Database Design Manual: using MySQL for Windows

Matthew Norman

Springer

Database Design Manual: using MySQL™ for Windows

Springer
London
Berlin
Heidelberg
New York
Hong Kong
Milan
Paris
Tokyo

Matthew Norman

Database Design Manual: using MySQL for Windows

Springer

Matthew Norman, BSc (Hons)
mysql@coldfusionfast.com

The publisher would like to acknowledge the support of John Cowell, De Monfort University, Leicester, Series Editor of the Essential series, for which this book was originally proposed.

British Library Cataloguing in Publication Data
Norman, Matthew, 1968-
Database design manual : using MySQL for Windows
1. Database design 2. Relational databases 3. SQL (Computer program language)
I. Title
005.7′565
ISBN 1852337168

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licences issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.

ISBN 1-85233-716-8 Springer-Verlag London Berlin Heidelberg
A member of BertelsmannSpringer Science + Business Media GmbH

© Springer-Verlag London Limited 2004

MySQL is a trademark of MySQL AB.

The use of registered names, trademarks etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.

The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

Typeset by Gray Publishing, Tunbridge Wells, UK
Printed and bound in the United States of America
34/3830-543210 Printed on acid-free paper SPIN 10901137

To Rowan, Little Roo and Zed.
I know I'd said I wouldn’t write another one

This page intentionally left blank

Contents

1 MySQL and Databases

About This Book

There are many books out there that describe MySQL and PHP. While most of them devote a lot of space to describing PHP, they only briefly describe MySQL. I wanted to write a book that was different from this, in that I wanted a book that described MySQL from a web perspective with only passing references to PHP. This is that book.

This book will describe the setup and use of a MySQL server and client and how to access, use and secure this system.

MySQL and PHP originally appeared on the Linux platform, so many books deal with it from a Linux perspective. This book will teach you about installing and using MySQL on the Microsoft Windows platform. However, much of the theory behind the software will be of use whatever platform you are using.

About MySQL

The MySQL website describes MySQL as the "world's most popular Open Source database." Its popularity is no doubt encouraged by the fact that if you need MySQL for non-commercial use, you can download a copy free from the website.

MySQL is nearly always bundled with the PHP web scripting language, and the two products are often seen mentioned together. Most Linux distributions come with MySQL and PHP as standard and MySQL has been ported for use to a variety of different platforms. Due to its bundling with PHP, MySQL is most often used as a database back end to a webserver.

Who Can Use This Book?

This book should appeal to anyone who needs more insight into how databases work in relation to the web. If you are completely new to using MySQL then this book will serve as a good introduction to the software. With an internet connection, you can easily obtain MySQL and associated tools at no cost.

This book will also be of benefit to someone wanting to learn SQL. While working through the examples you will get a firm understanding of the concepts behind structured query language and how you use it to communicate with databases.

If you are a web designer this book should also appeal if you want to see how the database can influence your web design. You can see what is going on behind the scenes with a web database.

If you are a hardened database programmer then this book may have a place in your library if you want to see what MySQL does specifically, or how it can interface with other systems.

The book also gives a brief introduction to database design, and shows how an efficiently designed database can improve the performance of MySQL.

Lots of books about SQL and databases start with a lot of theory. The theory normally describes the "relational model" and instils terror into all but the most technical reader! I'm not going to do this here. You do not have to learn about databases by looking at the theory. I want you to learn how to use MySQL, and by association Structured Query Language (SQL). You may pick up a bit of the theory on the way.

Databases and the Internet

As a web designer, databases can be of great use. In the early days of the internet, websites were just static things that required each page to be created manually. This may have been lucrative on an hourly rate but the more pages the site had the less sustainable they were. Imagine trying to mount a sales catalogue on the web; you may have had to create a page for every item in the catalogue. When databases began to get "glued" to websites, it suddenly meant that you could create a standard template page, which retrieved the product detail from the database and created a webpage "on the fly" for each product.

That is not the only use for a web database. As soon as you start to put products on line your users want to buy them! Another database can be used to log their orders, and even to process and pick their orders by the warehouse staff and reorder from suppliers.

Even the web itself requires databases to run. Search engines like Google use incredibly efficient databases that retrieve links to pages for us so simply that we take them for granted.

DBMS

MySQL is what is known as a Database Management System (DBMS). The management system decides how the data is stored, sorted and retrieved, as well as controlling user access to it. Every time a user retrieves data, deletes data, or adds more data, the DBMS deals with the request. The user cannot access the data files directly, he can only talk to the DBMS. The management system is a barrier that controls access to the underlying data.

Figure 1.1 illustrates how a request to the database has to go through the DBMS first, and cannot go directly to the database itself.

MySQL Databases

MySQL can control several databases at once. For instance, when you install MySQL, the system creates the system database which is named **mysql**. This database contains all of the

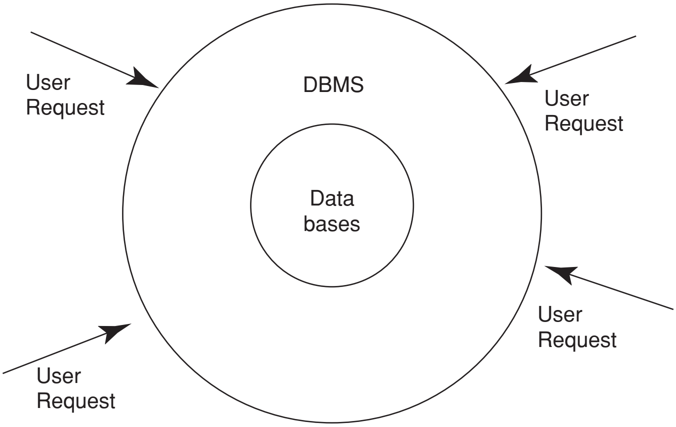


Figure 1.1 DBMS controls access to the underlying data.

data required to define any of the activities that MySQL has to perform. It stores details of other databases, users and all other files that the system uses to store data. It itself is a collection of data used for a specific purpose. This makes MySQL self-defining, in that the tables that it stores are used to define other tables that it stores.

When you are creating your own sets of data, create these in another database. In this book we will be using the **mysql** database to look at specific system functions, but all of the other data that we create will be stored in a database called **mysqlfast**. MySQL can easily control more than one database, so to prevent your tables being confused with system data, it is best to separate them by using different databases.

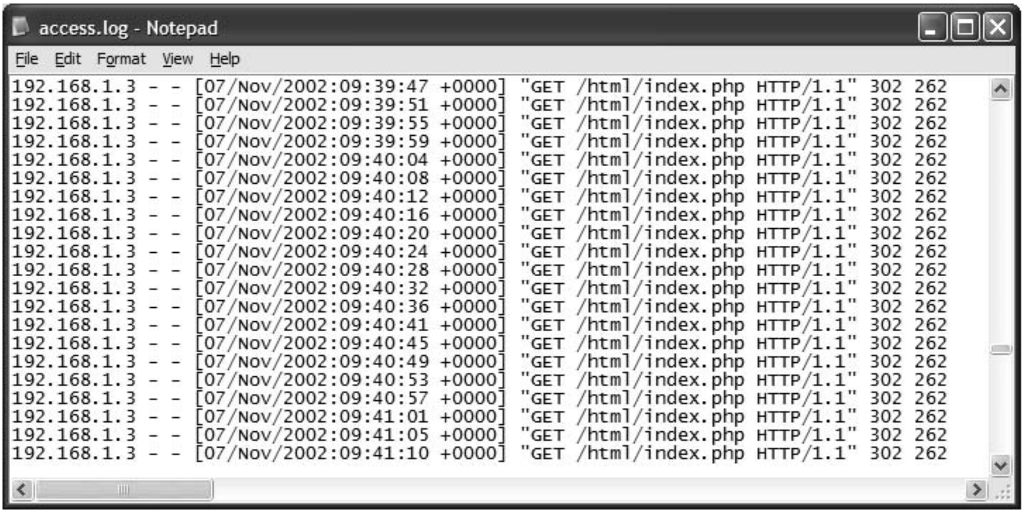
Tables

The database is a container to store your tables in. The table is a collection of data of a similar type. For instance, you could store all of the accesses to a website in one table, called *log*. You may have seen a log file such as the one shown in Figure 1.2 from an Apache webserver.

Figure 1.2 shows an excerpt from a log file. The file, called *access.log*, has a new line added to it every time a request is made to the Apache server. The entire file could be likened to a table within the database. All of the types of data that represent an access to the webserver are stored within this one file, or table, which has a name (access.log). In the same directory that this file is stored in is another file called *errors.log*. This file stores all of the errors that the Apache server logs and is really another table.

MySQL could be used to store all of the log entries and all of the error messages generated by the webserver.

The original database programs allowed you to set up little more than a table. Modern database systems are classed as relational, although there is much argument in the database fraternity as to what is relational and what is not. We will not enter this debate in this book. However, the term relational is useful in describing systems such as MySQL as the DBMS allows you to relate data in tables to data in other tables. In fact, some tables contain



```
192.168.1.3 - - [07/Nov/2002:09:39:47 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:39:51 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:39:55 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:39:59 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:40:04 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:40:08 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:40:12 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:40:16 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:40:20 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:40:24 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:40:28 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:40:32 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:40:36 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:40:41 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:40:45 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:40:49 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:40:53 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:40:57 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:41:01 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:41:05 +0000] "GET /html/index.php HTTP/1.1" 302 262
192.168.1.3 - - [07/Nov/2002:09:41:10 +0000] "GET /html/index.php HTTP/1.1" 302 262
```

Figure 1.2 An Apache access log file.

nothing more than data on how two or more other tables are related. In relational database theory, tables are called relations.

Columns

Any table will have a set of attributes, that is, a list of types of items that each table will store. In this book we will refer to this attribute as a column. In the old-style databases you would have called this a field.

If you look at Figure 1.2 you will see that each column contains a specific type of data. The first column contains the IP address of the machine that is accessing the webserver. The second and third columns appear to store nothing, which is represented as a hyphen character (-). The fourth column contains the date and time that the access took place. The fifth contains the HTTP command that was received and the columns continue.

Each column in a table has a unique name within that table, and that is how we refer to the column. So based on the information that we have from Figure 1.2 we could begin to define the table as follows:

```
Log(ip, accesstime, http_command)
```

Log is the name of the table, and **ip**, **accesstime** and **http_command** are the columns. A table does not need to have data within it to be classed as a table. A table with no content is still a table. This description of the database structure without actual data is known as meta-data. The meta-data also contains details about any restrictions of access to the database as well. You will also notice that each column contains data of a specific type. The first column is a collection of four numbers separated by stops. The next two could be strings, and then the last two columns in the figure are numbers. All data in a column must be of the same type. When you define the column, you specify the type of data within it. The next chapter will describe some of the types of data that you can store in each column.

Rows, Records and Tuples

A row, a record and a tuple are all different ways of referring to a line of data within a table. In this book I will generally use the term row. The tuple is the relational theory term for a row, and a record is the earlier form of databases term.

In Figure 1.2 the following could be described as a single row or record:

```
191.168.1.3 - - [07/Nov/2002:09:39:47 +0000] “GET /html/index.php
HTTP/1.1” 302 262
```

A lot of the data in the log is the same for each row; in fact, in the excerpt we have shown, the only thing that changes is the time. This is not actually a very efficient way of storing this information, as so much of it is static. We will deal with storing this in a better way in a later chapter.

Relations

We have already stated that the relational theory term for a table is a relation. However, tables in relational databases can have relationships between them. Look again at the last but one column of the log file in Figure 1.2. It contains the number 302. This column stores a number that represents the message that the system is logging. For instance, if you request a page that is not on a webserver, you may be familiar with receiving a 404 error. This number represents the error condition: *Not found.* In fact the 302 message represents the condition: *Moved temporarily.* Table 1.1 shows some more of these conditions

Table 1.1 Webserver status codes.

Code	Condition
200	OK
300	Multiple choices
302	Moved temporarily
401	Authorization required
403	Forbidden
404	Not found

Imagine that Table 1.1 was stored as a table in a database. Every time that the webserver added a row to its log table, it could log the status condition for the log entry by adding the string “Moved temporarily”, or it could add the numeric code from Table 1.1 to the log entry instead. Apache uses the latter approach. It includes a code in the log file that can be used to look up another value in another table. These two tables are therefore related to each other. They are related in that an entry in one table is a key to an entry in another.

Most databases claim to be relational ones. There is much debate as to what makes a database relational or not, and many claims that a database system is relational are disputed by others. We will not go into this argument here, but if someone points this out to you, do not be too concerned. Let these debates continue, and just use the software!

Client-server Systems

We have already described how MySQL is a database management system, in that it controls access to the database as well as how the database is physically ordered on the storage medium. You probably are very familiar with the concept of a client-server system. Figure 1.3 shows a simple diagram of one of these. Data is stored on the server, and when the client needs to access that data, it makes a request to the server. On receiving that request, the server takes the necessary action, and sends its response back to the client. The response could be the data, a request for more details, or an error message. One of the most widespread client-server systems is the internet. A remote client, often in another country from the server, makes a request for some form of data over the internet, which the server responds to.

The client and the server do not necessarily have to be on different machines. As you set up MySQL in the course of reading this book, you will set up the MySQL server software and the client software on the same machine. The system is still a client-server system even if it all runs in the same box.

One of the commonest uses of MySQL is as a database behind a website. When it is used in this way there are lots of client-server requests happening. Figure 1.4 illustrates this.

In Figure 1.4 you can see that first of all a user asks a web browser to look at a webpage. This request gets sent to the webserver. If that webpage is built using a scripting language such as PHP, the request gets forwarded to the script processor. If the script processor finds database commands (SQL) in the script it passes them to the DBMS, that is, the MySQL server. MySQL processes the SQL, which is sent back to the script processor as text. This text is used to generate the HTML along with other commands in the scripting language, and this gets sent back to the user in HTML, which finally gets processed by the web browser into a webpage. Although this seems like a complex route, it can happen very quickly, even when the servers are on different machines that the requests and responses have to be forwarded between. Each server is specifically optimized to do its own task quickly, so they all work well together.

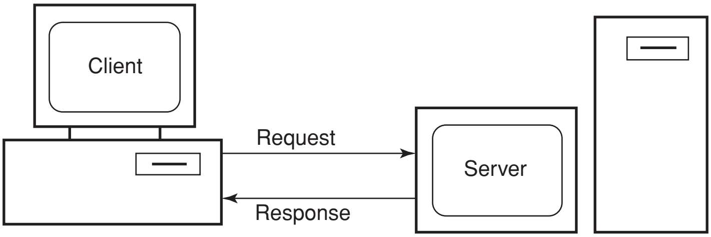


Figure 1.3 Client-server system.

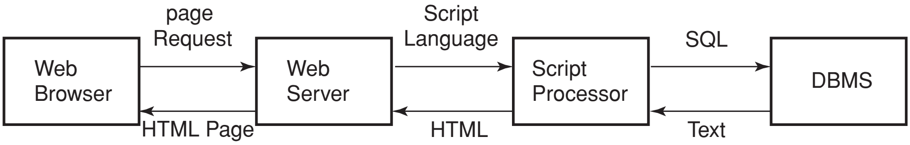


Figure 1.4 Client-server requests.

Structured Query Language

As databases became more prominent, some standard method of obtaining information from them was required. It does not really matter how the DBMS stores data or controls access to it, as long as you get the information that you want from it back in an acceptable way. So a standard way of communicating with database servers was needed.

The standard that most modern databases were built to support was ANSI SQL92. ANSI is the American National Standards Institute which sets the standards for the computer industry. Unsurprisingly, based on the name, the standard was agreed in 1992. It is tribute to the work of this organization that the standard has remained current for such a long time. Only relatively recently have new standards been agreed. SQL-99 or SQL:1999 came along next and is sometimes known as SQL-3. There is also a new standard called SQL:2003 or SQL:200n which is under development.

Most manuals on SQL servers compare their products with the SQL92 standard. I will not attempt to do this with MySQL, as this comparison can be found in the MySQL online manual. However, some of the MySQL commands are in addition to the basic SQL92 commands, and may not be available on other products. Also, because MySQL is an ongoing project, some SQL92 features are not implemented. A lot of the SQL92 commands are still present, but the code to make them work has yet to be developed.

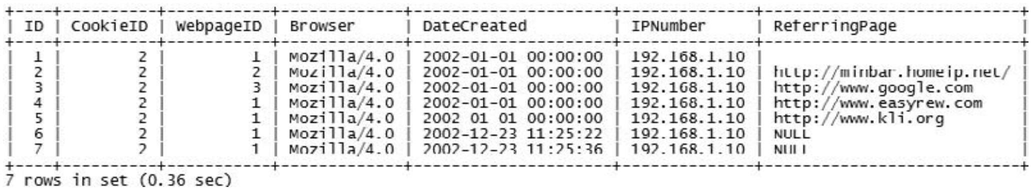
You need not worry about this too much though, and you can still use this book to learn about SQL in general as most of the commands are standard across all SQL platforms. Once you have worked through this book, you should be able to transfer your knowledge of SQL to other DBMS quickly.

Queries

The basic way of communicating with the SQL interpreter is with a query. A query is a piece of text that gets sent to the DBMS which contains instructions that the server can process. The query can contain commands to ask the server for data, to delete data, to create new data and new meta-data. Almost all communication with the DBMS is done through an SQL query. The only real exception to this is when the database server is in such difficulty that it will not function correctly, in which case other tools may be needed to repair the database.

The following is a basic query:

```
SELECT *
FROM log;
```



```
+----+----------+-----------+-------------+---------------------+--------------+---------------------------+
| ID | CookieID | WebpageID | Browser     | DateCreated         | IPNumber     | ReferringPage             |
+----+----------+-----------+-------------+---------------------+--------------+---------------------------+
|  1 |        2 |         1 | Mozilla/4.0 | 2002-01-01 00:00:00 | 192.168.1.10 |                           |
|  2 |        2 |         2 | Mozilla/4.0 | 2002-01-01 00:00:00 | 192.168.1.10 | http://minbar.homeip.net/ |
|  3 |        2 |         3 | Mozilla/4.0 | 2002-01-01 00:00:00 | 192.168.1.10 | http://www.google.com     |
|  4 |        2 |         1 | Mozilla/4.0 | 2002-01-01 00:00:00 | 192.168.1.10 | http://www.easyrew.com    |
|  5 |        2 |         1 | Mozilla/4.0 | 2002-01-01 00:00:00 | 192.168.1.10 | http://www.kli.org        |
|  6 |        2 |         1 | Mozilla/4.0 | 2002-12-23 11:25:22 | 192.168.1.10 | NULL                      |
|  7 |        2 |         1 | Mozilla/4.0 | 2002-12-23 11:25:36 | 192.168.1.10 | NULL                      |
+----+----------+-----------+-------------+---------------------+--------------+---------------------------+
7 rows in set (0.36 sec)
```

Figure 1.5 An output from a query.

The query is made up of a series of words and values terminated with a semicolon. The semicolon alerts the DBMS that it has come to the end of the query. You can normally send multiple queries to the database server by separating each one with a semicolon. In the basic command line MySQL client, the query is only executed once the semicolon is entered.

This query when sent to the DBMS, will cause it to examine the log table, work out all of the rows and columns in that table, and output them to the user, as shown in Figure 1.5.

Figure 1.5 also shows the output from a query. You will notice that the server has neatly tabulated the output into columns and rows for us, and even given a summary of how long the query took and how many rows it returned. The output in the figure is purely text based, as most of the communication between client and server is in this form. You can obtain graphical clients, however, which give you more control over the output. Where applicable, most of the examples within this book will be using a graphical client.

Installing and Testing MySQL 2

Obtaining MySQL

The files that you need to run the MySQL server on your Windows machine are available for download from the following address:

```
http://www.mysql.com/downloads
```

The files that you need to download are the current production release of MySQL (this book uses version 4.1) and the MySQLGUI. Although no longer under development, we use the MySQLGUI as it is a very simple client to use, and you can use it to display your queries with little complexity. The output from MySQLGUI is simple and clear and we use it for most of the figures in this book.

Installing the MySQL Server

You should now have a zip file of the MySQL server on your hard drive. You need to unzip this file to a temporary directory and run the setup program. Depending on the version of MySQL you are installing, you will be presented with a standard setup program similar to that shown in Figure 2.1.

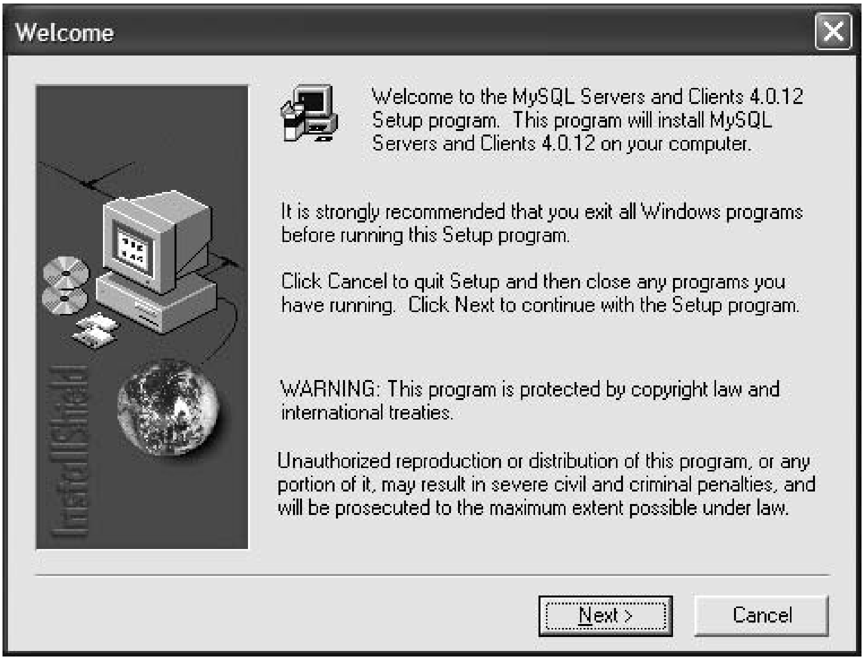
On the following screens, you can leave most of the options at their default. One of note is the installation directory, which is best to keep as:

```
C:\mysql
```

As changing the default directory requires other changes, it is recommended that you let the program install to the default directory to limit configuration errors in later use. This book has been written with the Typical installation selected.

Installing the Graphical Client Tool

The MySQLGUI graphical tools are distributed in a zip file in the same way as the main server. Unzip the MySQLGUI file to a temporary directory. The client tools do not have a



Welcome

Welcome to the MySQL Servers and Clients 4.0.12 Setup program. This program will install MySQL Servers and Clients 4.0.12 on your computer.

It is strongly recommended that you exit all Windows programs before running this Setup program.

Click Cancel to quit Setup and then close any programs you have running. Click Next to continue with the Setup program.

WARNING: This program is protected by copyright law and international treaties.

Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under law.

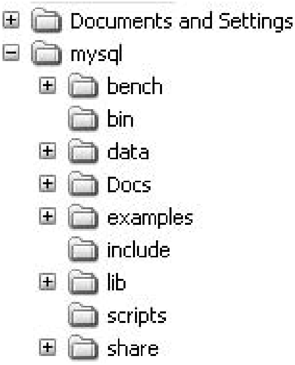
Figure 2.1 Beginning MySQL server setup.

setup program like the main server. All you need to do to run it is to find the *mysqlgui.exe* file and double click when you need to run it.

We can now begin exploring our installation.

MySQL Directory Structure

Figure 2.2 shows the directory structure that the MySQL setup installs.



```
⊞ Documents and Settings
⊟ mysql
    ⊞ bench
      bin
    ⊞ data
    ⊞ Docs
    ⊞ examples
      include
    ⊞ lib
      scripts
    ⊞ share
```

Figure 2.2 The MySQL server directory structure.

Most of these directories are concerned with the function of the server and can be ignored unless you specifically need to use them. The two directories of interest, however, are the:

- **Docs** directory, and the
- **bin** directory.

The **Docs** directory contains the manual and other help files for using the server, and the **bin** directory contains the executable files that you use to start and administer the server. We will now explore these directories.

The Online MySQL Manual

If you need quick online reference about MySQL you can use the manual files that are included in the installation. By default these are installed at:

```
C:\mysql\Docs
```

The files of note in this directory are:

- manual_toc.html
- manual.html

The manual_toc.html file is a web document that indexes the main manual.html file. Searching this usually provides the answer that you are looking for. If not, try searching the actual manual.html file itself. When this book refers to the online MySQL manual, it is referring to these files.

Starting the MySQL Service

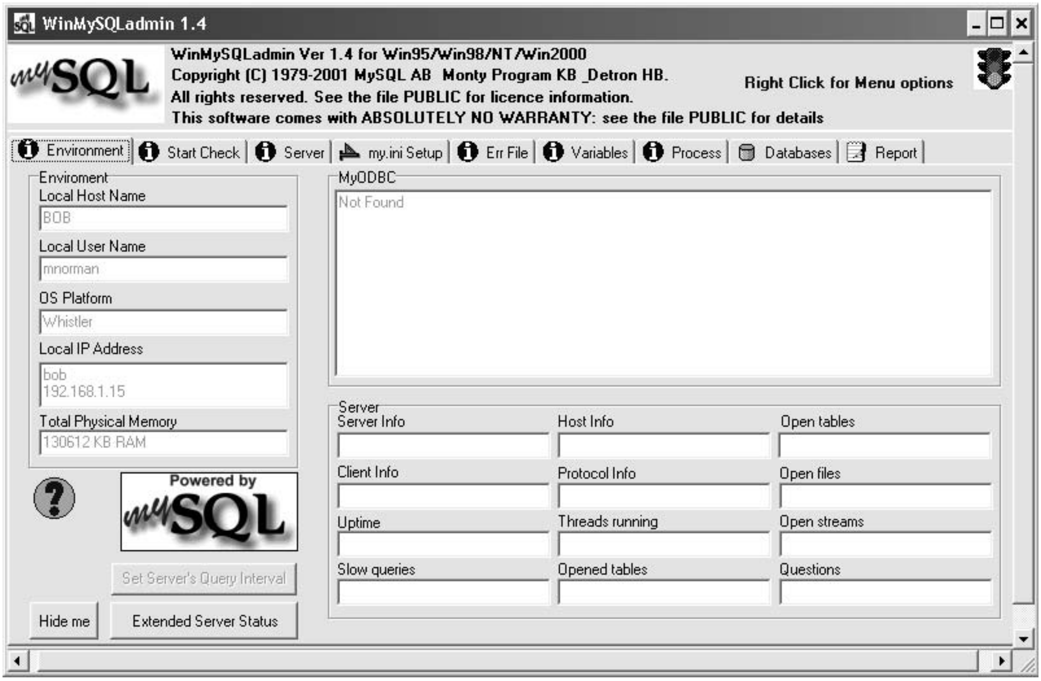
The default the directory where all of MySQL's executable files are stored is

```
C:\mysql\bin
```

To start the MySQL service, double click on the *winmysqladmin.exe* file. Momentarily the screen shown in Figure 2.3 will appear. Whenever the MySQL server is started on your PC this screen will appear for a few moments and then hide itself. We will return to this screen later.

The first time it is run the MySQL service will prompt you for some user credentials for the administrative user. We need to provide a username and password so use the following:

User:	root
Password:	sasquatch



WinMySQLadmin 1.4

WinMySQLadmin Ver 1.4 for Win95/Win98/NT/Win2000
Copyright (C) 1979-2001 MySQL AB Monty Program KB _Detron HB.
All rights reserved. See the file PUBLIC for licence information.
This software comes with ABSOLUTELY NO WARRANTY: see the file PUBLIC for details

Right Click for Menu options

Enviroment
Local Host Name: BOB
Local User Name: mnorman
OS Platform: Whistler
Local IP Address: bob 192.168.1.15
Total Physical Memory: 130612 KB RAM

MyODBC: Not Found

Figure 2.3 The first MySQL admin screen.

If you are not familiar with Linux, root is the standard administrator account on a Linux system. You can use other credentials here, but throughout the rest of the section we will assume that you have used the above. The information that you have typed will be stored in the MySQL start-up file so you will not have to provide this information in this way again unless you reinstall the server or otherwise remove the start-up file.

Depending on your operating system and the version of MySQL that you have used, the preceding process also installs a shortcut to WinMySQLadmin.exe in your start-up group. This ensures that the MySQL server starts up every time you restart your machine. You can also register mysql as a win32 service on certain operating systems. This is done by running the following at the dos prompt:

```
c:\mysql\bin\mysqld-nt —install
```

To verify that the server is running, MySQL puts an icon in the system tray. The icon resembles a traffic control, as shown in Figure 2.4. When the server is active the telltale shows a green light. When the server is loaded but not running a red light will show.

When you run your mouse over the telltale you are presented with a context menu, as shown in Figure 2.5. The **Show me** option will redisplay the screen shown in Figure 2.3 and is the only way that you can get back to this screen without starting another instance of the MySQL server.

One other option will be available to you, which will change depending on your operating system and provides a means of starting and stopping the service.

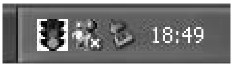


Figure 2.4 The MySQL system tray telltale icon.

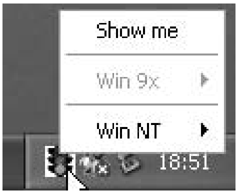


Figure 2.5 The MySQL telltale context menu.

MySQL Configuration Files

Before we test the service, we will use WinMySQLadmin to view the configuration file that it set up for us in the previous section. Run your mouse over the icon in the system tray as in Figure 2.4 and select **Show me**. The WinMySQLadmin form will appear again.

The form shown in Figure 2.3 has a series of tabs that give you access to various settings and information about your MySQL system. Click on the **my.ini Setup** tab. Figure 2.6 shows the startup file that was created for us earlier. It is usually stored in the *C:\windows* directory if you have chosen a standard installation. Any changes you make to this file can be saved by clicking the **Save Modification** button after making the changes.

Notice how the *ini* file contains various parameters, such as the location of the datastore, the IP address of the server, and at the end of the file, the username and password that you just entered.

As everything is now set up and running, we need to test that we can connect to the service.

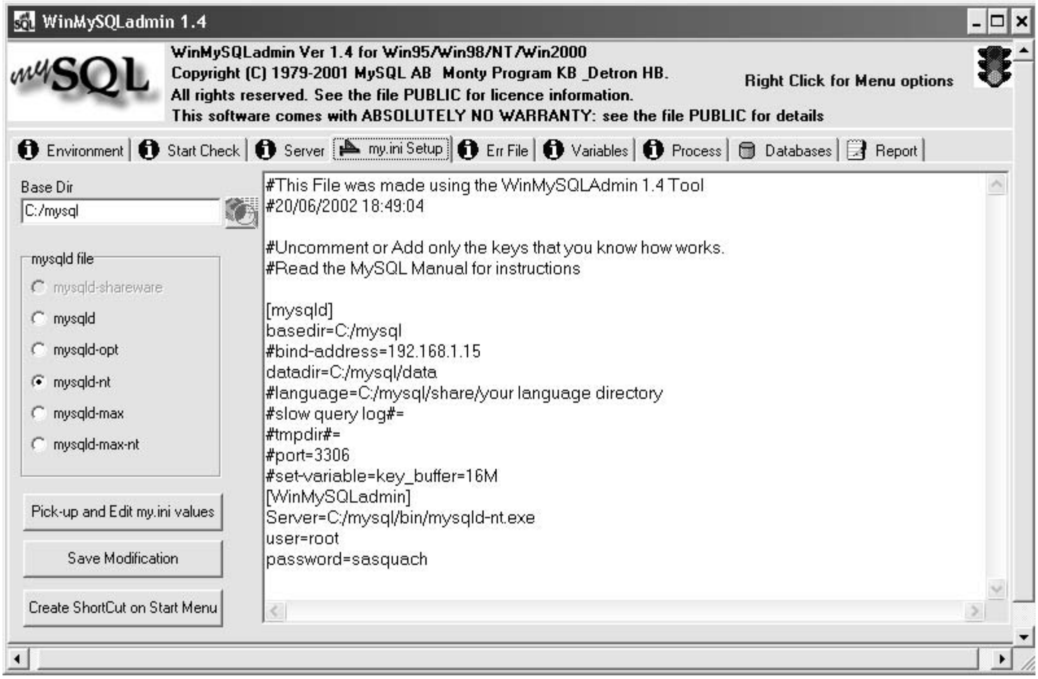
Connecting to the MySQL Service

We will start by connecting to the service in the most basic way, via the command line interface. Start a command box by selecting it from the Start menu or selecting Start/Run and typing command.com. Once you have this running, type:

```
cd \mysql\bin
```

to get you into the directory with the MySQL executables. First we have to run the MySQL command interpreter. Do this by typing:

```
mysql
```



WinMySQLadmin Ver 1.4 for Win95/Win98/NT/Win2000
Copyright (C) 1979-2001 MySQL AB Monty Program KB _Detron HB.
All rights reserved. See the file PUBLIC for licence information.
This software comes with ABSOLUTELY NO WARRANTY: see the file PUBLIC for details

Base Dir: C:/mysql

```
#This File was made using the WinMySQLAdmin 1.4 Tool
#20/06/2002 18:49:04

#Uncomment or Add only the keys that you know how works.
#Read the MySQL Manual for instructions

[mysqld]
basedir=C:/mysql
#bind-address=192.168.1.15
datadir=C:/mysql/data
#language=C:/mysql/share/your language directory
#slow query log#=
#tmpdir#=
#port=3306
#set-variable=key_buffer=16M
[WinMySQLadmin]
Server=C:/mysql/bin/mysqld-nt.exe
user=root
password=sasquach
```

Figure 2.6 Viewing the my.ini file.

You should now be beginning to see the results as shown in Figure 2.7. The next task is to connect to a database within MySQL. MySQL has some system databases built in, so we will use one of these for now. Type:

```
connect mysql
```

You should now be connected to the database, and all that remains is to have a look at one of the tables within the database. Again, there are some built in, and as a final test we will look at the user table. Type:

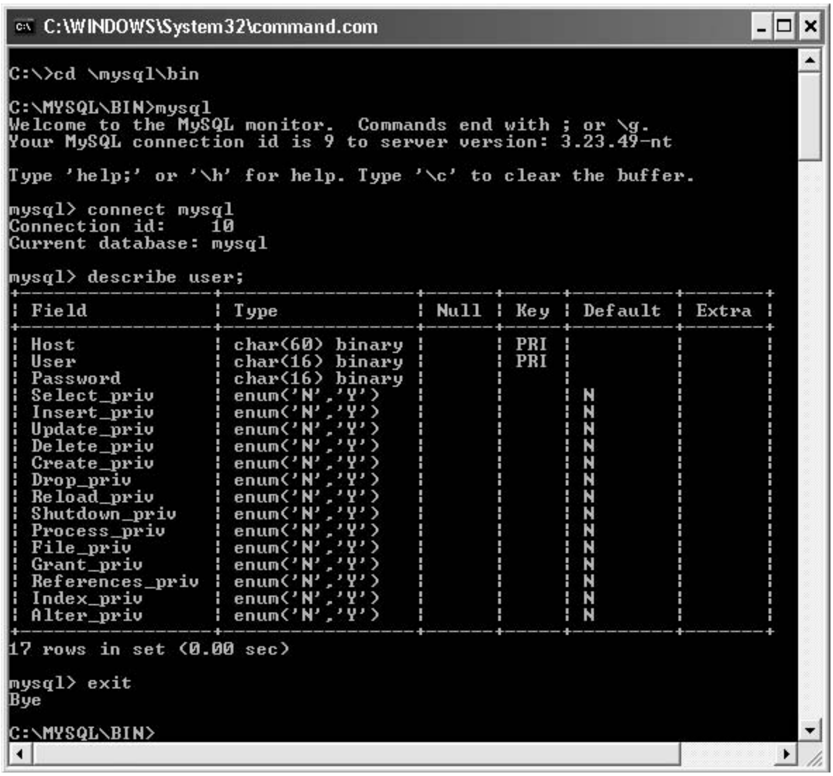
```
describe user;
```

If you have not forgotten to type the semicolon at the end, you will see the table in Figure 2.7. DESCRIBE and a table name will generate a description of each of the columns in the table. The test has worked and you have a running MySQL server!

To come out of the MySQL interpreter, type:

```
exit
```

Although this stops the interpreter, the service will still be running, waiting for the next connection.



```
C:\>cd \mysql\bin

C:\MYSQL\BIN>mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9 to server version: 3.23.49-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> connect mysql
Connection id:    10
Current database: mysql

mysql> describe user;
+-----------------+-----------------+------+-----+---------+-------+
| Field           | Type            | Null | Key | Default | Extra |
+-----------------+-----------------+------+-----+---------+-------+
| Host            | char(60) binary |      | PRI |         |       |
| User            | char(16) binary |      | PRI |         |       |
| Password        | char(16) binary |      |     |         |       |
| Select_priv     | enum('N','Y')   |      |     | N       |       |
| Insert_priv     | enum('N','Y')   |      |     | N       |       |
| Update_priv     | enum('N','Y')   |      |     | N       |       |
| Delete_priv     | enum('N','Y')   |      |     | N       |       |
| Create_priv     | enum('N','Y')   |      |     | N       |       |
| Drop_priv       | enum('N','Y')   |      |     | N       |       |
| Reload_priv     | enum('N','Y')   |      |     | N       |       |
| Shutdown_priv   | enum('N','Y')   |      |     | N       |       |
| Process_priv    | enum('N','Y')   |      |     | N       |       |
| File_priv       | enum('N','Y')   |      |     | N       |       |
| Grant_priv      | enum('N','Y')   |      |     | N       |       |
| References_priv | enum('N','Y')   |      |     | N       |       |
| Index_priv      | enum('N','Y')   |      |     | N       |       |
| Alter_priv      | enum('N','Y')   |      |     | N       |       |
+-----------------+-----------------+------+-----+---------+-------+
17 rows in set (0.00 sec)

mysql> exit
Bye

C:\MYSQL\BIN>
```

Figure 2.7 Testing the MySQL connection by command line.

Connecting with the Graphical Tool

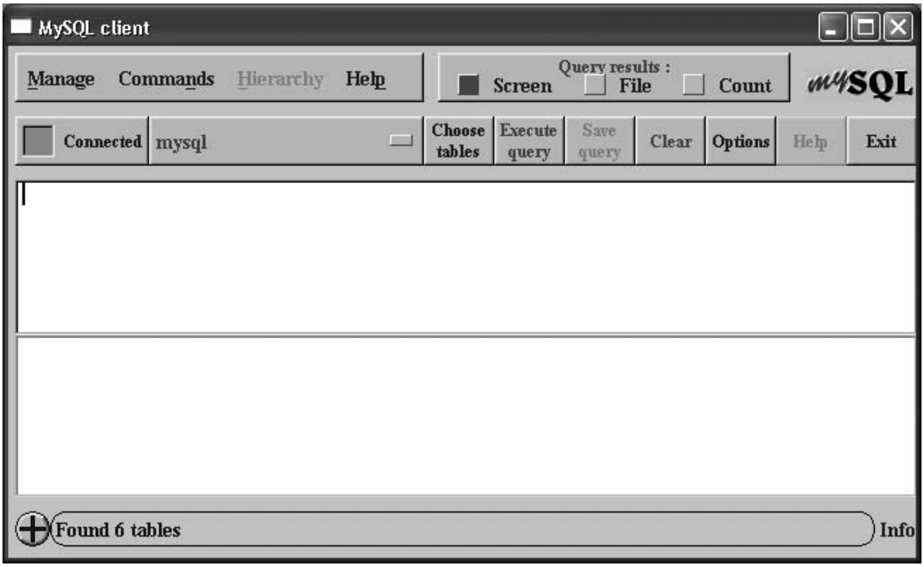
The final test of your system will be to connect to the MySQL server with the graphical tool that we installed earlier. We do this by selecting Run from the Start menu, and typing:

```
c:\mysql\bin\winmysqlgui.exe
```

The above assumes that this is where you copied the file to after unzipping it. If you created a shortcut to the file on your desktop you can just run it by double clicking on that instead.

You will see a splash screen for the product closely followed by a password dialogue. The graphical client assumes that you have already set up the username to log on to the server, and so only prompts you for the password. As we have not done this yet nothing that we type into the password box will log us on at the moment, so just press the **OK** button to bypass this prompt.

Figure 2.8 shows the main screen of the client. Your version will show an error message at the bottom as you have not yet connected to the server. To finish configuring the client,



Found 6 tables

Figure 2.8 WinMySQLGUI main screen.

click on the **Options** button and then on the **Client** tab. You will see a field on this screen called username. Type:

```
root
```

into this box and click the **Save** button. From the Manage menu, select the first option, **Connect**, to bring the password box up again. You will now be able to log in using the password:

```
sasquatch
```

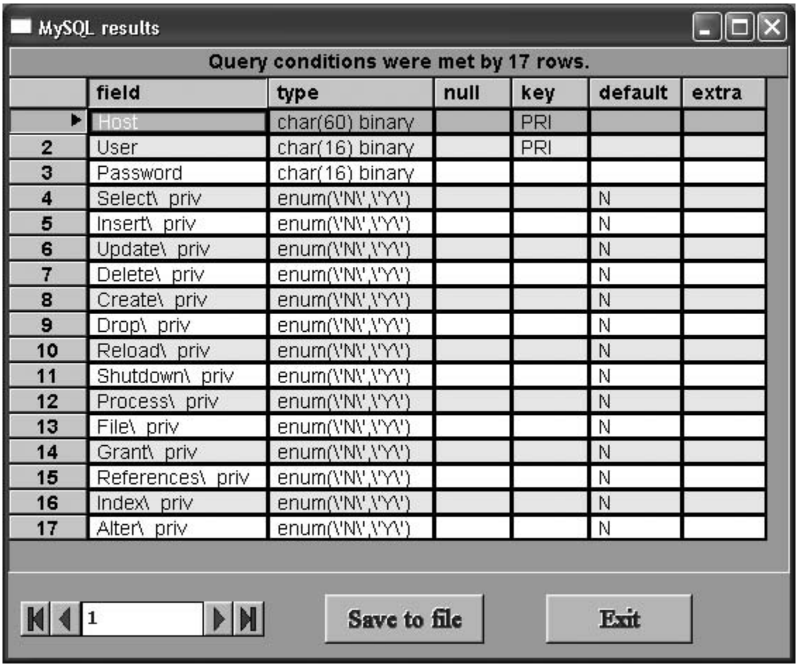
that we defined earlier. The screen should now look exactly the same as Figure 2.8.

You should now be connected to the database. To verify this, we will repeat the exercise that we tested the command line interface with. In the first text field on the screen, type:

```
describe user;
```

Unlike the command interface, purely pressing Return at the end of this command does not make it run. To execute the command you need to press the Query button on the button bar. Press this now. You will be rewarded with the very colourful screen shown in Figure 2.9.

If you compare Figure 2.7 with the output shown in Figure 2.9 you will see that this is displaying the same results, but in a clearer manner.



Query conditions were met by 17 rows.

	field	type	null	key	default	extra
►	Host	char(60) binary		PRI		
2	User	char(16) binary		PRI		
3	Password	char(16) binary				
4	Select\ priv	enum(\'N\',\'Y\')			N	
5	Insert\ priv	enum(\'N\',\'Y\')			N	
6	Update\ priv	enum(\'N\',\'Y\')			N	
7	Delete\ priv	enum(\'N\',\'Y\')			N	
8	Create\ priv	enum(\'N\',\'Y\')			N	
9	Drop\ priv	enum(\'N\',\'Y\')			N	
10	Reload\ priv	enum(\'N\',\'Y\')			N	
11	Shutdown\ priv	enum(\'N\',\'Y\')			N	
12	Process\ priv	enum(\'N\',\'Y\')			N	
13	File\ priv	enum(\'N\',\'Y\')			N	
14	Grant\ priv	enum(\'N\',\'Y\')			N	
15	References\ priv	enum(\'N\',\'Y\')			N	
16	Index\ priv	enum(\'N\',\'Y\')			N	
17	Alter\ priv	enum(\'N\',\'Y\')			N	

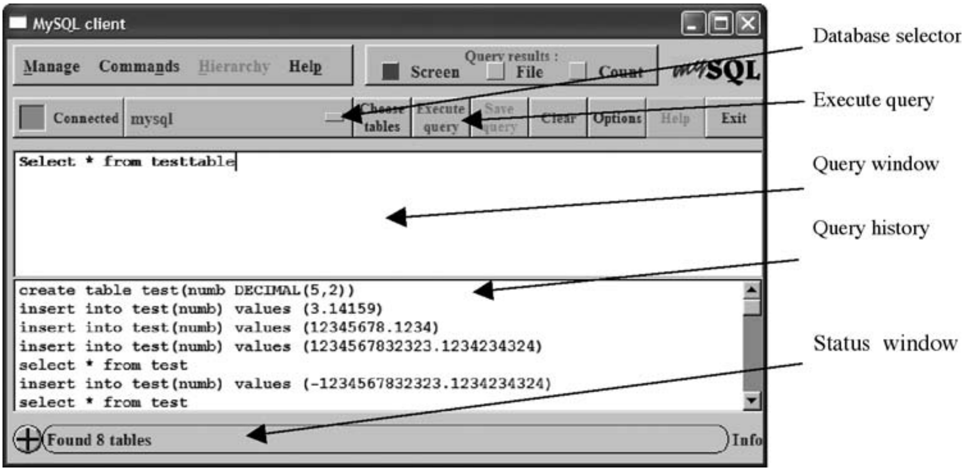
Figure 2.9 The output of the test query.

MySQLGUI Quick Tour

Throughout this book we will just use the MySQLGUI as a windows-based client to issue SQL commands and view the results. Most output from the MySQL server is in tabular form and will be similar to that displayed in Figure 2.9. If there is no tabular output such as error messages, this appears on the main GUI screen along with status messages such as the number of rows affected by an update query. Figure 2.10 shows some other useful parts of the graphical client.

To use the MySQL graphical client:

- Select the required database using the **Database Selector**. This will only contain the system-generated *mysql* database until you create your own. You can also set the default database that the client starts in in the options menu.
- Type the query that you wish to execute into the **Query Window**. You only need to type a semicolon if you are running more than one query at once.
- Once typed, click on the **Execute Query** button to run the query.
- If the query does not trigger another output box, check the text in the **Status Window**. If you have a syntax error or other problem that stops successful execution it will be explained here.



```
Select * from testtable
```

```
create table test(numb DECIMAL(5,2))
insert into test(numb) values (3.14159)
insert into test(numb) values (12345678.1234)
insert into test(numb) values (1234567832323.1234234324)
select * from test
insert into test(numb) values (-1234567832323.1234234324)
select * from test
```

Found 8 tables

Figure 2.10 The output of the test query.

- If your query has successfully executed it will be stored in the **Query History** window. Clicking on a query in the history will copy it back to the query window so you can execute it again. Sometimes you have to click on several of the queries before it copies it to the main window.

By default, the client will try to connect to a MySQL server running on the local machine, *localhost*. If you are connecting to a server that is not on the same machine as the client then you can set this up by clicking the options button and replacing *localhost* with the host name of the machine with the server installed. You will need to ensure that the username that you are using has access rights from your machine, which you can read about in Chapter 12.

3 How Shall We Store it? – Datatypes

The Datatype

All databases store “chunks” of information. The most efficient way of storing those chunks is to group them into similar chunks that can all be stored in a similar way.

This chapter will describe some of the ways that you can store data. You will need these methods to define your tables so that you can begin to store data within them.

