taken the results of both of the queries and joined them into one table, removing the duplicate row. (The two individual queries would both have returned row ID = 1.)

But that quick example does not illustrate the full potential of using the UNION keyword. You may realize that the above query could be re-written with a conventional WHERE clause as follows:

```
SELECT    *
FROM      log
WHERE     ID < 3 OR
          webpageID = 1
```

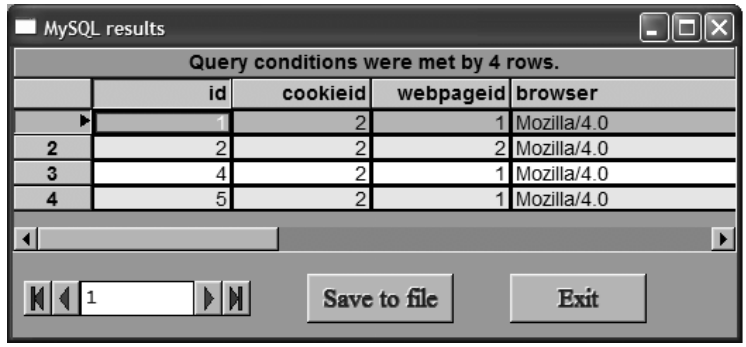

Figure 7.13 Joining two queries with UNION.

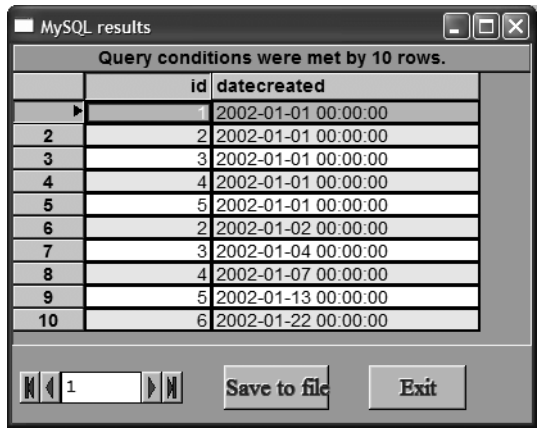

Figure 7.14 Two different tables joined with UNION.

The above can be re-written as it is selecting data from the same table. The beauty of the UNION command is that it can join data from different tables, as long as the columns you are choosing have similar datatypes. For example, both the *cookies* table and the *log* table have ID fields and date fields. How would we combine the ID and the dates from both tables into one results set? You could attempt it using a join as follows:

```
SELECT    log.ID, log.datecreated, cookies.cookieid, cookies.datecreated
FROM      log,cookies
```

However, the above will produce a large result set, as it is an unrestricted join, every row in the log table returned with a row from the cookies table producing 30 rows as a result. We just want the results with the rows from both of the two tables. We will try this query using a union:

```
SELECT    ID, datecreated
FROM log
UNION
SELECT    cookieid, datecreated
FROM      cookies
```

This produces a UNION of the two different tables on the *ID* and *datecreated* columns. The results are shown in Figure 7.14.

If you have been working through all of the examples in this book, your *cookie* table should contain 6 rows and your *log* table 5 rows. However, the result set, shown in the figure, only contains 10 rows. This is because the *datecreated* and the *ID* fields for both tables are the same for ID = 1. MySQL has therefore treated this as duplicate entry and removed one of the duplicates in the result set. As we have never specified the time when creating these fields, MySQL always defaults to 00:00:00. If our examples had used more accurate date/time fields, the difference in the time columns would have stopped this being treated as a duplicate row.