I will try to provide some examples in this chapter, but as you need information that you will get later in the book to fully understand what datatypes do, practical examples here are difficult. If you do not understand some of this chapter come back to it after you have run a few queries and created a few tables, and it should all make sense.

When you create a table, as well as deciding what you are going to call different groups of data, you also need to define the type of data that you are going to store within that group. The datatype (sometimes just called a type) is used to define the data that is being stored within the group. It also determines the sort of actions that you can do with that type of data.

There are many datatypes in programming languages, so if you have programmed before you probably will only need a cursory examination of this chapter (or the tables within) to familiarize yourself with MySQL datatypes. If you are new to programming and MySQL, then give this chapter some thought, as storing the data in the correct way will save you hours of re-design and coding later on in your database project.

This chapter will describe four sets of datatypes:

- numeric
- character
- date and
- other types.

The first three are the basic types that most data can be grouped into. MySQL breaks these three types down into further datatypes. We will also describe some other types that have special functions.

When you create a table to store data, when you name each column you also specify the datatype for that column. You specify the datatype using the keywords that we are about to describe. This chapter will just review these types; some examples of their use will be shown when we actually create some tables in the following chapters.

Numeric Types

A numeric datatype is used to store numbers. There are normally specific functions that you can perform on numbers and on most numeric datatypes.

Integers

An integer is an exact whole number. For example, the numbers 1 to 10 are all integers. If we are creating unique IDs for rows in our database, the ID field is usually an integer, which starts at 1 and gets bigger by one for each new row added. If you add, subtract and multiply integers you will get an integer as an answer.

Integers normally range from a negative number, through zero to a positive number. There are several datatypes that are used to store integers, depending on the range; the greater the range, the larger the amount of storage that it takes to keep the integer in a table.

Table 3.1, adapted from the online MySQL manual, shows the approximate range that each of the integer datatypes can represent.

Table 3.1 Integer datatype ranges.

Datatype	From	To	Storage in bytes per row
TINYINT	−128	+127	1
SMALLINT	−32 thousand	+32 thousand	2
MEDIUMINT	−8.3 million	+8.3 million	3
INT	−2 billion (10^{12})	+2 billion	4
BIGINT	−9 trillion (10^{18})	+9 trillion	8

If you look at Table 3.1 you can see that as we move down the table, the range of numbers becomes vast, with little change in the amount of storage that is required. If you are using an integer to store a primary key, the standard INT type will probably suffice if you are not expecting to store more than approximately 2 000 000 000 000 rows!

To use these types when declaring a datatype, they just follow the name of the column, for example:

```
columnname TINYINT
```

or:

```
columnname BIGINT
```

Formatting Integers

By specifying a number in brackets after an integer datatype, you can format the way that that type is returned in queries. For example:

```
columnname INT(10)
```

will specify the number stored padded with up to 10 leading spaces. So if you were to store the number 42 in an INT(10) datatype, the following would be returned:

```
••••••••42
```

Each • in the above represents a space. This is useful when you are outputting your numbers in columns, as it will make all of the column line up neatly (unless the number is bigger in this case than 999 999 999). A similar keyword, ZEROFILL when appended to the end of the integer declaration will make the returned number be padded out with zeros instead of spaces, so this time, the type declaration:

```
columnname INT(10) ZEROFILL
```

will produce the following when 42 is retrieved from it:

```
0000000042
```

These two variations of the datatype do not actually alter the number that is stored within the table; it just changes the formatting of the number when it is retrieved.

UNSIGNED

You will remember from Table 3.1 that the integer datatypes ranged from a negative number to a positive one. If you use the keyword UNSIGNED after declaring an integer, the column defined will only store numbers as positive integers. The range then goes from zero through to double the values in the To column of Table 3.1. For instance, in the following declaration:

```
columnname BIGINT UNSIGNED
```

numbers stored can range from 0 to 18 trillion. This is fine for using integers as unique IDs for rows, but if you start doing arithmetic with signed and unsigned datatypes it can lead to problems. Consult the online MySQL documentation to find out more about this behaviour.

BOOL

A Boolean value is one that can have two states, true or false, on or off, 0 or 1. Because it is such a simple definition, you only need to use one bit to store it, as a single bit can be either on or off.

In MySQL, you might expect BOOL to correspond to a Boolean value, that is a value of 0 or 1 (on or off.) Although BOOL can be used to store these values, it actually is the same as using the TINYINT(1) datatype. If you store a number bigger than 1 in a BOOL, then that number is stored as long as it is not larger than 127. If it is bigger than 127, then 127 is stored. If you are expecting this datatype to be a pure Boolean, it is not. If you refer back to Table 3.1 you will see that TINYINT takes up one byte to store. A byte is 8 bits, so using TINYINT takes eight times more storage than if BOOL was implemented correctly.

You can use the datatype BIT interchangeably with BOOL, they are both the type just described. If you want to store Boolean values it is best just to use TINYINT, and then within your code determine what number you need to store for on or off, 0 or 1. Bear in mind that if you are transferring your MySQL tables to another system that implements BOOL in a different way, the scripts that you need to write to export the data will need to create the actions necessary to reflect that difference. Further development of MySQL may well address this difference in a better way.

DECIMAL

The DECIMAL datatype is used for storing numbers that are not whole numbers, where the numbers after the decimal point are important. For instance, retailers often advertise prices that are a couple of cents or pennies less than the dollar or pound, for example, the dishwasher is $699.95, the computer is £599.99. Although these items cost the best part of $700 and £600 respectively, the fractional part of the number is important. If you stored the dishwasher price as an integer, rounding down, and sold 1000 of these on your website, your accounts would be out by $950. If you rounded the price up to $700, you still would have a discrepancy of $50. The more that you multiply these number, the more the rounding error costs (or makes!) you. Using an integer is therefore not recommended for storing numbers where the decimal precision is important, especially currency. An integer would be fine for storing the number of dishwashers that you have in stock, but not their prices.

MySQL provides the DECIMAL datatype to enable you to store precision fractions of numbers. You declare the DECIMAL datatype as follows:

```
columnname DECIMAL(precision, decimals)
```

- **precision** is the number of digits that are needed to store the complete number. If you are storing negative numbers the minus sign is included in these digits. If you do not store a negative number, a positive number can use this space and so the positive number can be a factor of 10 greater.
- **decimals** is the number of digits to store after the decimal point.

The *precision* part of this is especially confusing, so Table 3.2 shows some examples of the declaration and of what happens when you put different numbers into a column created with that declaration.

Table 3.2 Decimal datatype examples.

Declaration	Number inserted	Number stored
DECIMAL(5, 2)	3.141 59	3.14
DECIMAL(5, 2)	42	42.00
DECIMAL(5, 2)	–23 453.543	–999.99
DECIMAL(5, 2)	4932.32	4932.32
DECIMAL(5, 2)	493 212 343.3423	9999.99
DECIMAL(10, 5)	49 321 234 532.3423	999 999.999 99

The way that you define a decimal datatype may make more sense if you realize that the number is stored as a string. If you are not aware of what a string is, we will get to that in the next section. The decimal declaration defines a certain string length, and all numbers that are stored have to conform within that length, hence the rounding issues when you put in a number that falls out of range into one of these types.

This all may seem very complicated but DECIMAL is the only way that you can accurately store a number with a fractional part. You probably will only use it to store currency, but it is the best way of doing this.

Floating Point Numbers

The datatypes that have previously been mentioned are for storing exact values, either exact whole numbers or exact whole and fractions of numbers. Floating point numbers enable you to store a very wide range of numbers but with some levels of inaccuracy. The bigger the number gets, the less detail is stored about it. The whole topic of storing and dealing with floating point numbers is a large one which this book will not deal with.

I need to warn you now that even though I will not be going into great detail in the following sections, they can get a bit complex. I would not want you to get so confused with this section that you think that MySQL itself is too hard to learn – it is not – but these datatypes are! If you do not intend to store very large numbers in your databases you might just jump ahead to the character types, remembering that this section is here should you ever need to refer to it in a moment of courage later on!

FLOAT

MySQL supports floating point numbers with the FLOAT datatype. You can specify the precision of a floating point number as follows:

```
columnname FLOAT(precision)
```

precision in this instance is the number of digits that are to be stored. If you store up to 24 you are declaring a single precision number and between 25 to 53 it becomes a double precision number.

You can also specify a FLOAT with two attributes as follows:

```
columnname FLOAT(magnitude, decimals)
```

With this type of declaration the float is forced to be of the single precision type. The attributes are as follows:

- **magnitude** is the number of digits used to store the number.
- **decimals** is the number of decimal digits to be stored.

Table 3.3 shows some examples of numbers that have been stored using the FLOAT(5, 2) declaration. You will notice that it stores 3.141 59 and the smaller numbers accurately, with only the number of decimal places being rounded where specified. However, look at the larger numbers at the bottom of the table; as these have gone out of the precision range

Table 3.3 Single precision floats datatype ranges.

Declaration	Number inserted	Number stored
FLOAT(5, 2)	3.141 59	3.14
FLOAT(5, 2)	42	42.00
FLOAT(5, 2)	−23 453.543	−23 453.54
FLOAT(5, 2)	4932.32	4932.32
FLOAT(5, 2)	493 212 343.3423	493 212 352.00
FLOAT(5, 2)	49 321 234 532.3423	493 212 344 432.00

specified, the decimal places have just been set to zero, and the less significant part of the numbers stored is not the same as the number that was entered.

The idea behind this sort of datatype is that when handling very large numbers, the magnitude of the number is more important than the fine details of the number. For instance if you had $10 000 026.51, in the bank, the $10 million would be of more importance to you than the $26.51!

DOUBLE

Another floating point datatype is DOUBLE. This works in exactly the same way as the FLOAT declaration above when used with the two attributes as follows:

```
columnname DOUBLE (magnitude, decimals)
```

This time the number stored will obviously be of the double precision type, but the attributes are the same as follows:

- **magnitude** is the number of digits used to store the number.
- **decimals** is the number of decimal digits to be stored.

In other SQL texts you may see references to REAL and DOUBLE PRECISION datatypes. You can use these as well in MySQL but they are exactly the same as the DOUBLE datatype.

Character Types

The character datatypes are those used for storing characters and strings. The letters in this sentence are all characters, and the sentence itself is a string. In fact as I am sitting here typing this, Microsoft Word is probably thinking that this whole book is just a string of characters, all be it a big one. When specifying a string, you normally surround it with something, such as quote marks, to highlight to the MySQL server that what is inside the quotes should be treated differently to what is outside of them. This is something that we are used to in written text, for instance:

Gandalf spluttered, “Fly, you fools!”

In the above, the narrative is just in straightforward text, but the words that the character is speaking is surrounded with quote marks, so we can tell the difference between the spoken word and the story. MySQL needs to tell the difference between strings and the main syntax of queries so that it does not think that there are errors in the query.

CHAR

The CHAR datatype is used to define a string of fixed length as follows:

```
columnname CHAR(length)
```

length is the width of the string in characters, which can be between 0 and 253.

Once you declare a column with a CHAR value of fixed length, all of the strings that are stored in that column become that length. For example, suppose we used the following as a column declaration:

```
columnname CHAR(10)
```

If we were to store the string "Hello, World" in a column created like the one above, the system would only store "Hello, Wor", the first 10 characters of the string. All of the remaining characters are lost, and no error message is given. If you are storing a string less than 10 characters long, for instance, "Hello!", then MySQL will pad out the string with spaces to make it up to the 10 characters fixed length. So this will be stored as:

```
Hello!••••
```

where each • in the above represents a space. This can be an issue if you are searching CHAR columns. For instance, if you are searching for the word "Hello!" in a CHAR(10) column, nothing would be found, instead you have to search for "Hello!••••" which is what the datatype actually stores. Remember that each string has to be the length that the column declaration requires. We will not deal with this now, but in Chapter 6 there is a way to search for just the start of a string, which would allow you to find a string of less than the defined length of the column.

VARCHAR

The CHAR datatype described above only allows fixed length strings. As you will have seen, this can cause unexpected results in searching if you forget to pad out the search term to the required length of the column. The VARCHAR datatype gets around this problem by allowing a variable length of string up to 255 characters. You define a VARCHAR column as follows:

```
columnname VARCHAR(length)
```

length is the maximum width of the string in characters, which can be between 0 and 253.

Again, if we were to insert "Hello, World!" into this datatype when defined with a length of 10, we would get "Hello, Wor" stored. However, if we inserted the string "Hello!" into it, it would store "Hello!" with no extra spaces. Searching the column for "Hello!" would find the string with no further manipulation needed.

Fixed or Variable Length?

Although variable length datatypes as mentioned above make it easier for searching, using fixed length types have the advantage that they are faster to search. When you are designing your tables, if you want the searches to be faster you should consider using fixed length types.

Variable length types also can produce the need to perform maintenance on the table periodically. If you make lots of changes to a table with variable length rows, over time the table will become fragmented due to entries and space becoming de-allocated and re-allocated. MySQL provides some commands to remedy this which you can read about in Chapter 13 on Optimizing MySQL.

You should consider these issues when choosing the datatypes for your columns.

Storing Text

CHAR and VARCHAR are great for storing strings like people's names and addresses, but they have the limit that the string can be a maximum length of 255 characters. If we are storing things like the comments of a user to a website, or a book review, or other large blocks of text then this limit is prohibitive. The TEXT datatype gets around this limit by defining a string that has a much bigger limit. A TEXT column can contain strings that have a maximum of 65 535 characters. This should cover most things that a webpage user should type into a webpage! If we take the average word to be 5 characters long, that is still over 13 000 words. You define a TEXT column as follows:

```
columnname TEXT
```

It is easy to be tempted to define all columns that are going to contain strings as TEXT. Although this will stop you from running out of space when storing a string, it does place much more strain on the MySQL engine when sorting through lots of TEXT columns. It is much quicker to search through columns of CHAR and VARCHAR for small strings, because of the way that MySQL stores the TEXT datatype. If you want a fast database server, getting the datatypes correct is important.

MySQL also supports three other types of TEXT datatype: TINYTEXT, MEDIUMTEXT and LONGTEXT. Table 3.4 shows the attributes of all of the TEXT datatypes.

Table 3.4 Text datatypes.

Declaration	Maximum length of string in characters
columnname TINYTEXT	255
columnname TEXT	65 535
columnname MEDIUMTEXT	16 777 215
columnname LONGTEXT	4 294 967 295

You could just about store the entire text of this book in one MEDIUMTEXT database record!

BLOB

Don't worry, you are not losing your mind from looking at too many datatypes! A BLOB in MySQL is a Binary Large Object, and is used for storing data. For instance, you could store a jpeg image within a BLOB column. BLOBs conform to the same size constraints as the TEXT columns mentioned in Table 3.4, but are called TINYBLOB, BLOB, MEDIUMBLOB and LONGBLOB. A BLOB column is essentially the same as the corresponding TEXT column, but when you are searching or ordering it the BLOB will be case sensitive and the TEXT will not be.

Both BLOB and TEXT objects are really pointers to areas of storage on the server, and so are not physically stored within the table. This explains why operations on this type of column are slower than when searching normal column types.

Date Types

One of the commonest things that you store within tables are dates and times. An example of this would be a weblog, where every access to a website would have the date and time of that access stored within a text file or a log table in a database.

DATETIME

DATETIME is the date type that I use the most. This allows you to store the date and the time in one column. You define a DATETIME column simply as follows:

```
columnname DATETIME
```

Please refer to Chapter 11 on working with dates and times for details how to insert values into this column type.

When you retrieve a DATETIME value, MySQL returns a string of the format "YYYY-MM-DD HH:MM:SS" irrespective of how you have inserted the date/time in the first place. If you only insert the date part into such a column the HH:MM:SS are set by default to 00:00:00. If you insert a date that is invalid, such as 40 February 2003, MySQL stores this date as "0000-00-00 00:00:00" to show that there has been an error somewhere in the date. You can store values in this datatype for all dates and times between the year 1000 and the year 9999.

TIMESTAMP

TIMESTAMP works in a similar way to DATETIME, in that it stores both the date and time in a single column. However, the TIMESTAMP datatype has the added functionality that it will automatically update itself under certain conditions. This is useful if you want to keep a record of when a database row was created or last updated. Whenever you create a new row, or change the contents of a row, and do not explicitly change the contents of the first TIMESTAMP column in that row, this column will be automatically updated with the date and time of the change.

If, however, you explicitly set the value of the column when you are altering or creating the row, the TIMESTAMP column will not auto-update because you have specified what you want to go in it manually.

This can be a very useful column datatype as it will update itself without the need for you to create code that will update it for you.

You define a TIMESTAMP column with a value, as follows:

```
columnname TIMESTAMP(length)
```

length refers to the number of characters that the column will output when retrieving its value. For instance, of we were to store 28 February 2003, 01:50:59 in a TIMESTAMP(14) column, we would obtain the following if we were to retrieve this value at a later time:

```
20030228015059
```

Table 3.5 shows the other values that would be returned if we were to store the same date in timestamp columns declared with other lengths.

Table 3.5 TIMESTAMP datatype values.

Declaration	DateTime returned
columnname TIMESTAMP(2)	02 (YY)
columnname TIMESTAMP(4)	0202 (YYMM)
columnname TIMESTAMP(6)	020228 (YYMMDD)
columnname TIMESTAMP(8)	20020228 (YYYYMMDD)
columnname TIMESTAMP(10)	0202280150 (YYMMDDHHMM)
columnname TIMESTAMP(12)	020228015059 (YYMMDDHHMMSS)
columnname TIMESTAMP(14)	20020228015059 (YYYYMMDDHHMMSS)

The length attribute in the TIMESTAMP declaration only alters the size of the string that you get back when you retrieve the data. A TIMESTAMP row takes exactly the same storage space irrespective of the length attribute specified when you declare it.

DATE

The DATE datatype is used when you are concerned about storing a date but do not need to store the time as well. The DATE datatype is defined with no attributes as follows:

```
columnname DATE
```

You can insert a date into a date column using any of the methods described in Chapter 11, however the DATE datatype does have a limited range. Dates can be stored from 1 January 1000 through to 31 December 9999. That range should be adequate for most of us!

TIME

The TIME datatype is used to store a time value or period. It can range from –838 hours to 838 hours. This enables you not only to store the time of day, but also to store a long length of time in this column type.

A TIME column is declared without attributes as follows:

```
columnname TIME
```

Normally the output of the time column would be "HH:MM:SS", however when you are storing an hour value of over 99 hours the format will allow an extra hour digit to allow the measurement of this longer time period.

YEAR

The YEAR datatype allows you to store a year with very limited storage space. You specify a year column by using:

```
columnname YEAR
```

The YEAR that you store in this type of column can range from 1901 to 2153. The small range is due to the YEAR being stored in a single byte. The best way of inserting a date into this column is by specifying a string with 4 digits, for example:

```
1952
2010
2063
```

If you only use two digits, MySQL has to make a decision about what year you are referring to. For more information about how it chooses you can consult the section in the online MySQL manual. MySQL will always output the contents of the YEAR datatype with 4 digits to avoid confusion.

Other Types

All of the previous datatypes will have been used to store data that you are familiar with, like dates, times and numbers. MySQL also supports other types that you may be less familiar with. Two of these are described below.

SET

A set is a collection of things that are unique, the order of which is not important. For instance, the following is a set:

- Spoon
- Fork
- Knife

That set is exactly the same as the following set:

- Knife
- Fork
- Spoon

It does not matter if the elements of the set contain the items in a different order; if all of the elements are the same then the set is the same, irrespective of the order. However, the following is not a set as it contains a duplicate:

- Knife
- Spoon
- Fork
- Knife

The SET datatype in MySQL is used for storing elements of a set in each row of the table. The elements of the set are defined at the time that the column is declared and are stored as strings in the declaration. You can have a maximum of 64 elements in a MySQL set. To store occurrences of the set above you would use the following:

```
columnname SET(“Knife”,“Fork”,“Spoon”)
```

What this datatype actually does, is to allow each row of the column to store the following values:

- “”
- “Knife”
- “Fork”
- “Fork,Knife”
- “Spoon,Fork,Knife”
- “Spoon” etc.

The first item on the list above contains no elements, it is referred to as “the empty set” and it is still a valid element of the defined set. If you try and insert a string that is not within the set definition, then that is ignored. If you are adding more than one element to a set, then you should include them all in the same string, separated by commas.

SET is useful for storing the responses from an online questionnaire or anything that requires selections from a predefined list. This is useful as although when you retrieve entries from a SET column you get them as strings, they are actually stored as numeric values, thus saving storage space in the database.

ENUM

The MySQL ENUM datatype works in a similar way to the SET type described above. The ENUM type allows each row of its column to have a value chosen from a set list which you define at column creation time. Unlike a SET, which can have multiple elements of the set stored in each row, ENUM allows only one or none at all. SET also has a limit of 64 elements, but ENUM can have a maximum of 65 535 elements. You define an ENUM datatype as follows:

```
columnname ENUM("Knife","Fork","Spoon")
```

You will notice that this is very similar to the SET declaration. Each of the rows in a column defined as above could therefore contain only one of the following:

- “”
- “Knife”
- “Fork”
- “Spoon”

Again, this does not actually insert the string into the column, it inserts an index for the text based on the order that the strings were declared when the column was created. So if you were to insert “Fork” into a row, then the actual data stored would be 2, as this was the second element that was in the list when the column was defined. However, if you retrieve the data, MySQL realizes that the column is an ENUM type and so gives you the string “Fork” back.

This datatype is especially useful if you are storing the responses from static dropdown lists on webpages. As long as the contents of the dropdown list does not change, ENUM is the perfect way to store records of what has been selected.

This page intentionally left blank

4 Designing and Creating Tables

Redundant Data

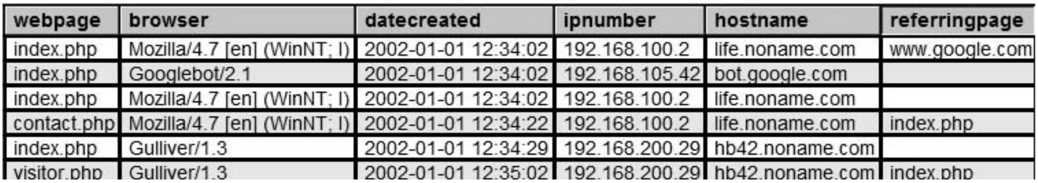
No matter how fast the database server is, it can only work well if it is processing efficiently organized data. Much of the speed and efficiency of modern database engines relies on the use of well-designed tables that are quick to process. One of the ways in which modern DBMS have improved the performance of databases is by eliminating redundant data.

Redundant data is data that is repeated unnecessarily within a table. Figure 4.1 shows a table that stores a record of accesses to a website. The client host details have been changed for anonymity.

Look at each column in Figure 4.1. Every column has data in it that is repeated. For instance, the webpage column has only three unique pages listed in it, and the index.php page is repeated in this column four times. Now look at the browser column. Not only is the Mozilla entry repeated three times it is also storing a lot of text for each row. Although lots of data is repeated, it is not necessarily all redundant.

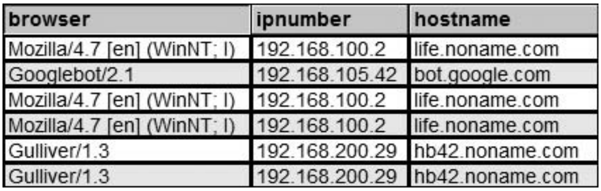
Data is classed as redundant if it can be removed from a table without loss of information. For instance, look at the extract from this table shown in Figure 4.2.

When viewed in isolation from the main table shown in Figure 4.1, the data shown in Figure 4.2 has more obvious redundancies. You will notice that in the six rows shown, only three of these rows are unique. When a remote machine accesses a webserver, it does this via its web browser software, using its IP number which is matched (usually) to its domain name. While that browser is looking at pages on your site, normally these three pieces of information remain the same for each page request. Only the page that they are looking at, the referring page and the time that they made the request would change.



webpage	browser	datecreated	ipnumber	hostname	referringpage
index.php	Mozilla/4.7 [en] (WinNT; I)	2002-01-01 12:34:02	192.168.100.2	life.noname.com	www.google.com
index.php	Googlebot/2.1	2002-01-01 12:34:02	192.168.105.42	bot.google.com	
index.php	Mozilla/4.7 [en] (WinNT; I)	2002-01-01 12:34:02	192.168.100.2	life.noname.com	
contact.php	Mozilla/4.7 [en] (WinNT; I)	2002-01-01 12:34:22	192.168.100.2	life.noname.com	index.php
index.php	Gulliver/1.3	2002-01-01 12:34:29	192.168.200.29	hb42.noname.com	
visitor.php	Gulliver/1.3	2002-01-01 12:35:02	192.168.200.29	hb42.noname.com	index.php

Figure 4.1 Example website log.



browser	ipnumber	hostname
Mozilla/4.7 [en] (WinNT; I)	192.168.100.2	life.noname.com
Googlebot/2.1	192.168.105.42	bot.google.com
Mozilla/4.7 [en] (WinNT; I)	192.168.100.2	life.noname.com
Mozilla/4.7 [en] (WinNT; I)	192.168.100.2	life.noname.com
Gulliver/1.3	192.168.200.29	hb42.noname.com
Gulliver/1.3	192.168.200.29	hb42.noname.com

Figure 4.2 Extract from Figure 4.1.

Reducing Redundant Data

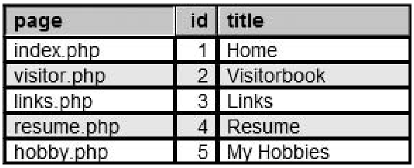
You may think that with the size and speed of modern hard drives there is little point worrying about storing too much text for each row. To an extent that is right, however it is not just the size of the database that may become an issue. The bigger the database is the longer it will take to utilize the data. If you are implementing a page counter on your webpage, and the whole of the log database is going to be counted every time a webpage is viewed, having a huge log table may well slow the site down considerably. Each time a page is viewed, the table gets larger, and so the site gets slower.

With the exception of the datecreated column, all of the other columns could have redundant data removed. Take for example the webpage column. This will store the name of the webpage that the log entry refers to. You will note that we immediately have a duplicate entry in the first two rows. Every time that the index page of the website is viewed, the current table will have the text "index.php" inserted into a new row.

To stop repeatedly storing redundant data, we need to move the text description of the webpage into another table, and find some way of relating the two entries to each other. Figure 4.3 shows what could be stored in the second table.

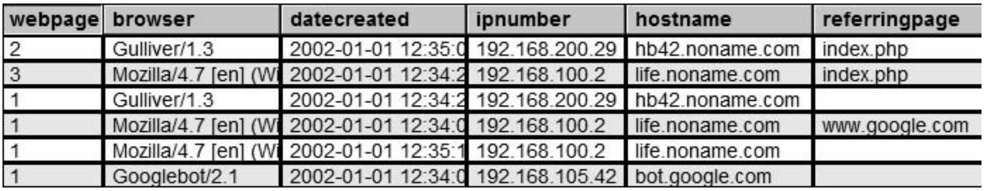
The Primary Key

If you examine Figure 4.3 you will see that we have created a new table which stores information about the webpage. We will call this the *webpage* table. It contains the name of the page, as well as the title, but it also contains a row of unique numbers, the ID column. This is known as the primary key. Each row in the webpage table can be identified by using its key. For example, if we were to see mention of ID 3 of the webpage table, we would know that it was referring to the *links.php* script, which is the Links page. This unique reference to the row in a table greatly reduces the chances of redundant data within your database tables. Figure 4.4 shows the website log table again, but this time we have changed the first webpage column.



page	id	title
index.php	1	Home
visitor.php	2	Visitorbook
links.php	3	Links
resume.php	4	Resume
hobby.php	5	My Hobbies

Figure 4.3 Example webpage table.



webpage	browser	datecreated	ipnumber	hostname	referringpage
2	Gulliver/1.3	2002-01-01 12:35:0	192.168.200.29	hb42.noname.com	index.php
3	Mozilla/4.7 [en] (Wi	2002-01-01 12:34:2	192.168.100.2	life.noname.com	index.php
1	Gulliver/1.3	2002-01-01 12:34:2	192.168.200.29	hb42.noname.com	
1	Mozilla/4.7 [en] (Wi	2002-01-01 12:34:0	192.168.100.2	life.noname.com	www.google.com
1	Mozilla/4.7 [en] (Wi	2002-01-01 12:35:1	192.168.100.2	life.noname.com	
1	Googlebot/2.1	2002-01-01 12:34:0	192.168.105.42	bot.google.com	

Figure 4.4 Website log with reduced redundant data.

Foreign Keys

In the example shown in Figure 4.4, instead of inserting a string to determine the webpage as we did in Figure 4.1, we have inserted the primary key from the webpage table in Figure 4.3. When you insert a primary key from one table into another, it is known as a *foreign key*. By using the foreign key to look up the row in the webpage table, we can get back more information about the row we are examining in the website table. We are able to recreate all of the information, without storing anything unnecessarily.

Figure 4.4 still has redundant data, though, so we will follow this example through by creating a cookie table. The cookie table will store the host information that we get from the site visitor. This table is shown in Figure 4.5.

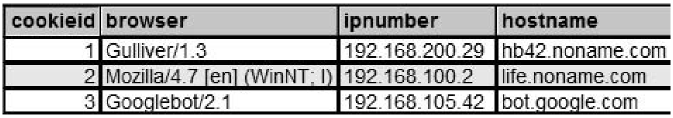
You can see now that Figure 4.5 contains the unique information that we got from each of the visitors that have viewed our webpages, and each of these visitors has a unique number, which we have conveniently called the *cookieID*. This is the primary key in the cookie table. The scripting language that we are using can use this number to set a cookie on the visitor's machine as well for future visits. We can now insert the cookieID as a foreign key into our weblog table, which we show again in Figure 4.6.

You will notice from Figure 4.6 that the table now contains much less redundant data. We still, however, have retained the ability to recreate that data, by looking up the foreign keys in the related tables. For instance, the first row of data in Figure 4.6 shows the following information:

- The date and time the page was viewed (**datecreated** – 2002-01-01 12:35:02).
- The page that linked to this page (**referringpage** – index.php).
- The cookie number of the client machine (**cookieID** – 1).
- The webpage number that was viewed (**webpage** – 2).

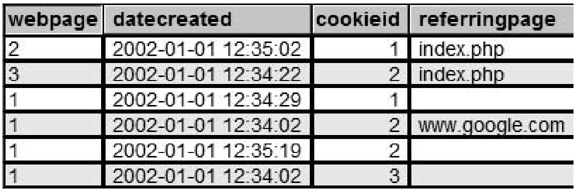
But by matching the foreign keys that we have stored with the rows in their parent tables, we can also ascertain:

- The browser that the client used (**cookie.browser** – Gulliver/1.3).



cookieid	browser	ipnumber	hostname
1	Gulliver/1.3	192.168.200.29	hb42.noname.com
2	Mozilla/4.7 [en] (WinNT; I)	192.168.100.2	life.noname.com
3	Googlebot/2.1	192.168.105.42	bot.google.com

Figure 4.5 Cookie table.



webpage	datecreated	cookieid	referringpage
2	2002-01-01 12:35:02	1	index.php
3	2002-01-01 12:34:22	2	index.php
1	2002-01-01 12:34:29	1	
1	2002-01-01 12:34:02	2	www.google.com
1	2002-01-01 12:35:19	2	
1	2002-01-01 12:34:02	3	

Figure 4.6 Weblog with even less redundant data.

- The client's host name (**cookie.hostname** – hb42.noname.com).
- The client's IP number (**cookie.ipnumber** – 192.168.200.29).
- The script name of the page viewed (**webpage.page** – visitor.php).
- The title of the page viewed (**webpage.title** – VisitorBook).

We refer to columns in external tables by giving the table name followed by a period and then the column name. So you can see from the above example that we have actually stored the same amount of information, without actually storing any data unnecessarily. When we want to recreate all of the stored data across several tables, we use an SQL join command. This will be explained in a later chapter.

Redundant or Not?

Look at Figure 4.6 again. Is there still redundant data in this table? In the cookie column, the first line and the third line contain the *cookieID* of 1. If we were to remove this value, as it is repeated, we would lose the link to the cookie table, and thus lose some information about the page view. When a foreign key is repeated within a column, it usually does not signify redundant data. So there is no redundant data within the *cookieID* column now.

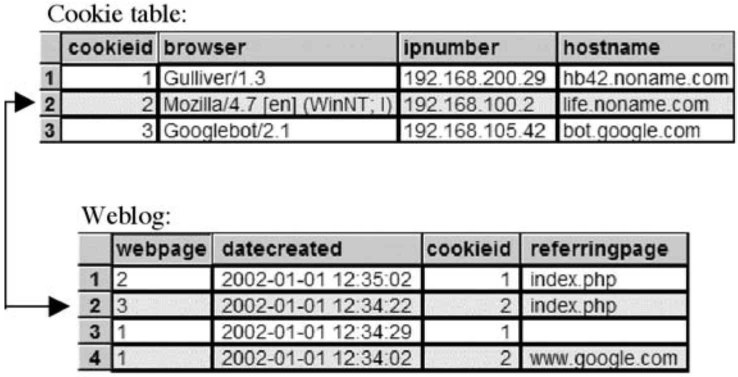
Of course, you may notice that Figure 4.6 still has some redundant data in it, as there are still repetitions in the referring page column. However, no other information about the referring page needs to be stored, so creating another table just to log this may well be an unnecessary overhead. On this occasion we will leave this data as it is and not store it in another table.

Referential Integrity

Creating relationships between tables using primary and foreign keys to remove redundant data is reasonably straightforward. However, when you wish to alter or delete records in related tables you can come across problems with referential integrity.

For example, look at the two tables shown in Figure 4.7. If we were to remove row 2 from the *weblog* table, this would be allowable as no other table in our example makes reference to this row. The referential integrity of our data is maintained. We are removing some of the data, but the rest of the database remains sound.

If we were to remove row 2 from the *cookie* table we have a problem. We would lose some information about the Mozilla client, but what about the rest of the database? In this case



Cookie table:

	cookieid	browser	ipnumber	hostname
1	1	Gulliver/1.3	192.168.200.29	hb42.noname.com
2	2	Mozilla/4.7 [en] (WinNT; I)	192.168.100.2	life.noname.com
3	3	Googlebot/2.1	192.168.105.42	bot.google.com

Weblog:

	webpage	datecreated	cookieid	referringpage
1	2	2002-01-01 12:35:02	1	index.php
2	3	2002-01-01 12:34:22	2	index.php
3	1	2002-01-01 12:34:29	1	
4	1	2002-01-01 12:34:02	2	www.google.com

Figure 4.7 References must remain consistent.

row 2 of the weblog table still refers to row 2 of the cookie table which we have just removed. If we tried to join the two tables together to get back all of the information stored between them, the system would not be able to find the reference to the Mozilla cookie, so the join would fail. Removal of row 2 from the cookie table has compromised the referential integrity of the rest of the database. The only way that we could remove row 2 of the cookie table is if we removed every row that contained a reference to it in all related tables. We have to remove all of the foreign key references to this row first, and then remove the primary key from the cookie table. There is a specific type of command used for deleting data while maintaining referential integrity called a CASCADE delete. This will be touched upon in a later chapter.

Most DBMS will try to maintain and enforce referential integrity, but this will only work if you have correctly told the system where the relationships between tables are. If you are using a scripting language such as PHP, it will be possible to enforce referential integrity in your code, and not with the database. However, this will only work if your code works, and so it is much better to let the DBMS enforce the rules.

MySQL will only enforce referential integrity if you are using a specific type of table. The default table on most installations will be of the MyISAM type. This type does not enforce referential integrity. If you are creating tables which need referential integrity enforced, you will need to create InnoDB tables. The online MySQL manual contains a section on ensuring that your MySQL server supports these tables.

NULL

Finally, before I show you how to create a table, we need to deal with NULL. A NULL entry is placed in a column when there is no value stored in that column. The NULL is a special symbol which signifies that no value has been inserted yet. A NULL is not like the empty string (“”), because the empty string is a string with no contents. NULL is the absence of content. Because of this you cannot match a NULL in comparisons.

When creating a table, you can specify that a column cannot contain a NULL by using NOT NULL after the column type. If you are specifying a column that is a foreign key, the

NOT NULL command implies that the row *must* have a relationship with a row in another table.

Your primary key column would normally also be specified as NOT NULL, as each value in this column must be unique.

CREATE DATABASE

Before we create a table we have to create a database. The database is the container in which we store the tables. You create the database using the following command:

```
CREATE DATABASE databasename
```

Once you have created the database you have to indicate to MySQL which database you will be using. You do this by means of the following:

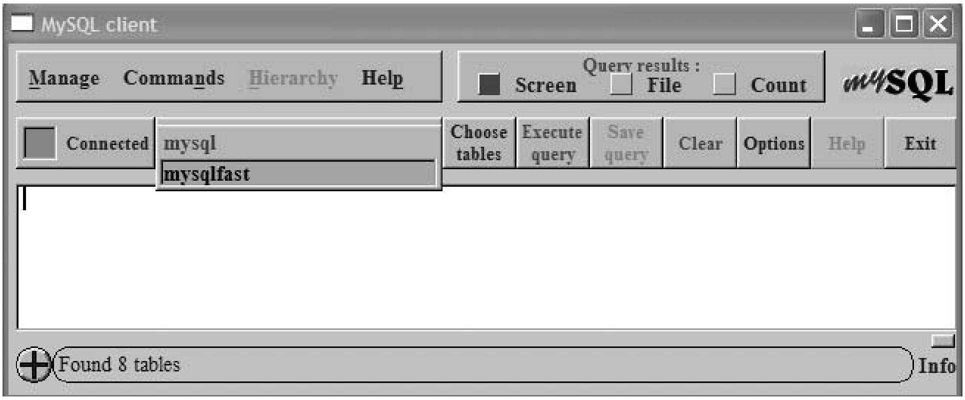
```
USE databasename
```

As has been previously discussed, the default database that MySQL creates to store all of its system tables is called mysql. It is best if you create a separate database to store the example tables used in this book. We will call this database MySQLfast. We can select this database to use by issuing the following commands:

```
CREATE DATABASE MySQLfast;
USE MySQLfast;
```

If you create the new database using the graphical client, it does not immediately appear in the dropdown list of databases. To get it to appear after creating the database, close down the client and restart it. It should now be selectable from the dropdown list, as shown in Figure 4.8.

Selecting a database from the dropdown list in the graphical client is exactly the same as issuing the USE database command.



mysql
mysqlfast

Found 8 tables

Figure 4.8 Selecting a database from the dropdown list.

CREATE TABLE

After you have designed your database, you need to define the tables in MySQL. The main way to do this is with the CREATE TABLE command which is used as follows:

```
CREATE TABLE tablename (columnname type options,
                        columnname type options, … otheroptions)
```

In the above tablename and columnname are self-evident, and type is the datatype of the column as we described in Chapter 3.

options are various options that you can apply to the column, such as the following:

- NOT NULL which prohibits the use of NULL within the column.
- AUTO_INCREMENT which will add a value to the value of this column in the previously added row, and insert the new value into this row automatically. This normally defaults to an increment of 1, and is useful for automatically creating a new, unique primary key value.
- DEFAULT which when followed by a value allows to set the default value that the column will be created with if not specified when inserting data.

So we will now create a table to store our website's webpages. We will store the following:

- The ID of the webpage, which we will automatically create, and will be the primary key and a medium integer.
- The title of the page, which will be the text that we put between the <TITLE></TITLE> tags in the HTML document. This will be a variable length string of 50 characters maximum.
- The actual content of the webpage, which will be inserted between the <BODY></BODY> tags of the HTML document. We will use the text type for storing this, as it will enable us to store a large amount of text if needed.

Make sure you are logged into the correct database by typing the following into the graphical client:

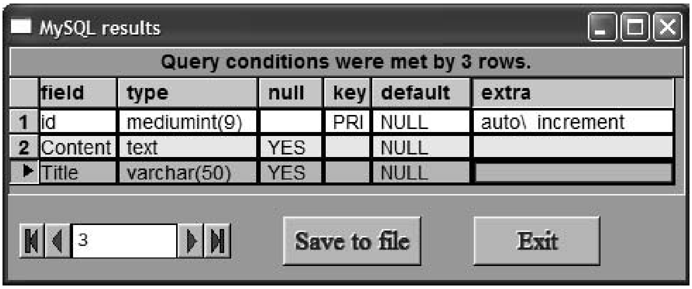
```
USE MySQLfast
```

We can now create the webpage table as follows:

```
CREATE TABLE webpage ( id MEDIUMINT NOT NULL AUTO_INCREMENT,
                       Content TEXT,
                       Title VARCHAR(50),
                       PRIMARY KEY (id));
```

Running the query above creates the table, but does not give you any other feedback unless you have typed something in wrong. To check that the database is correctly created, run the following:

```
DESCRIBE webpage
```



Query conditions were met by 3 rows.

	field	type	null	key	default	extra
1	id	mediumint(9)		PRI	NULL	auto\ increment
2	Content	text	YES		NULL	
▶	Title	varchar(50)	YES		NULL	

Figure 4.9 DESCRIBE the newly created table.

Figure 4.9 shows the results of this. You can see that the table has been created, and the ID column is the primary key and there is an auto-increment column.

You will notice that after we defined the list of columns we added the PRIMARY KEY option, which set the column that we specified as the primary key for this table. You can also specify a column as a primary key by putting these words directly after the type definition for a column.

We will demonstrate this by creating another table for the website log as follows:

```
CREATE TABLE Log ( ID MEDIUMINT
                   NOT NULL
                   AUTO_INCREMENT
                   PRIMARY KEY,
                   CookieID MEDIUMINT,
                   WebpageID MEDIUMINT,
                   Browser VARCHAR(50),
                   DateCreated DATETIME,
                   IPNumber VARCHAR(15),
                   ReferringPage VARCHAR(255)  )
```

At this stage I've decided to store the visitor browser information within the log file, and not as a separate table. We are, however, only storing the IP number of the client machine. If we need to find the host name we can look it up using a DNS server.

Identifying Foreign Keys

Our webpage table did not have any foreign keys, but our log table does. To define the foreign key we have to insert the following into our create table definition:

```
FOREIGN KEY (column) REFERENCES parenttable (primarycolumn)
```

In this command:

- **column** is the name of the column that contains the foreign key in this table.
- **primarycolumn** is the name of the primary key column that this foreign key references.

- **parenttable** is the name of the table where the column resides.

We can add this to the CREATE definition for our log table as follows:

```
CREATE TABLE Log (  ID MEDIUMINT
                       NOT NULL
                       AUTO_INCREMENT
                       PRIMARY KEY,
                    CookieID MEDIUMINT,
                    WebpageID MEDIUMINT,
                    Browser VARCHAR(50),
                    DateCreated DATETIME,
                    IPNumber VARCHAR(15),
                    ReferringPage VARCHAR(255),
                    FOREIGN KEY (WebpageID ) REFERENCES Webpage (ID),
                    FOREIGN KEY (CookieID ) REFERENCES Cookies (CookieID)
                    )
                    TYPE = InnoDB
```

Figure 4.10 shows this table being described.

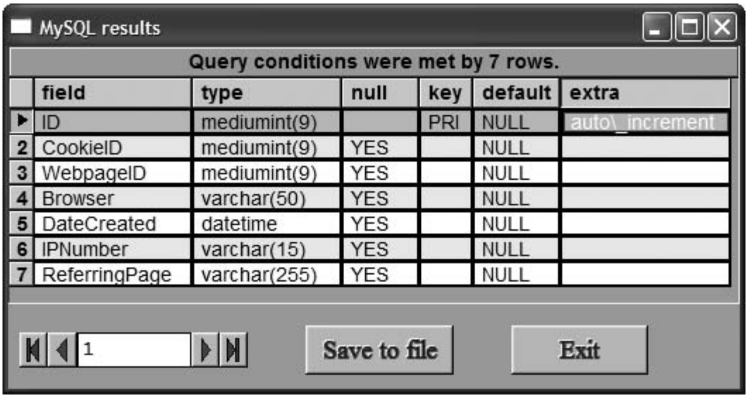
If you try to create this table after you have run the previous script you will get a *Table already exists* error. You can add the words:

```
IF NOT EXISTS
```

directly after the:

```
CREATE TABLE
```

statement to stop MySQL generating an error; however this does not replace the table, it just suppresses the error if it already exists, so it is of questionable use. Other versions of



Query conditions were met by 7 rows.

	field	type	null	key	default	extra
►	ID	mediumint(9)		PRI	NULL	auto_increment
2	CookieID	mediumint(9)	YES		NULL	
3	WebpageID	mediumint(9)	YES		NULL	
4	Browser	varchar(50)	YES		NULL	
5	DateCreated	datetime	YES		NULL	
6	IPNumber	varchar(15)	YES		NULL	
7	ReferringPage	varchar(255)	YES		NULL	

Figure 4.10 DESCRIBE Log table.

SQL allow you to issue the command CREATE OR REPLACE when you are creating a table, which will replace the existing table if needed. MySQL does not yet support this function.

You will notice that the line:

```
TYPE = InnoDB
```

has been added at the end of the table definition. This is the table type that you need to use if you want MySQL to support referential integrity.

Create from Select Statement

You can also create tables based on the output of a select statement. The format is as follows:

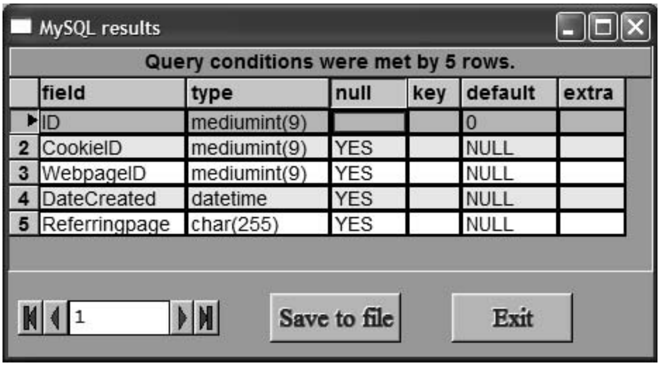
```
CREATE TABLE tablename AS selectstatement
```

We will describe the select statement in detail in a later chapter. The select statement allows you to select some or all of the column and rows in a table. For instance, if we decided that we wanted to make a different log table with all of the client details removed, we could do it with the following script:

```
CREATE TABLE Log2 AS SELECT      ID,
                          CookieID,
                          WebpageID,
                          DateCreated,
                          Referringpage
                FROM log
```

The select statement that we use here requests those column names that do not contain client details. After creating this table, describing it gives the results shown in Figure 4.11.

Compare Figure 4.11 with Figure 4.9. You will notice that the new table has fewer columns, as we expected. The new table has been created purely from the column names and datatypes of the select statement, and took no notice of the other column options our



Query conditions were met by 5 rows.

	field	type	null	key	default	extra
▶	ID	mediumint(9)			0	
2	CookieID	mediumint(9)	YES		NULL	
3	WebpageID	mediumint(9)	YES		NULL	
4	DateCreated	datetime	YES		NULL	
5	Referringpage	char(255)	YES		NULL	

Figure 4.11 DESCRIBE Log2 table.

original table had, such as the primary key and auto-increment options. If you need to add these options to an existing table, see the detail on ALTER TABLE in Chapter 8.

CREATE TABLE LIKE

MySQL version 4.1 gives us an extra function that solves the problem of the column attributes not being copied over when creating a table from a SELECT statement. This is used as follows:

```
CREATE TABLE destination LIKE source
```

This will create a new destination table that is exactly based on the source table's structure. Create a log3 table as follows:

```
CREATE TABLE log3 LIKE log
```

If you describe this table and compare it with the original log table, you will notice that all of the column attributes have now been copied over.

We will create some other tables as needed through the course of this book.

This page intentionally left blank

5 Populating the Database

INSERT

The main method of putting data into our table is by using the SQL INSERT command. The most commonly used format of this command is as follows:

```
INSERT     INTO    tablename (columnname, columnname, …)
           VALUES (value, value, …);
```

The insert command adds a new record to the table, putting the date given into the fields specified. You must ensure that all of the column names specified after the tablename have corresponding entries in the VALUES section. The values must be of the same datatype as the column they are going to be inserted into.

Any columns that are not specified in the statement are left at their default value if you specified it, or set to NULL.

To demonstrate, we will begin to add some data into the database that we described earlier. To start with we will add data to the webpage table, which will store the title and content of our website. All websites have a home page so we will start by inserting a home page. Start up the graphical tools and log in. Type the following to ensure that we are connected to the correct database:

```
USE MySQLfast
```

You can also ensure that you are connected to the MySQLfast database by checking that the client says *MySQLfast* next to the word *Connected* on the second row of controls on the client. If it says *mysql* you are in the wrong database, so pull down the menu by clicking on the square next to *mysql* and select *MySQLfast*. Once you are connected to the correct database you can type the following in the query box:

```
INSERT     INTO webpage (Title)
           VALUES (“Home”)
```

Click onto the **Execute Query** button to run the query. Unless there is an error there will not be any output to show you what you have just done. If you do have an error message it is likely that you have not connected to the database, or you have not yet created the webpage table as described in the previous chapter. If the latter is the case, please review the create table section in Chapter 4.

To view the contents of the database, type the following and click on **Execute Query**:

```
SELECT * FROM webpage
```

We will review the SELECT command in great depth in the next chapter, but for now all this query is saying is "give me everything from the table called webpage". The results are shown in Figure 5.1.

If you look at Figure 5.1 you will see that although we only inserted the word Home, MySQL has auto-generated the value 1 for the ID column. We also did not specify a value for the content column so this has been set to NULL as it contains nothing at the moment.

If you remember our example database that we described in the previous chapter we had a table for storing the entries in a visitor's book for our website. This implies that there would be a page to view or add entries to a visitor's book, so let's add an entry for that page by executing the following:

```
INSERT    INTO webpage (Title, Content)
          VALUES  ("Visitorbook","Please add a comment to my visitor book")
```

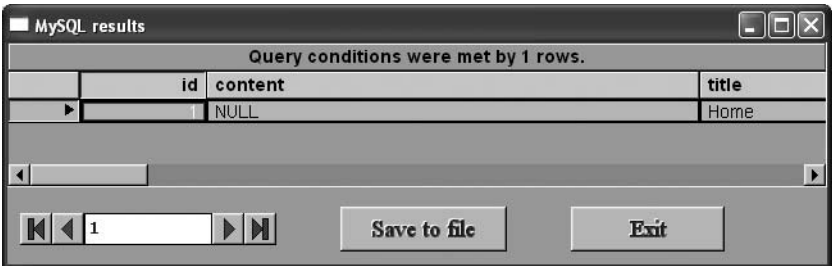
Notice that on this occasion we have chosen to add some content for a page. Again, if you want to see the results of this, execute:

```
SELECT      * FROM webpage
```

INSERT – Multiple Rows

We could carry on inserting various names for pages using the above commands, but if you have many rows to add to a table you can speed up the process by adding multiple rows after the VALUES keyword as follows:

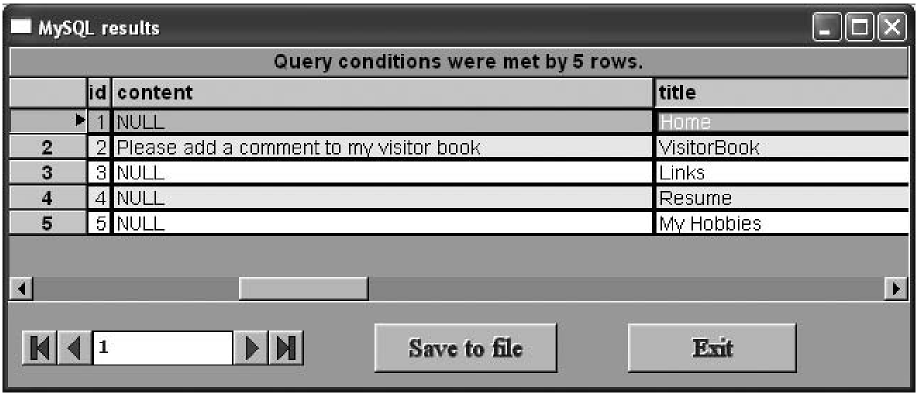
```
INSERT    INTO webpage (Title)
VALUES    ("Links"),
          ("Resume"),
          ("My Hobbies")
```



Query conditions were met by 1 rows.

id	content	title
1	NULL	Home

Figure 5.1 The webpage table.



Query conditions were met by 5 rows.

	id	content	title
▶	1	NULL	Home
2	2	Please add a comment to my visitor book	VisitorBook
3	3	NULL	Links
4	4	NULL	Resume
5	5	NULL	My Hobbies

Figure 5.2 The webpage table.

When you are inserting multiple rows into a database you place each row inside parentheses and separate rows with a comma. Select all of the rows from your webpage table again. It should look like Figure 5.2.

INSERT – All Columns

An alternative form of the INSERT statement misses out the column name list as follows:

```
INSERT    INTO    tablename
VALUES (value, value, …);
```

In this case you must specify a value for each of the columns in the table in the VALUES section, in the order which you originally created the table.

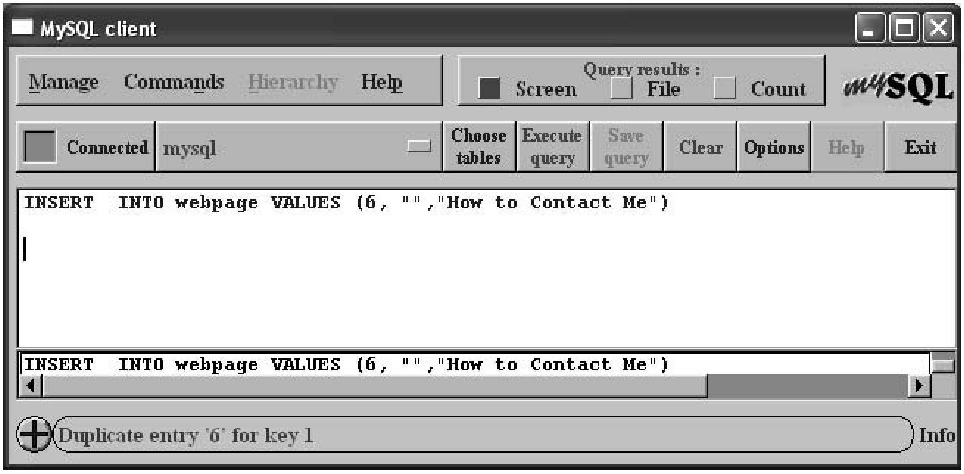
You have less control over the inserting of data using this method, and it can have problems associated with it. For example, if you have a column that is auto-generating (AUTO_INCREMENT) to provide your row with a unique primary key, inserting a number over this may well cause the table to have a duplicate primary key, which is not allowed. It may work, but may lead to errors in the processing of your data later on. If we look at Figure 5.2 again, we can see that we can safely add a row with the ID number 6 as there is not another row with this ID. The following will add a row with that ID:

```
INSERT INTO webpage VALUES (6, “”,”How to Contact Me”)
```

This should have successfully added another row into your table. As a further demonstration of how hard-coding an auto-generating field can lead to errors, run the above query a second time. Figure 5.3 shows the results of this.

Notice how the last line of Figure 5.3 has the error message:

```
Duplicate entry ‘6’ for key 1.
```



```
INSERT  INTO webpage VALUES (6, "","How to Contact Me")
```

```
INSERT  INTO webpage VALUES (6, "","How to Contact Me")
```

Duplicate entry '6' for key 1

Figure 5.3 Error caused by duplicate primary key.

When you tried to add another row to the webpage table that had an ID of 6, MySQL realized that the ID column is the primary key of the database and so wouldn't allow you to insert a key with that value a second time.

INSERT – Columns from a Query

You can also insert rows that are output from another table. The command takes the following format:

```
INSERT    INTO tablename
          SELECT rows
          FROM anothertablename
          WHERE condition;
```

To demonstrate the above command, we will copy our table to another one. First we need to create a new table with the same columns as the webpage table:

```
CREATE TABLE webpage2 (ID MEDIUMINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
Content text, Title text)
```

Now we will copy the data from webpage to webpage2 by executing the following:

```
INSERT INTO webpage2
SELECT * from webpage
```

When using the above command, the table that you are selecting from must have the same number of columns as the table that you are inserting into. For instance, if you try the following:

```
INSERT INTO webpage2
SELECT title FROM webpage
```

you will get the error:

```
Column count does not match value count at row 1.
```

To ensure that your columns match, you can specify a column list after the first table name as follows:

```
INSERT INTO webpage2 (title, content)
SELECT title,content FROM webpage
```

As long as you have the same number of columns in the SELECT statement as you do in the table you are inserting into, you can use just about any code for the SELECT statement. The following will search for rows that have the word *Home* in the title field and insert them into the webpage2 table:

```
INSERT INTO webpage2 (title, content)
SELECT title,content FROM webpage WHERE title = ‘Home’
```

To view the contents of this new table, execute the following:

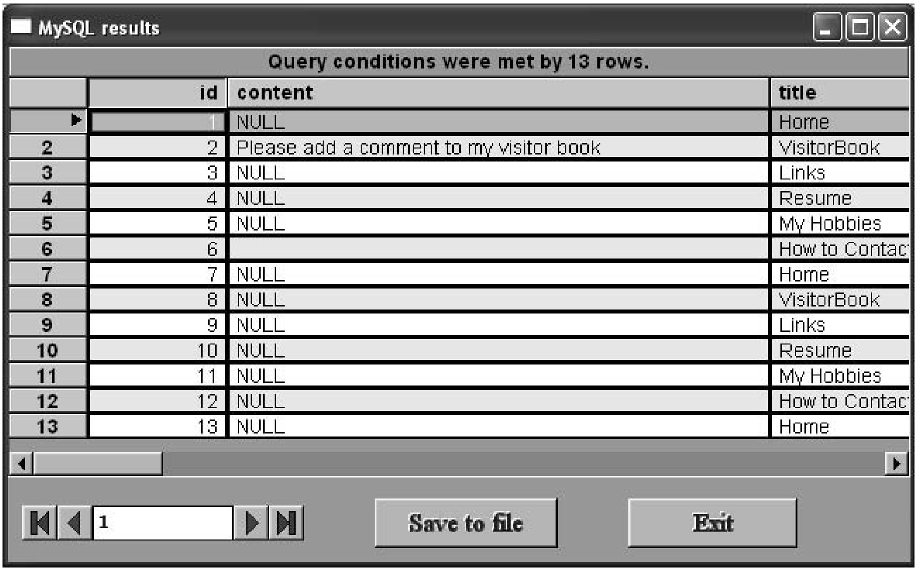
```
SELECT * FROM webpage2
```

Figure 5.4 shows the current content of our webpage2 table.

If you look at Figure 5.4, you will see that we have a unique primary key for each row of the table. You can also see that the content column sometimes has NULL in it, where no data was specified in the content string, an empty string as in row 6, and text data as in row 2. All are equally valid entries. You can also see that in the title column we now have duplicate page entries. This is acceptable in our demonstration database but would probably have little use if we were going to use this table to power a website. We will remove this temporary table in Chapter 9.

Inserting Data with Scripts

Another way to get data into our database is to run a script. A script is just a collection of SQL queries separated by colons and stored in a text file.



Query conditions were met by 13 rows.

	id	content	title
▶	1	NULL	Home
2	2	Please add a comment to my visitor book	VisitorBook
3	3	NULL	Links
4	4	NULL	Resume
5	5	NULL	My Hobbies
6	6		How to Contac
7	7	NULL	Home
8	8	NULL	VisitorBook
9	9	NULL	Links
10	10	NULL	Resume
11	11	NULL	My Hobbies
12	12	NULL	How to Contac
13	13	NULL	Home

Figure 5.4 The webpage2 table populated from another table.

For example, the following file is a collection of commands that creates a table called Cookies and fills it with some sample data:

```
USE          mysqlfast;
CREATE TABLE Cookies (CookieID MEDIUMINT
                      NOT NULL
                      AUTO_INCREMENT
                      PRIMARY KEY,
             DateCreated DATETIME);
INSERT INTO Cookies (DateCreated) VALUES (“2002-01-01”);
INSERT INTO Cookies (DateCreated) VALUES (“2002-01-02”);
INSERT INTO Cookies (DateCreated) VALUES (“2002-01-04”);
INSERT INTO Cookies (DateCreated) VALUES (“2002-01-07”);
INSERT INTO Cookies (DateCreated) VALUES (“2002-01-13”);
INSERT INTO Cookies (DateCreated) VALUES (“2002-01-22”);
SELECT * FROM Cookies;
```

Unfortunately, WinMySQLGUI does not support the running of script files, so we need to use the mysql command line to execute the script. Type the above script into a text editor such as Notepad and save it as *scriptimport.sql* in the directory:

```
C:\mysql\bin
```

The file needs to be saved within that directory as it is hard to select files from directories other than the one that MySQL us running from. Start up a command prompt, by selecting Start/Run and typing command. Then type:

```
cd \mysql\bin
```

to get to the binary directory and then type:

```
mysql
```

to run the client. Once the client is running you can then run the script. You do this by using the command **source** and giving the command a filename as follows:

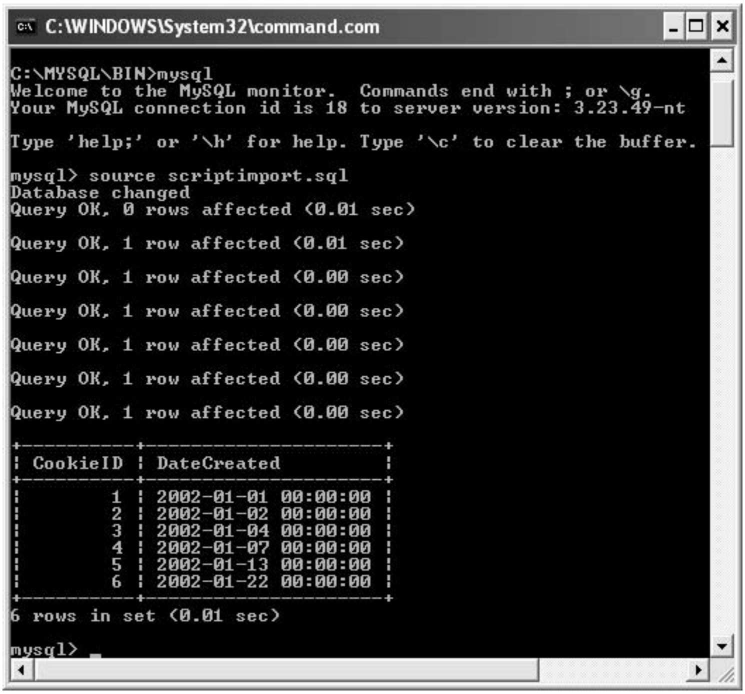
```
source scriptimport.sql
```

This will run the script and the output can be seen in Figure 5.5.

You will notice from the last line of our script that we added:

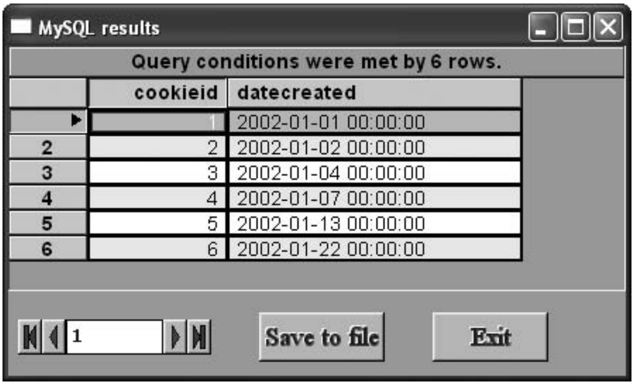
```
SELECT * FROM Cookies
```

at the end of the script. Because of this we have a sample output at the end of the query. You can clearly see that 6 rows have been created, the dates have been inserted and the auto-generated cookieIDs have been created. To prove that MySQL is a multi-user system, load



```
C:\MYSQL\BIN>mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 18 to server version: 3.23.49-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> source scriptimport.sql
Database changed
Query OK, 0 rows affected (0.01 sec)

Query OK, 1 row affected (0.01 sec)

Query OK, 1 row affected (0.00 sec)

Query OK, 1 row affected (0.00 sec)

Query OK, 1 row affected (0.00 sec)

Query OK, 1 row affected (0.00 sec)

Query OK, 1 row affected (0.00 sec)

+----------+---------------------+
¦ CookieID ¦ DateCreated         ¦
+----------+---------------------+
¦        1 ¦ 2002-01-01 00:00:00 ¦
¦        2 ¦ 2002-01-02 00:00:00 ¦
¦        3 ¦ 2002-01-04 00:00:00 ¦
¦        4 ¦ 2002-01-07 00:00:00 ¦
¦        5 ¦ 2002-01-13 00:00:00 ¦
¦        6 ¦ 2002-01-22 00:00:00 ¦
+----------+---------------------+
6 rows in set (0.01 sec)

mysql> _
```

Figure 5.5 Running a script from the command line.



MySQL results

Query conditions were met by 6 rows.

	cookieid	datecreated
▶	1	2002-01-01 00:00:00
2	2	2002-01-02 00:00:00
3	3	2002-01-04 00:00:00
4	4	2002-01-07 00:00:00
5	5	2002-01-13 00:00:00
6	6	2002-01-22 00:00:00

Figure 5.6 The cookie table viewed in the graphical tools.

the graphical tools, connect to the *mysqlfast* database and run the query again. Figure 5.6 shows the results which compared with Figure 5.5 are identical.

Directly Inserting Data

Another way that we can insert data into tables is by using the MySQL command LOAD DATA. This file loads data into a specified table from a text file. The basic format is as follows:

```
LOAD DATA INFILE ‘filename’
INTO TABLE tablename (Column1, Column2, …)
```

Filename is the name of the text file with the data that you want to import. Tablename is the table to insert the data into. The columns are the columns that you want to insert, in the order that they appear within your text file. For example, the following is a heavily edited section of a weblog file:

```
ID,DateCreated,Browser,IPNumber,CookierID,WebPageID,ReferringPage
1,2002-01-01,Mozilla/4.0,192.168.1.10,2,1,Typed
2,2002-01-01,Mozilla/4.0,192.168.1.10,2,2,http://minbar.homeip.net/
3,2002-01-01,Mozilla/4.0,192.168.1.10,2,3,http://www.google.com
4,2002-01-01,Mozilla/4.0,192.168.1.10,2,1,http://www.easyrew.com
5,2002-01-01,Mozilla/4.0,192.168.1.10,2,1,http://www.kli.org
```

If we wanted to import that file into our log table, we would need to save the above as a text file. Call the file *log.sql* and save it in:

```
C:\mysql\data\mysqlfast
```

By default, LOAD DATA expects the file to be on the same machine as the MySQL server, within the directory that contains the information about the current database, in our case, *mysqlfast*. If we were running the client and server on separate machines, and wanted the file to be on the client machine, we would need to insert the word LOCAL before INFILE to specify that it is on the local machine. We do not need to do this in our example.

If we look at the text file again, we can see that the first line is a list of the column names in the order of the file. This is useful information to us but not to the LOAD DATA command. We could delete that first row but that would mean changing the contents of our log file which may be needed by another program, so we need to add the line:

```
IGNORE 1 LINES
```

at the end of our query to get the command to bypass the first line.

You will notice that our file has commas separating the columns. This is one standard of representing data in text files, called a “csv” or Comma Separated Variable file. By default, though, the LOAD DATA command looks for a tab to separate columns. We therefore have to tell LOAD DATA to view the separators as commas. This is done by adding the following to the query:

```
FIELDS TERMINATED BY ‘,’
```

As you may expect, as there is a FIELDS TERMINATED command there is also a RECORDS TERMINATED command for setting the way that the record’s end is represented. As LOAD DATA’s default setting for a record’s end is the newline, the same as our text file, we don’t have to specify this.

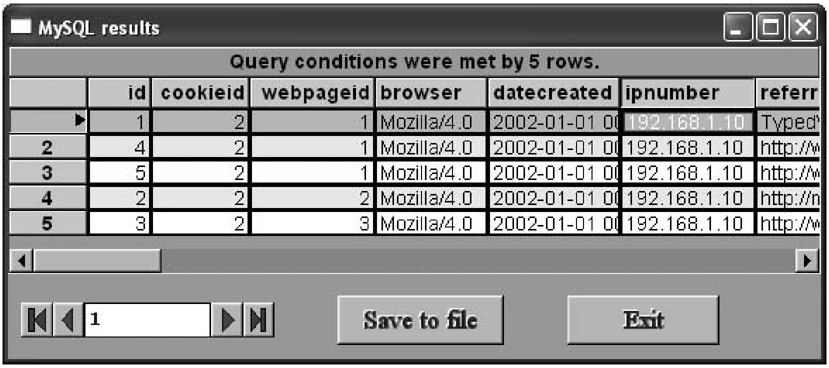
If you remember when we created the Log table in the previous chapter, we used the following command:

```
CREATE TABLE Log (  ID MEDIUMINT
                       NOT NULL
                       AUTO_INCREMENT
                       PRIMARY KEY,
                    CookieID MEDIUMINT,
                    WebpageID MEDIUMINT,
                    Browser TEXT,
                    DateCreated DATETIME,
                    IPNumber TEXT,
                    ReferringPage TEXT    )
```

By default, LOAD DATA expects the columns in the text file to be in the same order as when the table was created. If you compare this with the text file you will see that the column order is different. LOAD DATA allows you to specify the order of columns to import by just giving it a list of the rows.

We now have enough information to insert our log file into the Log table. The command that we will use to do this is as follows:

```
LOAD DATA INFILE ‘log.csv’
     INTO TABLE Log
     FIELDS TERMINATED BY ‘,’
     IGNORE 1 LINES
           (ID,
           DateCreated,
           Browser,
           IPNumber,
           CookieID,
           WebPageID,
           ReferringPage )
```



MySQL results

Query conditions were met by 5 rows.

	id	cookieid	webpageid	browser	datecreated	ipnumber	referr
▶	1	2	1	Mozilla/4.0	2002-01-01 0	192.168.1.10	Typed'
2	4	2	1	Mozilla/4.0	2002-01-01 0	192.168.1.10	http://w
3	5	2	1	Mozilla/4.0	2002-01-01 0	192.168.1.10	http://w
4	2	2	2	Mozilla/4.0	2002-01-01 0	192.168.1.10	http://n
5	3	2	3	Mozilla/4.0	2002-01-01 0	192.168.1.10	http://w

Figure 5.7 The Log table populated by LOAD DATA.

Figure 5.7 shows the results of the above query, when we select everything from the Log table.

The above process can seem like a lot of effort to put 5 rows in a table; we have had to manipulate the load command and probably could have inserted that many rows in the same time by using INSERT statements. The effort, however, would have been much more worth if we were inserting a full log file into our database. The file that the above example file was taken from had over 38 000 rows, which could be inserted into our database with the same effort as we have expended in the above example. This is where the power of LOAD DATA comes into its own.

We may also have been importing the data from another database system, such as Oracle or SQL Server. If this was the case it would be easier to output the data from that DBMS in a format that the LOAD DATA command instantly recognizes as default; for instance using tabs as column separators and making the order of the columns the same as our create table command. The more you can manipulate your data into a standard format, the easier it is to import into other systems.

Now that we have some data in our database we can begin to see the techniques involved in retrieving the data.

Retrieving the Data 6

SELECT

We have spent quite a while over the last few chapters thinking about, designing and populating our database. Now we have got some data in there, we need to get it back in a controlled way.

Probably the most commonly used SQL command is the SELECT statement which returns data from the database. The SELECT statement has various different formats, which we will describe one by one in this chapter with examples.

The simplest form of the select statement is as follows:

```
SELECT      *
FROM   tablename
```

The above statement will return everything from your table. The asterisk in this context means all columns. So the query means: "Give me all of the columns and all of the rows of the table tablename".

For example, to run this query on the webpage table would require the following code:

```
SELECT      *
FROM   webpage
```

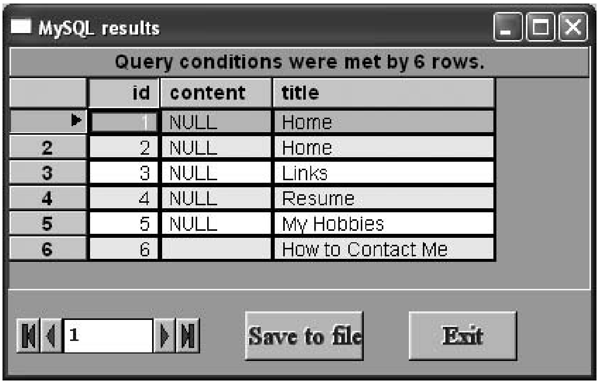
Figure 6.1 shows the results of running this query.

With the data that is currently within our webpage table, the SELECT * results fit neatly into the results shown in Figure 6.1. In practice, however, we rarely have this limited amount of data. For example, our weblog table will quickly become populated by data that will fill many screens. We can see an example of this already if we run the same query on the log table:

```
SELECT      *
FROM   log
```

Figure 6.2 shows the results of this simple query.

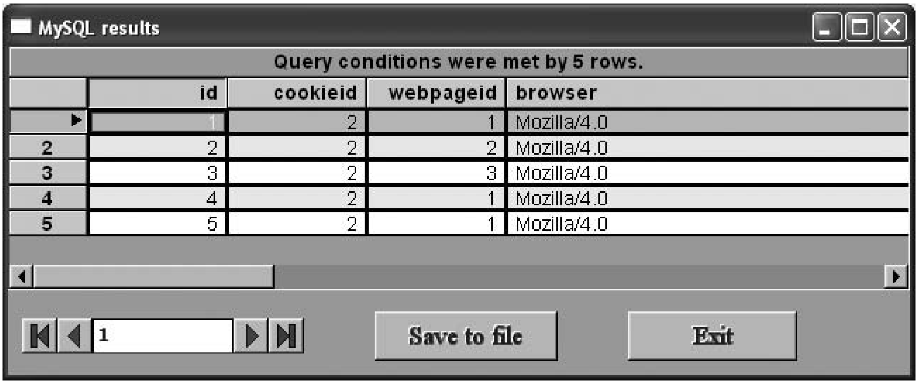
All of the data in the log table has been returned, as can be seen by scrolling across the results. However, if we want to find out on which dates certain pages were viewed, scrolling



Query conditions were met by 6 rows.

	id	content	title
▶	1	NULL	Home
2	2	NULL	Home
3	3	NULL	Links
4	4	NULL	Resume
5	5	NULL	My Hobbies
6	6		How to Contact Me

Figure 6.1 Select * from webpage.



Query conditions were met by 5 rows.

	id	cookieid	webpageid	browser
▶	1	2	1	Mozilla/4.0
2	2	2	2	Mozilla/4.0
3	3	2	3	Mozilla/4.0
4	4	2	1	Mozilla/4.0
5	5	2	1	Mozilla/4.0

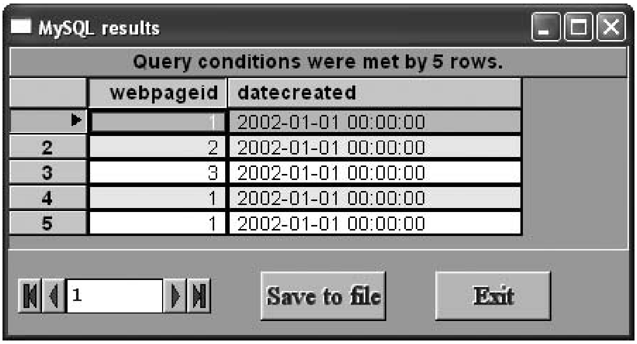
Figure 6.2 A wide set of results from the log table.

across is slow and could possibly lead to errors in reading the results if we slipped our eyes to the wrong line while scrolling. It is much better to get exactly the data that we want. The SELECT command allows you to specify the columns that you want returned as in the following example:

```
SELECT    webpageid,datecreated
FROM      log
```

You will notice that the above query has the table names separated by commas. If we run this query we get the results shown in Figure 6.3, which give us just the data that we want.

When you use the SELECT statement to return columns it divides your table vertically, retrieving only the specific columns that answer your query. You can specify any of your columns with the SELECT statement, including all of them, but the asterisk means less typing. You can also select the same column more than one time, as well as selecting columns from different tables. You may not yet see why you would want to use the latter two methods, but we will return to them later.



Query conditions were met by 5 rows.

	webpageid	datecreated
▶	1	2002-01-01 00:00:00
2	2	2002-01-01 00:00:00
3	3	2002-01-01 00:00:00
4	1	2002-01-01 00:00:00
5	1	2002-01-01 00:00:00

Figure 6.3 Two columns returned by the SELECT command.

SELECT DISTINCT

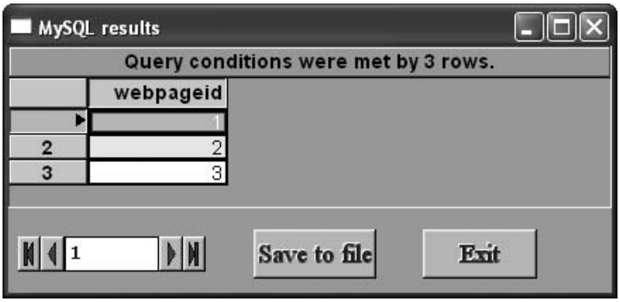
SELECT DISTINCT is a method of returning a unique set of data from your table. It is best to only apply this SELECT to a specific column in your table. The format of this query is as follows:

```
SELECT     DISTINCT columnname
FROM       tablename
```

For example, you may remember that one of the pages that we have on our website was our resume. If you refer back to Figure 6.1 you can see that this was page ID 4. We can use SELECT DISTINCT to give us a list of the pages that have been viewed in our weblog table. We don't need the quantity or number of times each page has been viewed, that can come later, but we just need a list of pages that have been viewed. The following query will give us such a list:

```
SELECT     DISTINCT webpageid
FROM       log
```

The results of the above query are shown in Figure 6.4.



Query conditions were met by 3 rows.

	webpageid
▶	1
2	2
3	3

Figure 6.4 SELECT DISTINCT on the Log table.

If you examine Figure 6.4 you will see that although the Log table contained more than three rows, the SELECT DISTINCT query has returned only three, and all of them contain a unique page ID. Unfortunately, none of these rows contain the page ID 4, so no one has viewed the resume webpage yet. It looks like I will not be changing jobs in the near future. In a later chapter we will see how to count specific hits per page, but you can see from the above how the simple addition of the DISTINCT command can instantly present our data in a manner that is much easier to use. You will also realize that if our website had thousands of hits in the log, this query would still produce a few rows that were easy to understand.

WHERE

We have seen how to select specific columns of data using the SELECT columnname query. This slices the table vertically. The next command will slice the table horizontally, selecting specific rows. This is done by matching the data within a record with a set condition that is specified in the WHERE clause, as follows:

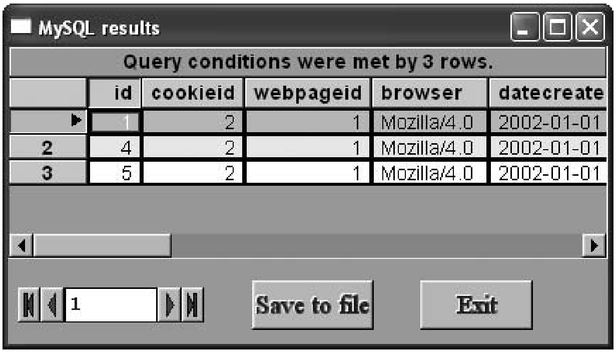
```
SELECT     items
FROM       tablename
WHERE      condition
```

The WHERE clause isn't only used with select statements; it can be used on any occasion where we wish to limit a query to only apply to certain matching rows.

To illustrate the WHERE clause, the following query should tell us all the times that the Home page (ID = 1) has been viewed in our weblog:

```
SELECT     *
FROM       log
WHERE      webpageid = 1
```

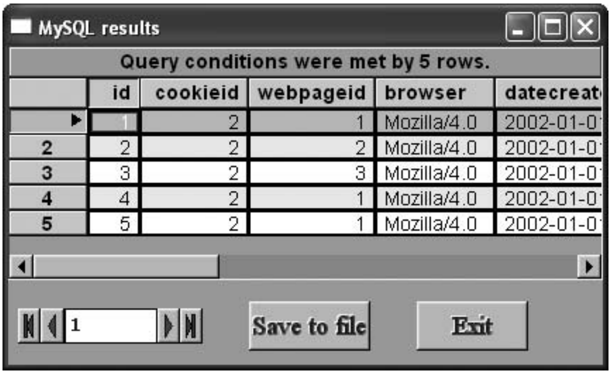
Figure 6.5 show the results of this query. The query is saying: "Give me all of the rows that contain a 1 (the ID of the Home page) in the webpageid column." Compare this with



Query conditions were met by 3 rows.

	id	cookieid	webpageid	browser	datecreate
▶	1	2	1	Mozilla/4.0	2002-01-01
2	4	2	1	Mozilla/4.0	2002-01-01
3	5	2	1	Mozilla/4.0	2002-01-01

Figure 6.5 WHERE returns columns that match a criteria.



MySQL results

Query conditions were met by 5 rows.

	id	cookieid	webpageid	browser	datecreat
▶	1	2	1	Mozilla/4.0	2002-01-0
2	2	2	2	Mozilla/4.0	2002-01-0
3	3	2	3	Mozilla/4.0	2002-01-0
4	4	2	1	Mozilla/4.0	2002-01-0
5	5	2	1	Mozilla/4.0	2002-01-0

Figure 6.6 Without the WHERE clause.

Figure 6.6 which is the results of the SELECT statement when the WHERE clause is omitted.

The condition in the WHERE clause can check for one condition or many, and can comprise many different formats. For example, if we were searching for a text string we would put the string within quote marks as follows:

```
SELECT     *
FROM       webpage
WHERE      title = “Home”
```

The current version of MySQL will accept strings within quote marks (“”) or apostrophes (‘’). This may change in future versions but whichever you decide to use make sure that you consistently use the same throughout all of your queries.

MySQL Operators

As well as =, MySQL supports other operators. Some of these are shown in Table 6.1.

Table 6.1 Some MySQL Operators.

Operator	Meaning
=	equal
<>	not equal
!=	not equal
<=	less than or equal
<	less than
>=	greater than or equal
>	greater than
<=>	null safe equal

LIKE

When you are searching for a string using the equals sign (=), SQL only searches for the exact string. If you need to search for a string that began with *Home*, you need to use the LIKE operator and pattern matching. To test this, we will add another row to our webpage table which will point to a webpage that contains happy memories of where we grew up. Add the row with the following query:

```
INSERT    INTO     webpage (Title, Content)
          VALUES   ("Home is where the Heart is",
                   "This page contains happy memories from my childhood")
```

Select everything from the webpage table after running the query above just to check that the new row has been added. You should now see several rows that have a Title column that begin with the word *Home*. We will now try and retrieve all of those columns.

If you have programmed before, the temptation is to try the following query:

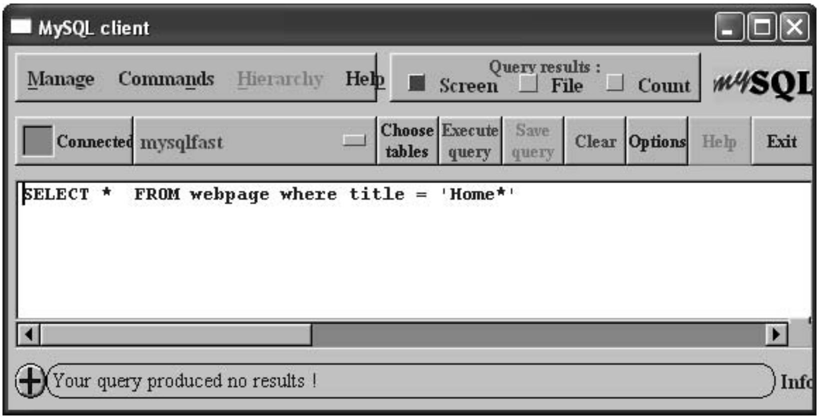
```
SELECT    *
FROM      webpage
WHERE     Title = "Home*"
```

Figure 6.7 shows that when you run this query it doesn't produce what you would expect. All that is given is the error message that the query produced no results.

To get that query working the way that we want, we need to use the LIKE operator in the condition as follows:

```
SELECT    *
FROM      webpage
WHERE     Title LIKE "Home%"
```

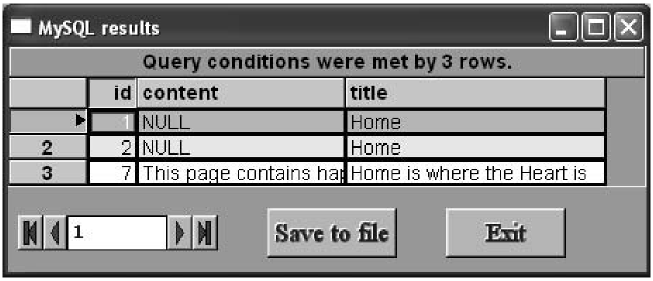
Figure 6.8 shows the results of running this query.



```
SELECT *  FROM webpage where title = 'Home*'
```

Your query produced no results !

Figure 6.7 Equals will not work with fuzzy matching.



Query conditions were met by 3 rows.

	id	content	title
▸	1	NULL	Home
2	2	NULL	Home
3	7	This page contains ha	Home is where the Heart is

Figure 6.8 Use LIKE to fuzzy match.

You will have noticed that we did not use the asterisk as the match all characters as you may be familiar with from other languages. SQL uses the percent sign (%) to signify that any number of characters can be matched. Putting the percent sign at the end of the string we are looking for makes SQL match anything that begins with the string. To match anything with *Home* at the end we would use:

```
SELECT     *
FROM       webpage
WHERE      Title LIKE “%Home”
```

And if we were looking for a string with the word *Home* somewhere in it, we would put the percent sign at both ends of the string as follows:

```
SELECT     *
FROM       webpage
WHERE      Title LIKE “%Home%”
```

Before you run the query above, try and work out what it will select from our table. Why do you think it works the way it does?

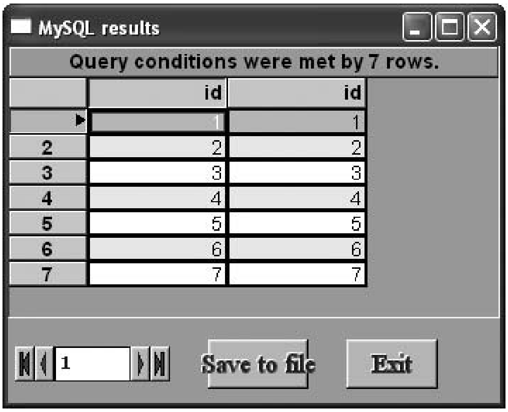
As well as using the percent sign to match multiple characters, you can also use the underscore (_) to match single characters as follows:

```
SELECT     *
FROM       log
WHERE      browser LIKE “Mozilla/_._%”
```

This will match any version of the Mozilla browser, no matter what browser description follows.

SELECT AS (aliases)

On occasions, you may need to select the same column twice in a query. An example of an application of this will occur in the chapter on aggregate functions. The simplest way to select a column twice is by quoting it twice in the column list as follows:



Query conditions were met by 7 rows.

	id	id
▶	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7

Figure 6.9 Selecting the same column twice.

```
SELECT     id,id
FROM       webpage
```

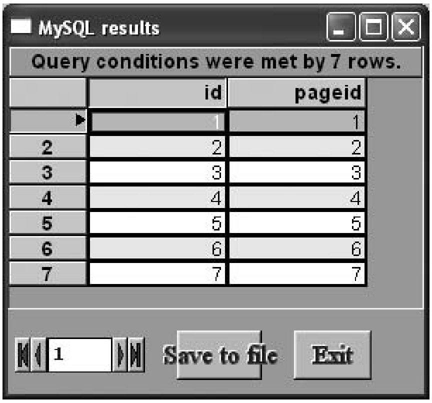
As you will expect, the above query produces a result as shown in Figure 6.9.

However, it may be that later in the query we need to perform some calculation on one of those columns. How are we to tell the difference between the two? As both columns are called *id* that would be difficult. To solve this problem SQL allows the use of aliases, that is, the renaming of a column by use of the AS keyword.

To use this we would modify the above code as follows:

```
SELECT     id,id AS pageid
FROM       webpage
```

You can compare the different results of the two queries by examining Figures 6.9 and 6.10.



Query conditions were met by 7 rows.

	id	pageid
▶	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7

Figure 6.10 Selecting the same column twice with an alias.

You can see that the names of the columns are now different. There are many more uses of the AS keyword. We use it to great extent with aggregate functions. It also will appear a few times in the rest of this chapter.

BETWEEN

The BETWEEN keyword is used when you need to get data back from your query that matches a range of values. For instance, you may want to see how many webpages were viewed last week (between two dates), or that have been viewed at least ten but not more than fifty times each.

BETWEEN is followed with the first value in the range, the AND keyword and the second value, as follows:

```
SELECT    columns
FROM      tables
WHERE     column BETWEEN firstvalue AND secondvalue
```

For example, every new visitor to our website generates a new cookie. We store these cookies in our cookie table. However, we know that the first four cookies that were created were just for testing purposes. So we need a query to look for cookies within a range. This can be achieved as follows:

```
SELECT    cookieid
FROM      cookies
WHERE     cookieid BETWEEN 5 AND 10
```

Figure 6.11 shows the results of the above query. You will note that the query has only produced two rows with the IDs 5 and 6.

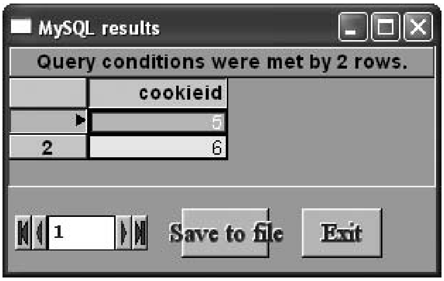
Although this is a very simple example, the BETWEEN keyword is very powerful as we will see later on when we look at processing dates.

NOT

The NOT keyword when applied to a condition will give the opposite answer. All conditions give a true or false result, and NOT changes the result to the opposite. To demonstrate this we will apply this to the query we just discussed which returned all rows with the ID between 5 and 10. Add NOT to the query as follows:

```
SELECT    cookieid
FROM      cookies
WHERE     cookieid NOT BETWEEN 5 AND 10
```

Figure 6.11 showed that the first query picked the IDs 5 and 6. Compare this with Figure 6.12 which shows the results when NOT is applied. A different set of rows has been selected this time.

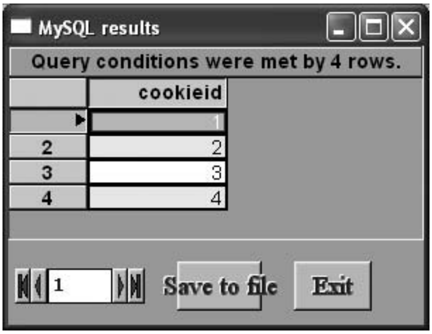


MySQL results

Query conditions were met by 2 rows.

	cookieid
▶	5
2	6

Figure 6.11 BETWEEN selects from a range.



MySQL results

Query conditions were met by 4 rows.

	cookieid
▶	1
2	2
3	3
4	4

Figure 6.12 NOT applied to the BETWEEN query.

As another example, we will look at what happens if you select a single row from a table. We will use the cookie table again, so select a single row as follows:

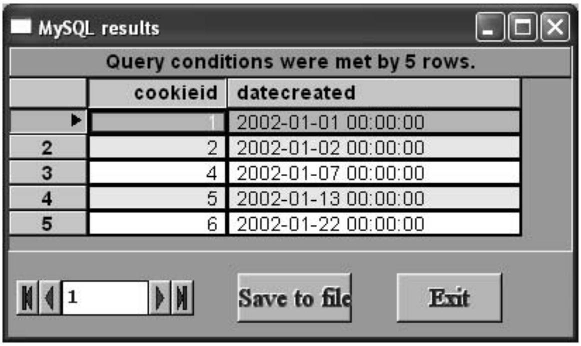
```
SELECT    *
FROM      cookies
WHERE     cookieid = 3
```

This will definitely select a single row as we are selecting from the primary key column, so there will be no other rows with the ID of 3. Run the query and notice what is returned. Now add a NOT to the condition as follows:

```
SELECT    *
FROM      cookies
WHERE     NOT (cookieid = 3)
```

You need to add the parentheses to ensure that the NOT works with the whole condition. With this change, what will the query now select? Figure 6.13 shows the results.

Notice how the NOT has radically changed the results of the query; instead of returning a single row, it now returns everything in the table apart from the specified row.



Query conditions were met by 5 rows.

	cookieid	datecreated
▶	1	2002-01-01 00:00:00
2	2	2002-01-02 00:00:00
3	4	2002-01-07 00:00:00
4	5	2002-01-13 00:00:00
5	6	2002-01-22 00:00:00

Figure 6.13 NOT applied to a single row query.

Why do we need the parentheses in the above script? Can you work out what would happen without them and then edit your query and test your theory? Why do you think this happens?

CONCAT

CONCAT allows you to customize your queries output any way you wish. Normally if you execute a *SELECT tablename* all that you will get back are the unprocessed contents of these records. This will not always be exactly what you need, so you can alter the output with CONCAT. To demonstrate this we will add a table for logging visitor book entries to our website. We will have to run a few scripts to set this up.

First we will add an entry for the visitor's page in our webpage table:

```
INSERT     INTO     webpage (Title, Content)
           VALUES   (“Visitor book”,
                    “Please sign my visitor’s book by filling in the details
                    on this page”)
```

Now we will create a visitorbook table and add some fictitious entries:

```
CREATE TABLE visitorbook
  (EntryID MEDIUMINT AUTO_INCREMENT NOT NULL PRIMARY KEY,
  CookieID MEDIUMINT,
  EntryDate DATETIME,
  EntryText TEXT,
  FirstName TEXT,
  MiddleName TEXT,
  LastName TEXT) ;
```

```
INSERT INTO visitorbook
  (CookieID,EntryDate,FirstName,MiddleName,LastName,EntryText)
  VALUES (1, “2002-01-22”,“Rowan”,“”,“Norman”,“Just testing!”),
  (1, “2002-02-01”,“Reuben”,“”,“Norman”,“I dont know what to type”),
  (1, “2002-02-01”,“Pat”,“”,“Shah”,“Thought I would leave a message”),
  (1, “2002-02-05”,“Lucy”,“”,“Garrett”,“Where has Princess gone?”),
  (1, “2002-02-15”,“Zephen”,“”,“”,“I like the site but what about graphics”),
  (1, “2002-02-16”,“Matthew”,“”,“Norman”,“Just testing!”);
```

You may notice that, for speed, we have used the TEXT datatype for most of the columns. This is not very efficient but we will remedy this in a later chapter. Just create the table as it is for now. Once you have populated the visitorbook table, check your work with:

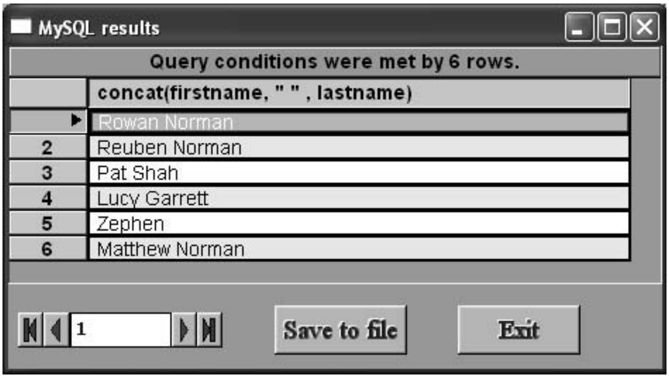
```
SELECT     *
FROM       visitorbook
```

This should give you six rows if you have copied the script in correctly. Now we will use CONCAT to merge the firstname and lastname tables into one column. Execute the following:

```
SELECT     CONCAT(FirstName, “ “ , LastName)
           FROM visitorbook
```

This query says: “Give me the firstname, a space, and the lastname all in one column, for every row in the visitorbook.” Each element inside the CONCAT parentheses is separated by commas. If you have to include a string, such as the space character, then you need to surround it with quotation marks. Figure 6.14 shows the results of this query; all names are displayed neatly in a single column, with a space in between.

If you don’t have results as shown in Figure 6.14, it is probably because you have put a space in between the CONCAT keyword and the parentheses. CONCAT is a function, and all



Query conditions were met by 6 rows.

	concat(firstname, " " , lastname)
►	Rowan Norman
2	Reuben Norman
3	Pat Shah
4	Lucy Garrett
5	Zephen
6	Matthew Norman

Figure 6.14 CONCAT formats your results.

function keywords that are followed by parentheses should have them directly after the keyword, without a space.

Notice also the name of the column at the top of Figure 6.14. This name is the CONCAT function we issued within the query. Although this name makes it obvious what the column is, if we had to continually refer to this within other queries or php code it would soon become tedious. So here is a good example of using the AS keyword to give the results a different name:

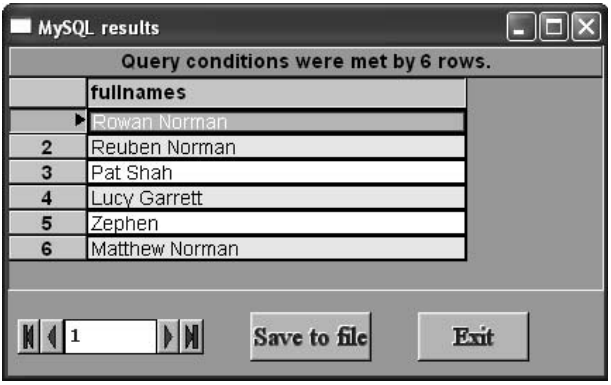
```
SELECT      CONCAT(FirstName, “ “ , LastName) AS fullnames
            FROM visitorbook
```

Figure 6.15 shows the results of just adding AS fullnames to our query. Our column now has a meaningful name.

We can demonstrate this one step further. On our website we will eventually show visitors all of the visitor book entries. We will probably have to format the data we get back from the query extensively to make it fit for presentation on a webpage. This might require much formatting within the scripting language that the page is written in. So we will use the CONCAT function of SQL to do some of the formatting for us, saving some time and work later on. Here is a sample script:

```
SELECT    CONCAT(“In our visitor book, “,
                  FirstName,
                  “ “ ,
                  LastName,
                   “ says: “,
                  Entrytext )
          AS entries
          FROM visitorbook
```

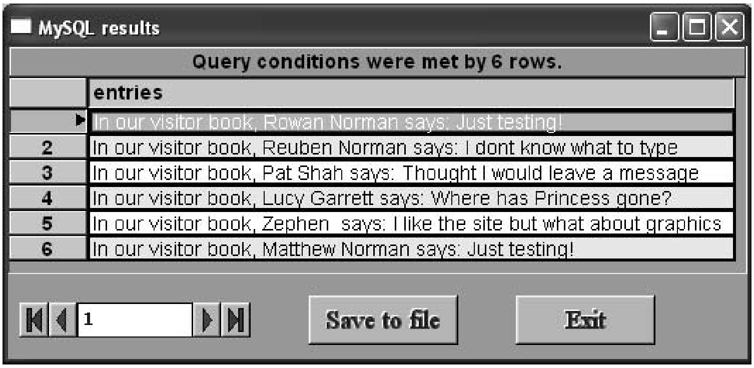
As the CONCAT has multiple elements here, I have over-formatted the script so you can see what is happening. There are lots of commas and speech marks that can get quite con-



Query conditions were met by 6 rows.

	fullnames
▶	Rowan Norman
2	Reuben Norman
3	Pat Shah
4	Lucy Garrett
5	Zephen
6	Matthew Norman

Figure 6.15 CONCAT and AS for readable results.



Query conditions were met by 6 rows.

	entries
▶	In our visitor book, Rowan Norman says: Just testing!
2	In our visitor book, Reuben Norman says: I dont know what to type
3	In our visitor book, Pat Shah says: Thought I would leave a message
4	In our visitor book, Lucy Garrett says: Where has Princess gone?
5	In our visitor book, Zephen says: I like the site but what about graphics
6	In our visitor book, Matthew Norman says: Just testing!

Figure 6.16 CONCAT supports complex formatting.

fusing, so have a look at the results in Figure 6.16 to see what it does. Now that you've viewed the results, the script should make more sense.

So you can see how you can use CONCAT to format your query's output, getting SQL to do the work for you to save scripting time later on. You can put multiple elements inside the CONCAT function. Also remember that you are not limited to using just a CONCAT function in a select statement, you can also use other column names at the same time, for example:

```
SELECT    CONCAT(FirstName, “ “ , LastName) AS names, EntryID, Entrytext
          FROM visitorbook
```

Before running this script, see if you can work out what results it will produce.

ORDER BY

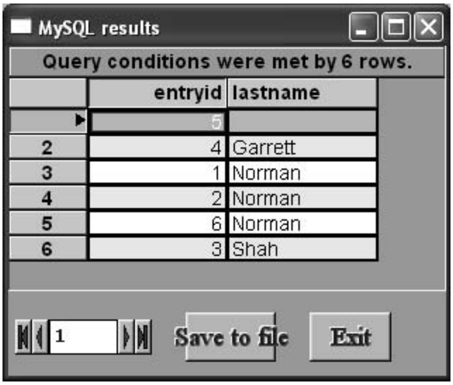
One of the things that you often wish to do with databases is sort the results. MySQL sorts tables using the ORDER BY clause in the following format:

```
SELECT    columnnames
FROM      visitorbook
ORDER BY  columnnames
```

For example, to sort all of our visitorbook entries by last name we would run:

```
SELECT    entryid, lastname
FROM      visitorbook
ORDER BY  lastname
```

Figure 6.17 shows the results of this query. Notice how the rows have been returned in a different order to which they were created.



Query conditions were met by 6 rows.

	entryid	lastname
▶	5	
2	4	Garrett
3	1	Norman
4	2	Norman
5	6	Norman
6	3	Shah

Figure 6.17 ORDER BY sorts columns in ascending order.

By default, ORDER BY sorts in ascending order. You can also specify this by adding the keyword ASC after the clause. To get the sort to go in descending order we add the keyword DESC instead, as follows:

```
SELECT    entryid, lastname
FROM      visitorbook
ORDER BY  lastname DESC
```

Figure 6.18 shows the results in descending order. You are not limited to just sorting by one column, and you do not even have to return the column you are sorting from the query. Here is an example of this which sorts by the *lastname* but only returns the *entryID* and *firstname*:

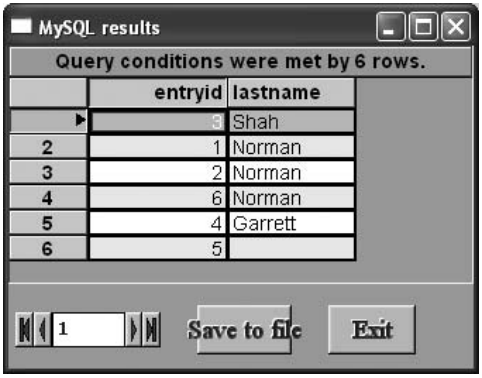
```
SELECT    entryid, firstname
FROM      visitorbook
ORDER BY  lastname, firstname
```

Compare Figure 6.19 with Figure 6.17. The list is now sorted first on the *lastname* column irrespective of whether it is returned or not. After that it is sorted by the *firstname* column. This still leaves the three Normans together in the middle of the query, but then sorts their first names alphabetically as well.

As we have just been discussing the ORDER BY clause, this might seem like a good place to discuss the GROUP BY clause. However, as much as the two clauses sound similar the GROUP BY clause has a totally different use and so will be discussed in Chapter 10.

IN

Often in a query you have to search for a specific match, or a match that is less, or more, than a set value. However, sometimes you may need to search for values that don't easily fit

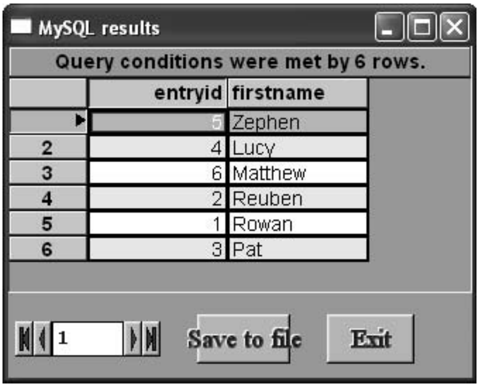


MySQL results

Query conditions were met by 6 rows.

	entryid	lastname
▶	3	Shah
2	1	Norman
3	2	Norman
4	6	Norman
5	4	Garrett
6	5	

Figure 6.18 ORDER BY . . . DESC reverses the sort order.



MySQL results

Query conditions were met by 6 rows.

	entryid	firstname
▶	5	Zephen
2	4	Lucy
3	6	Matthew
4	2	Reuben
5	1	Rowan
6	3	Pat

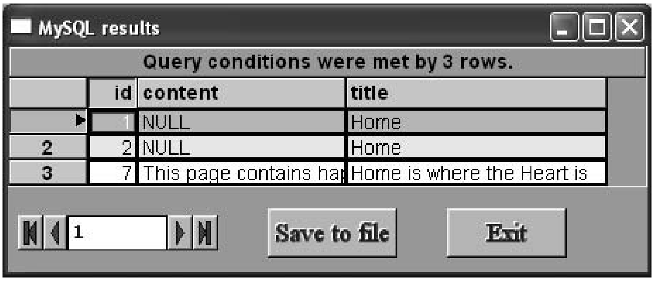
Figure 6.19 ORDER BY on a different column than that returned.

into a simple clause. When this is needed, you can use the IN keyword to search against a set of criteria. This command works as follows:

```
SELECT    columns
FROM      tablename
WHERE     column IN (values)
```

In the above query, values is a comma-separated set of values. As an example of this we will again select all of the webpages that begin with the word *Home* from our webpage table. Using LIKE this command is:

```
SELECT    id, title
FROM      webpage
WHERE     title LIKE “Home%”
```



Query conditions were met by 3 rows.

	id	content	title
▶	1	NULL	Home
2	2	NULL	Home
3	7	This page contains ha	Home is where the Heart is

Figure 6.20 Selecting the *Home* pages.

Figure 6.20 shows the above script in action. It also shows the three different IDs that begin with *Home*; 1, 2 and 7. We will now use those IDs to select the rows:

```
SELECT    id, title
FROM      webpage
WHERE     id IN (1,2,7)
```

Running the above query will produce the same results as shown in Figure 6.20 but they have been selected in a completely different way. We can select the rows that are not in that ID set by adding a NOT to the clause as follows:

```
SELECT    id, title
FROM      webpage
WHERE     id NOT IN (1,2,7)
```

This will return the complementary set of rows. Note where the NOT keyword appears in the clause. As different positions cause different result sets, it is always a good idea to run the query through an SQL client before you use it in a website so that you can ensure that you are going to get the results that you expect.

Subqueries

Version 4.1 of MySQL now includes the ability to perform subqueries. A subquery is a query that is run to derive a value that is fed into the main query. For instance, the following query was used to extract the three rows shown in Figure 6.20:

```
SELECT    id, title
FROM      webpage
WHERE     title LIKE “Home%”
```

We can modify this so that it will only extract the id of these rows as follows:

```
SELECT     id
FROM       webpage
WHERE      title LIKE “Home%”
```

This will produce a value set of (1, 5, 7). To extract all of the log entries using this value set we can use the following:

```
SELECT     *
FROM       log
WHERE      webpageid IN (1,2,7)
```

However, in the above we are hard coding the value set that we are searching for. We are manually entering the values that we got for the first query into the second. If we were to change the value that we are searching for within the first query, then the value set returned may differ, and the results in the second query will be invalid unless we re-write the second query with the new values.

To automate the passing of values between these queries, we use the output of one query as a subquery in another query as follows:

```
SELECT     *
FROM       log
WHERE      webpageid IN (SELECT id FROM webpage WHERE title LIKE “Home%”)
```

The subquery must be contained in brackets and must return values that can be used by the parent query.

Joining Tables 7

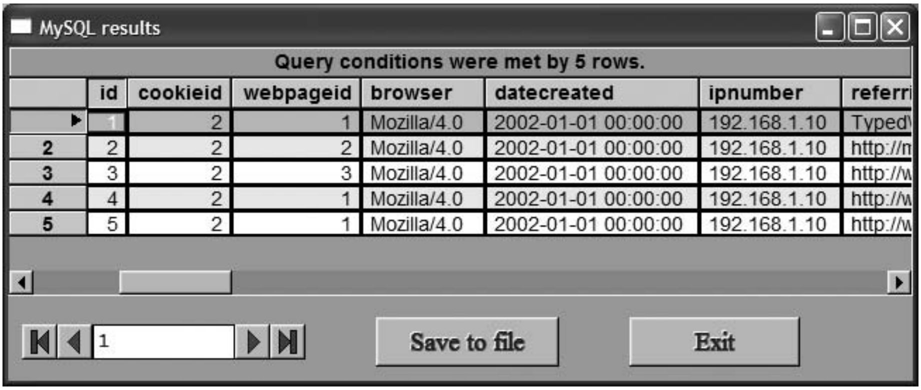
Information in Multiple Tables

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.



Query conditions were met by 5 rows.

	id	cookieid	webpageid	browser	datecreated	ipnumber	referri
▶	1	2	1	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	Typed\
2	2	2	2	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http://n
3	3	2	3	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http://w
4	4	2	1	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http://w
5	5	2	1	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http://w

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
```

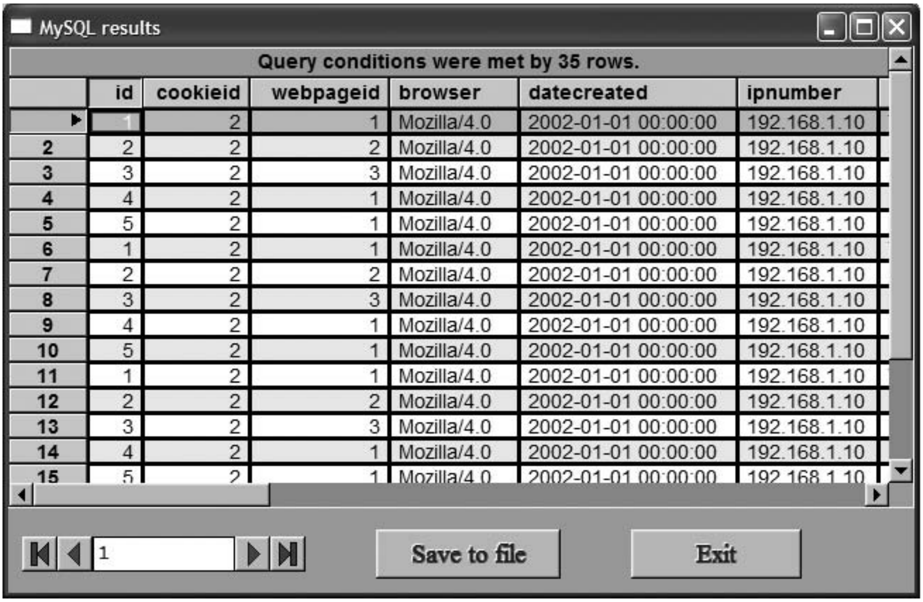
However, that seldom produces the effect that you might expect. Try joining the *Log* table to the *webpage* table as follows:

```
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 × 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



Query conditions were met by 35 rows.

	id	cookieid	webpageid	browser	datecreated	ipnumber
▶	1	2	1	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10
2	2	2	2	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10
3	3	2	3	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10
4	4	2	1	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10
5	5	2	1	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10
6	1	2	1	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10
7	2	2	2	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10
8	3	2	3	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10
9	4	2	1	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10
10	5	2	1	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10
11	1	2	1	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10
12	2	2	2	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10
13	3	2	3	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10
14	4	2	1	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10
15	5	2	1	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10

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.

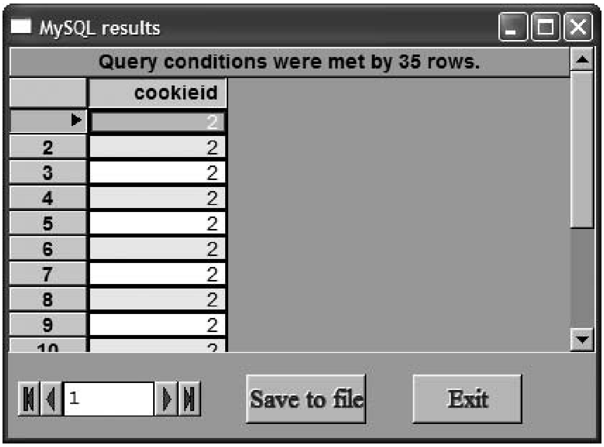
Inner Joins

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
```



Query conditions were met by 35 rows.

	cookieid
►	2
2	2
3	2
4	2
5	2
6	2
7	2
8	2
9	2
10	2

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
```

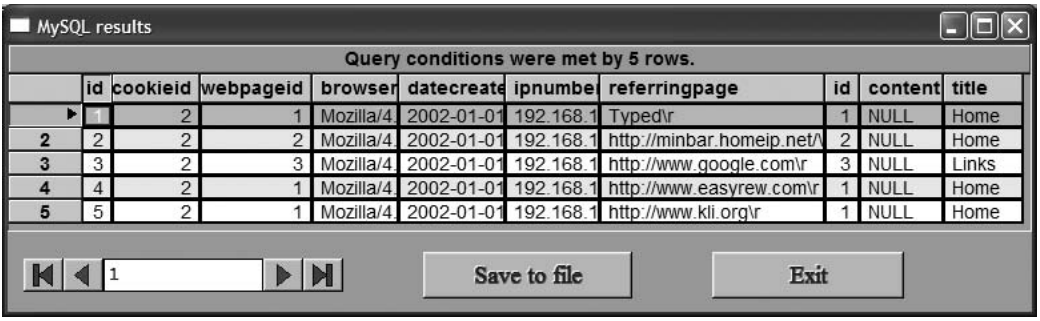
Figure 7.4 shows the results of the above query. Notice how the WHERE clause restricted the number of rows of the query to 5, and got all and more of the data that we want on each row.

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.



Query conditions were met by 5 rows.

	id	cookieid	webpageid	browser	datecreate	ipnumbe	referringpage	id	content	title
▶	1	2	1	Mozilla/4.	2002-01-01	192.168.1	Typed\r	1	NULL	Home
2	2	2	2	Mozilla/4.	2002-01-01	192.168.1	http://minbar.homeip.net/\	2	NULL	Home
3	3	2	3	Mozilla/4.	2002-01-01	192.168.1	http://www.google.com\r	3	NULL	Links
4	4	2	1	Mozilla/4.	2002-01-01	192.168.1	http://www.easyrew.com\r	1	NULL	Home
5	5	2	1	Mozilla/4.	2002-01-01	192.168.1	http://www.kli.org\r	1	NULL	Home

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
```

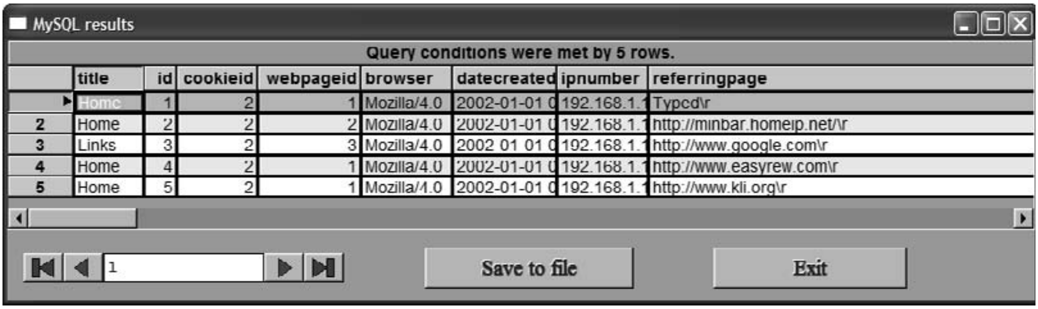
will just select the single column *title* from the *webpage* table. On a previous query we selected all columns from all tables using the asterisk. You can see that using the asterisk to select everything would be the same as using,

```
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
```

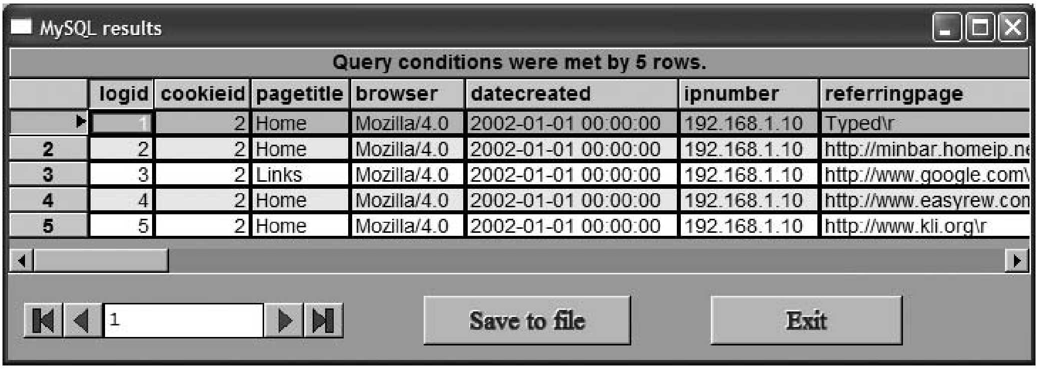
Figure 7.6 shows the results of our completed query. We've given the *log*'s *id* column the alias *logid* in the output to save it getting confused with any other *id* columns elsewhere. We



Query conditions were met by 5 rows.

	title	id	cookieid	webpageid	browser	datecreated	ipnumber	referringpage
▸	Homc	1	2	1	Mozilla/4.0	2002-01-01 0	192.168.1.1	Typcd\r
2	Home	2	2	2	Mozilla/4.0	2002-01-01 0	192.168.1.1	http://minbar.homeip.net/\r
3	Links	3	2	3	Mozilla/4.0	2002 01 01 0	192.168.1.1	http://www.google.com\r
4	Home	4	2	1	Mozilla/4.0	2002-01-01 0	192.168.1.1	http://www.easyrew.com\r
5	Home	5	2	1	Mozilla/4.0	2002-01-01 0	192.168.1.1	http://www.kli.org\r

Figure 7.5 An equi-join with selected columns.



Query conditions were met by 5 rows.

	logid	cookieid	pagetitle	browser	datecreated	ipnumber	referringpage
▸	1	2	Home	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	Typed\r
2	2	2	Home	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http://minbar.homeip.n
3	3	2	Links	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http://www.google.com\
4	4	2	Home	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http://www.easyrew.con
5	5	2	Home	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http://www.kli.org\r

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.

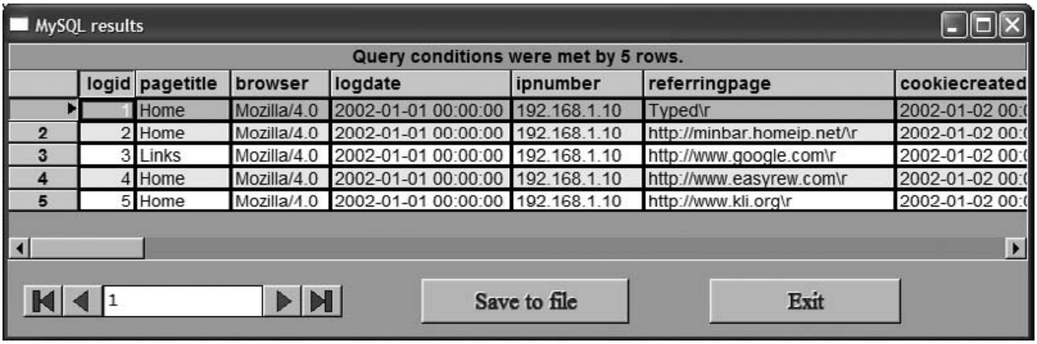
You will also notice that we do not have to return a column that we use in the match conditions. In our example, we used *webpage.id* in the match query but filtered that column out of the actual results by not specifying it in our output column list after the SELECT keyword.

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.



Query conditions were met by 5 rows.

	logid	pagetitle	browser	logdate	ipnumber	referringpage	cookiecreated
▶	1	Home	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	Typed\r	2002-01-02 00:
2	2	Home	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http://minbar.homeip.net/\r	2002-01-02 00:
3	3	Links	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http://www.google.com\r	2002-01-02 00:
4	4	Home	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http://www.easyrew.com\r	2002-01-02 00:
5	5	Home	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http://www.kli.org\r	2002-01-02 00:

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:

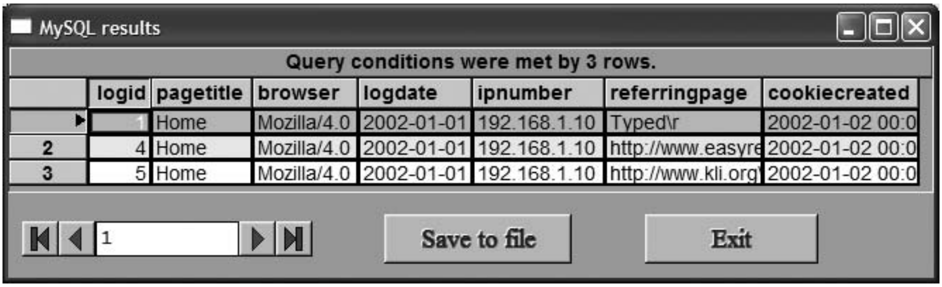
```
SELECT     log.id AS logid,
           webpage.title AS pagetitle,
           log.browser,
           log.datecreated AS logdate,
           log.ipnumber,
           log.referringpage,
           cookies.datecreated AS cookiecreated
FROM       log, webpage, cookies
WHERE      webpage.id = log.webpageid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



Query conditions were met by 3 rows.

	logid	pagetitle	browser	logdate	ipnumber	referringpage	cookiecreated
▶	1	Home	Mozilla/4.0	2002-01-01	192.168.1.10	Typed\r	2002-01-02 00:0
2	4	Home	Mozilla/4.0	2002-01-01	192.168.1.10	http://www.easyre	2002-01-02 00:0
3	5	Home	Mozilla/4.0	2002-01-01	192.168.1.10	http://www.kli.org\	2002-01-02 00:0

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:

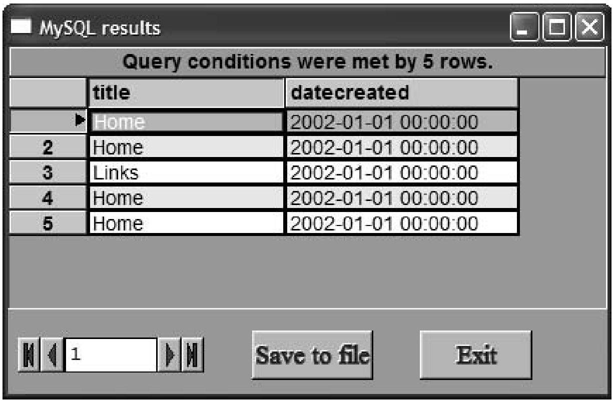
```
SELECT     columns
FROM       firsttable
           LEFT JOIN    secondtable ON
                        firsttable.column = secondtable.column
```

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.

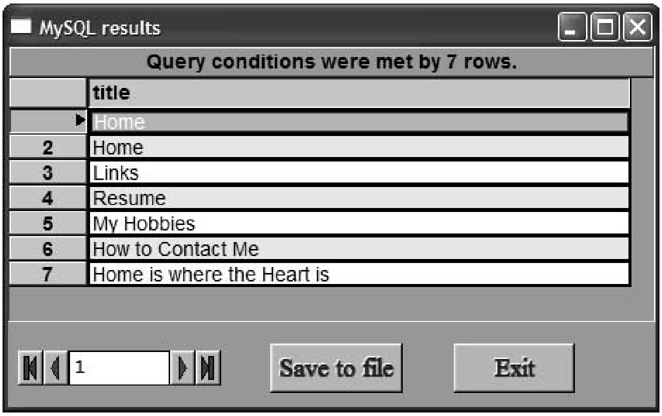
What we are trying to display is a list of all of the webpages that we have on our site and a relevant log entry for each page. However, if we look at Figure 7.9 we can see that we have



Query conditions were met by 5 rows.

	title	datecreated
▶	Home	2002-01-01 00:00:00
2	Home	2002-01-01 00:00:00
3	Links	2002-01-01 00:00:00
4	Home	2002-01-01 00:00:00
5	Home	2002-01-01 00:00:00

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



Query conditions were met by 7 rows.

	title
▶	Home
2	Home
3	Links
4	Resume
5	My Hobbies
6	How to Contact Me
7	Home is where the Heart is

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.

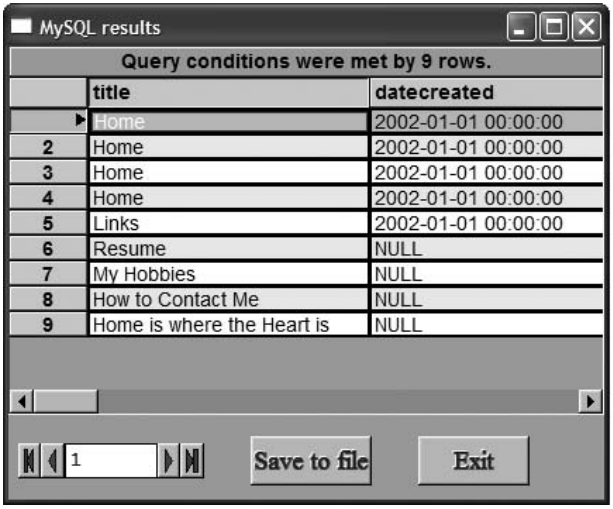
If we look at the query again, we can see why we call it a left join:

```
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
```



Query conditions were met by 9 rows.

	title	datecreated
▶	Home	2002-01-01 00:00:00
2	Home	2002-01-01 00:00:00
3	Home	2002-01-01 00:00:00
4	Home	2002-01-01 00:00:00
5	Links	2002-01-01 00:00:00
6	Resume	NULL
7	My Hobbies	NULL
8	How to Contact Me	NULL
9	Home is where the Heart is	NULL

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

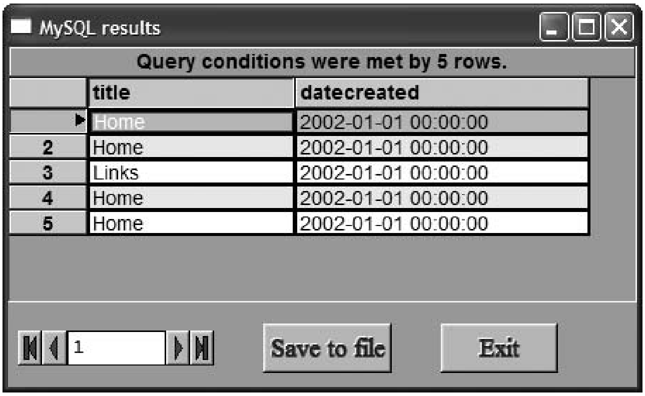
As we have a left join, it follows that we will also have a right join. The format of a right join is similar to that of a left join, as below:

```
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.



Query conditions were met by 5 rows.

	title	datecreated
▶	Home	2002-01-01 00:00:00
2	Home	2002-01-01 00:00:00
3	Links	2002-01-01 00:00:00
4	Home	2002-01-01 00:00:00
5	Home	2002-01-01 00:00:00

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
```

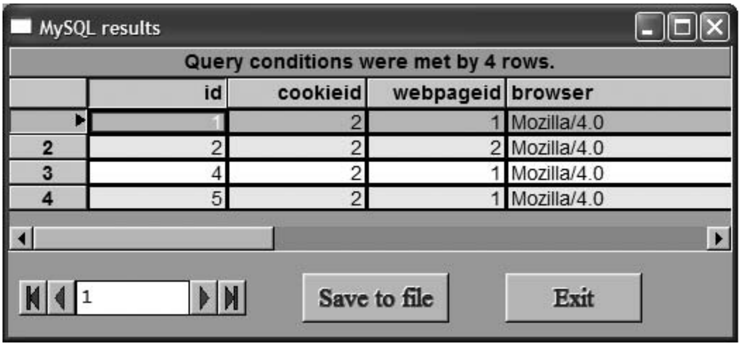
We can demonstrate this quickly by combining a query that selects all log IDs less than 3 from the log table with another that selects all log IDs that refer to the Home page as follows:

```
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:

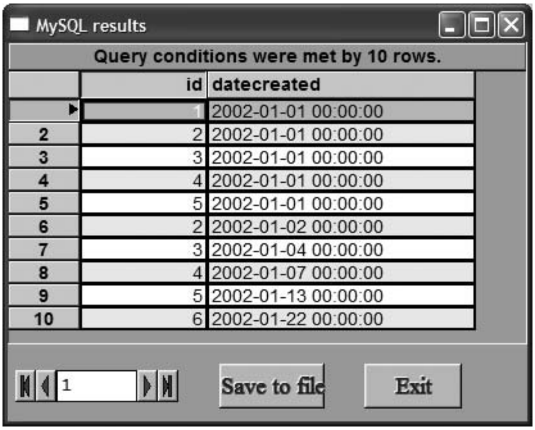
```
SELECT    *
FROM      log
WHERE     ID < 3 OR
          webpageID = 1
```



Query conditions were met by 4 rows.

	id	cookieid	webpageid	browser
▶	1	2	1	Mozilla/4.0
2	2	2	2	Mozilla/4.0
3	4	2	1	Mozilla/4.0
4	5	2	1	Mozilla/4.0

Figure 7.13 Joining two queries with UNION.



Query conditions were met by 10 rows.

	id	datecreated
▶	1	2002-01-01 00:00:00
2	2	2002-01-01 00:00:00
3	3	2002-01-01 00:00:00
4	4	2002-01-01 00:00:00
5	5	2002-01-01 00:00:00
6	2	2002-01-02 00:00:00
7	3	2002-01-04 00:00:00
8	4	2002-01-07 00:00:00
9	5	2002-01-13 00:00:00
10	6	2002-01-22 00:00:00

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.

8 Changing Data

Altering Data

Once you have populated your tables with data, that is seldom the end of the process. Although most websites accumulate much data that does not change over time, such as page access logs and cookie tables, other data changes. On a website that allows users to log on, for example, users may wish to change their password or their email address. On a database-driven website, changes to the contents of the pages will require changes to the underlying database that contains the pages.

SQL therefore has various commands that allow you to change the data that you have previously stored in your tables. This chapter contains some common commands that are used for this purpose:

- UPDATE allows you to change the contents of data within a table.
- ALTER allows you to make changes to the table structure once you have created it.
- REPLACE prevents errors when creating things that may already exist.

We will now describe these commands.

UPDATE

UPDATE allows you to change the contents of parts of a table without altering its structure. The basic format of the UPDATE command is as follows:

```
UPDATE    tablename
SET       columnname=numericvalue,
          columnname=”textvalue”
WHERE     condition
```

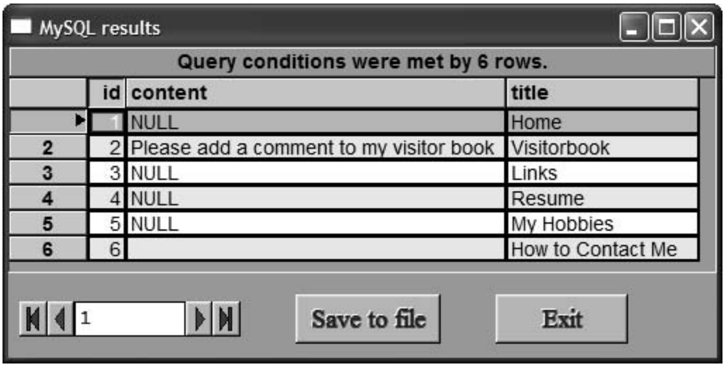
As you work through the book you might have found it frustrating that our *webpage* table contains hardly any content, just the name of each page. Select everything from the *webpage* table to remind yourself of its limited contents. Figure 8.1 shows the table.

You will notice from Figure 8.1 that the content column is largely filled with NULLs. We will use the UPDATE keyword to remedy this and give it some more relevant data. The following script will add some text to the content column:

```
UPDATE       webpage
SET          content = “My email is someone@nowhere.com”
```

The result of running the above is shown in Figure 8.2. You will notice that the UPDATE command does not give you the normal results grid, as the command changes data but does not return it. The only feedback that you get is in the status bar at the bottom of the box.

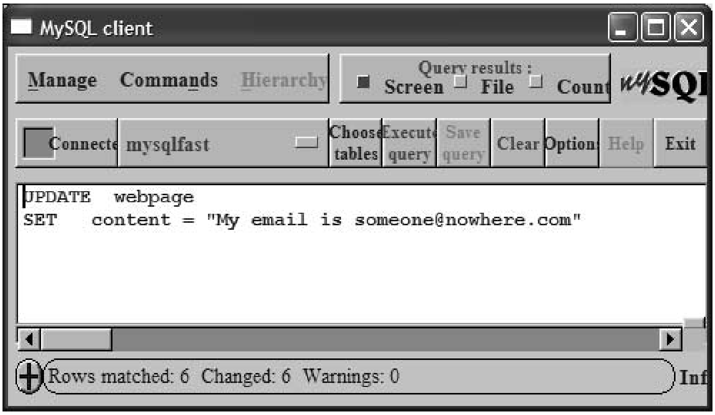
You will see in Figure 8.2 that for the above query it has matched and changed six rows. This could be a problem as we only wanted to change the one row for the *Contact Me* page. This example has been used to show how dangerous the UPDATE query can be without the WHERE clause. Select everything from the *webpage* table again and you will see the results of the unrestricted UPDATE, as in Figure 8.3. All of the content column cells have been replaced with our email address.



Query conditions were met by 6 rows.

	id	content	title
▶	1	NULL	Home
2	2	Please add a comment to my visitor book	Visitorbook
3	3	NULL	Links
4	4	NULL	Resume
5	5	NULL	My Hobbies
6	6		How to Contact Me

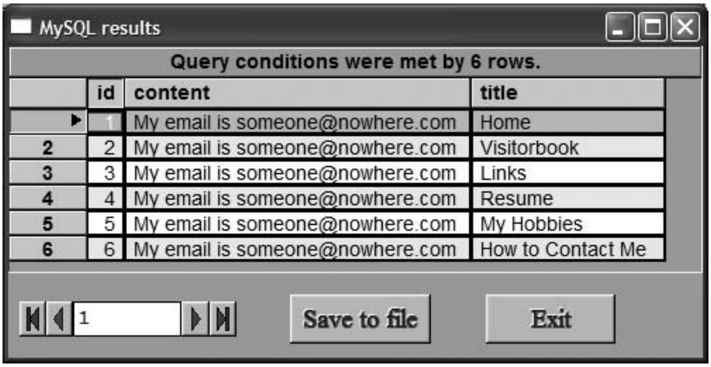
Figure 8.1 Our content lacking webpage table.



```
UPDATE   webpage
SET   content = "My email is someone@nowhere.com"
```

Rows matched: 6 Changed: 6 Warnings: 0

Figure 8.2 When updating, feedback appears on the last line.



Query conditions were met by 6 rows.

	id	content	title
▶	1	My email is someone@nowhere.com	Home
2	2	My email is someone@nowhere.com	Visitorbook
3	3	My email is someone@nowhere.com	Links
4	4	My email is someone@nowhere.com	Resume
5	5	My email is someone@nowhere.com	My Hobbies
6	6	My email is someone@nowhere.com	How to Contact Me

Figure 8.3 The damage caused by omitting the WHERE.

Thankfully, our example database only has six rows, only one of which had some meaningful data, so we will not have to do much work to fix this problem. Imagine, however, the amount of work that would be involved if we had used a similar query on a commercial website with hundreds or thousands of pages! This example shows that it is best to get into the habit of always using a WHERE clause with the UPDATE query. Try to think of the query as *UPDATE*, *SET*, *WHERE* as opposed to *UPDATE*, *SET* and it may save you a lot of work. I hope this is the last time you will see this happen and you will not make this mistake when it matters!

To fix this, we will first clear out the wrong data by using the UPDATE without the WHERE for one last time:

```
UPDATE     webpage
SET        content = “”
```

The above query sets all of the cells in the content column to the empty string. Now we will re-run the UPDATE query for row 6 but this time use the WHERE clause:

```
UPDATE     webpage
SET        content = “My email is someone@nowhere.com”
WHERE      ID = 6
```

After running the query above, select everything from the *webpage* table and you should have the more satisfactory results as shown in Figure 8.4. We have still lost the data that was in row 2 but we will remedy that next. Notice that by using the WHERE clause to match the primary key field (*ID*) of the table we were able to restrict our update to exactly the row that we wished to change.

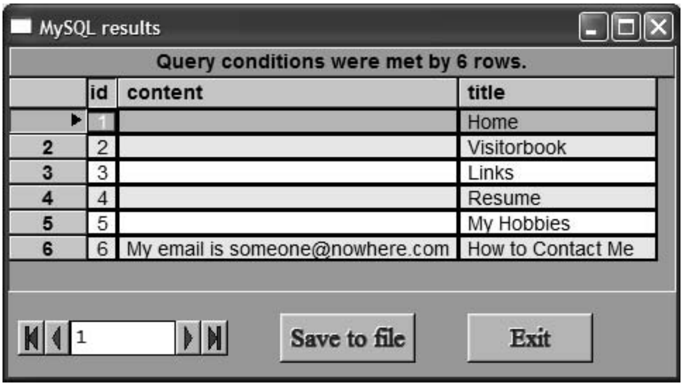
Now that we have successfully changed the data in one column, we can add some content to all of the rows in the *webpage* table one by one. The following script will do this for you:

```
UPDATE    webpage
SET       content = “Welcome to my MySQL based website”
WHERE     ID = 1;

UPDATE    webpage
SET       content = “Please add a comment to my visitor book”
WHERE     ID = 2;

UPDATE    webpage
SET       content = “Here are some links you may find of use”
WHERE     ID = 3;

UPDATE    webpage
SET       content = “Here is what I have done with my work life up till now”
WHERE     ID = 4;

UPDATE    wbpage
SET       content = “My hobbies are, web programming, anorak spotting and
            fish collecting “
WHERE     ID = 5;
```

Figure 8.5 shows what the webpage table will look like after you have selected everything from it again. Our website’s database now has some content.

ALTER

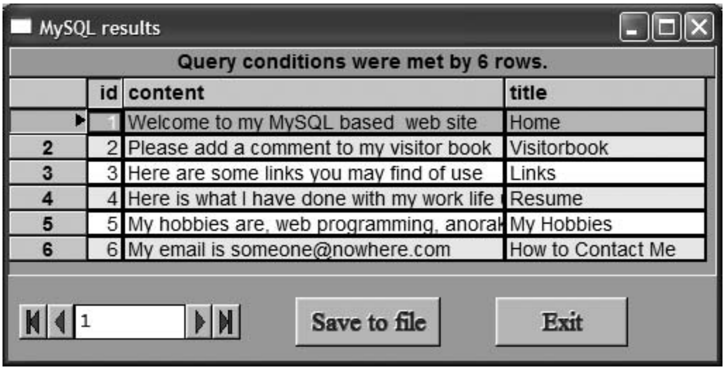
The ALTER command is a strange command to have in SQL as it can be argued that it is not needed. This command allows you to change the structure of a table after it has been created. The argument is that if you have correctly completed the analysis and design of your database, you should not need to alter its structure at a later time. However, the real world is seldom like this, we make mistakes, and goals and requirements change, so it is a good



Query conditions were met by 6 rows.

	id	content	title
►	1		Home
2	2		Visitorbook
3	3		Links
4	4		Resume
5	5		My Hobbies
6	6	My email is someone@nowhere.com	How to Contact Me

Figure 8.4 Fixing our unconstrained UPDATE.



Query conditions were met by 6 rows.

	id	content	title
▶	1	Welcome to my MySQL based web site	Home
2	2	Please add a comment to my visitor book	Visitorbook
3	3	Here are some links you may find of use	Links
4	4	Here is what I have done with my work life (	Resume
5	5	My hobbies are, web programming, anorak	My Hobbies
6	6	My email is someone@nowhere.com	How to Contact Me

Figure 8.5 The correctly updated webpage table.

thing that we have the ALTER command. The general format of an ALTER query is as follows:

```
ALTER TABLE   tablename
commandtype   columnname parameters
```

The ALTER TABLE command can do several things, depending on what you type in the *commandtype* position. Commandtype can be one of the following:

- ADD adds a new property or column to the table.
- RENAME changes the name of the table.
- CHANGE changes the name of a column and its datatype.
- MODIFY, changes the datatype of a column.
- DROP removes the column from the table completely.

The next few sections will demonstrate these different types of ALTER query with our *visitorbook* table. To remind ourselves of this table, run the following:

```
DESCRIBE    visitorbook
```

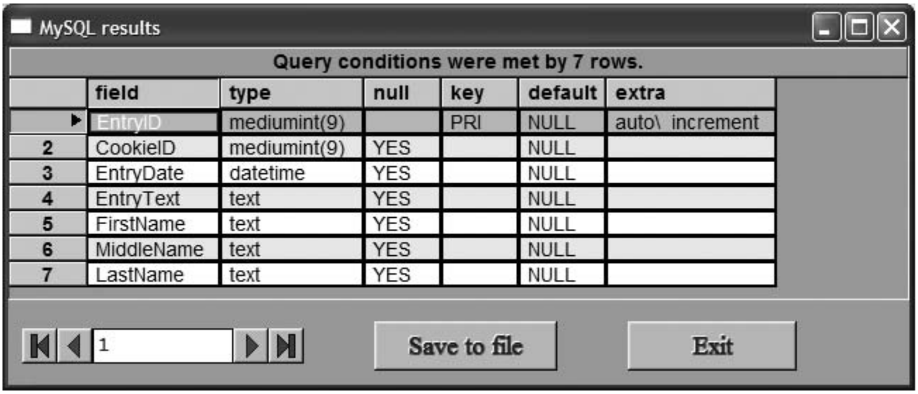
Figure 8.6 shows the current format of the *visitorbook* table. As we created this in a rush and not in our chapter on the design of databases, we made a few mistakes. We will change some of the columns and add an extra one using ALTER.

As a final reminder, Figure 8.7 shows the current content of the table, after selecting everything from it.

ALTER TABLE ADD

The ALTER TABLE ADD query has the following basic format:

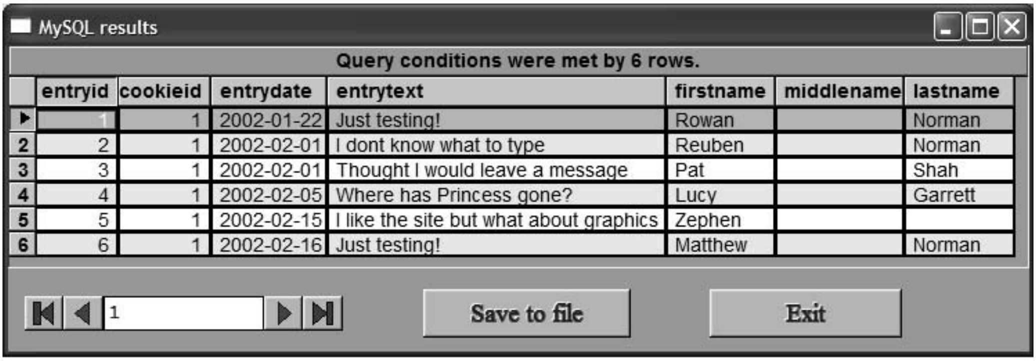
```
ALTER TABLE   tablename
ADD           columnname datatype
```



Query conditions were met by 7 rows.

	field	type	null	key	default	extra
▶	EntryID	mediumint(9)		PRI	NULL	auto\ increment
2	CookieID	mediumint(9)	YES		NULL	
3	EntryDate	datetime	YES		NULL	
4	EntryText	text	YES		NULL	
5	FirstName	text	YES		NULL	
6	MiddleName	text	YES		NULL	
7	LastName	text	YES		NULL	

Figure 8.6 The current format of the visitorbook table.



Query conditions were met by 6 rows.

	entryid	cookieid	entrydate	entrytext	firstname	middlename	lastname
▶	1	1	2002-01-22	Just testing!	Rowan		Norman
2	2	1	2002-02-01	I dont know what to type	Reuben		Norman
3	3	1	2002-02-01	Thought I would leave a message	Pat		Shah
4	4	1	2002-02-05	Where has Princess gone?	Lucy		Garrett
5	5	1	2002-02-15	I like the site but what about graphics	Zephen		
6	6	1	2002-02-16	Just testing!	Matthew		Norman

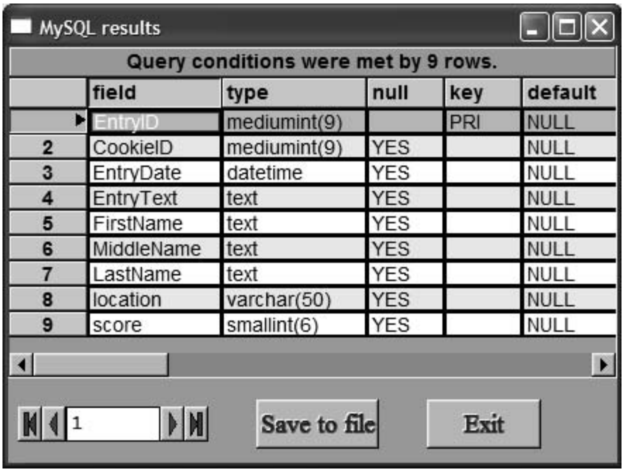
Figure 8.7 The current content of the visitorbook table.

On our visitorbook page on the website, it would be nice to ask the visitor to give us some idea of where they are based geographically. We will store this in a location column. We will also allow them to rate our site with a score between 1 and 20. For the location column, we need to store a string. It is hard to guess the length of this directly but let us say this would be no more than 50 characters long. Therefore we use the following to add a location column to the table:

```
ALTER TABLE   visitorbook
ADD           location VARCHAR(50)
```

For the score column, we will only be storing a limited range of numbers, so we can just use a SMALLINT datatype to store the score. We add the extra column as follows:

```
ALTER TABLE   visitorbook
ADD           score SMALLINT
```



MySQL results

Query conditions were met by 9 rows.

	field	type	null	key	default
▶	EntryID	mediumint(9)		PRI	NULL
2	CookieID	mediumint(9)	YES		NULL
3	EntryDate	datetime	YES		NULL
4	EntryText	text	YES		NULL
5	FirstName	text	YES		NULL
6	MiddleName	text	YES		NULL
7	LastName	text	YES		NULL
8	location	varchar(50)	YES		NULL
9	score	smallint(6)	YES		NULL

Figure 8.8 The two new columns in the visitorbook table.

You may notice from the above that the format to add a table follows the same format as when we were creating a table. We can add other parameters to the command, should they be needed, that allow the column to become a foreign key, or allow nulls. For our examples we do not need to specify any additional parameters. Have a look at the chapter on creating databases to remind you of these extra parameters if you need to. Describe the *webpage* table again and you will see the results shown in Figure 8.8.

You will see from Figure 8.8 that the visitorbook table now contains the two new rows.

The ADD keyword can also be used to add other things to the table, such as primary keys, as well as indexes. To add a primary key to a table that does not already have one, use:

```
ALTER TABLE   tablename
ADD           PRIMARY KEY columnname
```

Of course, the column that you are setting the primary key for must have unique data in each field for the column to be specified as such. To add an index to a column use:

```
ALTER TABLE   tablename
ADD           INDEX indexname (columnname)
```

ALTER TABLE RENAME

The ALTER TABLE RENAME query allows you to change the name of a table. It has the following basic format:

```
ALTER TABLE   tablename
RENAME        newtablename
```

Therefore, to rename the *visitorbook* table to *visitorsbook*, we would execute the following query:

```
ALTER TABLE   visitorbook
RENAME        visitorsbook
```

And to change it back:

```
ALTER TABLE   visitorsbook
RENAME        visitorbook
```

RENAME only allows you to change the name of the whole table. One way that you can change the name of a column is by using CHANGE

ALTER TABLE CHANGE

The ALTER TABLE CHANGE query allows you to change the name of a column as well as its datatype. It has the following basic format:

```
ALTER TABLE   tablename
CHANGE        oldcolumnname newcolumnname datatype
```

CHANGE requires you to specify the old and new column names even if you are not changing the name of the column, in which case they can both be the same. Likewise, it requires you to specify the datatype even if you are not changing it.

For example, our column *lastname* could be confusing to some cultures, so we will change it to *familyname* instead. To do this, run the following query:

```
ALTER TABLE   visitorbook
CHANGE        lastname familyname TEXT
```

Notice how we still have to specify the datatype even though we are not changing it. While we are looking at that column though, do you see a problem with the datatype? We are using a TEXT field, which can contain strings of varying length, to store a string that in all probability is only tens of characters long. Let us make the arbitrary decision that a family name will never be longer than 40 characters, so change the type to a VARCHAR using the following script:

```
ALTER TABLE   visitorbook
CHANGE        familyname familyname VARCHAR(40)
```

The CHANGE command allows you to change the column name and datatype at the same time as well by specifying the two different column names and the new datatype.

You could change the *firstname* column to a VARCHAR as well with the following:

```
ALTER TABLE   visitorbook
CHANGE        firstname firstname VARCHAR(40)
```

However, the next command shows an easier way to change the datatype if that is the only change you wish to make to the column.

ALTER TABLE MODIFY

MySQL has given us the ALTER TABLE MODIFY query that saves us a bit of typing if we are only going to change the datatype. This has the following basic format:

```
ALTER TABLE   tablename
MODIFY        columnname datatype
```

To use MODIFY to change the datatype of a column we no longer have to type the column name twice, so to change the firstname column we can use:

```
ALTER TABLE   visitorbook
MODIFY        firstname VARCHAR(40)
```

We have not looked at the format of our table for a while, so Figure 8.9 shows our changes to date with a describe visitorbook query.

ALTER TABLE DROP

The ALTER TABLE DROP query has the following basic format:

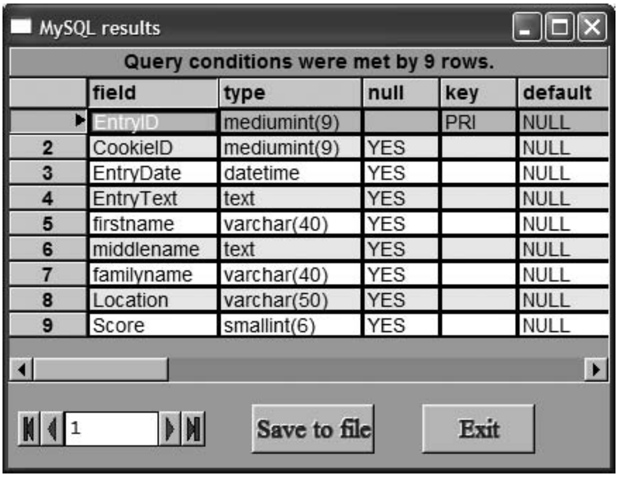
```
ALTER TABLE   tablename
DROP          columnname, columnname
```

You can delete one column or several by separating the column names with commas.

Looking at Figure 8.9 may make you wonder why we did not change the datatype of the *middlename* column. This column is awaiting a worse fate! If you turn back to Figure 8.7 you will notice that there is no data being stored at all in the *middlename* column, and with hindsight, we have no real need to obtain that information from our website visitors. Currently that will be just wasting space within our database. We can remove the *middlename* column by using the following script:

```
ALTER TABLE   visitorbook
DROP          middlename
```

If you run that script you may be startled by how quickly it drops the column. Bear in mind that when you remove columns it is not like a modern interface which happily asks



Query conditions were met by 9 rows.

	field	type	null	key	default
▶	EntryID	mediumint(9)		PRI	NULL
2	CookieID	mediumint(9)	YES		NULL
3	EntryDate	datetime	YES		NULL
4	EntryText	text	YES		NULL
5	firstname	varchar(40)	YES		NULL
6	middlename	text	YES		NULL
7	familyname	varchar(40)	YES		NULL
8	Location	varchar(50)	YES		NULL
9	Score	smallint(6)	YES		NULL

Figure 8.9 Our ALTERed visitorbook table.

you if you are sure that you want to delete something. There are no second chances, all of the column's data will be destroyed when you drop it as soon as you issue the command.

You can also use the ALTER TABLE DROP query to remove an index or a primary key from a table. To remove the primary key use:

```
ALTER TABLE   tablename
DROP          PRIMARY KEY
```

This does not remove the actual column, but removes the primary key flag from that column.

To remove an index use:

```
ALTER TABLE   tablename
DROP INDEX    indexname
```

Dropping the primary key or the index on a table is not as dangerous as dropping a whole column, as when you do this you will not lose any data, and the index or primary key can be added back on again afterwards if you need to.

Get it Right the First Time!

You may have noticed a lot of disk activity when you were running some of these ALTER TABLE commands. This is because it is very hard for the database system to change a table that has already been created. The way that it gets around this problem can be very processor intensive, as the server creates a new table with the changed column, copies the data from the old table to the new one, deletes the old table and finally renames the new table to the same name as the old one! On a big table, running an ALTER command may take a long time, all the more reason why you should endeavour to create the table accurately in the first instance.

Many of these commands are also a MySQL addition to the original ANSI SQL92 definition, so may not work on other database management systems that are only compliant with this standard.

UPDATE Revisited

I previously warned of the dangers of doing an unrestricted update, that is, an UPDATE without the WHERE clause. However, I will now show you a time when such an update is useful. When we were adding columns to our visitorbook table we added a score column. It would be useful to add some data to that column for use later on. We could just set all of the columns to the same value, but instead we will use the data stored in one column to 'seed' the data in the score column. Again, we could just copy from one column to the next but we will change it slightly by multiplying the source column to get the new one. This sounds complex, so run the following script:

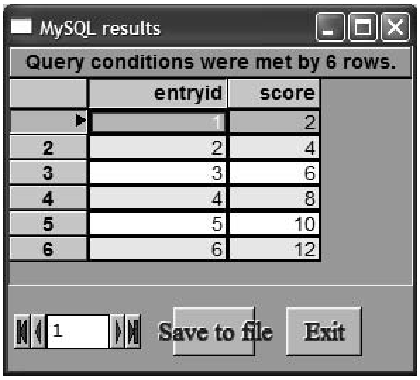
```
UPDATE    visitorbook
SET       score=entryID * 2
```

For each row, this query will take the value of the entryID field, multiply it by 2, and store it in the score field of that same row. To view the results, run the following:

```
SELECT    entryID, score
FROM      visitorbook
```

Figure 8.10 shows the results of our update.

Looking at Figure 8.10, you can see that we now have a different score in each row of the table, and each score is twice the value of the *entryID*. We just used the *entryID* as a changing number to seed the score column, to show that an unrestrained UPDATE does not necessarily have to result in all the updated fields having the same value.



MySQL results

Query conditions were met by 6 rows.

	entryid	score
▶	1	2
2	2	4
3	3	6
4	4	8
5	5	10
6	6	12

Figure 8.10 UPDATE calculating values for each row.

Although we are using this example to 'fix' the score of our website, this type of update can be useful if we realize that we have accumulated a lot of data that needs changing. For instance, in an e-commerce system, you may have been storing the total cost for each order without sales tax, and then decide that it should be with tax. It would be easy enough to change your PHP code to store the total value plus the tax, but what about the data already in the table? You can create a new column in the table for the new total, use an unrestricted update to multiply the old total column by the tax amount and place the data in the new column. You then drop the old column, rename the new one and your data is now stored correctly.

REPLACE

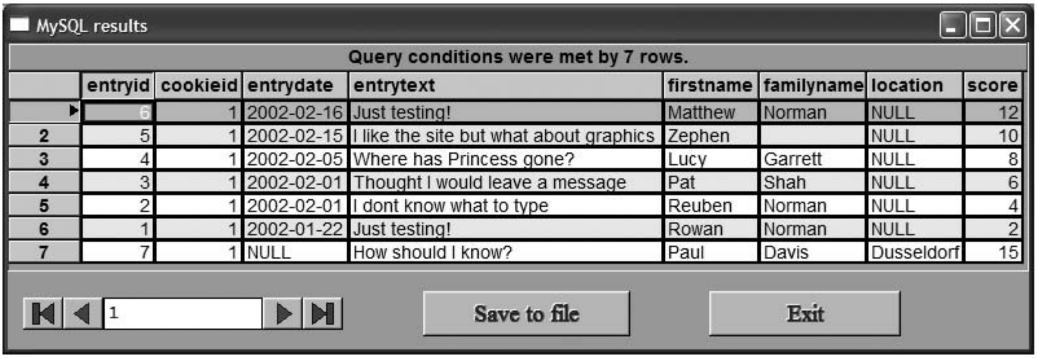
REPLACE will insert data into a table, but if you supply a primary key that is the same as one that is already in the table, then REPLACE will replace that row. It has the same format as the INSERT query, and also can be used in a similar way to the UPDATE query, but without the WHERE clause. The first format for REPLACE is as follows:

```
REPLACE   table ( column, column …)
VALUES    value, value …
```

If one of the columns that you specify is the primary key for the table, and the value of that field does not match a key that is already in that table, then the above works exactly as a standard INSERT query. If, however, the primary key matches one that already exists, it will delete that row, and replace it with the data that you have just provided. To demonstrate, we will add a new row to the visitorbook table using REPLACE, and then change that row using REPLACE a second time. Run the following:

```
REPLACE   visitorbook ( entryid,
                        cookieid,
                        firstname,
                        familyname,
                        entrytext,
                        score,
                        location)
VALUES    (7,
          1,
          'Paul',
          'Davis',
          'How should I know?',
          15,
          'Dusseldorf')
```

Now select everything from the visitorbook table. The results are shown in Figure 8.11. Before adding this row the last row's primary key was 6, so specifying 7 for this row has done exactly the same as an INSERT would have.



Query conditions were met by 7 rows.

	entryid	cookieid	entrydate	entrytext	firstname	familyname	location	score
►	6	1	2002-02-16	Just testing!	Matthew	Norman	NULL	12
2	5	1	2002-02-15	I like the site but what about graphics	Zephen		NULL	10
3	4	1	2002-02-05	Where has Princess gone?	Lucy	Garrett	NULL	8
4	3	1	2002-02-01	Thought I would leave a message	Pat	Shah	NULL	6
5	2	1	2002-02-01	I dont know what to type	Reuben	Norman	NULL	4
6	1	1	2002-01-22	Just testing!	Rowan	Norman	NULL	2
7	7	1	NULL	How should I know?	Paul	Davis	Dusseldorf	15

Figure 8.12 REPLACE removes unspecified fields.

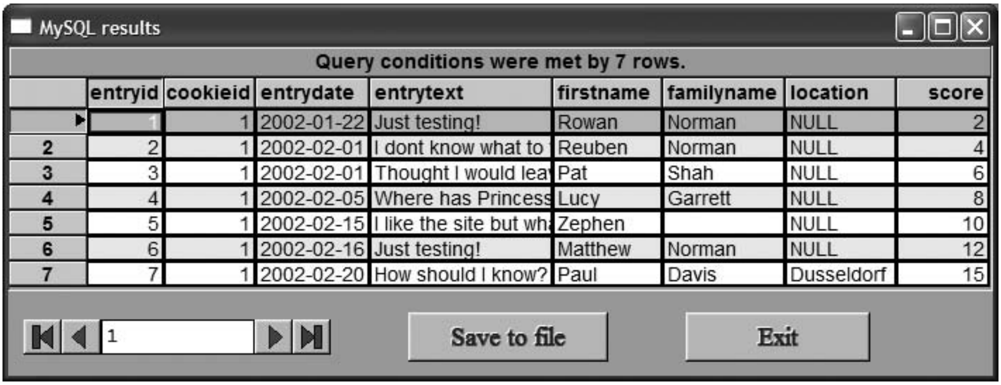
You may notice that we missed out the date on the last entry. If you have used the UPDATE query, the temptation would be to run the following to add a date to the row:

```
REPLACE   visitorbook (entryid, entrydate)
VALUES    (7, '2002-02-20')
```

However, Figure 8.12 shows the results of this. Notice that instead of replacing the entrydate field in the row, it has replaced the whole row, but only added the entrydate to the new row that it has inserted. This shows the difference between UPDATE and REPLACE. UPDATE can selectively change a field within a row, whereas REPLACE will delete the row that it is replacing, and create a new row with only the fields that you specify.

So to add the date to the row we just added with REPLACE, we need to essentially run the whole query again as follows:

```
REPLACE   visitorbook ( entryid,
                        entrydate,
                        cookieid,
                        firstname,
                        familyname,
                        entrytext,
                        score,
                        location)
VALUES    (7,
          '2002-02-20',
          1,
          'Paul',
          'Davis',
          'How should I know?',
          15,
          'Dusseldorf')
```



MySQL results

Query conditions were met by 7 rows.

	entryid	cookieid	entrydate	entrytext	firstname	familyname	location	score
▶	1	1	2002-01-22	Just testing!	Rowan	Norman	NULL	2
2	2	1	2002-02-01	I dont know what to	Reuben	Norman	NULL	4
3	3	1	2002-02-01	Thought I would lea	Pat	Shah	NULL	6
4	4	1	2002-02-05	Where has Princess	Lucy	Garrett	NULL	8
5	5	1	2002-02-15	I like the site but wh	Zephen		NULL	10
6	6	1	2002-02-16	Just testing!	Matthew	Norman	NULL	12
7	7	1	2002-02-20	How should I know?	Paul	Davis	Dusseldorf	15

Figure 8.13 Replacing a row with REPLACE.

Figure 8.13 shows the results of running the above query. We now have a row that has all of the fields populated.

We can also run the REPLACE query with the alternative format:

```
REPLACE    table
SET        column=value,
           column=value
```

I personally prefer using this format of the command as it saves you having to spend time ensuring that your column list is in the same order as your value list.

CREATE or REPLACE

You can also use the REPLACE command when you are creating a table, function or datatype. When you are creating one of these objects, if you use:

```
CREATE OR REPLACE object parameters
```

instead of just CREATE, the command will delete an existing object before creating a new one. This stops an error if the object already exists, and also stops you having to delete the object first if you do want to replace it.

DUPLICATE KEY UPDATE

When you are inserting data into a table with a primary key, you can have problems if you insert a key that already exists. For example, if we were inserting a new row into our visitor book shown in Figure 8.13, we could use the following:

```
INSERT into visitorbook (entryid, entrydate, cookieid, firstname,
                         familyname, entrytext, score, location)
VALUES    (7,’2002-02-20’,1,’Paul’,’Davis’,’How should I
know?’,15,’Dusseldorf’)
```

However, you will notice that we already have a row with the primary key of 7, so this insert will stop with an error as follows:

Duplicate entry ‘7’ for key 1

There is a new clause in MySQL version 4.1 that allows the insert to update the duplicate key if the primary key already exists. To use this we re-write the query as follows:

```
INSERT into visitorbook (entryid, entrydate, cookieid, firstname,
                         familyname, entrytext, score, location)
VALUES    (7,’2002-02-20’,1,’Paul’,’Davis’,’How should I
          know?’,15,’Dusseldorf’)
ON DUPLICATE KEY UPDATE entryid = entryid + 1
```

In this case, the INSERT will run without an error, and the row that used to have a primary key of 7 will have been replaced with a primary key of 8. This works in this example because before the insert a row with the primary key of 8 did not exist. If there was such a row then a similar error would be generated for row 8.

This page intentionally left blank

9

Deleting our Data

Keeping Data Accurate

After spending so much time designing our databases and putting data into them, it seems a pity to discuss deleting them, but deleting records and tables is still an important part of using MySQL. If we are trying to keep our data valid and accurate we will have to remove data from time to time. This chapter will show you how to remove rows from tables, and then how to remove whole tables.

You may remember back in Chapter 5 we created the *webpage2* table by inserting data from the *webpage* table. If for some reason you no longer have the *webpage2* table in your database, you can create it again quickly by running the following:

```
CREATE TABLE webpage2 (ID MEDIUMINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
                       Content text, Title text)
```

Now copy the data into this table by executing the following:

```
INSERT INTO webpage2
SELECT * from webpage
```

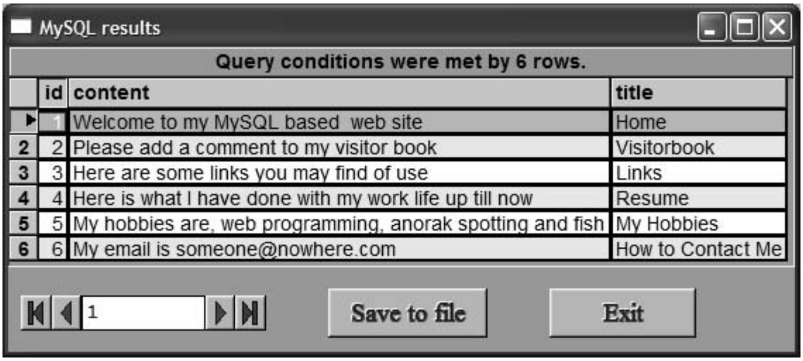
Select everything from *webpage2* and you will see the sacrificial table in Figure 9.1.

DELETE FROM

To DELETE rows from a table, you use the following query:

```
DELETE FROM    tablename
WHERE          condition
```

As we demonstrated earlier, this command will quickly and unsympathetically remove everything from a table without a WHERE clause, so be careful when you are using it. For instance, if we wanted to delete the visitorbook row from our *webpage2* table we would use the following:



Query conditions were met by 6 rows.

	id	content	title
▶	1	Welcome to my MySQL based web site	Home
2	2	Please add a comment to my visitor book	Visitorbook
3	3	Here are some links you may find of use	Links
4	4	Here is what I have done with my work life up till now	Resume
5	5	My hobbies are, web programming, anorak spotting and fish	My Hobbies
6	6	My email is someone@nowhere.com	How to Contact Me

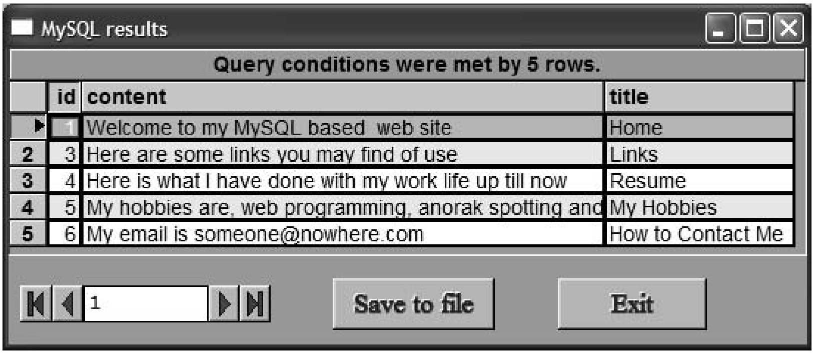
Figure 9.1 The webpage2 table.

```
DELETE FROM    webpage2
WHERE          id=2
```

As you see from Figure 9.2, this has cleanly removed the row. As we were referring to a row specifically by using its primary key, it was safe enough to delete it as we did.

The problem comes when you want to delete a few rows using some pattern matching. You must be sure that you know what you are about to remove. One of the ways to ensure that you are deleting the right rows is to try and select them with the same criteria first. For instance, if we wanted to delete the *How to contact me* row with pattern matching, test the match first with a SELECT as follows:

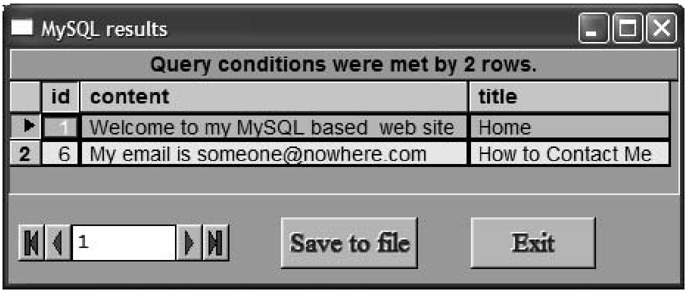
```
SELECT     *
FROM       webpage2
WHERE      title LIKE ‘Ho%’
```



Query conditions were met by 5 rows.

	id	content	title
▶	1	Welcome to my MySQL based web site	Home
2	3	Here are some links you may find of use	Links
3	4	Here is what I have done with my work life up till now	Resume
4	5	My hobbies are, web programming, anorak spotting and	My Hobbies
5	6	My email is someone@nowhere.com	How to Contact Me

Figure 9.2 The webpage2 table after removing a row.



Query conditions were met by 2 rows.

	id	content	title
▶	1	Welcome to my MySQL based web site	Home
2	6	My email is someone@nowhere.com	How to Contact Me

Figure 9.3 The webpage2 table after deleting rows using pattern matching.

You will notice from Figure 9.3 that if we were to have deleted using this match we would have removed two rows. Was this what you wanted to do with the match? A simple deletion of a row may have resulted in our removing the Home page of our website.

Finally, you can use DELETE to completely clear all of the rows from a table as follows:

```
DELETE FROM webpage2
```

Run the above and then try to select everything from the table. You will just receive a message saying that your query produced no results. Describing *webpage2* shows that the table is still there, but it no longer has any data in it.

DROP TABLE

There comes a time when you not only need to delete the contents of a table but you need to remove the table as well. To do this, use the DROP command as follows:

```
DROP TABLE tablename
```

So to finally remove all traces of *webpage2*, execute the following query:

```
DROP TABLE webpage2
```

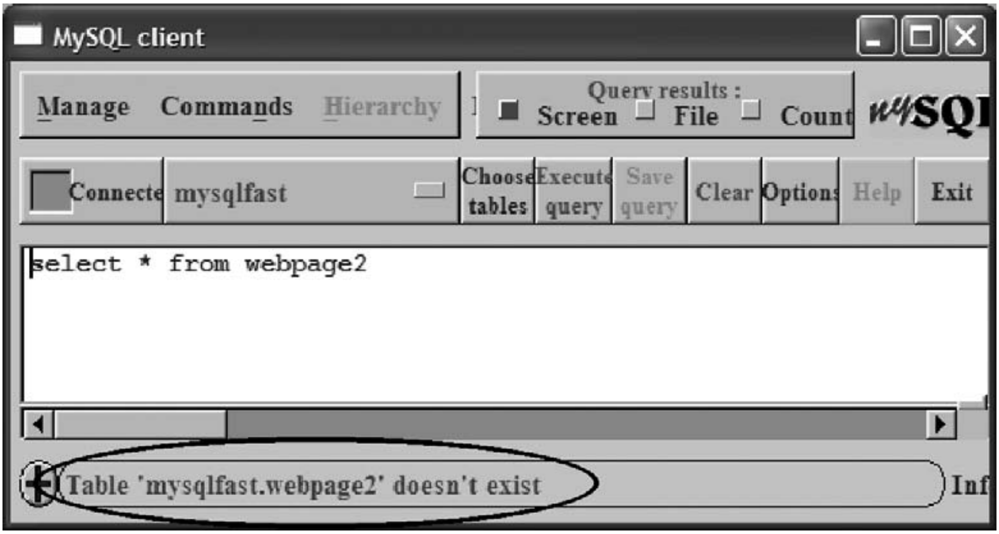
If you now try and select anything from *webpage2*, you will get the error shown in Figure 9.4.

You can DROP multiple tables by specifying a list of them as follows:

```
DROP TABLE table1, table2, table3, etc
```

If you are using a MySQL table type that checks for referential integrity, the DROP TABLE command may generate errors if other tables make reference to rows in the table being dropped. To drop a table and all others that reference it, you would use:

```
DROP TABLE tablename CASCADE
```



```
select * from webpage2
```

Table 'mysqlfast.webpage2' doesn't exist

Figure 9.4 The webpage2 table has been dropped.

The CASCADE does not function in the current version of MySQL but if you are using this book to learn SQL for other systems it is useful to know about. Newer versions of MySQL may well implement this function.

DROP

The DROP command also can be used to delete entire databases. To do this you omit the TABLE keyword as follows:

```
DROP        databasename
```

We will not demonstrate this command because we want to use our database for the rest of the book!

Aggregate Functions 10

Count

Aggregate functions in MySQL allow you to calculate a summary of values from a column and from groups of values within a column. As usual, the best way to demonstrate this concept is with an example.

Count in MySQL is a function that allows you to count the number of specified items in a table. One way that you can use count is as follows:

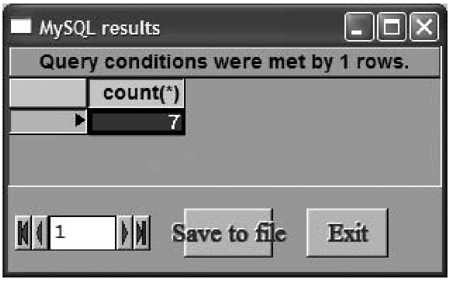
```
SELECT     count(columnname)
FROM       tablename
```

We can use the above statement to count the number of rows within a table by using the following script:

```
SELECT     count(*)
FROM       visitorbook
```

If you execute that query you will get the very simple results shown in Figure 10.1.

You may wonder why you would bother performing a query like this when you can just look at the top row of the output to tell you how many rows came from your query. However, when you are accessing these functions from within a script language, like PHP, then it is more useful. For instance, the following script counts all of the hits in our log table, thus producing a counter for the whole of the website:



Query conditions were met by 1 rows.

count(*)
7

Figure 10.1 Counting rows in a table.

```
SELECT    count(*) as PageHits
FROM      log
```

Figure 10.2 shows the result of that query. Notice that we used an alias to make the output a little bit more meaningful than the column title that the first query showed in Figure 10.1. If you were to write a script to execute that query on the first page of your website, then it would be a wonderful way of implementing a website page counter. That query will show all of the hits on all of the pages of our site, as long as you have remembered to make each page log a hit to it in the database!

If you wanted to implement a page counter for each individual page, then we can use the same script but this time restrict the search with a WHERE clause like this:

```
SELECT    count(*) as PageHits
FROM      log
WHERE     webpageID=2
```

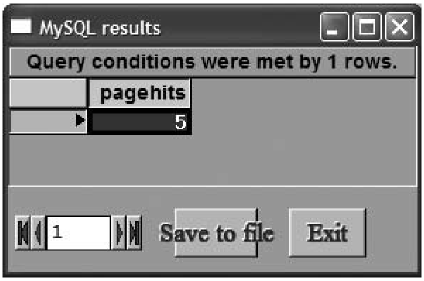
For instance, in our *webpage* table, the *webpageID* of the visitorbook page is 2, so the script above will count the number of times that an entry has been put in the database for that page. Executing that query on each page where we want a counter to appear will enable the same table, log, to be used to generate counters for every page on the site.

Now we have encountered our first aggregate function, it’s time to show you something that makes it even more powerful, the GROUP BY clause.

GROUP BY

GROUP BY is a clause that is added after a select statement that allows you to get extra functionality from an aggregate function. The GROUP BY clause has to be placed after a where clause if you include one. You use the GROUP BY clause as follows:

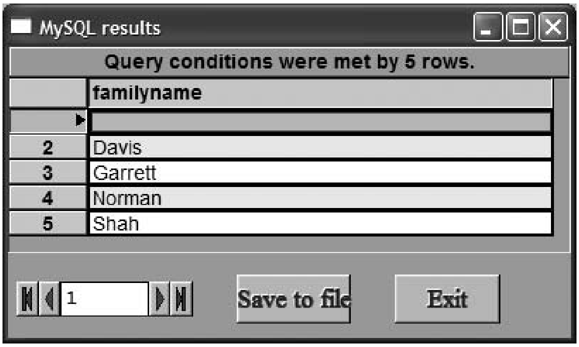
```
SELECT    aggregatefunction(columnname), columns
FROM      tablename
GROUP BY  columnname
```



Query conditions were met by 1 rows.

	pagehits
▶	5

Figure 10.2 Counting all the hits on our website.



Query conditions were met by 5 rows.

	familyname
▶	
2	Davis
3	Garrett
4	Norman
5	Shah

Figure 10.3 GROUP BY without an aggregate.

On its own, the GROUP BY clause does not do much. Consider the results of the following query shown in Figure 10.3:

```
SELECT    familyname
FROM      visitorbook
GROUP BY  familyname
```

If this query stops with an error it is probably because you have not followed scripts in Chapter 8, where we renamed the lastname column to familyname.

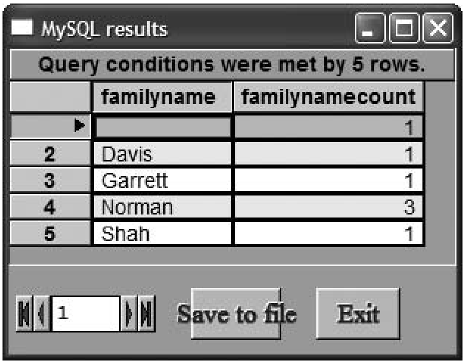
At first glance Figure 10.3 looks as if the GROUP BY has done little more than a SELECT DISTINCT, in that it has picked the unique list of last names from the visitor book. If you remember from the last time that we looked at that table there were three Normans, a Garrett, a Davis, a Shah and someone with no last name. The query has just picked one instance of each of them. The power of the GROUP BY clause comes when we attach it to a query with an aggregate function. For example, take the following:

```
SELECT    familyname, count(familyname) as familynamecount
FROM      visitorbook
GROUP BY  familyname
```

Figure 10.4 shows the results of that query. GROUP BY has done more than just select the unique entries of the *familyname* column, it has grouped each similar entry into a set of values that the aggregate functions can work with.

To demonstrate further, if we wanted to apply the GROUP BY clause to the query that counted website hits, we can use:

```
SELECT    id, count(*)
FROM      log
GROUP BY  webpageid
```



Query conditions were met by 5 rows.

	familyname	familynamecount
▶		1
2	Davis	1
3	Garrett	1
4	Norman	3
5	Shah	1

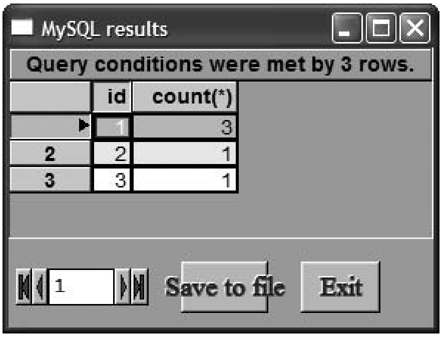
Figure 10.4 Counting groups with GROUP BY.

Figure 10.5 shows the results. Notice that on this occasion we have not grouped by the clause that we are counting, but that does not matter, the count is really just counting rows, not columns. So the query is saying: "Give me the number rows of the table that refer to each specific webpage".

Now this is getting powerful; the simple query has shown that our Home page has had three hits, and our *visitorbook* and *links* page have each received one hit. Imagine how useful this data would be for a table that contained tens of thousands of hits on hundreds of webpages. If we were performing such a query and wanted to restrict the results to the first 10 pages, then we would add a WHERE clause as follows:

```
SELECT     id, count(*)
FROM       log
WHERE      webpageid <=10
GROUP BY   webpageid
```

Notice the position of the WHERE clause. If you put it after the GROUP BY you get an error message.



Query conditions were met by 3 rows.

	id	count(*)
▶	1	3
2	2	1
3	3	1

Figure 10.5 Counting groups with GROUP BY.

Of course, the results in Figure 10.5 are not very user friendly. To get a more readable output, try the following query:

```
SELECT     webpage.title, count(*) as numberofhits
FROM       log LEFT JOIN webpage
                    ON log.webpageid=webpage.id
GROUP BY   log.webpageid
```

Figure 10.6 shows what happens when we use the aggregate GROUP BY along with a LEFT JOIN to get the webpage names and an alias to get a meaningful column name.

We have now confirmed the results that we had guessed at in Figure 10.5, by making our output more readable and meaningful. We could use the output of this query with little reformatting on a webpage that showed the full statistics for our website.

MAX

The MAX function calculates the maximum value from a set of numeric values. One way of using this function is:

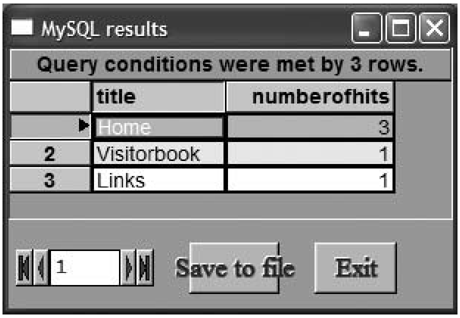
```
SELECT     MAX(score)
FROM       visitorbook
```

This will find the highest value of score that someone has entered into the visitor book. You can see the result of running this query in Figure 10.7.

If you have a column that uses AUTONUMBER to create its ID as a primary key, you can use a function to tell which row was the last added, by selecting the highest ID as follows:

```
SELECT     MAX(cookieID)
FROM       cookies
```

If for some reason you want to manually add a key to the table without using AUTONUMBER, you can modify the above query as follows:



Query conditions were met by 3 rows.

	title	numberofhits
▶	Home	3
2	Visitorbook	1
3	Links	1

Figure 10.6 A count, GROUP BY, alias and LEFT JOIN.

```
SELECT     MAX(cookieID) +1
FROM       cookies
```

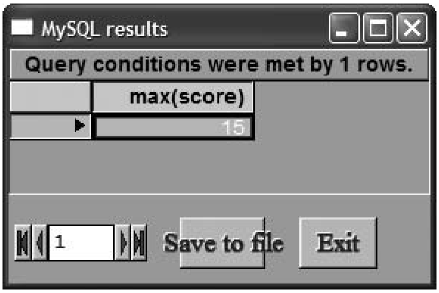
This will add one to the highest primary key that there is, and you can insert the results straight into the table as the new primary key.

MIN

The MIN function is the opposite of MAX in that it calculates the minimum value from a set of numeric values. So to find the lowest score that our visitors gave our website we would use:

```
SELECT     MIN(score)
FROM       visitorbook
```

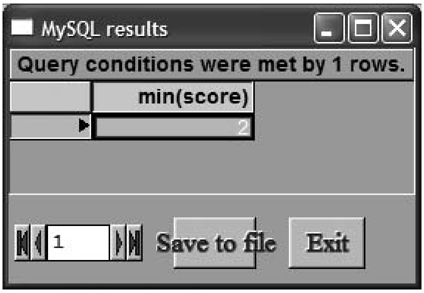
The results are shown in Figure 10.8. The minimum score that any of our visitors has given the website yet is 2 out of 20.


MySQL results

Query conditions were met by 1 rows.

max(score)
15

Figure 10.7 Finding the maximum of a set of values.


MySQL results

Query conditions were met by 1 rows.

min(score)
2

Figure 10.8 Finding the minimum of a set of values.

SUM

SUM finds the total value by adding together a set of numeric values. To add up all of the scores that our visitors gave our website we would use:

```
SELECT     SUM(score)
FROM       visitorbook
```

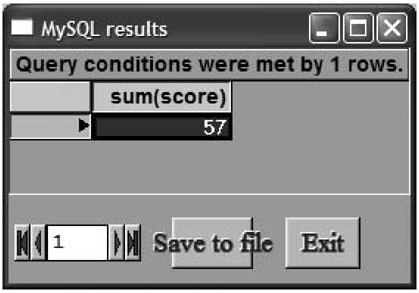
The results are shown in Figure 10.9.

Using Multiple Aggregates

You are not limited to just using one aggregate in a query. For instance, if we wanted to calculate the average score given by a visitor, we can use an aggregate to sum the score divided by a count of the scores given, as follows:

```
SELECT     SUM(score) / COUNT(score) AS average
FROM       visitorbook
```

Have a look at Figure 10.10 to see the average score for our site based on the data that was in the tables.

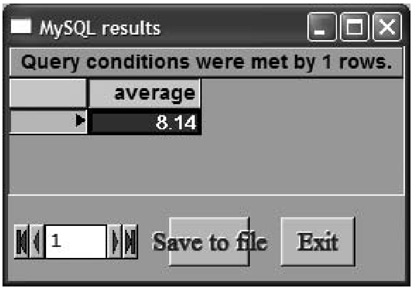


MySQL results

Query conditions were met by 1 rows.

	sum(score)
▶	57

Figure 10.9 The sum of the scores on the visitorbook.



MySQL results

Query conditions were met by 1 rows.

	average
▶	8.14

Figure 10.10 The average of the scores on the visitorbook.

We can even use the following to find the average score given by people of the same family name:

```
SELECT     familyname, SUM(score) / COUNT(score) AS average
FROM       visitorbook
GROUP BY   familyname
```

Figure 10.11 shows the query above in action. It gives the name and works out the average score for each name. It also shows that my family are not showing favouritism by giving me top marks! You may notice from the query that the select statement asks for a standard column name, *familyname*, and a number derived from aggregate statements. Although it is quite acceptable to ask for multiple aggregates, as follows:

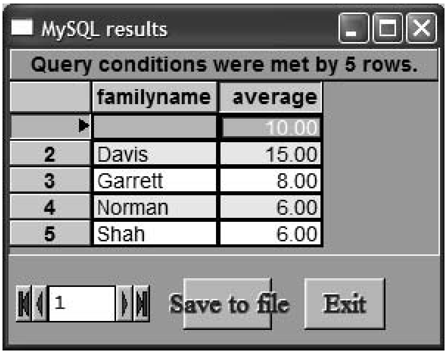
```
SELECT     SUM(score) AS sum,
           COUNT(score) AS count,
           max(score) AS max
FROM       visitorbook
```

you are not allowed to mix column names and aggregates in the same selection set without using GROUP BY. For instance, the following script will generate the error shown in Figure 10.12:

```
SELECT     familyname, SUM(score) / COUNT(score) AS average
FROM       visitorbook
```

The full error states, "Mixing of GROUP columns (MIN(),MAX(),COUNT()...) with no GROUP columns is illegal if there is no GROUP BY clause."

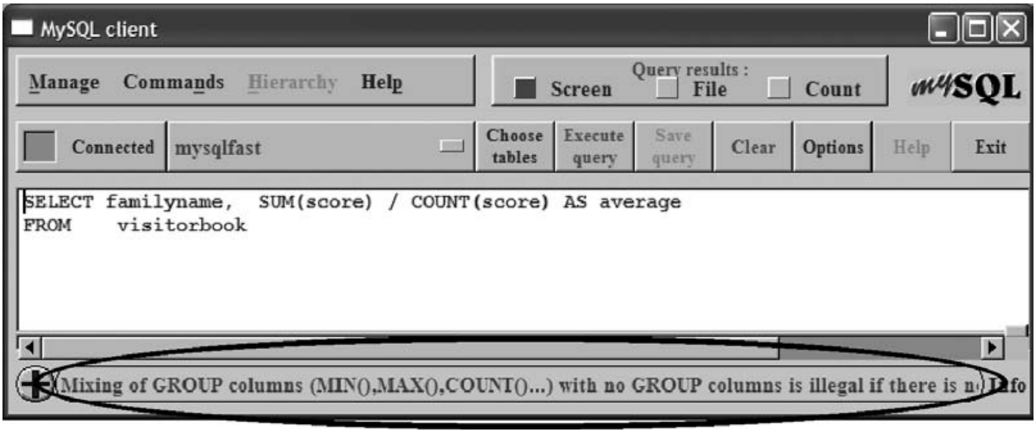
So although you can use multiple aggregates together in a SELECT statement, if you are mixing them with column names you must remember to include a GROUP BY clause to get the query to work.



Query conditions were met by 5 rows.

	familyname	average
▶		10.00
2	Davis	15.00
3	Garrett	8.00
4	Norman	6.00
5	Shah	6.00

Figure 10.11 The average of the scores for each family.



```
SELECT familyname,  SUM(score) / COUNT(score) AS average
FROM    visitorbook
```

Mixing of GROUP columns (MIN(),MAX(),COUNT()...) with no GROUP columns is illegal if there is n

Figure 10.12 Columns mixed with aggregates must use GROUP BY.

AVG

Although we calculated the average manually in the previous section, it is such a useful tool that there is a built-in function in MySQL. The AVG function calculates the average value of a set of numeric values. So to find the average that our visitors gave our website we would use:

```
SELECT    AVG(score)
FROM      visitorbook
```

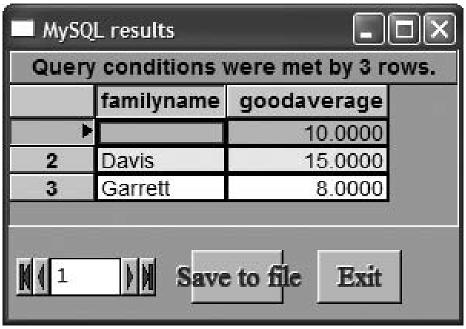
This query will give similar results to those shown in Figure 10.10, although to a higher precision than the manual calculation.

HAVING

When using the WHERE clause in an aggregate query, we have already mentioned that it has to be placed before the GROUP BY clause as it restricts the query before it has been sorted into groups. The HAVING clause is placed after the GROUP BY clause, and allows you to restrict after the group and by an aggregate. Examine the following query:

```
SELECT    familyname, AVG(score) AS goodaverage
FROM      visitorbook
GROUP BY  familyname
HAVING    AVG(score) >= 8
```

Similar to the previous query, this one calculates and displays the average score for each family, but then restricts the results, using the HAVING clause, to groups whose average is



Query conditions were met by 3 rows.

	familyname	goodaverage
▶		10.0000
2	Davis	15.0000
3	Garrett	8.0000

Figure 10.13 The average of the scores for each family.

greater than or equal to 8. These results are shown in Figure 10.13, and you can compare these with the results of the query without the HAVING clause that are shown in Figure 10.11.

Working with Dates and Times 11

Using Date and Time

Putting a date and time column in a database is a simple task. Populating a row with the current date and time is even easier in many scripting languages. However, getting the date and time back, although sometimes confusing, can be an extremely powerful way of analysing your data, and getting more from your tables.

NOW

To obtain the current date and time, use the NOW() function. The function requires the open and close brackets even though you are not passing any value to it. To quickly obtain the current date and time you can use a SELECT as follows:

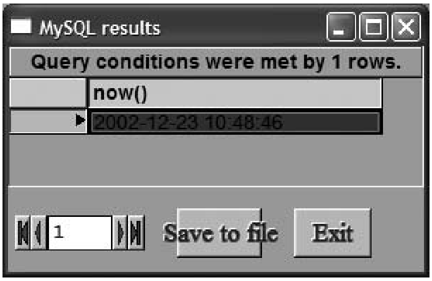
```
SELECT NOW()
```

This will produce the output shown in Figure 11.1.

When we created the Log table, we manually filled it with sample data. When the Log table is in use on the system, we will want the date and time to be automatically added into the table, so we can use the following to accomplish this:

```
INSERT INTO    Log       ( DateCreated,
                         Browser,
                         IPNumber,
                         CookieID,
                         WebPageID)
               VALUES    ( NOW(),
                         'Mozilla/4.0',
                         '192.168.1.10',
                         2,
                         1)
```

The above will use the NOW() function to insert the current system date and time into the newly created row. If you have been keeping up with the examples, this should have created a row with the auto-ID of 6, so to view this row we can issue the following query:



Query conditions were met by 1 rows.

now()
2002-12-23 10:48:46

Figure 11.1 SELECT NOW().

```
SELECT     id, datecreated
FROM       log
WHERE      ID= 6
```

Figure 11.2 shows the results, yours will show a different date and time!

CURRENT_DATE

CURRENT_DATE gives you the system date for today. It differs from NOW() in that it does not require the brackets and only outputs the date. You can use the following to find the current date quickly:

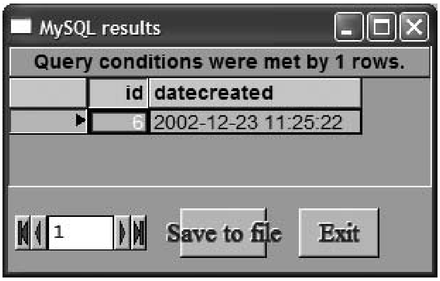
```
SELECT       CURRENT_DATE
```

Figure 11.3 shows the way that CURRENT_DATE is output in a query.

CURRENT_TIME

CURRENT_TIME returns the system time. Again it differs from NOW() in that it does not require the brackets and only outputs the time. You can use the following to find the current time quickly:

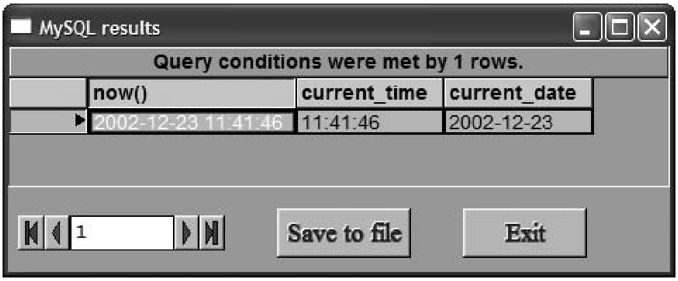
```
SELECT       CURRENT_TIME
```



Query conditions were met by 1 rows.

id	datecreated
6	2002-12-23 11:25:22

Figure 11.2 An inserted NOW().



Query conditions were met by 1 rows.

now()	current_time	current_date
2002-12-23 11:41:46	11:41:46	2002-12-23

Figure 11.3 Selecting the current date and time.

Figure 11.3 shows the way that CURRENT_TIME is output in a query. You can insert the current time into a table using a similar technique to that which we demonstrated with the NOW() function.

YEAR

YEAR() allows you to extract the year from a given date. You need to pass a date to the YEAR function as follows:

```
YEAR(aDate)
```

aDate is any object that contains a date part. So we could use the following to output the current year in a select statement:

```
SELECT     YEAR( NOW() )
           AS This_year_is
```

Figure 11.4 shows this query along with the next function. We have passed NOW() in as the date, which will actually pass in the date and time, but the YEAR() function ignores the extra data and works with just the date portion of the NOW() output.

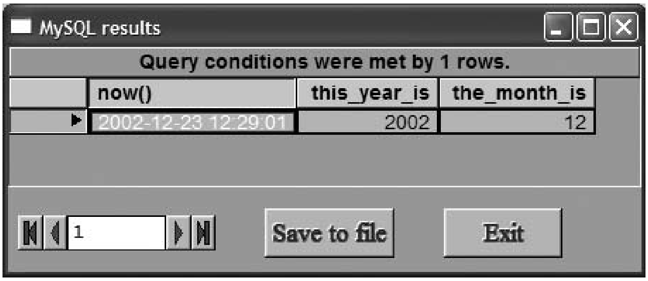
MONTH

MONTH() works in the same way as YEAR() but outputs the numerical value corresponding to the month of the date given. The function takes the following format:

```
MONTH(aDate)
```

aDate is any object that contains a date part. So we could use the following to output the current month number in a select statement:

```
SELECT     MONTH( NOW() )
           AS The_month_is
```



MySQL results

Query conditions were met by 1 rows.

now()	this_year_is	the_month_is
2002-12-23 12:29:01	2002	12

Figure 11.4 Selecting the current year and month.

As previously, we have combined this query with the results from the YEAR function which are shown in Figure 11.4.

MONTHNAME

Using MONTH() to find the month number as shown above is fine if you are wanting to perform some month calculations, but if we are presenting this to the user we may wish to output the month name in words. We do this by using the MONTHNAME() function, which takes the following format:

```
MONTHNAME(aDate)
```

As usual, **aDate** is any object that contains a valid date. We will demonstrate this by a different query this time. The following query will show the months of all the entries in the log table:

```
SELECT     MONTHNAME(datecreated)
FROM       Log
```

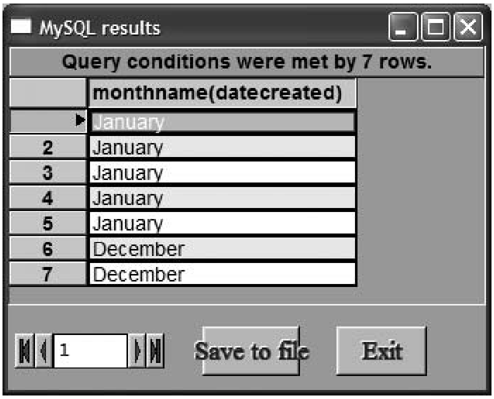
Figure 11.5 shows our log table with the month, as opposed to the exact date and time, that each entry was created. Notice how on this occasion we inserted the current row's *DateCreated* field into the function as the date. The query has processed the function for each row in the table. Depending on how closely you have followed the INSERT statements in this book, your table may have more or fewer rows.

DAYNAME

DAYNAME() will return the day that a given date falls on. You need to pass the function a date as follows:

```
SELECT     DAYNAME( NOW() )
```

Figure 11.6 shows the results when combined with the next two examples. You will see from the figure that I am writing this on 23 December 2002, which is a Monday.



Query conditions were met by 7 rows.

	monthname(datecreated)
▶	January
2	January
3	January
4	January
5	January
6	December
7	December

Figure 11.5 MONTHNAME() outputting the log table.

DAYOFWEEK

DAYOFWEEK() will return a number based on the day that the current date falls on from Table 11.1.

Table 11.1 DAYOFWEEK returned values.

Day	Value returned
Sunday	1
Monday	2
Tuesday	3
Wednesday	4
Thursday	5
Friday	6
Saturday	7

DAYOFWEEK needs you to pass it an object with a date component, as with other date functions that we have discussed previously. To obtain the numeric day of the week for today, you would issue the following query:

```
SELECT      DAYOFWEEK( NOW() )
```

Figure 11.6 shows the results from that query. You can see that the day is Monday and its DAYOFWEEK value is therefore 2.

WEEKDAY

WEEKDAY() works in a similar way as DAYOFWEEK() except it indexes the day names in a different way. It returns values for the day of the week based on the values in Table 11.2.

Table 11.2 WEEKDAY returned values.

Day	Value returned
Monday	0
Tuesday	1
Wednesday	2
Thursday	3
Friday	4
Saturday	5
Sunday	6

To obtain the numeric weekday for today, you would issue the following query:

```
SELECT      WEEKDAY( NOW() )
```

Figure 11.6 shows the results from that query. You can see that the day is Monday and its WEEKDAY VALUE is therefore 0.

DAYOFMONTH

The DAYOFMONTH() function returns the day number of a given month. We use the function as follows:

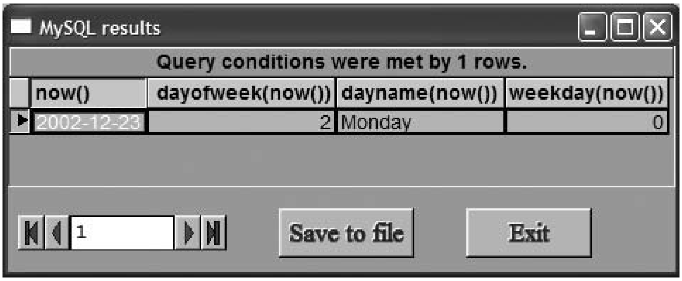
```
DAYOFMONTH( aDate )
```

By passing the output of the NOW() function into DAYOFMONTH, you can obtain the current day number. Figure 11.7 shows this at the time of writing, which is 23 December.

DAYOFYEAR

The DAYOFYEAR() function returns a count of the number of days since 1 January for a given date. We use the function as follows:

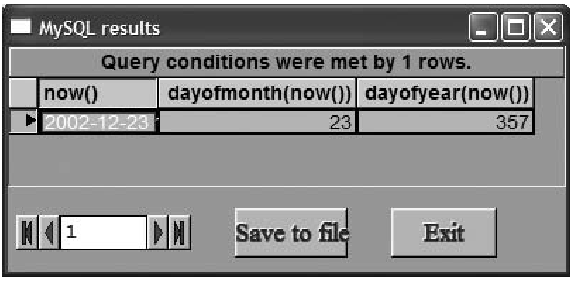
```
DAYOFYEAR( aDate )
```



Query conditions were met by 1 rows.

now()	dayofweek(now())	dayname(now())	weekday(now())
2002-12-23	2	Monday	0

Figure 11.6 DAYNAME(), DAYOFWEEK() and WEEKDAY().



Query conditions were met by 1 rows.

now()	dayofmonth(now())	dayofyear(now())
2002-12-23	23	357

Figure 11.7 DAYOFMONTH() and DAYOFYEAR().

aDate is any object that contains a date part. By passing the output of the NOW() function into DAYOFYEAR, you can obtain the number of days since the start of the year for a given date. Figure 11.7 shows the output of the DAYOFYEAR() function when used in a SELECT statement. DAYOFYEAR() is useful when calculating with dates, as it provides a numerical version of the date to work with. You can use functions like this to calculate things like ages, although there are other functions that make this easier.

WEEK

WEEK() returns the number of weeks since the start of the year for a given date. As usual, all that it requires as a parameter is an object with a date part as follows:

```
WEEK( aDate )
```

So, to find the current week number we would use:

```
SELECT      WEEK( NOW() )
```

Figure 11.8 shows the WEEK() function in action. You can see that it is nearly the last week of the year at time of writing.

Incidentally, depending on the day that the year starts on, it is possible to have the beginnings of a week 53, so if 53 is returned from the WEEK() function, it is not an error.

Using the above format, WEEK() calculates the week number using the assumption that the first day of the week is Sunday. If you want the calculation to be based on a Monday, you use the amended format:

```
WEEK( aDate, 1 )
```

There are other peculiarities when the week falls around the year end; is it the first week of this year or the last week of last year? Have a look at the Date and Time Functions section of the MySQL online manual for further details.

YEARWEEK

YEARWEEK() returns the year and the week number as one big number. You use the function by passing it a value which has a date component as follows:

```
YEARWEEK( aDate )
```

Figure 11.8 shows this function working in week 51 of 2002. I am sure that this function is of use to someone; however, I have yet to find a use for it myself. It would be interesting to find out what others are using it for.

QUARTER

QUARTER() returns a number ranging between 1 and 4 depending on the part of the year that it is given. The number returned is based on Table 11.3.

Table 11.3 QUARTER returned values.

Between:	And:	Value returned
1 January	31 March	1
1 April	30 June	2
1 July	30 September	3
1 October	31 December	4

QUARTER() works like other date functions in that you supply it an object with a date part as follows:

```
QUARTER( aDate )
```

Figure 11.8 shows the function working in the fourth quarter of 2002.

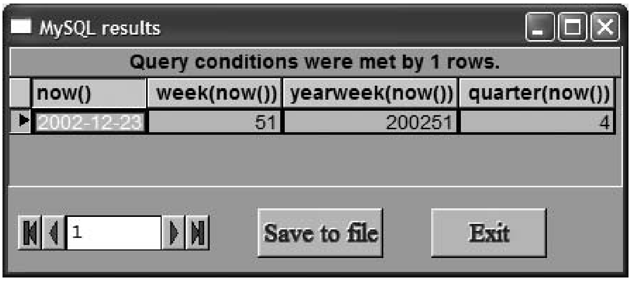
HOUR

HOUR() returns the hour for a given time object, based on the 24-hour clock. You pass the time value to it as follows:

```
HOUR( aTime )
```

To obtain the current hour, we can use the following:

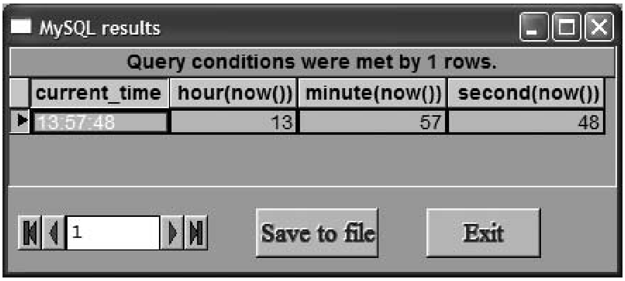
```
SELECT MINUTE( NOW() )
```



Query conditions were met by 1 rows.

now()	week(now())	yearweek(now())	quarter(now())
2002-12-23	51	200251	4

Figure 11.8 WEEK(), YEARWEEK() and QUARTER().



Query conditions were met by 1 rows.

current_time	hour(now())	minute(now())	second(now())
13:57:48	13	57	48

Figure 11.9 HOUR(), MINUTE() and SECOND().

Figure 11.9 shows this function running along with some other time functions.

MINUTE

MINUTE() returns the minutes past the hour for a given time object. You pass the time value to it as follows:

```
MINUTE( aTime )
```

To obtain the current minute past the hour, we can use the following:

```
SELECT MINUTE( NOW() )
```

Figure 11.9 shows this query in action.

SECOND

SECOND() returns the seconds in the minute of a given time object. You pass the time value to it as follows:

```
SECOND( aTime )
```

To obtain the seconds part of the current time, we can use the following:

```
SELECT SECOND( NOW() )
```

Figure 11.9 shows the results for this at the time of writing.

DATE_ADD

There is no real need to add dates together, however adding a period of time to a date is useful. The DATE_ADD() function allows you to add a time period to a date. The format of DATE_ADD() is as follows:

```
DATE_ADD( aDate, INTERVAL timeperiod )
```

The timeperiod will be a value, followed by the type of time period. This sounds complex, so as usual we will demonstrate by example. To find out what date it will be on this day next week, we can add 7 days to the current date with the following:

```
DATE_ADD(‘2002-12-23’, INTERVAL 7 DAY )
```

You will note that we have typed the date here, instead of using NOW() to show you an alternative way of entering the date. Figure 11.10 shows this running, with the column name of nextweek. So the date this time next week will be 30 December.

You can also add other time periods. For instance, the following will return the date that it will be three months from today:

```
DATE_ADD(‘2002-12-23’, INTERVAL 3 MONTH )
```

The above is shown in the *threemonths* column of Figure 11.10; three months from today will be 23 March 2003. The complete script to generate Figure 11.10 is as follows:

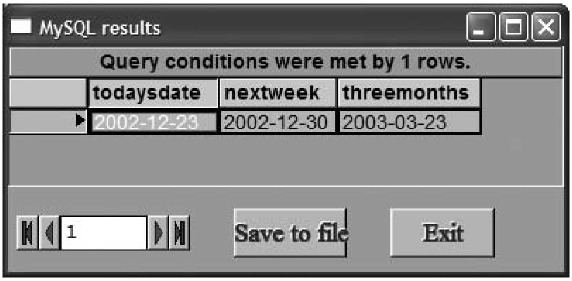
```
SELECT    ‘2002-12-23’ AS TodaysDate,
          DATE_ADD(‘2002-12-23’, INTERVAL 7 DAY ) AS NextWeek,
          DATE_ADD(‘2002-12-23’, INTERVAL 3 MONTH ) AS ThreeMonths
```

As well as adding day, month and year periods to a date, you can also add time periods as well. For instance, to add six hours to the current date we would use:

```
DATE_ADD(‘2002-12-23’, INTERVAL 6 HOUR )
```

You can see this in Figure 11.11, under the *6hrs* column. Notice though that because we only specified a date in the creation string, it has taken the default value of 00:00:00, midnight, as the time, and added six hours to that. The *6fromnow* column shows the results when we use the NOW() function to generate a date that also has a time part. This time, adding six hours gives us a date and time for tomorrow. Again, to recreate the results of Figure 11.11, you can use the following:

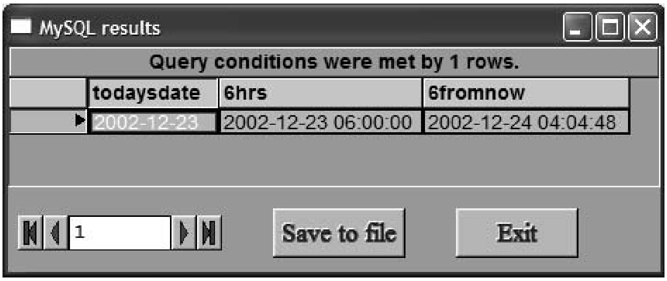
```
SELECT    ‘2002-12-23’ AS TodaysDate,
          DATE_ADD(‘2002-12-23’, INTERVAL 6 HOUR ) AS 6Hrs,
          DATE_ADD(NOW(), INTERVAL 6 HOUR ) AS 6fromnow
```



Query conditions were met by 1 rows.

todaysdate	nextweek	threemonths
2002-12-23	2002-12-30	2003-03-23

Figure 11.10 DATE_ADD() adds day periods to given dates.



Query conditions were met by 1 rows.

todaysdate	6hrs	6fromnow
2002-12-23	2002-12-23 06:00:00	2002-12-24 04:04:48

Figure 11.11 DATE_ADD() adds time periods to given dates.

Table 11.4 shows some of the period types that you can use with DATE_ADD() and other related functions to specify your time period.

Table 11.4 Some time period types.

Type	Value format	Timeperiod example
SECOND	numeric	INTERVAL 5 SECOND
MINUTE	numeric	INTERVAL 5 MINUTE
HOUR	numeric	INTERVAL 5 HOUR
DAY	numeric	INTERVAL 5 DAY
MONTH	numeric	INTERVAL 5 MONTH
YEAR	numeric	INTERVAL 5 YEAR
HOURSECOND	"hours:minutes:seconds"	INTERVAL "05:10:00" HOURSECOND
HOURMINUTE	"hours:minutes"	INTERVAL "05:10" HOURMINUTE
DAYHOUR	"days hours"	INTERVAL "5 12" DAYHOUR

A full listing of these types is in the MySQL online documentation.

ADDDATE

The ADDDATE() function works exactly the same as the DATE_ADD() function just described. The format is therefore:

```
ADDDATE( aDate, INTERVAL timeperiod )
```

Please refer to the DATE_ADD section above for further information.

DATE_SUB

The DATE_SUB() function works similarly to the DATE_ADD() function, but it subtracts the time period instead. The format is:

```
DATE_SUB( aDate, INTERVAL timeperiod )
```

The timeperiod will be a value, followed by the type of time period from Table 11.4. For example, to find out what the date was this time last week, we can use:

```
DATE_SUB(‘2002-12-23’, INTERVAL 7 DAY )
```

And we can take time away from a date as follows:

```
DATE_SUB(‘2002-12-23’, INTERVAL 1 SECOND )
```

Figure 11.12 shows these two functions running, using the following script:

```
SELECT    ‘2002-12-23’ AS TodaysDate,
          DATE_SUB(‘2002-12-23’, INTERVAL 7 DAY ) AS lastweek,
          DATE_SUB(‘2002-12-23’, INTERVAL 1 SECOND ) AS yesterday
```

SUBDATE

The SUBDATE() function works exactly the same as the DATE_SUB() function just described. The format is therefore:

```
SUBDATE( aDate, INTERVAL timeperiod )
```

Please refer to the DATE_SUB section above for further information.

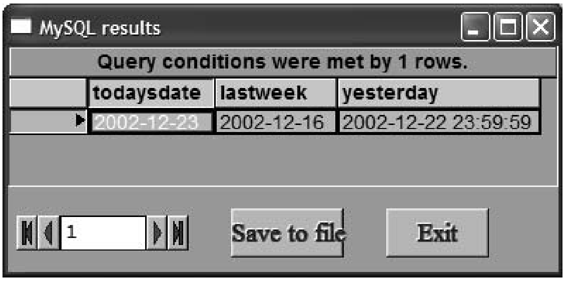
TO_DAYS

The TO_DAYS() function converts a date into the number of days since year 0. You pass the function a date as follows:

```
TO_DAYS(aDate)
```

So to convert today’s date into days we can use:

```
TO_DAYS(‘2002-12-23’) as ToDays
```

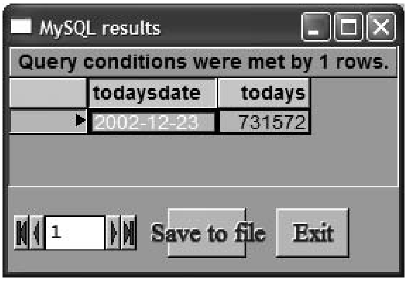


MySQL results

Query conditions were met by 1 rows.

todaysdate	lastweek	yesterday
2002-12-23	2002-12-16	2002-12-22 23:59:59

Figure 11.12 DATE_SUB() subtracts time periods from given dates.



Query conditions were met by 1 rows.

	todaysdate	todays
▶	2002-12-23	731572

Figure 11.13 The number of days since year 0.

Figure 11.13 shows this function running within a SELECT query.

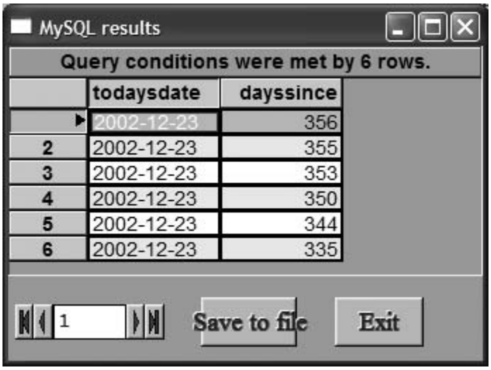
Again, this can be useful when you need to perform some calculation using dates. For instance, our cookie table contains the date that the cookie was created, which is the first day that the user would have accessed the site. The following query could be used to ascertain how many days have passed since that first access:

```
SELECT     NOW() AS todaysdate,
           TO_DAYS(NOW())-TO_DAYS (datecreated) AS dayssince
FROM       cookies
```

Figure 11.14 shows the query running. You'll notice that it has been nearly a year since the creation of most of the cookies.

On a webpage, you would probably want to restrict that query to a single row by selecting the ID of the user's cookie with a WHERE statement, for instance:

```
SELECT     NOW() AS todaysdate,
           TO_DAYS(NOW())-TO_DAYS (datecreated) AS dayssince
FROM       cookies
WHERE      ID=1
```



Query conditions were met by 6 rows.

	todaysdate	dayssince
▶	2002-12-23	356
2	2002-12-23	355
3	2002-12-23	353
4	2002-12-23	350
5	2002-12-23	344
6	2002-12-23	335

Figure 11.14 TO_DAYS() used for date calculation.

FROM_DAYS

FROM_DAYS() is the complementary function to TO_DAYS() in that it converts a given number of days back into a date since year 0. You pass the function a date as follows:

```
FROM_DAYS(Days)
```

Days is the number of days that you want the function to use. So we could use the following to find out which date was 300 000 days after the start of year 0:

```
SELECT ‘2002-12-23’ AS TodaysDate,
FROM_DAYS(300000) AS FromDays
```

Figure 11.15 shows that it was 16 May 821 – quite a long time ago!

Again, you can use this function in day calculations. For instance, we can re-write one of our previous DATE_ADD() examples as follows:

```
SELECT     ‘2002-12-23’ AS TodaysDate,
           FROM_DAYS(TO_DAYS( NOW() )+7) AS NextWeek
```

So this query:

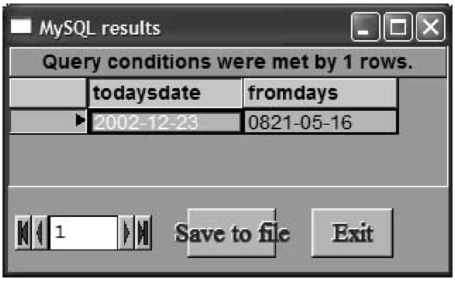
- Calculates the current date using NOW().
- Converts the current date into days since year 0 with TO_DAYS().
- Adds 7 to this figure, to get to next week.
- And then converts the total back into a date.

Of course, the DATE_ADD() function does this more efficiently, however you can see how these two functions can be used when you are calculating with dates.

TIME_FORMAT

TIME_FORMAT() allows you to change the way that a given time is displayed. The format of the statement is:

```
TIME_FORMAT(aDate,format)
```



Query conditions were met by 1 rows.

todaysdate	fromdays
2002-12-23	0821-05-16

Figure 11.15 300 000 days from year 0!

aDate is any object that contains a time part.

format is a string that contains the required parts of the date, using the following symbols:

- %H – the hour, in the 24-hour clock format, with a leading zero
- %k – the hour, in the 24-hour clock format, without a leading zero
- %l – the hour, between 1 and 12
- %h – the hour, between 1 and 12, with a leading zero
- %i – the minutes
- %r – the full time, in the 12-hour clock (hh:mm:ss) with AM/PM shown
- %T – the full time in the 24-hour clock (hh:mm:ss)
- %S – the seconds
- %p – AM or PM

So an alternative format for the CURRENT_TIME function would be:

```
TIME_FORMAT(NOW(),“The time is: %T”)
```

If you like the time to be output that way, but prefer to use hyphens instead of colons, you could use:

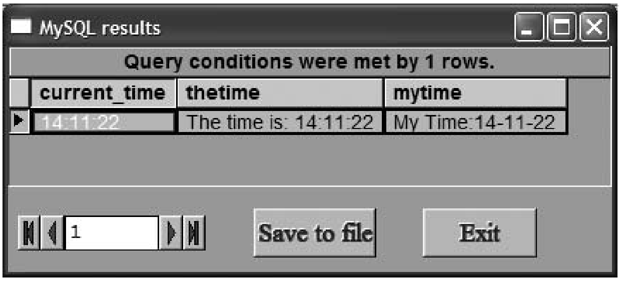
```
TIME_FORMAT(NOW(),”My Time:%H-%i-%S”)
```

In the first example, we have just used the symbol %T, which has output one of the standard forms of the time format. Compare this with the second example, where, to get more flexibility, we have used three symbols to refer to each part of the time specifically. We can then easily insert any other characters (such as our hyphens) at any point in the format string. We have also added other text within the format string, to give us a readable output from the function. Figure 11.16 shows these functions working.

DATE_FORMAT

DATE_FORMAT() allows you to change the way that a given date and time is displayed. The format of the statement is:

```
DATE_FORMAT(aDate,format)
```



Query conditions were met by 1 rows.

current_time	thetime	mytime
14:11:22	The time is: 14:11:22	My Time:14-11-22

Figure 11.16 Formatting the output of time.

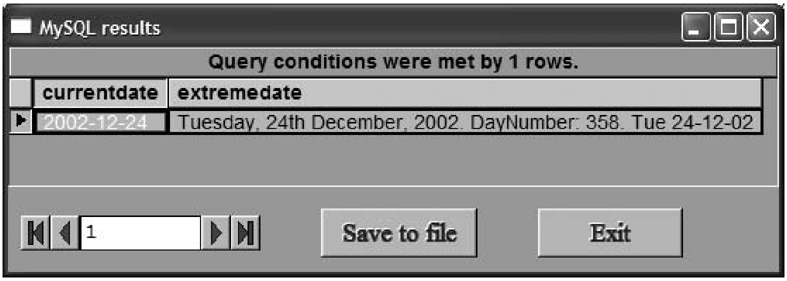
aDate is any date object or string that can be converted into a date.
format is a string that contains the required parts of the date, using all of the previously mentioned time symbols with the addition of the following:

- %M – full text of month name
- %W – day of the week, written in full
- %D – day of the month with added text (i.e. 1st, 2nd, 3rd)
- %Y – the year, using 4 digits
- %y – the year, using 2 digits
- %a – abbreviated day of the week
- %d – day of the month with leading zero
- %e – standard day of the month
- %m – month number with leading zero
- %c – standard month number
- %b – month name abbreviated to three characters
- %j – day of year
- %w – day of the week in numbers, starting with Sunday = 0

We can demonstrate most of these in one query as follows:

```
SELECT    CURRENT_DATE AS currentdate,
          DATE_FORMAT( NOW(),
                      “%W, %D %M, %Y. DayNumber: %j. %a %d-%m-%y”)
```

I could have included time parts in that query as well, but then the results may have not fitted in Figure 11.17. The query has used different date parts, punctuation spacing and text, to format the date in our chosen way. Although it is quite hard to remember all of the symbols for formatting dates and time – and there are more described in the on-line manual – one way is as follows. *D*, *M* and *Y* are easily remembered as Day, Month and Year. If they are written in capitals then it is the full un-abbreviated version of the date part. If you write them in lower case, *d*, *m* and *y*, you get the shortened format for the part. Don’t forget to include the percent sign (%) in front of any of the symbols.



MySQL results

Query conditions were met by 1 rows.

currentdate	extremedate
2002-12-24	Tuesday, 24th December, 2002. DayNumber: 358. Tue 24-12-02

Figure 11.17 Formatting the date function.

Working with Dates

Dates can be very problematic in computer systems. For instance, depending on your location, the date may be formatted in different ways, the date may be stored in different ways – which prompted all the worry about the year 2000 bug, and dates may be incorrectly understood, for example:

```
01-02-01
```

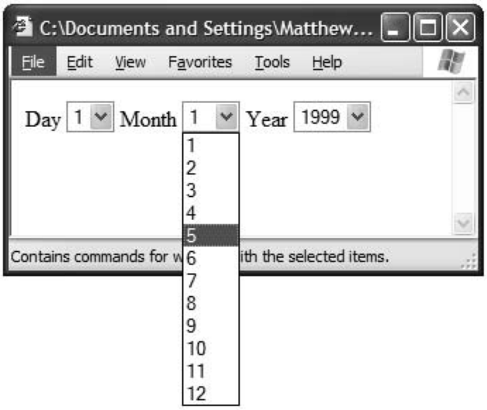
could be understood as:

- 1 February 2001
- 1 February 1 (year 1)
- 2 January 2001
- 2 January 1, etc.

So when using dates you need to make a decision about how you want to store them and then stay with it, to save errors being built into your data by the use of wrongly understood data. One colleague of mine always stores dates as a number of days from year 0, using something like TO_DAYS() to calculate this. You can then use other functions to format the output of the date to whatever you want and avoid any problems.

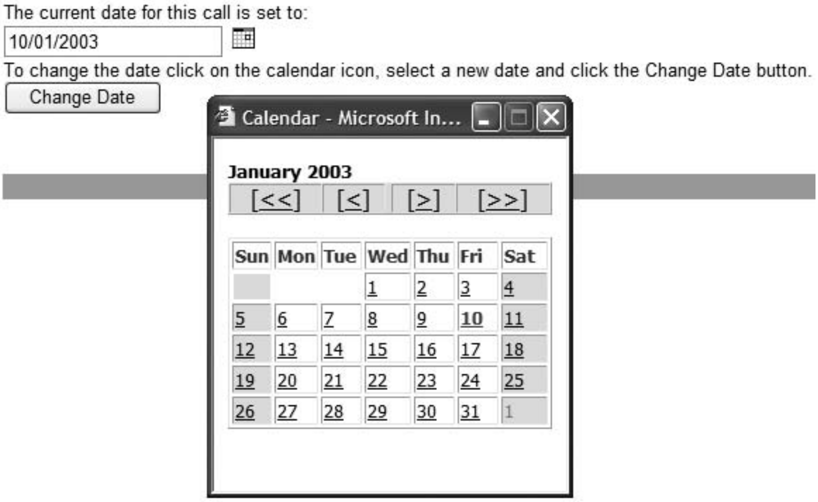
Another issue is the input of dates on webpages. Again a user can enter a date in many different ways. One way I have seen to avoid this is to have a series of dropdown boxes for the day, month and year, as shown in Figure 11.18. This prevents the user from entering the wrong data but will not stop them entering dates such as 31 February, which does not exist.

JavaScript functions are available to pick dates from calendars and personally I have found these to be most successful for avoiding input errors. An example of one is shown in Figure 11.19. Instead of clicking directly on a date field, this is marked read only and you have to click on an icon that invokes the calendar popup. Once you select the date the popup vanishes and the date is stored as you want it to be in the read-only field.



C:\Documents and Settings\Matthew...
File Edit View Favorites Tools Help
Day 1 Month 1 Year 1999
1
2
3
4
5
6
7
8
9
10
11
12
Contains commands for w ith the selected items.

Figure 11.18 Entering dates through dropdowns.



The current date for this call is set to:

10/01/2003

To change the date click on the calendar icon, select a new date and click the Change Date button.

January 2003

Sun	Mon	Tue	Wed	Thu	Fri	Sat
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1

Figure 11.19 Selecting dates from a calendar.

Common Date Queries

Date queries can be very complicated to create, and if you are struggling with one, get help! A brief search on the internet can reveal that someone has already written a query that if not of immediate use may well put you on the correct path again.

Also, remember the on-line MySQL manual that installed along with MySQL. This contains more examples of common queries which can be very useful. One of note is a method to calculate ages, which is not reproduced again here, but can be found under the Date and Time Functions section of the manual (Section 6.3.4 in the MySQL 4.11 alpha manual).

Securing the Database 12

Protecting Valuable Data

So far in this book we have been using our database as a single user. We are logging in as a root user with no restrictions on what we do to our data. We can create tables and databases whenever we want, and we can delete data and whole tables and databases just as quickly, with no restrictions. This is fine for a single user who takes responsibility for his actions, but what if many people are using the data?

MySQL is a multi-user system, so many people can access the data at one time. If you have ever used any multi-user system where everyone has the highest level of access then you will know that even when there is no malicious activity, files disappear, mistakes are made, and data and time are lost. When you give someone access to your data, you should therefore restrict them to the minimum level of access that is possible for them to perform their task. That way any damage that they may cause is limited.

So MySQL provides a method of creating and maintaining user accounts, and then granting specific levels of access to these users. This chapter will describe some of the methods of creating users so that the data will be protected.

To create a user within MySQL you GRANT them access to a resource. At that time you provide a string to identify that user to the system (the username) and a string to authenticate that user.

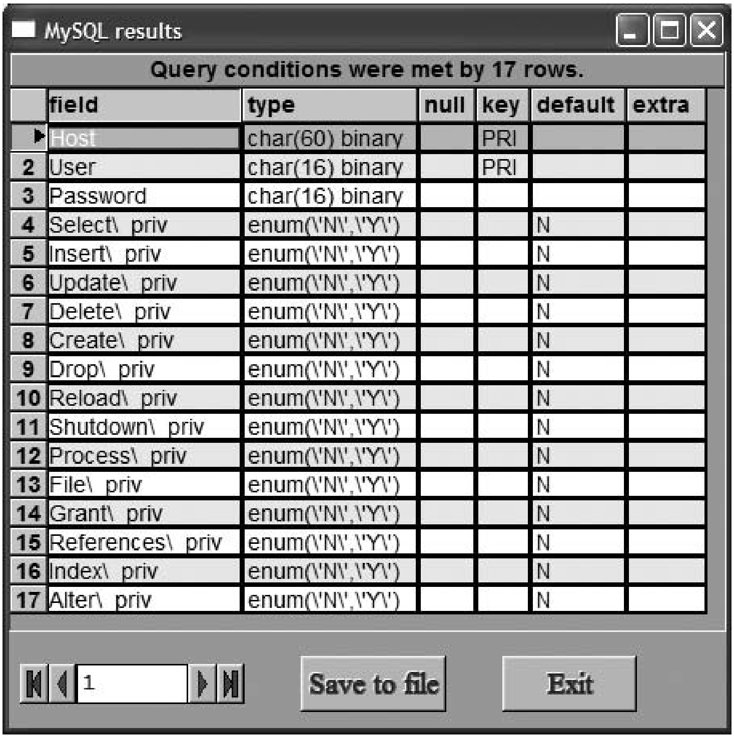
The User Table

Users are actually stored within a table within MySQL. To look at the table, make sure that you are in the *mysql* database (the rest of the chapters use the *mysqlfast* database) by selecting it from the database dropdown or typing:

```
use mysql
```

The *mysql* database contains the system tables which MySQL needs to function. The user table is one table within this database. We created all of the other tables in this book in a separate database so that we could clearly distinguish our tables and system tables from each other. You can view the user table by executing the following query:

```
describe user
```



Query conditions were met by 17 rows.

	field	type	null	key	default	extra
▶	Host	char(60) binary		PRI		
2	User	char(16) binary		PRI		
3	Password	char(16) binary				
4	Select\ priv	enum(\'N\',\'Y\')			N	
5	Insert\ priv	enum(\'N\',\'Y\')			N	
6	Update\ priv	enum(\'N\',\'Y\')			N	
7	Delete\ priv	enum(\'N\',\'Y\')			N	
8	Create\ priv	enum(\'N\',\'Y\')			N	
9	Drop\ priv	enum(\'N\',\'Y\')			N	
10	Reload\ priv	enum(\'N\',\'Y\')			N	
11	Shutdown\ priv	enum(\'N\',\'Y\')			N	
12	Process\ priv	enum(\'N\',\'Y\')			N	
13	File\ priv	enum(\'N\',\'Y\')			N	
14	Grant\ priv	enum(\'N\',\'Y\')			N	
15	References\ priv	enum(\'N\',\'Y\')			N	
16	Index\ priv	enum(\'N\',\'Y\')			N	
17	Alter\ priv	enum(\'N\',\'Y\')			N	

Figure 12.1 Format of the user system table.

Figure 12.1 shows the format of this table. Note that there are rows for the username and password, as well as rows for specific access privileges.

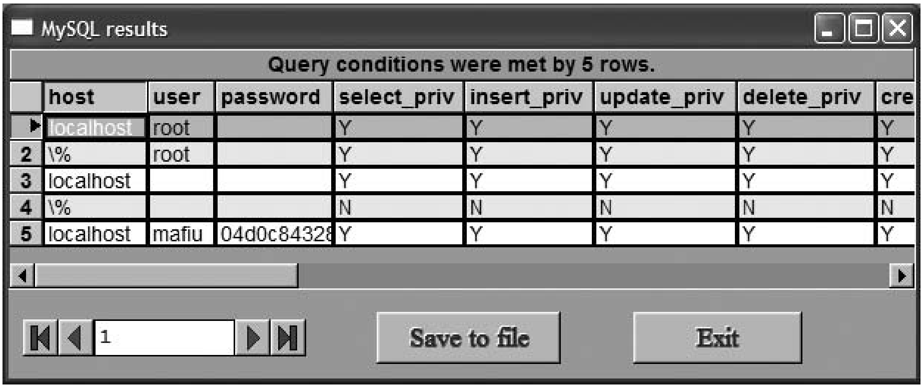
The content of the table is revealed by the following:

```
SELECT      *
FROM        USER
```

Figure 12.2 shows some of the contents of the table. You will see that during the production of this book there has only really been one user created, with other root users that are system generated.

You will also see that the root user in this instance does not have a password and the password for the user called mafiu is not the one that you may have been typing in throughout the book. For security, MySQL encrypts a password before it gets stored within a table. You can see the benefit of this when other users can access the user table and view each other's passwords.

When you grant a privilege to a user, if the user does not already exist it will create an entry in the user table. If they already exist then the privilege will be added to the existing user.



Query conditions were met by 5 rows.

	host	user	password	select_priv	insert_priv	update_priv	delete_priv	cre
▶	localhost	root		Y	Y	Y	Y	Y
2	\%	root		Y	Y	Y	Y	Y
3	localhost			Y	Y	Y	Y	Y
4	\%			N	N	N	N	N
5	localhost	mafiu	04d0c84328	Y	Y	Y	Y	Y

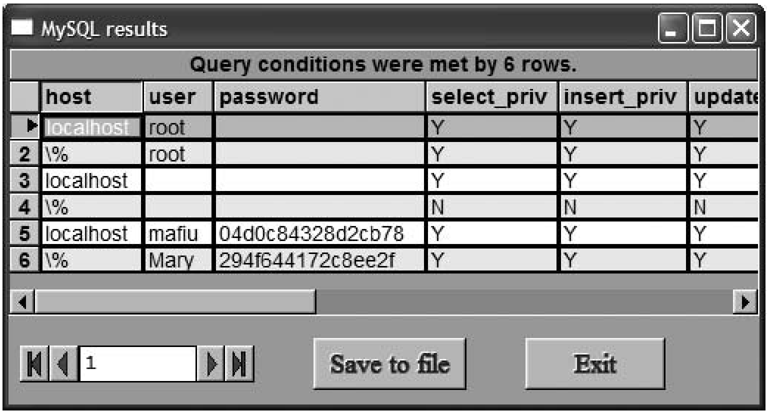
Figure 12.2 Content of the user system table.

Creating a User

To create a new user then, and give them the greatest possible access, we would use:

```
GRANT     ALL PRIVILEGES
          ON *.* TO Mary@localhost
          IDENTIFIED by “yeti”
```

That does not immediately produce any results, but if you select everything from the USER table again you will see the results in Figure 12.3. Notice that we now have a Mary entry in the table, with a scrambled password.



Query conditions were met by 6 rows.

	host	user	password	select_priv	insert_priv	updat
▶	localhost	root		Y	Y	Y
2	\%	root		Y	Y	Y
3	localhost			Y	Y	Y
4	\%			N	N	N
5	localhost	mafiu	04d0c84328d2cb78	Y	Y	Y
6	\%	Mary	294f644172c8ee2f	Y	Y	Y

Figure 12.3 The new user in the user table.

Connecting as a Different User

Because we are using the GUI logged in as root, we will use the command line MySQL monitor to log in as this user. Open up a dos box, and type the following:

```
c:
cd \mysql\bin
```

to get you into the directory with the MySQL executables.

Normally, to run the MySQL monitor you would just type mysql. However, this logs in as the root user with full access. We need to tell the monitor to log in as our new Mary user. Do this by typing:

```
mysql —user=Mary —password=yeti
```

Now we need to connect to a database within MySQL. Type:

```
connect mysql
```

You should now be connected to the database, and all that remains is to have a look at one of the tables within the database. Type the following to view the user table:

```
SELECT * FROM user;
```

Remember that when you are in the MySQL monitor you need to add the semicolon at the end of the line to make the query execute. We may have got out of the habit of this while using the graphical client. If you forget to add the semicolon you can just add it to the new line to trigger the query that you typed on the previous line. Figure 12.4 shows all of the above in action including a forgotten semicolon.

The table output in Figure 12.4 is a bit messy which is why you may be glad that we normally use the graphical client. You can, however, see that we have now logged in with our new user and examined a table. We will now deny the Mary user access to this table using the graphical client again. Switch back to this, but leave the dos box open so that we can return to it in a few moments.

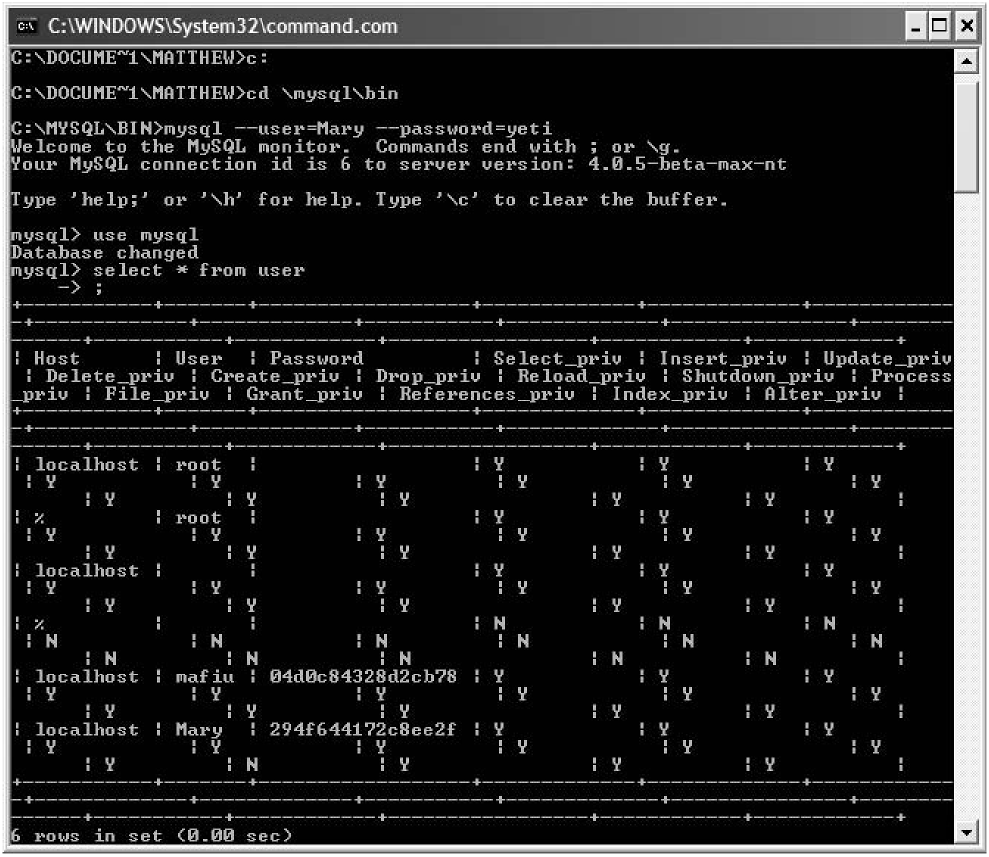
Restricting a User's Access

In the graphical client, logged in as root, type:

```
REVOKE ALL PRIVILEGES ON *.*
FROM Mary@localhost
```

There is no point in selectively denying access to a table when the user has access to everything, so the above removes the access anything privilege on the account.

Back at the command prompt, if you select everything from the user table you will still get the table returned. This is because the REVOKE command just issued is a global command; it is saying revoke Mary's access to everything. Changes to global privileges only



```
C:\DOCUME~1\MATTHEW>c:

C:\DOCUME~1\MATTHEW>cd \mysql\bin

C:\MYSQL\BIN>mysql --user=Mary --password=yeti
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6 to server version: 4.0.5-beta-max-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use mysql
Database changed
mysql> select * from user
    -> ;
+-----------+-------+------------------+-------------+-------------+------------
-+-------------+-------------+-----------+-------------+---------------+--------
------+-----------+------------+-----------------+------------+------------+
| Host      | User  | Password         | Select_priv | Insert_priv | Update_priv
 | Delete_priv | Create_priv | Drop_priv | Reload_priv | Shutdown_priv | Process
_priv | File_priv | Grant_priv | References_priv | Index_priv | Alter_priv |
+-----------+-------+------------------+-------------+-------------+------------
-+-------------+-------------+-----------+-------------+---------------+--------
------+-----------+------------+-----------------+------------+------------+
| localhost | root  |                  | Y           | Y           | Y
 | Y           | Y           | Y         | Y           | Y             | Y
      | Y         | Y          | Y               | Y          | Y          |
| %         | root  |                  | Y           | Y           | Y
 | Y           | Y           | Y         | Y           | Y             | Y
      | Y         | Y          | Y               | Y          | Y          |
| localhost |       |                  | Y           | Y           | Y
 | Y           | Y           | Y         | Y           | Y             | Y
      | Y         | Y          | Y               | Y          | Y          |
| %         |       |                  | N           | N           | N
 | N           | N           | N         | N           | N             | N
      | N         | N          | N               | N          | N          |
| localhost | mafiu | 04d0c84328d2cb78 | Y           | Y           | Y
 | Y           | Y           | Y         | Y           | Y             | Y
      | Y         | Y          | Y               | Y          | Y          |
| localhost | Mary  | 294f644172c8ee2f | Y           | Y           | Y
 | Y           | Y           | Y         | Y           | Y             | Y
      | Y         | N          | Y               | Y          | Y          |
+-----------+-------+------------------+-------------+-------------+------------
-+-------------+-------------+-----------+-------------+---------------+--------
------+-----------+------------+-----------------+------------+------------+
6 rows in set (0.00 sec)
```

Figure 12.4 Logging in to the monitor with our new user.

come into effect when a user logs in. As Mary was logged in when we changed her privileges, she still can access everything. To log out, at the command prompt type:

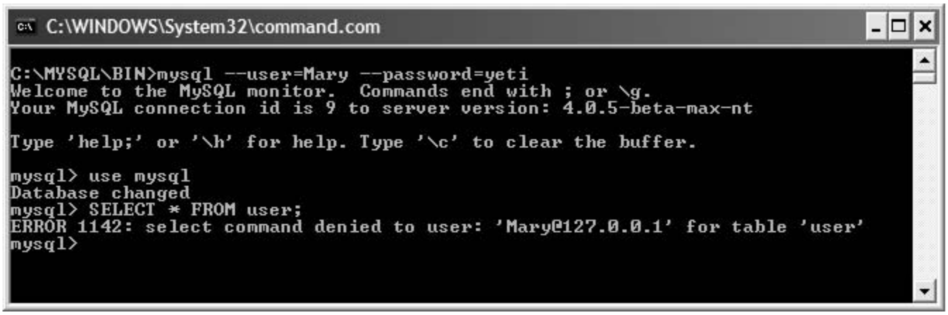
```
exit
```

and then type:

```
mysql —user=Mary —password=yeti
```

to log in again. Now change to the *mysql* database and select everything from the user table again as follows:

```
use mysql;
SELECT * from user;
```



```
C:\MYSQL\BIN>mysql --user=Mary --password=yeti
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9 to server version: 4.0.5-beta-max-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use mysql
Database changed
mysql> SELECT * FROM user;
ERROR 1142: select command denied to user: 'Mary@127.0.0.1' for table 'user'
mysql>
```

Figure 12.5 Mary now has no access to the user table.

Figure 12.5 shows that Mary no longer has the access that she once had. Instead of returning the table, MySQL just returns:

```
 Error 1142: select command denied to user: ‘Mary@127.0.0.1’ for table
‘user’
```

If you are wondering about the number 127.0.0.1, this is just the IP (Internet Protocol) number that has been allocated for *localhost*.

Even though the user table has encrypted passwords, it is best not to give normal users access to it. When given sample passwords and user accounts, it is possible to use software tools to obtain the password. As you already know the passwords in this table and book there is little point in trying to hide them, so we will give Mary access to view the contents of that table again. In the graphical client, type:

```
GRANT SELECT ON mysql.user
TO Mary@localhost
```

Back on the MySQL monitor, try to select from the user table again. This time you should get the whole table back. Notice that because this is a table-level privilege, this time changes to the user’s access became effective immediately. Now let’s be more malicious. On Mary’s login, type the following:

```
DELETE FROM user WHERE user =’mafiu’;
```

Mary will get the error message:

```
ERROR 1142: delete command denied to user: ‘Mary@127.0.0.1’ for table
‘user’
```

This is where the GRANT function becomes very useful. If Mary still had all privileges, she would have just deleted a user. As we had only given her access to the user table for SELECTing, a DELETE query threw an error, saving the day. The safest rule is to give a user the minimal access required for them to complete their tasks.

To see what has happened to the user system table after these GRANT and REVOKE commands, issue the following query in the graphical client:

```
SELECT *
FROM user
WHERE user = 'Mary'
```

Notice that although the user table is a system table, it is still a table that you can apply any query to, as long as you have the access rights. Although we revoked the ALL PRIVILEGES right, there is still an entry for Mary in the user table, as we can see by looking at Figure 12.6.

You will see from the small selection of privilege columns in Figure 12.6 that these columns have now all been set to N. If you have this on your screen, scroll across and you will see that all of the other columns have been set to N as well. Mary has therefore lost her global rights to everything. If this is the case, how does the system know that she has access to the user table to do SELECT queries?

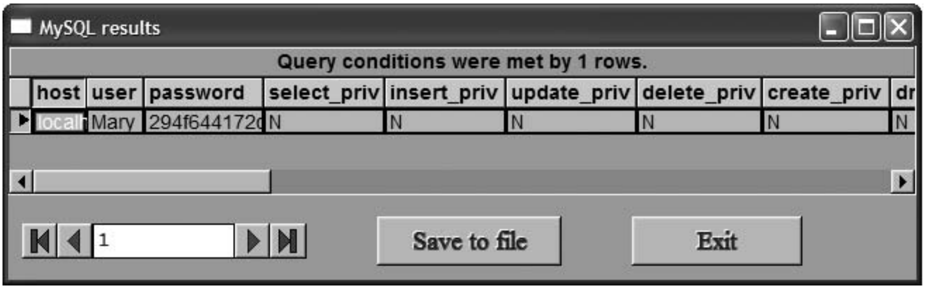
The tables_priv Table

Access rights to a specific table are stored within the *tables_priv* table. This table has slightly different properties from the *user* table. Figure 12.7 describes this table, which you may like to compare with the user table shown in Figure 12.1.

You will notice that instead of having columns specifically for each privilege, this table has a column where you can explicitly set which keywords will work on the specified table: *tables_priv*. If we SELECT everything from this table now, the results are as shown in Figure 12.8. Notice that Mary has an entry that allows her to view (SELECT) the user table. It also shows you who granted that access, in the grantor column, useful when you are tracking down a problem with access.

What will happen if we allow Mary some further access to the user table, say allowing her to insert a new user? The following script will allow this:

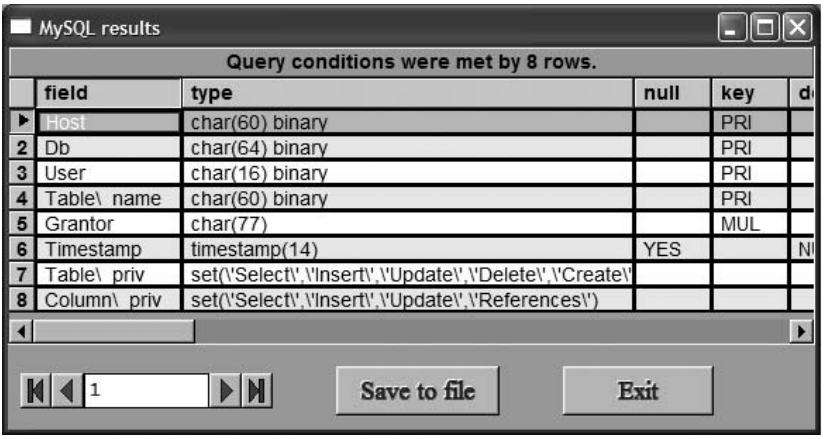
```
GRANT INSERT on mysql.user
TO Mary@localhost
```



Query conditions were met by 1 rows.

host	user	password	select_priv	insert_priv	update_priv	delete_priv	create_priv	dr
localh	Mary	294f644172d	N	N	N	N	N	N

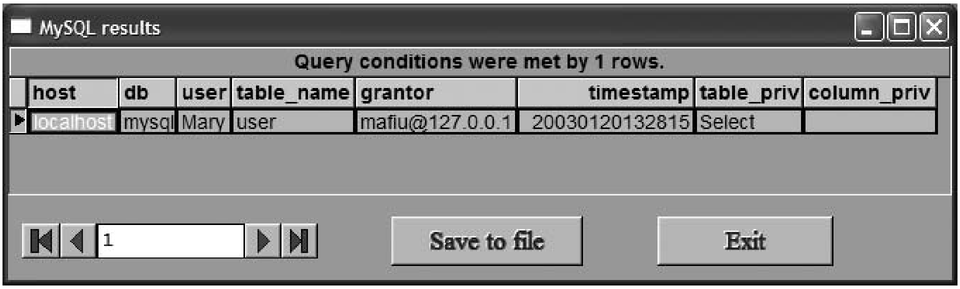
Figure 12.6 Mary's new entry in the user table.



Query conditions were met by 8 rows.

	field	type	null	key	d
▶	Host	char(60) binary		PRI	
2	Db	char(64) binary		PRI	
3	User	char(16) binary		PRI	
4	Table\ name	char(60) binary		PRI	
5	Grantor	char(77)		MUL	
6	Timestamp	timestamp(14)	YES		N
7	Table\ priv	set(\'Select\',\'Insert\',\'Update\',\'Delete\',\'Create\'			
8	Column\ priv	set(\'Select\',\'Insert\',\'Update\',\'References\')			

Figure 12.7 DESCRIBE tables_priv.



Query conditions were met by 1 rows.

	host	db	user	table_name	grantor	timestamp	table_priv	column_priv
▶	localhost	mysql	Mary	user	mafiu@127.0.0.1	20030120132815	Select	

Figure 12.8 Mary's entry in the tables_priv table.

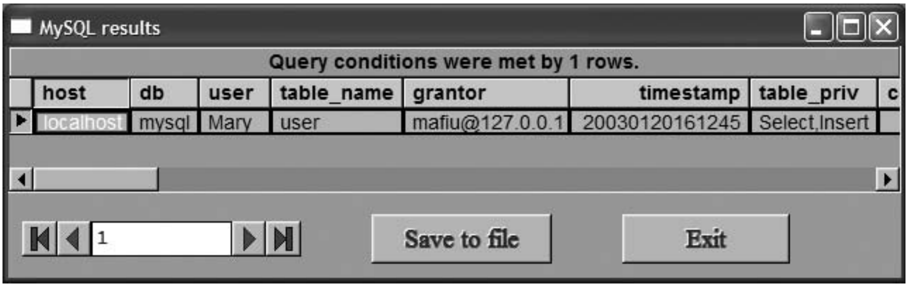
Figure 12.9 shows what happens if we view the *tables_priv* table after running the GRANT query above. If you compare it with Figure 12.8 you will see that instead of adding another row for the INSERT privilege it has appended the first entry. The *tables_priv* column now reads Select and Insert.

The Grant Tables

The *user* and *tables_priv* tables are known as the grant tables. These tables control everyone's access to your MySQL server. There are five grant tables in all, as shown in Table 12.1.

All of these tables are stored within the *mysql* database, so you can refer to them directly when using GRANT or REVOKE by adding *mysql.* to the front of the table name, i.e. *mysql.db*, *mysql.tables_priv*, *mysql.columns_priv*, etc.

Generally, you do not need to know what is going on within all of these tables, as long as you understand how the GRANT and REVOKE queries work. However, it is useful to realize that MySQL uses its own tables to store the privileges in, and you can access these using the normal query mechanisms.



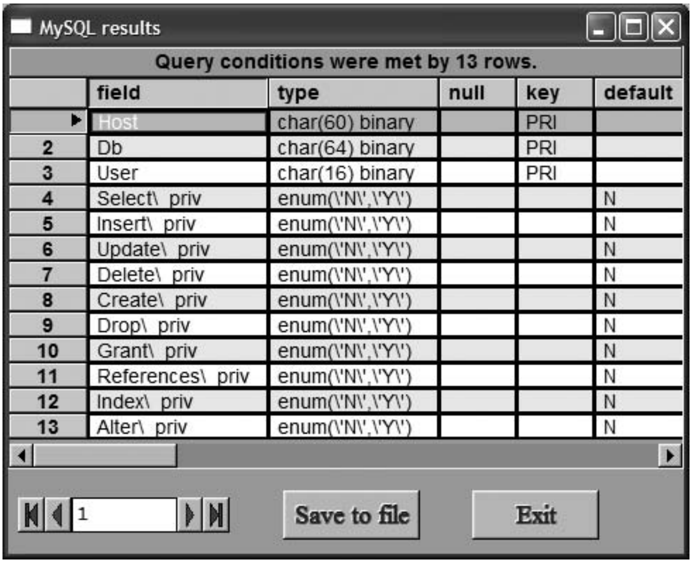
Query conditions were met by 1 rows.

	host	db	user	table_name	grantor	timestamp	table_priv	c
▶	localhost	mysql	Mary	user	mafiu@127.0.0.1	20030120161245	Select,Insert	

Figure 12.9 Mary's amended entry in tables_priv.

Table 12.1 The MySQL grant tables.

Table name	Uses
user	Stores the global privileges for the whole server (examined in Figure 12.1).
db	Stores the privileges for a specific database on the server (examined in Figure 12.10).
host	Stores database-specific privileges for the specified host.
tables_priv	Stores the privileges for access to a given table within a specific database (examined in Figure 12.7).
columns_priv	Stores the privileges for access to a given column within a specific table within a database.



Query conditions were met by 13 rows.

	field	type	null	key	default
▶	Host	char(60) binary		PRI	
2	Db	char(64) binary		PRI	
3	User	char(16) binary		PRI	
4	Select\ priv	enum(\'N\',\'Y\')			N
5	Insert\ priv	enum(\'N\',\'Y\')			N
6	Update\ priv	enum(\'N\',\'Y\')			N
7	Delete\ priv	enum(\'N\',\'Y\')			N
8	Create\ priv	enum(\'N\',\'Y\')			N
9	Drop\ priv	enum(\'N\',\'Y\')			N
10	Grant\ priv	enum(\'N\',\'Y\')			N
11	References\ priv	enum(\'N\',\'Y\')			N
12	Index\ priv	enum(\'N\',\'Y\')			N
13	Alter\ priv	enum(\'N\',\'Y\')			N

Figure 12.10 The db table.

FLUSH PRIVILEGES

There is one point that you need to bear in mind though. When you use the GRANT and REVOKE queries, the system realizes what you are doing and so makes any access changes immediately. Some changes, such as global privileges, will take effect the next time the user connects to the server. However, if you use normal SQL queries to change a user's privileges, the system just treats these as normal database queries. The user's access rights however *do not* change immediately. To make the server realize that you have just committed some privilege changes, you have to type the following:

```
FLUSH PRIVILEGES
```

This alerts the MySQL server to a privilege change and makes it reload the grant tables.

GRANT

GRANT is used to give a privilege to a user. If details of that user do not exist in any of the grant tables then a new user is created. GRANT can therefore be used to create users. GRANT takes the following format:

```
GRANT     privilege ON systemobject
TO        username
          IDENTIFIED BY 'password'
```

In the above:

- **username** and **password** are the credentials that the user needs to connect to the database. Note that you need the quote marks around the password string. The IDENTIFIED BY clause is optional, but can be used to change the password of a user that already exists. If this is omitted when GRANTing a privilege to a user that doesn't exist, the user is created without a password.
- **privilege** is the action that you are going to allow the user to perform. Table 12.2 shows some of the privileges that are available.
- **systemobject** is what the user will be performing that access on. It is usually used with the following format:

```
database.table
```

so *.* refers to all tables in all databases on this server.

REVOKE

REVOKE is used to remove a given a privilege from a user. REVOKE will fail if the user has not been previously granted the privilege. REVOKE takes the following format:

```
REVOKE    privilege ON systemobject
FROM      username
```

Table 12.2 Some privileges to use with GRANT.

Privilege	Allows the user to:
ALL PRIVILEGES	Use all privileges available on the given object
ALTER	Change table data structures with ALTER TABLE
CREATE	Create tables
CREATE TEMPORARY TABLES	Create temporary tables
DELETE	Remove rows from tables with DELETE
DROP	Remove tables with DROP
EXECUTE	Run a stored procedure, however stored procedures are not implemented in the current version of MySQL
FILE	Read and write files
INDEX	Create and remove indexes
INSERT	Insert data into tables
LOCK TABLES	LOCK tables that they have the SELECT privilege on
RELOAD	Use the FLUSH command
SELECT	Retrieve data from tables using SELECT
SHUTDOWN	Shut down the MySQL server
UPDATE	Change the contents of tables with UPDATE

In the code:

- **privilege** is the action that you are revoking from the user and uses Table 12.2 in the same way as the GRANT command.
- **systemobject** is what the user is being denied access to.

If you revoke a privilege successfully you get no feedback from the MySQL server. However, if you revoke a privilege that a user does not have, for example:

```
REVOKE     SELECT ON mysql.tables_priv
FROM       Mary@localhost
```

you will get the error:

```
There is no such grant defined for user 'Mary' on host 'localhost' on
table 'user'
```

If you REVOKE all privileges that have been previously given to a user, this does not delete that user from the user table, but they will have no further access to the system until GRANTed more privileges. The user will stay valid until they are explicitly deleted from the user table.

Restricting Users

Bear in mind that users of your MySQL database will very probably be different from any users that authenticate to a web system that you are building. Your MySQL table will probably have your administrative user, root, which is used for creating your tables and any manual administrative tasks, and a general web user, say, web_anonymous, that is used by the webserver to access the tables. Depending on what your web system does, you will give the anonymous user limited privileges, such as:

```
GRANT      SELECT
ON         mysqlfast.webpage
TO         web_anonymous@localhost;

GRANT        INSERT
ON           mysqlfast.log
TO           web_anonymous@localhost;

GRANT        INSERT,SELECT
ON           mysqlfast.cookies
TO           web_anonymous@localhost;


GRANT        INSERT,SELECT
ON           mysqlfast.visitorbook
TO           web_anonymous@localhost;
```

These queries will allow the webserver to:

- retrieve the contents of webpages,
- add entries to the log table when a page is viewed,
- check the existence of a cookie and create a new one if necessary, and
- create new visitorbook entries and view existing ones.

You will see from the chapters that follow the way that you embed the user credentials into the scripting language that you are using to access MySQL and build the webpages with. If you only allow the web_anonymous user these limited privileges, there is little that the webserver can do to damage your data. For instance, with the set of privileges above, it would be impossible for that user to remove a webpage, i.e.

```
DELETE FROM webpage
WHERE title = ‘Home’
```

as the user does not have DELETE access on the webpage table. Unless you wrote a section of the website to edit and delete webpages, you would have to log in as root and manually delete the page using SQL to remove the home page if required.

Other Passwords

Although the examples we have been using in this book have referred to a fictional website, many sites use databases to authenticate users. For instance, search engines like Yahoo allow you not just to search, but also to log into them to gain more facilities. It is likely that your username, password and many other things that you enter into the site are being stored behind the scenes in a database. If you are designing such a system you should give some thought to how you are going to store your passwords. The temptation is to store them as plain text strings, but that is not the safest way of doing it. If you look back at Figure 12.6 you will remember that SQL automatically encrypted the password so that it was not stored in an obvious way within the user table.

So if you are creating your own user table for your website, MySQL has provided an encrypt function that allows you to code your passwords before you store them in the table. This function is called PASSWORD(), and is used as follows:

```
PASSWORD(‘password_string’)
```

The function will return an encrypted version of the password string, which can then be inserted into your database table. For example, the following insert statement would insert an encrypted password into a table containing your web users:

```
INSERT INTO webuser (username,
                     password,
                     email)
VALUES ( ‘mountainman’,
         PASSWORD(‘bigfoot’),
         ’someone@nowhere.com’)
```

In reality, of course, instead of running the query as above, you would have inserted variables obtained from web form fields into the table, and you would have encrypted the variable using PASSWORD, not just the typed text.

Interestingly, there is no DECRYPT function. This is to stop you doing:

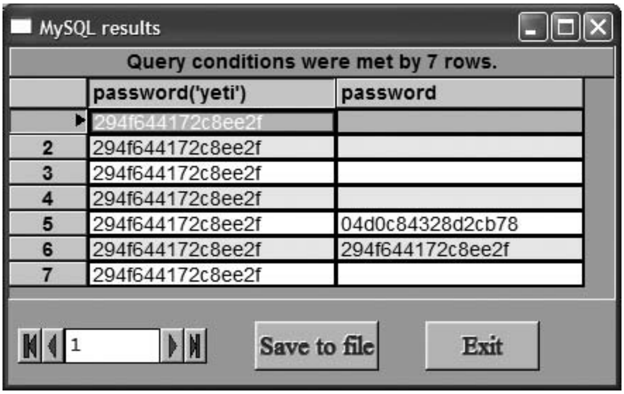
```
SELECT user,
DECRYPT(password) FROM mysql.user
```

This obviously would be a huge security risk. So how do you check to see if someone has entered the same password that is in the database? Well, everytime that you encrypt the same text string with PASSWORD, it always produces the same encrypted string. For instance, run the following query:

```
SELECT    PASSWORD(‘yeti’), password
FROM      user
```

This query will encrypt the word yeti, and output it along with all of the password strings stored in the user table. Figure 12.11 shows the results.

You may remember when we GRANTed some rights to the Mary user, we identified her by the password *yeti*. Look at the results in Figure 12.11 and you should see that the 6th row down contains the same string in both columns. So you can compare encrypted strings to see if they match. If an existing user typed his password into a webpage, we can encrypt it and use it in a where clause to check that the password typed is the same as the one stored. For example, the following:



	password('yeti')	password
▶	294f644172c8ee2f	
2	294f644172c8ee2f	
3	294f644172c8ee2f	
4	294f644172c8ee2f	
5	294f644172c8ee2f	04d0c84328d2cb78
6	294f644172c8ee2f	294f644172c8ee2f
7	294f644172c8ee2f	

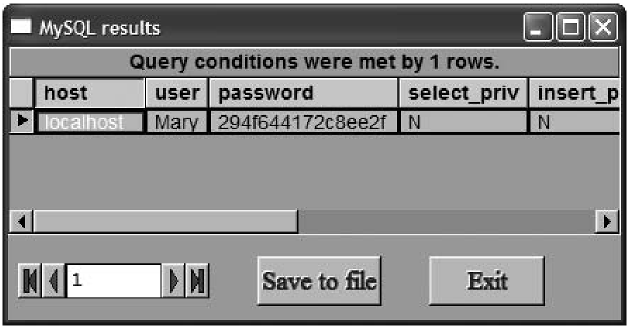
Figure 12.11 Password encryption.

```
SELECT     *
FROM       mysql.user
WHERE      user=”Mary” and password=PASSWORD(‘yeti’)
```

will give the results shown in Figure 12.12, returning one row, proving that the user and password are correct. However, if you ran the following:

```
SELECT     *
FROM       mysql.user
WHERE      user=”Mary” and password=PASSWORD(‘abominable’)
```

the query would produce no results, as the Mary user’s password is not *abominable*, the encrypted version is not the same as the stored encrypted version, they therefore do not match and so the password is proved to be incorrect. To check for a valid username and password combination in your scripting language, simply check if you have more than 0 records returned from the query. Again, if this was being used on a webpage we would be



	host	user	password	select_priv	insert_p
▶	localhost	Mary	294f644172c8ee2f	N	N

Figure 12.12 Matching encrypted passwords.

passing form variables into the SQL statement as opposed to hard coding them as we have in the queries above.

In the above example we used the *mysql.user* table to demonstrate encrypted password checking. On a production system, however, it would be best to implement a separate table in a separate database to store your user credentials.

This is a very simple way of securing your stored passwords, but is not the only security consideration that you may need to take when authenticating users to webpages. If your application needs to be secure, then consider implementing a secure web connection as well as the encrypted passwords, as in our example the password would be sent in plain text from the web form to the MySQL server. Once it is there and encrypted it is safer, but if intercepted on the way will still cause a breach of your security system.

This page intentionally left blank

13 Optimizing MySQL

Tables Get Bigger

All of the tables that we have created during the course of this book have had less than 10 rows. We have already talked about how MySQL is a very fast way of handling data, but when we are using such small amounts of data there is little opportunity to test its speed. However, if we were storing the log of a very popular website in database tables, the 30 000 hits that site may receive in a day would soon gather a huge amount of data. As soon as we need to process large amounts of data we can introduce delays into our systems. To minimize these delays and to make our query processing as fast as possible, we need to give thought to how well our queries are written and what we can do to improve their speed.

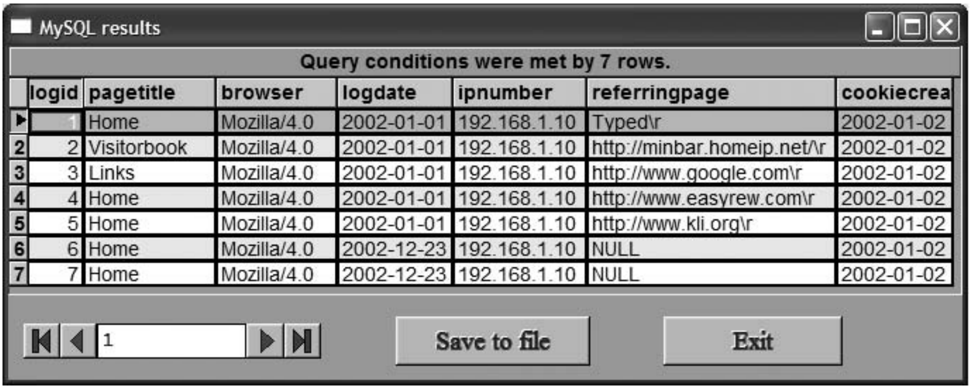
How Well Does It Run?

In Chapter 7, we dealt with joining tables together. This can be a complex process that can be very processor intensive on large tables. You may remember that we looked at the following script:

```
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
```

If you remember that chapter, this script looked at each entry in the log table, showed the name of the page they were looking at, and showed the first date that that cookie user viewed the site. Figure 13.1 shows that query in action. With the limited number of rows that we have in the tables involved, the query produces seven rows of results.

Look back at the script that generated the results in Figure 13.1. Although the results are quite small, the script is reasonably complex. But what is actually going on behind the



Query conditions were met by 7 rows.

logid	pagetitle	browser	logdate	ipnumber	referringpage	cookiecrea
1	Home	Mozilla/4.0	2002-01-01	192.168.1.10	Typed\r	2002-01-02
2	Visitorbook	Mozilla/4.0	2002-01-01	192.168.1.10	http://minbar.homeip.net/\r	2002-01-02
3	Links	Mozilla/4.0	2002-01-01	192.168.1.10	http://www.google.com\r	2002-01-02
4	Home	Mozilla/4.0	2002-01-01	192.168.1.10	http://www.easyrew.com\r	2002-01-02
5	Home	Mozilla/4.0	2002-01-01	192.168.1.10	http://www.kli.org\r	2002-01-02
6	Home	Mozilla/4.0	2002-12-23	192.168.1.10	NULL	2002-01-02
7	Home	Mozilla/4.0	2002-12-23	192.168.1.10	NULL	2002-01-02

Figure 13.1 The output from a join of three tables.

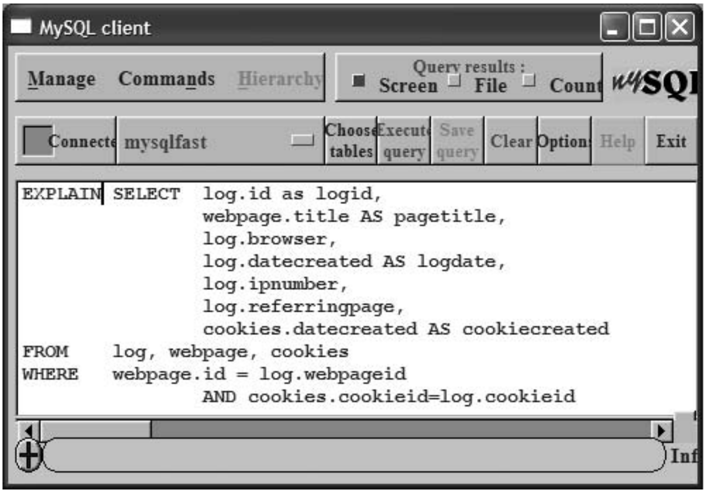
scenes? If we were to run the same script with a log table that contained 1 000 000 rows, would that script run quickly or could we make it run more efficiently?

Examining Queries

If you have just entered the above query into the MySQL client, click the cursor to the start of the query and type the word EXPLAIN in front of the query, as shown in Figure 13.2.

EXPLAIN can be run on a table or on a query. Running it on a table, as follows:

```
EXPLAIN tablename
```



```
EXPLAIN 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 13.2 Explaining a query.

gives the same results as describing a table as follows:

```
DESCRIBE tablename
```

EXPLAIN comes into its own when you apply it to a query as we have done in Figure 13.2. Simply add the keyword EXPLAIN in front of the query that you wish to analyse. When we EXPLAIN our three table join query we get the results shown in Figure 13.3.

When you EXPLAIN a query, the MySQL server does not actually run the query, but gives an approximation of what it would have to do were you to run it. This means that if you were about to run a query on a huge amount of data, you could EXPLAIN the query first to make sure that it would run in the best way.

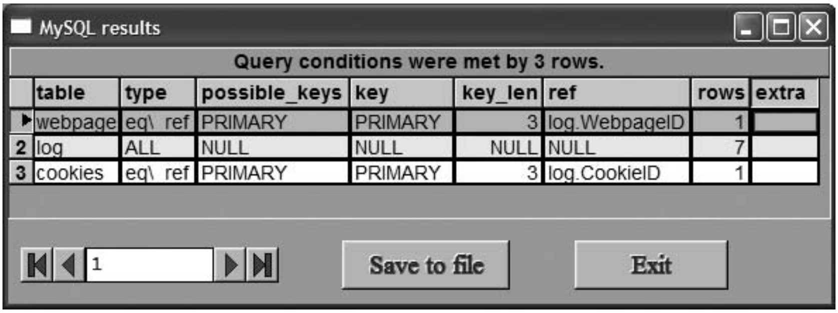
The first column of the EXPLAIN, **table**, shows the tables that the query will have to reference to show the results. In this case it is three tables, webpage, log and cookies. Of course we do not actually need to use EXPLAIN to tell us this, but the rest of the columns are of more use!

The second column, **type**, signifies the type of join that MySQL will have to do to process the row. The log table will use the ALL join, and the webpage and cookies tables will use the eq_ref join. The full set of joins that can appear here are shown in Table 13.1. This table runs from the slowest and most process-intensive join to the fastest, most efficient one.

The third column in the EXPLAIN output, **possible_keys**, shows the keys that MySQL hopes to use to execute the query. This column is called possible because the query might

Table 13.1 Join types used by the EXPLAIN facility.

Join type	Explanation
ALL	The query will have to scan all rows of the table to complete this query.
index	The index of this table is used to complete this query.
range	An index is used to select a range of rows from this table for scanning by the query.
ref	If the join cannot match one unique row, a set of rows from this table is read for every set of rows that match the join condition of the other tables.
eq_ref	One row from this table is read for every set of rows from the other tables when a single row can be matched by the join.
const	If the table only has one row, it is only read from once.
system	If the table only has one row and is read from once and is also a system table, this join type is noted by the EXPLAIN.



Query conditions were met by 3 rows.

	table	type	possible_keys	key	key_len	ref	rows	extra
▶	webpage	eq\ ref	PRIMARY	PRIMARY	3	log.WebpageID	1	
2	log	ALL	NULL	NULL	NULL	NULL	7	
3	cookies	eq\ ref	PRIMARY	PRIMARY	3	log.CookieID	1	

Figure 13.3 Sample explain results.

not actually use all of the keys in the search. The keys are independent of the order of the tables, so some keys may never actually be processed. If one of these columns is NULL, you may be able to optimize the query further by adding an index to the table. We will do that later in the section on indexes.

The fourth column in the EXPLAIN output, **key**, shows the key that MySQL actually used to perform the query. If no key is used then the entry is NULL. If there is a PRIMARY entry for a column in the possible_keys column which is NULL in this column then there may be a way of optimizing the query further. The optimize table section will show how to do this.

The fifth column in the EXPLAIN output, **key_len**, gives us the length of the key that was used. You can use this column to see if MySQL is using more than one column when a multiple column key exists.

The sixth column in the EXPLAIN output, **ref**, shows which columns were used by the key join. You will notice from Figure 13.3 that the log row contains a NULL for this column, showing that no key was used for this part of the join.

The seventh column in the EXPLAIN output, **rows**, shows the estimated number of rows that the query will use to fulfil the join. This is not necessarily the number of rows that the query will return, but the number that the query will need to look at to execute. If you multiply all of the row values together, you get an estimate of the total number of rows examined by the query. The next section will examine this further.

The last column in the EXPLAIN output, **extra**, gives you any extra information about how the join will be carried out. Table 13.2 shows what you might find in this column.

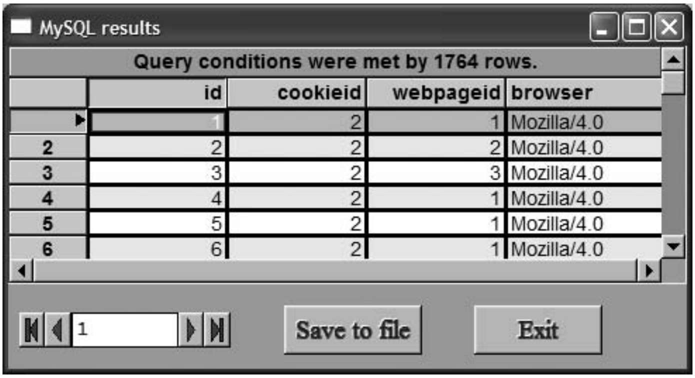
Table 13.2 Possible entries in the extra column.

Entry	Explanation
Distinct	As soon as the first match is found no more searching will be done within this table.
Not exists	An optimized LEFT JOIN will be used and as soon as a matching column is found no more rows will be examined in the table.
Range checked for each record	MySQL tries to find an index for each row in the set of rows from the preceding tables and uses this index if available to obtain the rows.
Using filesort	This takes two passes as it has to sort the rows to retrieve them in the second pass but needs to work out how to sort these rows out in the first pass.
Using index	The index alone is used to retreive the required join data without any of the rows being scanned.
Using temporary	The query needs to create a temporary table to hold the results while processing. If you do a GROUP BY on a different set of columns than an ORDER BY this will occur.
Using where	A where clause is used to select the rows required by the query.

Looking at Cross-joins Again

You may remember that in Chapter 7 we talked about performing a cross join which produced a lot of rows from tables with only a few rows. We can demonstrate this by running the following:

```
SELECT *
FROM log, cookies, webpage, visitorbook
```



Query conditions were met by 1764 rows.

	id	cookieid	webpageid	browser
▶	1	2	1	Mozilla/4.0
2	2	2	2	Mozilla/4.0
3	3	2	3	Mozilla/4.0
4	4	2	1	Mozilla/4.0
5	5	2	1	Mozilla/4.0
6	6	2	1	Mozilla/4.0

Figure 13.4 An inefficient cross join.

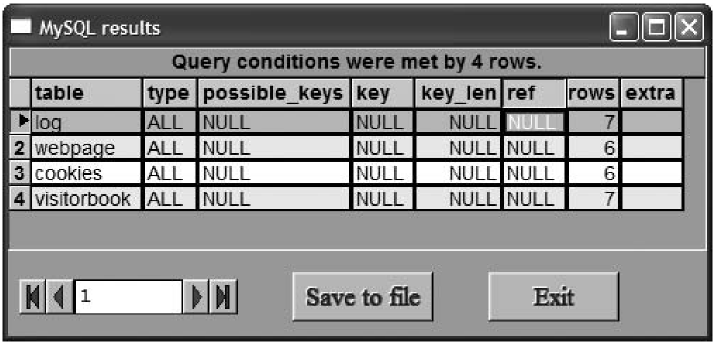
Figure 13.4 shows the output of cross joining the log, cookies, webpage and visitorbook tables together. If you are performing a cross join without knowledge of the number of rows in your tables, it is useful to use EXPLAIN and view the **rows** column to estimate the number of rows you may get returned. Figure 13.5 shows the EXPLAIN running on this query as follows:

```
EXPLAIN SELECT *
FROM log, cookies, webpage, visitorbook
```

All of **rows** column values multiplied together (7 × 6 × 6 × 7) comes to 1764, which is the same as the number of rows returned in Figure 13.4.

We have already discussed how cross joins are inefficient; to demonstrate, we will run the query above with a WHERE clause to restrict the data that we are obtaining from the query as follows:

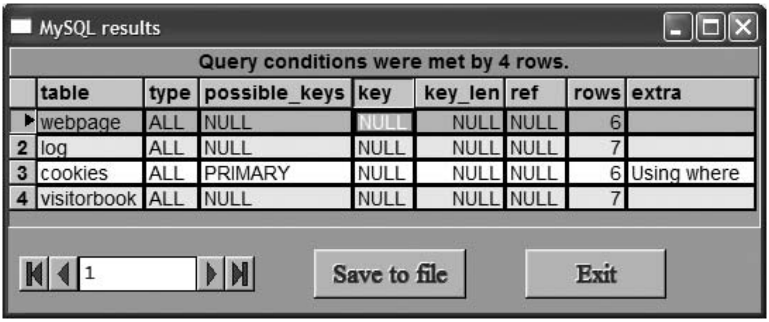
```
SELECT *
FROM webpage, log, cookies, visitorbook
WHERE cookies.cookieid > 3
```



Query conditions were met by 4 rows.

	table	type	possible_keys	key	key_len	ref	rows	extra
▶	log	ALL	NULL	NULL	NULL	NULL	7	
2	webpage	ALL	NULL	NULL	NULL	NULL	6	
3	cookies	ALL	NULL	NULL	NULL	NULL	6	
4	visitorbook	ALL	NULL	NULL	NULL	NULL	7	

Figure 13.5 EXPLAIN on the cross join.



Query conditions were met by 4 rows.

	table	type	possible_keys	key	key_len	ref	rows	extra
▶	webpage	ALL	NULL	NULL	NULL	NULL	6	
2	log	ALL	NULL	NULL	NULL	NULL	7	
3	cookies	ALL	PRIMARY	NULL	NULL	NULL	6	Using where
4	visitorbook	ALL	NULL	NULL	NULL	NULL	7	

Figure 13.6 EXPLAIN on a restricted cross join.

If you run the above script, you will get approximately 800 rows back. You would therefore assume that adding the WHERE clause made the query more efficient. However, running EXPLAIN on the query gives the results shown in Figure 13.6.

You will notice that although the actual query only returned around 800 rows, running the EXPLAIN shows that 1764 rows need to be examined in order to execute the query, so it is just as inefficient as the cross join before the WHERE restriction.

Section 3.2.1 of the online MySQL manual that comes with the product gives a thorough walk-through of using EXPLAIN to optimize a complex query.

Indexing

When we looked at the **possible_keys** column of the EXPLAIN output, I said that if we had a NULL in this column the query could be made more efficient by using an index. An index is a data structure that will speed up the retrieval of rows when applied to a certain search criterion. In other words, if you are searching for something in a table, the search will generally be quicker if an index exists for the column that the data is contained in.

For instance, say you have a log database full of tens of thousands of records which contains data for a whole year of web accesses. The data is stored in a sorted form, as data is inserted into the table in chronological order; so all of the dates are ascending. If you need to search for all of the page hits in June, the server has to sequentially examine all of the rows for the proceeding months until it finds the entries for June. It probably needs to check all of the rows created in the months after June as well if it does not realize that the data is stored in date order.

If we were to add an index to the date column in this table, when searching for June, the MySQL server would examine the index and find that the table is sorted and realize that it can start the retrieval approximately 60 000 records into the table. This might actually be half-way through May, but this means that much fewer rows need to be examined before finding the June entries. Then, as soon as the server comes across a July entry the search can stop, because if the dates are sorted, no more June entries can occur once the July ones begin. If the table contains approximately 10 000 rows per month, the index will have saved the server from looking at over 100 000 rows, that is, the January to May rows and the July to December ones.

CREATE INDEX

The above is an extremely simple explanation of an index, and there are many types of indexes. If you want to understand these more then you should obtain one of the many database theory books that are available. However, if you want to quickly add an index to a table, you can do it without much thought by using the following:

```
CREATE INDEX indexname
ON tablename (columnname)
```

If you are creating an index on a PRIMARY KEY column, where all of the data in the column will be unique, you can create a unique index as follows:

```
CREATE UNIQUE INDEX indexname
ON tablename (columnname)
```

You can also use the ALTER command to add an index to a table as follows:

```
ALTER TABLE tablename
ADD INDEX indexname (columnname)
```

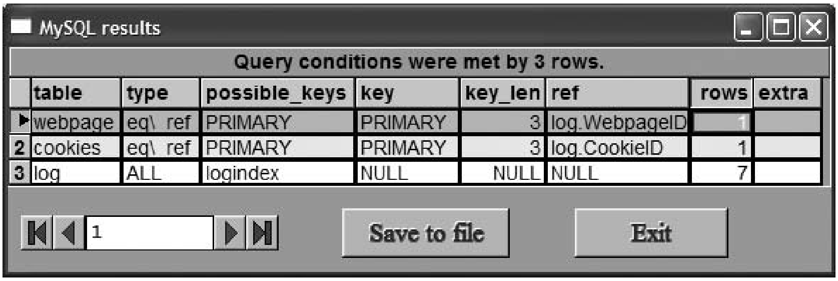
On all three of the scripts above, the *columnname* can actually refer to one or more columns, separated by commas, as you can have an index that refers to multiple columns.

One of the EXPLAIN commands that we issued earlier showed that the join command would not use a primary key as a possible key for the join on the log table. This would mean that we could speed the query up by using an index on the log table. If we will be running this query often, we can optimize it by adding the index. The query will be using the *webpageid* column to join, so we will add an index to the table as follows:

```
ALTER TABLE log
ADD INDEX logindex (webpageid)
```

We need to EXPLAIN the query again to see if the new index has made any difference to the query, so run it again as follows:

```
EXPLAIN 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
```



Query conditions were met by 3 rows.

	table	type	possible_keys	key	key_len	ref	rows	extra
▶	webpage	eq\ ref	PRIMARY	PRIMARY	3	log.WebpageID	1	
2	cookies	eq\ ref	PRIMARY	PRIMARY	3	log.CookieID	1	
3	log	ALL	logindex	NULL	NULL	NULL	7	

Figure 13.7 Optimizing a query by adding an index.

If you compare the results from this EXPLAIN shown in Figure 13.7, with the results from the previous EXPLAIN in Figure 13.3, you will notice that the *log* row now has the index entry in the **possible_keys** column instead of NULL. So if we were running this query on a huge set of data, the index would optimize this query, and make it run better.

Indexes not only make a difference to a join, in fact they can also make a difference to any retrieval of rows where you are restricting the output by values in a column. If the index refers to a column that is being used by the query, then the retrieval will be quicker.

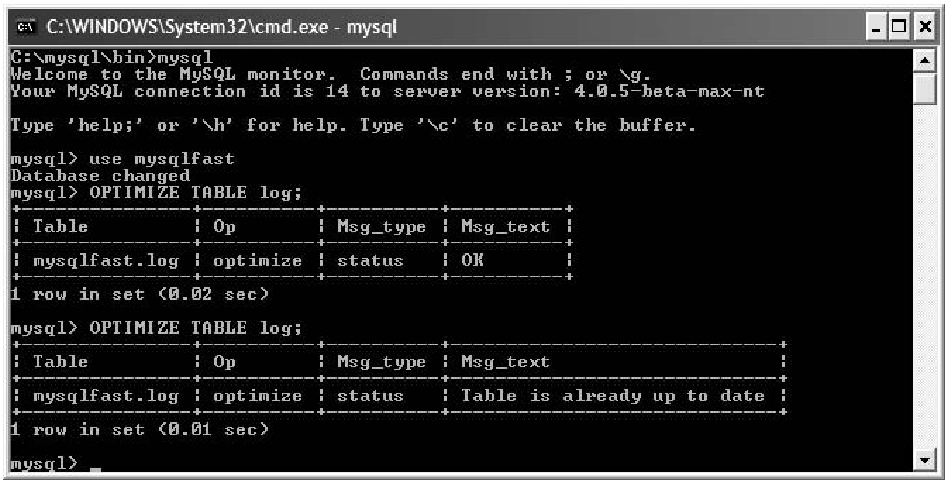
OPTIMIZE TABLE

As mentioned when discussing datatypes, the constant insertion and deletion of records in variable length column types causes the table to become fragmented due to entries and space becoming de-allocated and re-allocated. This fragmentation can cause query operations on the table to be slower than expected. To remedy this, use the following:

```
OPTIMIZE TABLE tablename
```

You can optimize more than one table at once by separating the table names with commas. OPTIMIZE TABLE will defragment the table's datafile and recover any wasted space. It first repairs any split or deleted rows if it finds them, then sorts the index pages if they are out of order, and then updates the table's statistics if the index sort did not complete the repair. Because optimizing a table can involve moving large parts of the table around, MySQL stops other processes from changing the table while it is running. The complex data manipulation could be hindered by another user trying to insert something into the table during the operation.

For some reason, the graphical MySQL client does not run the OPTIMIZE TABLE command, so the output when run in the MySQL monitor is shown in Figure 13.8. You will notice from the figure that if the table is already optimized, MySQL does not attempt to optimize it again, as we can see from the second time that we run the command in the figure. If this is the case, it lets the user know that the table is up to date. The insertion or deletion of a row in the table will allow you to run the OPTIMIZE TABLE command again should you require it. However, there is little need to run this command immediately after you have just run it.



```
C:\mysql\bin>mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 14 to server version: 4.0.5-beta-max-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use mysqlfast
Database changed
mysql> OPTIMIZE TABLE log;
+---------------+----------+----------+----------+
| Table         | Op       | Msg_type | Msg_text |
+---------------+----------+----------+----------+
| mysqlfast.log | optimize | status   | OK       |
+---------------+----------+----------+----------+
1 row in set (0.02 sec)

mysql> OPTIMIZE TABLE log;
+---------------+----------+----------+-----------------------------+
| Table         | Op       | Msg_type | Msg_text                    |
+---------------+----------+----------+-----------------------------+
| mysqlfast.log | optimize | status   | Table is already up to date |
+---------------+----------+----------+-----------------------------+
1 row in set (0.01 sec)

mysql> _
```

Figure 13.8 Optimizing a table.

CHECK TABLE

The CHECK TABLE command allows you to check one or a series of your tables for errors. The command is used as follows:

```
CHECK TABLE tablename, tablename, tablename, … options
```

Use as many **tablenames** as you like separated with commas.
options can be one of the following:

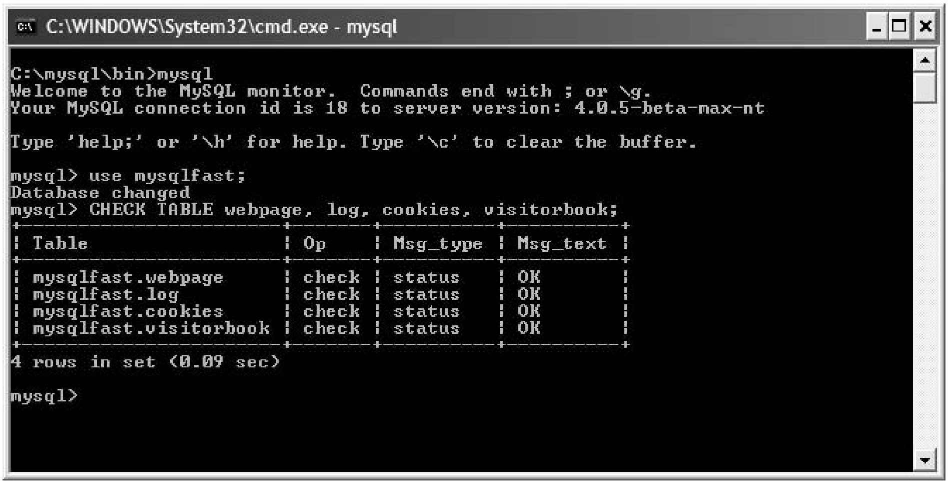
- QUICK does not check for wrong links.
- FAST just checks tables which have not been closed in the correct way.
- CHANGED checks rows as above and tables that have changed since the last time that CHECK TABLE was run.
- MEDIUM checks the rows in the table for correct deleted links, generates a checksum for the rows and verifies this from the key checksums.
- EXTENDED ensures that the table is 100% correct and can take some time to run.

The CHECK TABLE command will return a table with the results of the check in it. A sample output for some of the tables we have used in this book is shown in Figure 13.9.

You will notice that the output provides the name of the table, the operation, which is always check, a message type and the text of the message.

The message type can be either:

- status
- error
- info, or
- warning.



```
C:\mysql\bin>mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 18 to server version: 4.0.5-beta-max-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use mysqlfast;
Database changed
mysql> CHECK TABLE webpage, log, cookies, visitorbook;
+-----------------------+-------+----------+----------+
| Table                 | Op    | Msg_type | Msg_text |
+-----------------------+-------+----------+----------+
| mysqlfast.webpage     | check | status   | OK       |
| mysqlfast.log         | check | status   | OK       |
| mysqlfast.cookies     | check | status   | OK       |
| mysqlfast.visitorbook | check | status   | OK       |
+-----------------------+-------+----------+----------+
4 rows in set (0.09 sec)

mysql>
```

Figure 13.9 Checking tables.

The last row for each table name should always be a status message reading OK. If it does not then there may be some problems with the table, and so you should run a REPAIR TABLE to try and remedy this fault.

REPAIR TABLE

As we have just explained, after running a CHECK TABLE you may well have some errors in the output. Also if the MySQL server has been performing strangely, or unexpectedly been crashing, this may be due to errors somewhere in the table's data. If this is the case, after checking your table, you may need to run REPAIR TABLE on it to correct any problems. You do this using the following command:

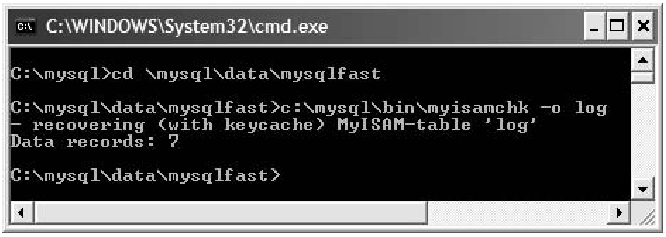
```
REPAIR TABLE tablename, tablename, tablename, … options
```

As you can get many rows of feedback from a table repair, it is best to only repair one table at a time.
The options can be:

- QUICK only tries to repair the index tree of the table.
- EXTENDED creates the indexes row by row.
- USE_FRM recreates the table from the *.frm* file in the tables directory.

You will get a similar table of results from REPAIR TABLE as you did with CHECK TABLE. If the last row of results does not show an OK message, you need to try and repair MySQL from the command line, not through the monitor. To do this, start a dos session, change the directory to the directory where your database is stored, i.e.

```
cd \mysql\data\mysqlfast
```



```
C:\mysql>cd \mysql\data\mysqlfast

C:\mysql\data\mysqlfast>c:\mysql\bin\myisamchk -o log
- recovering (with keycache) MyISAM-table 'log'
Data records: 7

C:\mysql\data\mysqlfast>
```

Figure 13.10 Repairing a table with myisamchk -o.

and then type:

```
c:\mysql\data\bin\myisamchk -o tablename
```

tablename is the name of the table that you are trying to repair. This process and the results can be seen in Figure 13.10.

ANALYZE TABLE

The ANALYZE TABLE command is another tool to allow you to optimize your MySQL tables, which is used as follows:

```
ANALYZE TABLE tablename, tablename, tablename, …
```

ANALYZE TABLE will examine the key distribution within the table and store the results. These keys are used when you are joining the table with others, and if the key distribution is stored this way, it will make some joins run quicker.

Again, this will generate a table of results similar to those shown in Figure 13.9.

This page intentionally left blank

Sharing MySQL data with MyODBC 14

It's Not Just For MySQL

As you probably realized before you bought this book, databases are immensely useful things. They can provide real-time statistics of webservers, store the whole contents of a website or product catalogue for easy updating, provide management information, produce mail shots to clients, and the list goes on.

Just because all of your data is stored in MySQL does not mean that that is where it has to stay. One of the beauties of MySQL is that you can share the data between not just multiple users but multiple applications as well. A common use of MySQL is to power database-driven PHP websites, but that is not the end of the matter. Although PHP and MySQL are often seen working together, do not think that you are confined to using just PHP for MySQL websites. MySQL have provided standard database drivers which allow you to use MySQL data with any program that can talk to these drivers. While your MySQL database is serving your webserver, there is nothing to stop it also providing data obtained from your website to populate mail-shots in Microsoft Word, or driving other web application tools like ASP and ColdFusion.

More Standards

If you have been in the computing industry for some time, you may remember the problems that used to occur when you were setting printers up. Before Windows, every person who wrote software had to customize their software to make it compatible with their programs. Buying a printer could often mean hours of configuring and guessing to get it to work with your program. This changed when printer drivers started to be used. You then made your software talk to a virtual printer device, and it would print without having to make any real changes to your software. All the printer manufacturers would need to do is to produce drivers to make their printer work with the virtual drivers, as opposed to drivers for every piece of software that needed to print.

The same concept of drivers has been applied to many aspects of computing, and on Windows systems, databases can be used by other programs by means of ODBC drivers.

Open Database Connectivity

Open Database Connectivity (or ODBC) is a method of sharing data between databases and other applications. This is done by means of ODBC drivers which have been written by the database vendors, and the ability to communicate with that driver in SQL by the software that needs to use the database.

As well as being used on Linux and with PHP, MySQL is able to interface to other systems using ODBC. ODBC is a method of allowing a program to talk to a database. All the program has to do is talk to the ODBC driver, which handles any specific requirements of the database server. No matter what the DBMS, as long as the correct ODBC driver exists for it then the application can talk to it.

Data Sources

The ODBC connection between your specific database and the software that is talking to it is called a *data source*. You will usually refer to that data source by its name. All database queries are sent to the data source, which uses the ODBC driver to talk to the database server. Data obtained from the request is then passed back to the client software.

You can have many data sources on the client machine, and some of the sources can point to database servers on remote machines. To differentiate though, they must all have unique names.

Obtaining MyODBC

The ODBC driver that MySQL uses is called MyODBC. This can be obtained from the downloads section of the www.MySQL.com website which can currently be found at the following link:

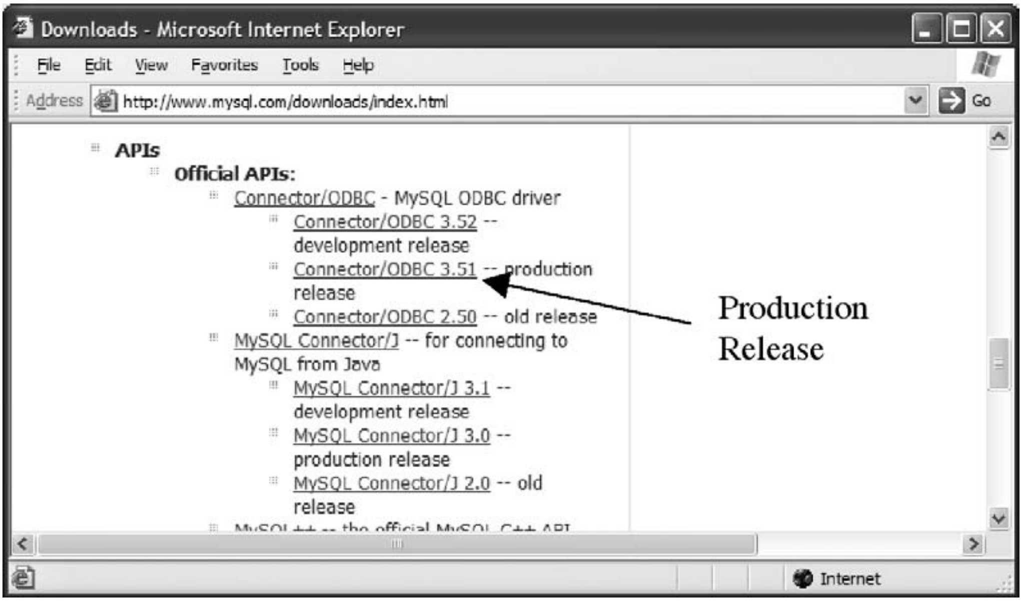
```
http://www.mysql.com/downloads/index.html
```

Figure 14.1 shows the page at the above URL. If you scroll down you will see the link to MyODBC under the Official APIs section. There are usually three types of releases under this section, a current working release, an old release and a release that is being tested. It is usually best to go for the current, stable release which in the figure is production release 3.51. Click on the Connector/ODBC 3.51 link and your web browser will take you to another page which allows you to select the operating system that you will be using MyODBC with and finally download the driver.

Installing MyODBC Drivers

After downloading, double click on the MyODBC .exe file that you just obtained to start installation. You will be presented with a screen as shown in Figure 14.2.

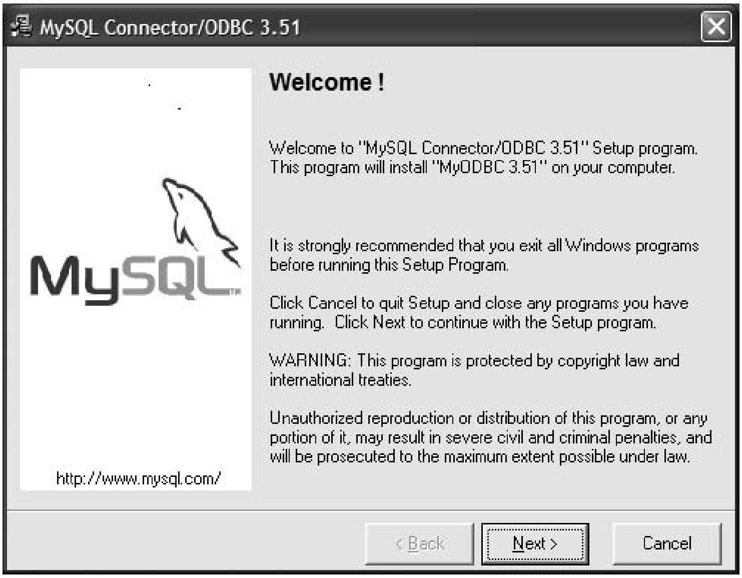
After reading the messages, click on **Next** on the screen shown in Figure 14.2. There are no other options to choose in this setup so you can just click **Next** on the following screens unless you want to read the other messages that you are given. When you get to the screen shown in Figure 14.3, you have completed installing the MyODBC drivers.


Downloads - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.mysql.com/downloads/index.html
Go
Internet

- APIs
  - Official APIs:
    - Connector/ODBC - MySQL ODBC driver
      - Connector/ODBC 3.52 -- development release
      - Connector/ODBC 3.51 -- production release
      - Connector/ODBC 2.50 -- old release
    - MySQL Connector/J -- for connecting to MySQL from Java
      - MySQL Connector/J 3.1 -- development release
      - MySQL Connector/J 3.0 -- production release
      - MySQL Connector/J 2.0 -- old release
    - MySQL++ -- the official MySQL C++ API

Production Release

Figure 14.1 Downloading MyODBC.



Welcome !

Welcome to "MySQL Connector/ODBC 3.51" Setup program. This program will install "MyODBC 3.51" on your computer.

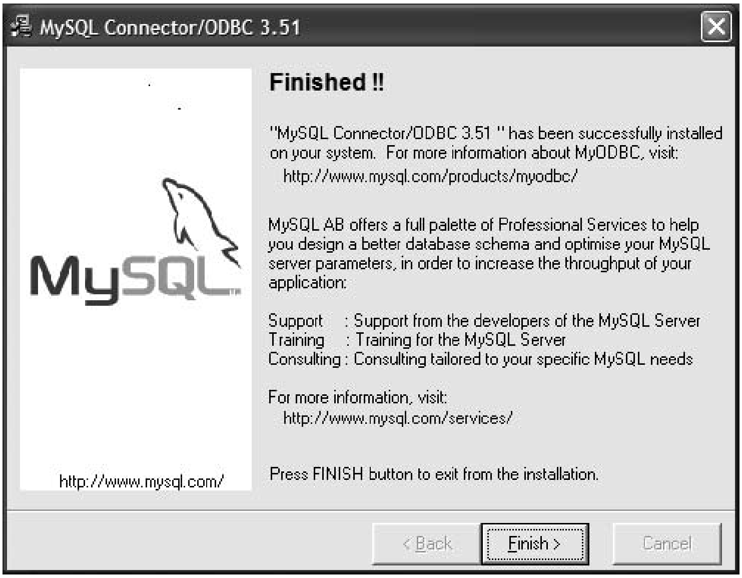
It is strongly recommended that you exit all Windows programs before running this Setup Program.

Click Cancel to quit Setup and close any programs you have running. Click Next to continue with the Setup program.

WARNING: This program is protected by copyright law and international treaties.

Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under law.

Figure 14.2 Installing MyODBC.



MySQL Connector/ODBC 3.51

Finished !!

"MySQL Connector/ODBC 3.51 " has been successfully installed on your system. For more information about MyODBC, visit:
http://www.mysql.com/products/myodbc/

MySQL AB offers a full palette of Professional Services to help you design a better database schema and optimise your MySQL server parameters, in order to increase the throughput of your application:

Support : Support from the developers of the MySQL Server
Training : Training for the MySQL Server
Consulting : Consulting tailored to your specific MySQL needs

For more information, visit:
http://www.mysql.com/services/

Press FINISH button to exit from the installation.

Figure 14.3 Successful installation of MyODBC.

Setting Up a Data Source

The next step in using MyODBC is setting up your data source. This can be done directly in Windows by using the data source control in control panels. Select Control Panel from your start menu. Depending on your version of Windows, you may see the Data Sources or ODBC control straight away. In Windows XP you first need to open up the Administrative Tools section of the control panel to find the correct icon. The icon is shown in Figure 14.4.

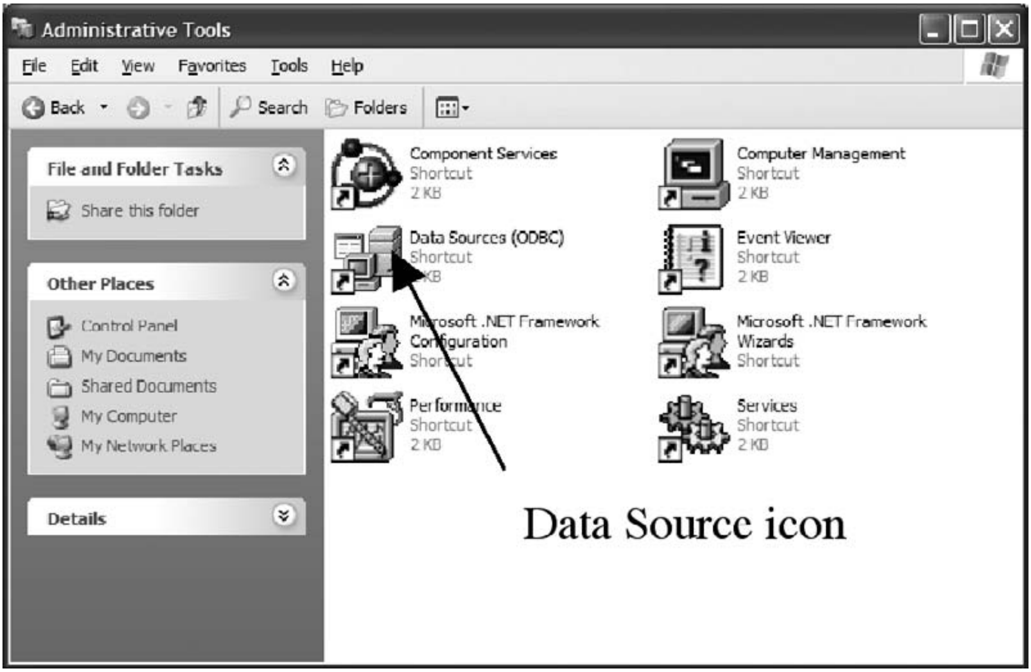
Double click on the data source or ODBC icon in your control panel. The data source control is shown in Figure 14.5.

You will notice that there may already be some data sources set up. You can have two different types of data source, a user data source and a system data source. The user data source is only available to the current user who is logged in, whereas the system data source is available to anyone who is using the machine. Although there are some security issues with using a system data source, this is the type that we will be using for our example. Click on the **System DSN** tab and click on **Add**.

You will then be presented with a list of drivers, as shown in Figure 14.6, which will vary depending on what software you have previously installed on your machine. Scroll down until you find the MySQL entry that corresponds to the drivers you have just installed and click on it. Next, click on the **Finish** button. This does not actually finish the data source setup but takes you to the options for the new data source as shown in Figure 14.7. If you are using an earlier version of MyODBC, this may well look different.

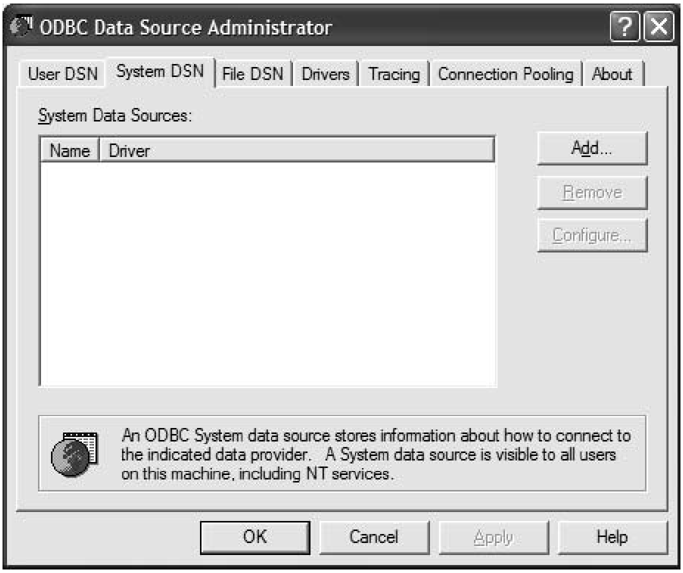
Fill in the fields in this control as shown in the figure, referring to the following:

- The **Data Source Name** will be the name that you refer to the data source with within your code.



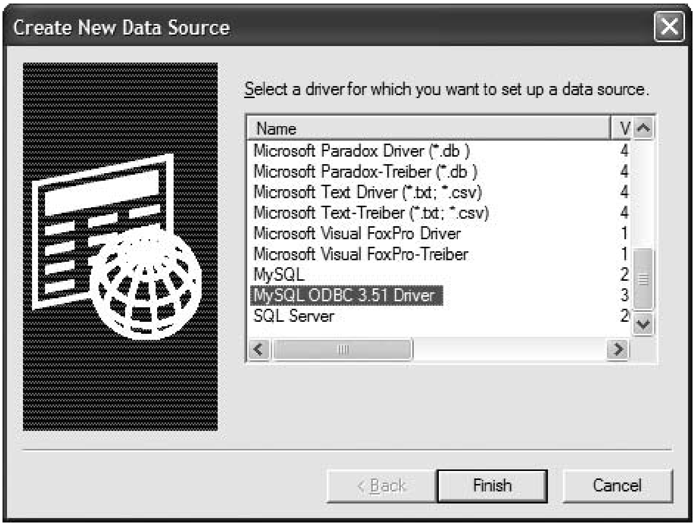
Data Source icon

Figure 14.4 Running ODBC setup.



An ODBC System data source stores information about how to connect to the indicated data provider. A System data source is visible to all users on this machine, including NT services.

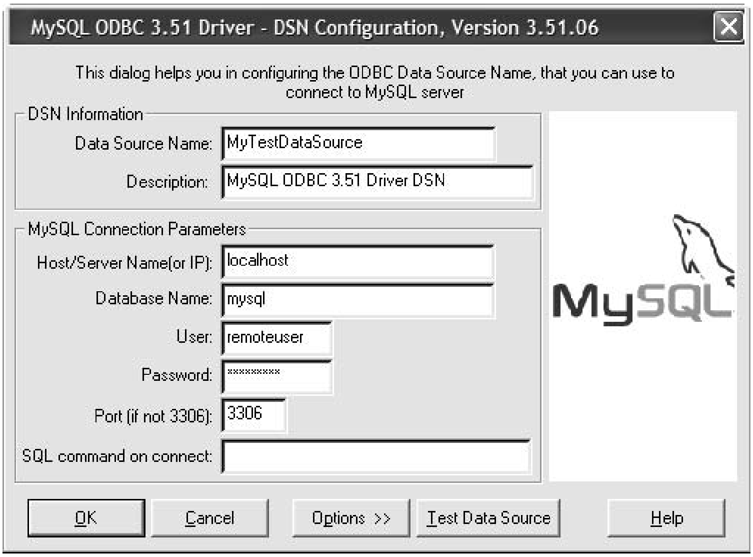
Figure 14.5 The data source control panel.



Select a driver for which you want to set up a data source.

Name	V
Microsoft Paradox Driver (*.db)	4
Microsoft Paradox-Treiber (*.db)	4
Microsoft Text Driver (*.txt; *.csv)	4
Microsoft Text-Treiber (*.txt; *.csv)	4
Microsoft Visual FoxPro Driver	1
Microsoft Visual FoxPro-Treiber	1
MySQL	2
MySQL ODBC 3.51 Driver	3
SQL Server	2

Figure 14.6 Selecting the MyODBC driver.



This dialog helps you in configuring the ODBC Data Source Name, that you can use to connect to MySQL server

DSN Information

Data Source Name: MyTestDataSource

Description: MySQL ODBC 3.51 Driver DSN

MySQL Connection Parameters

Host/Server Name(or IP): localhost

Database Name: mysql

User: remoteuser

Password: *********

Port (if not 3306): 3306

SQL command on connect:

Figure 14.7 MyODBC data source options.

- The **Description** field is not used within your code, so it is not that important; however, if you are setting up this data source on a system that has other data sources, give it a meaningful description.
- The **Host/Server Name** field needs to contain the name of the machine that your MySQL server is running on. If you just have this on your local machine, you can use *localhost*.

If you are connecting to a remote machine with the server on then specify it using its domain name, i.e. *mysql.domain.com*. You can also use the IP number of the machine, i.e. *192.168.1.100*

- The **Database Name** field is used to select the MySql database that you are connecting to. For this example we will use the system *mysql* database. This is safe if you are just using your MySQL database to run through the examples in this book, but if you are on a live server with other users I would recommend creating a new database to connect your data source to for security reasons.
- The **User** and **Password** fields are the user credentials that MyODBC uses to connect to the MySQL server. You can use any valid username and password, but again for the same reasons as above it may be better to use a user that has been specifically created for ODBC MySQL access. It would be a bad idea to use your root account, as this means that the datasource, if it could connect to the system mysql database, could have full access to view and change all of your MySQL data.
- Unless you have changed it from the default settings, leave the **Port** field at the default setting of *3306*.
- The **SQL command on connect** field is used to automatically run a command as soon as the data source makes a connection to the MySQL server. Leave this blank for our example.

If you are using the ODBC source to connect to a server on another machine, the user must have rights to connect from this machine in the host column of the user table. For instance, if the machine with the MyODBC connection had the IP number 192.168.1.1, and the MySQL server had the number 192.168.1.100, the following would apply:

- A user created with the following commands would *not* be able to connect:

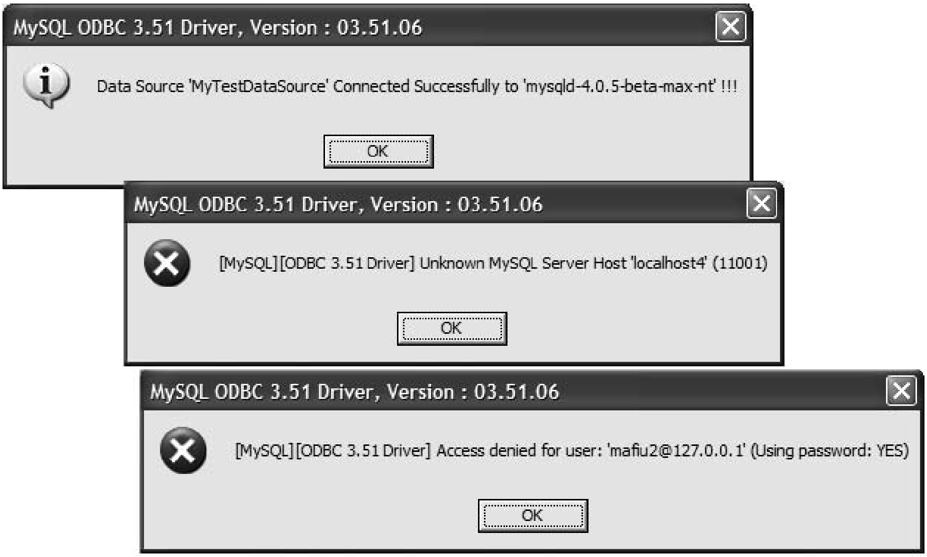
```
GRANT SELECT ON mysql.* TO remoteuser@192.168.1.100;

GRANT SELECT ON mysql.* TO remoteuser@localhost;
```

- A user created with the following commands *would* be able to connect:

```
GRANT SELECT ON mysql.* TO remoteuser@*;

GRANT SELECT ON mysql.* TO remoteuser@192.168.1.1;
```

If you are using a dedicated webserver and dedicated MySQL server for your website on separate boxes, it is best to clearly define the data source user so that it can only connect from the specific webservers that require it by specifying the server name or IP address and not using wildcards for the host entries. This would secure your setup to stop remote users connecting to your MySQL server from other computers. Remember that if the data source is being used by a webserver, and a user on another computer, say 192.168.1.200, connects to that webserver, the data source connection to the MySQL server will come from the webserver, not the machine 192.168.1.200.

On version 3.3.1 of MyODBC you also have a button that allows you to test your connection to the data source. It is easier to test the connection now, at this control, rather than try-



MySQL ODBC 3.51 Driver, Version : 03.51.06

Data Source 'MyTestDataSource' Connected Successfully to 'mysqld-4.0.5-beta-max-nt' !!!

MySQL ODBC 3.51 Driver, Version : 03.51.06

[MySQL][ODBC 3.51 Driver] Unknown MySQL Server Host 'localhost4' (11001)

MySQL ODBC 3.51 Driver, Version : 03.51.06

[MySQL][ODBC 3.51 Driver] Access denied for user: 'mafiu2@127.0.0.1' (Using password: YES)

Figure 14.8 Some MyODBC data source errors.

ing to debug connection problems in the program that is accessing the data source. Figure 14.8 shows three of the possible responses from clicking on the **Test Data Source** button.

The first error in Figure 14.8 shows the type of error that you may get if you have specified the host name of the MySQL server incorrectly. The last error shown in the figure can occur if the username or password is incorrect, or if the user does not have access rights from the machine where the data source is trying to connect from.

Once your data source has tested correctly, you will be able to refer to it and use it in other programs that can communicate with data sources.

Example Connections

To demonstrate how to use the MyODBC data source from other programs, we will show examples of its use with ASP and with ColdFusion.

ASP, or Active Server Pages is a technology that allows you to create dynamic webpages by means of scripts that run on a server. ASP was first implemented by Microsoft when it came built in to its webserver, IIS which comes bundled as part of many of Microsoft's modern operating systems.

ColdFusion was originally produced by Allaire Corporation who have now merged with Macromedia, Inc. ColdFusion again is a server side scripting language which allows the creation of dynamic webpages on your server by means of tags that you insert into HTML code.

The following two examples will show how you can use MyODBC to connect these two products back to your MySQL server to utilize your data.

It should be mentioned at this point that I am simply going to show how to connect to the data source and process some of the data that is returned. These examples will barely scratch the surface of using ASP and ColdFusion which are immensely powerful products in their own right. If you wish to investigate these further, they are covered in other books published by Springer.

MySQL and ASP

There are several ways in which you can connect your ASP script to a MyODBC data source. This section will show one way using the *MyTestDataSource* data source that we created earlier in this chapter using VBSCRIPT.

A webpage that is going to be processed by ASP will reside somewhere in your web space of your webserver and have the extension .asp, for example:

```
c:\inetpub\wwwroot\myexample.asp
```

To send this file to the ASP interpreter, it has to be requested through the webserver using an internet browser by going to the following location:

```
http://localhost/myexample.asp
```

The .asp file is a collection of ASP code and HTML tags. Some of the ASP code will itself output HTML code.

The following code, when executed through an ASP server, will connect to our data source and output some of its data:

```
<% @LANGUAGE=”VBSCRIPT” %>

<HTML>
<TITLE>Asp MySQL MyODBC connection</TITLE>
<BODY>

<%
set MyODBCConnection = Server.CreateObject(“ADODB.Connection”)
set MyRecordSet = Server.CreateObject(“ADODB.Recordset”)

MyODBCConnection.Open “DSN=MyTestDataSource”
MyRecordSet.ActiveConnection = MyODBCConnection
MyRecordSet.Open “SELECT * FROM user WHERE user=’mafiu’”
%>

<TABLE border=1>

<%
while not MyRecordSet.EOF
response.write “<TR><TD>” & MyRecordSet.fields(0) & “</TD><TD>” &
MyRecordSet.fields(1) & “</TD></TR>”
MyRecordSet.MoveNext
```

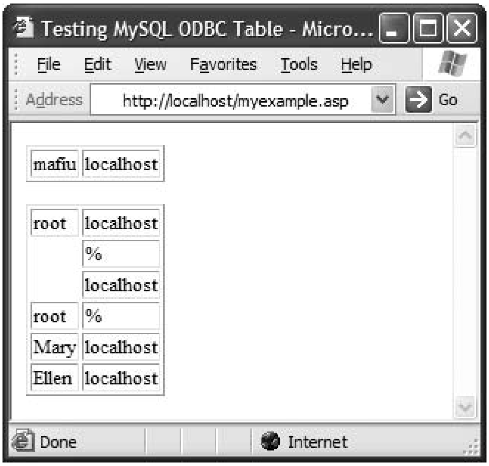
```
wend
%>

</TABLE>
<BR>
<TABLE border=1>

<%
MyRecordSet.Close
MyRecordSet.Open “SELECT * FROM user WHERE user !=’mafiu’”
while not MyRecordSet.EOF
response.write “<TR><TD>” & MyRecordSet(“Host”) & “</TD><TD>” &
MyRecordSet(“User”) & “</TD></TR>”
MyRecordSet.MoveNext
wend
%>

</TABLE>
</HTML>
```

The output of this code is shown in Figure 14.9. Let us examine this code briefly to see what each part does and how it is used.

To begin, we have to tell ASP which scripting language we will be using to define our page; this is done with the following command:

```
<% @LANGUAGE=”VBSCRIPT” %>
```

This lets the server know that we will be using VBSCRIPT within this page. All text within the <% … %> tags is processed by ASP, all other text within the file is output as straight HTML to the browser. Therefore we can start by setting up a basic HTML page with the following:



http://localhost/myexample.asp

mafiu	localhost

root	localhost
	%
	localhost
root	%
Mary	localhost
Ellen	localhost

Figure 14.9 ASP accessing the MySQL user table.

```
 <HTML>
<TITLE>Asp MySQL MyODBC connection</TITLE>
<BODY>
```

This just defines the page as HTML for the browser and sets the page title to be displayed. Now for some ASP, as signified by the <%:

```
<%
set MyODBCConnection = Server.CreateObject("ADODB.Connection")
```

This creates an object that allows for a connection to a database and calls it *MyODBCConnection*. This object can be used elsewhere by sending it commands by appending them to the name of the object. We see this in use further down the page. Next we define another object that handles sets of records using the following:

```
set MyRecordSet = Server.CreateObject("ADODB.Recordset")
```

This time we have called the object *MyRecordSet*. Now we have created two objects needed for the connection to the database, we can start sending commands to them. The first of these commands connects to the *MyTestDataSource* by sending the open command to the *MyODBCConnection* object:

```
MyODBCConnection.Open "DSN=MyTestDataSource"
```

Now that this is open, we can connect our RecordSet object to the open data source:

```
MyRecordSet.ActiveConnection = MyODBCConnection
```

Then finally in this part of the ASP script, we can actually send an SQL command to the data source with:

```
MyRecordSet.Open "SELECT * FROM user WHERE user='mafiu'"
%>
```

This page will create a table that has a row for every row returned from the query. We do this by looping through the rows returned by the record set in ASP. However, we only want the loop to run on the rows of the table itself, so the actual definition of the table is outside the loop, in standard HTML as follows:

```
<TABLE border=1>
```

Now for the loop. For each row of the table we want to output an HTML table row as follows:

```
<TR><TD>Host column</TD><TD>User column</TD></TR>
```

The RecordSet object has a command that returns a true if you have reached the end of the record set. Switching to ASP again, we set up a while loop that will keep going until the end of the record set is reached:

```
<%
while not MyRecordSet.EOF
```

To output HTML from within the ASP script, we use the **response.write** command. You use this command by sending a string to it in quotes, or a function or object that returns a string. As the next line needs a string made up of both text and the output of the query, we can turn them all into one string by using the & operator. The script continues:

```
response.write "<TR><TD>" & MyRecordSet.fields(0) & "</TD><TD>" &
MyRecordSet.fields(1) & "</TD></TR>"
```

There are several ways of accessing the columns that are returned by the query. On this occasion we send the fields command to the record set, with the number of the field specified by using:

```
MyRecordSet.fields(0)
```

This selects the first column returned by the query, which in this example is the host column. Field 2 is the user column. On this occasion the fields that are returned are in the order which they were created, as we used the wildcard (*) in the select query to obtain them. However, if we had sent the query:

```
SELECT user,host
FROM user;
```

then the columns would be returned in the select order. Bear this in mind when you are using the fields command with the record set, as you may get unexpected results from not specifying the exact column name.

Now that we have output the table row for this row of the record set, we can move to the next record in the set by using the following:

```
MyRecordSet.MoveNext
```

This does not end the while loop, just moves to the next record. To trigger the end of the while loop we use:

```
wend
%>
```

This will return to the top of the while loop and if the end of the record set has not been reached repeat the processing for the next row in the set. If we are at the end of the set, the

%> shows we have finished processing the ASP for the moment, so we can complete the table by closing the table tag:

```
</TABLE>
```

Next we will perform another query which will select a different set of records, but we will access the fields specifically as opposed to using their number as above. First we add a line break and then open a table tag again. Then we can start a new batch of ASP code:

```
<BR>
<TABLE border=1>

<%
```

Before we can send a new query to the record set, we have to close the existing record set with the following:

```
MyRecordSet.Close
```

Then we send the new query and start a new while loop:

```


MyRecordSet.Open “SELECT * FROM user WHERE user !=’mafiu’”
while not MyRecordSet.EOF
```

Again, we output the table data. This time, however, we specify the column name of the query directly by using *MyRecordSet(“columnname”)* as follows:

```
response.write “<TR><TD>” & MyRecordSet(“Host”) & “</TD><TD>” &
MyRecordSet(“User”) & “</TD></TR>”
```

We move to the next record and end the loop in the same way as before:

```
MyRecordSet.MoveNext
wend
%>
```

And finally, we send the closing tags for the HTML script.

```
</TABLE>
</HTML>
```

This section may have left you rather confused, as we are really describing a programming language to which a whole book should be dedicated. This will have given you a taster of how to use ASP with the MyODBC source. If you have access to a webserver with ASP, you may wish to try and use this file to access other data or queries to give yourself more experience.

In summary, you connect to a data source in ASP by:

- Defining an ODBC Connection object.
- Defining a Record Set object.
- Opening the data source through the ODBC Connection Object.
- Linking the Record Set to the ODBC Connection.
- Sending the query to the database through the record set.
- Processing the record set in a loop, outputting columns by name or by their index.
- If you need to send a new query to the same connection, close the recordset and re-open it with the new query.
- ASP script is defined within an .asp file by surrounding it with <% ... %>. Any other text within the file is output directly as HTML.

The connection details described in this section are not specific to MySQL and MyODBC; as long as you have the correct data source name and the same tables, this will work with an ODBC data source from any database vendor.

MySQL and ColdFusion

Connecting to a data source in ColdFusion is done by using ColdFusion Markup Language tags (CMFL) embedded within HTML code. When the ColdFusion server encounters CMFL tags it processes the results and returns them as text to be send back to the webserver.

ColdFusion allows you to set up a data source remotely by using its remote administrator website. Although we have already set up a data source using the Microsoft Control Panel, we will demonstrate how you can do this through ColdFusion. The next section will use ColdFusionMX running on localhost as a demonstration.

Creating a Data Source in ColdFusion

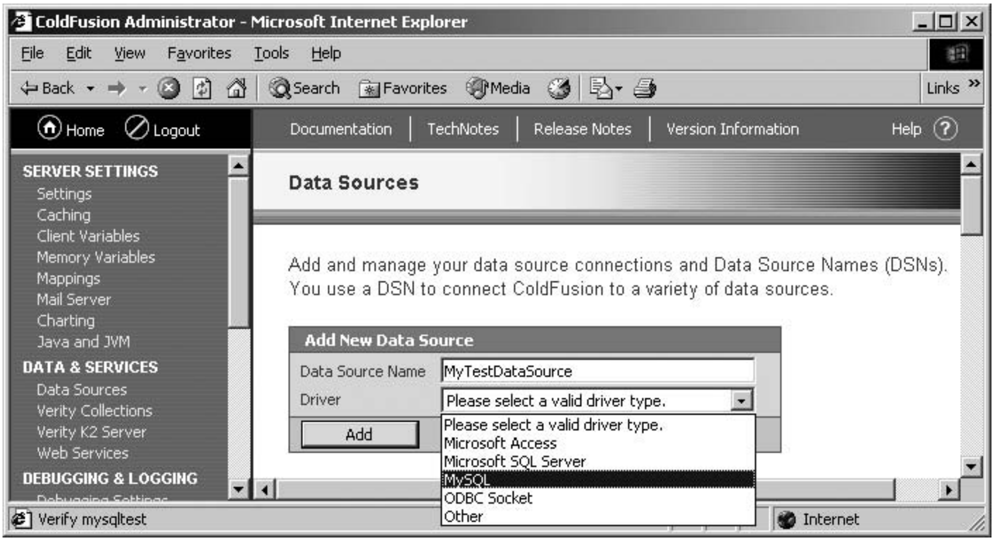
The ColdFusionMX administrator is accessed by pointing an internet browser at the following location:

```
http://localhost/CFIDE/Administrator
```

After entering your password, you are presented with a menu which has a **Data Sources** link. Clicking on this will give you access to the data source setup page. Part of this page is shown in Figure 14.10.

Type in the name of the data source you wish to create on the page shown in Figure 14.10. Then select the MySQL driver from the driver menu. Once selected, click the **Add** button to take you to the page shown in Figure 14.11.

Figure 14.11 shows the dialogue that allows you to specify the details for the MyODBC connection. You can fill this out using the same details and warnings that we used in the section on setting up the data source with the Microsoft ODBC control panel. Once you submit this form, the source is immediately tested by ColdFusion. If you get a page such as



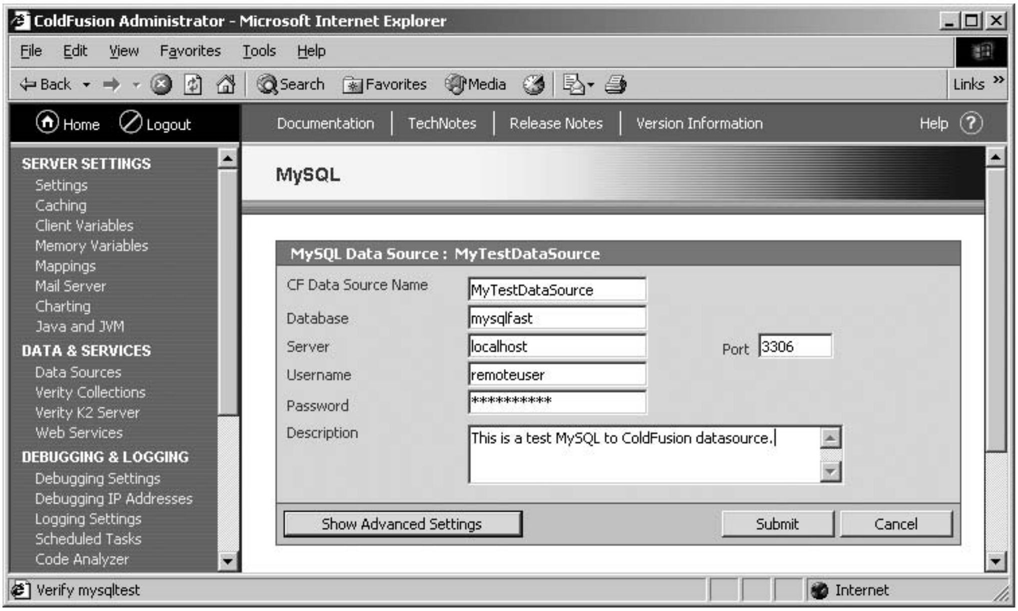
Data Sources

Add and manage your data source connections and Data Source Names (DSNs). You use a DSN to connect ColdFusion to a variety of data sources.

Add New Data Source

Data Source Name	MyTestDataSource
Driver	MySQL

Figure 14.10 Adding a data source in ColdFusionMX.

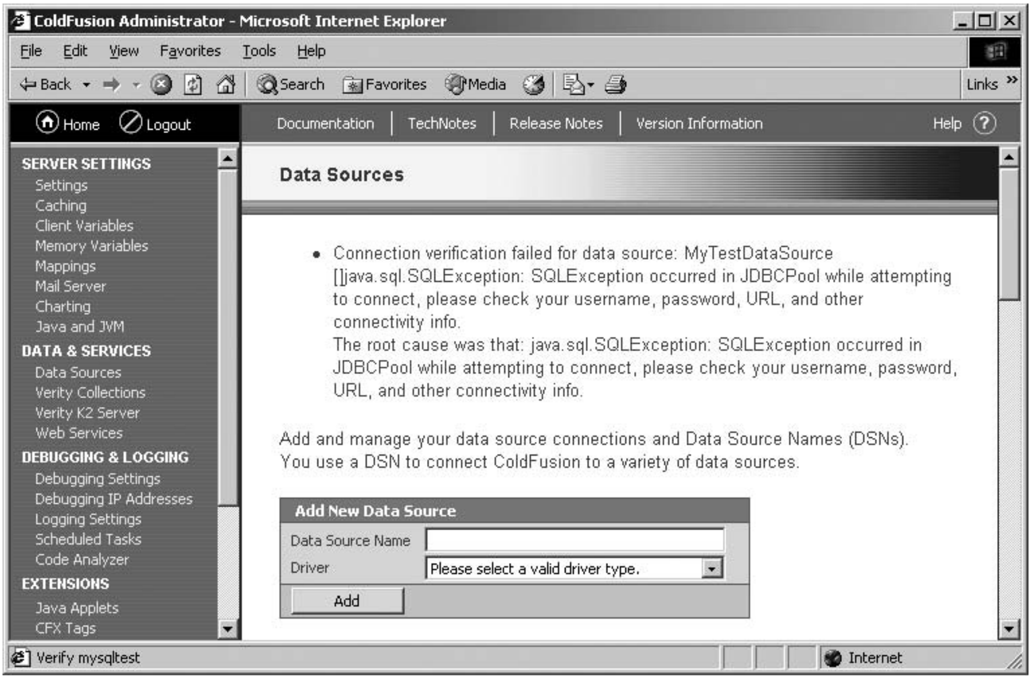


MySQL

MySQL Data Source : MyTestDataSource

CF Data Source Name	MyTestDataSource
Database	mysqlfast
Server	localhost
Port	3306
Username	remoteuser
Password	**********
Description	This is a test MySQL to ColdFusion datasource.

Figure 14.11 Specifying the data source in ColdFusionMX.



- Connection verification failed for data source: MyTestDataSource
  []java.sql.SQLException: SQLException occurred in JDBCPool while attempting to connect, please check your username, password, URL, and other connectivity info.
  The root cause was that: java.sql.SQLException: SQLException occurred in JDBCPool while attempting to connect, please check your username, password, URL, and other connectivity info.

Add and manage your data source connections and Data Source Names (DSNs). You use a DSN to connect ColdFusion to a variety of data sources.

Data Source Name
Driver: Please select a valid driver type.

Figure 14.12 A data source connection error.

the one shown in Figure 14.12, you have an error in your settings, so use the Back button on your browser and check that you have entered everything correctly and that the user has the correct access from the machine that the ColdFusion server is running on.

If all is working correctly with the data source then you will see a page similar to Figure 14.13. If this is the only data source on the system, you should see the OK after your data source name to show that it is verified. If you have other data sources installed, you may have to hunt around a bit for it.

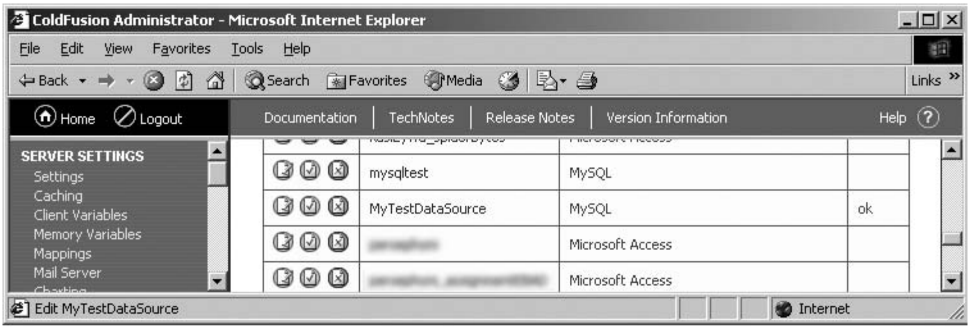
Accessing the Data Source in ColdFusion

A webpage that is going to be processed by ColdFusion will reside somewhere in your web space of your webserver and have the extension .cfm, for example:

```
c:\inetpub\wwwroot\index.cfm
```

This file is often referred to as a ColdFusion template file. To send this file to the ColdFusion server it has to be requested through the webserver using an internet browser by going to the following location:

```
http://localhost/index.cfm
```


ColdFusion Administrator - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Links »
Home Logout
Documentation | TechNotes | Release Notes | Version Information
Help
SERVER SETTINGS
Settings
Caching
Client Variables
Memory Variables
Mappings
Mail Server
Internet

☐ ☑ ☒	mysqltest	MySQL	
☐ ☑ ☒	MyTestDataSource	MySQL	ok
☐ ☑ ☒	[redacted]	Microsoft Access	
☐ ☑ ☒	[redacted]	Microsoft Access	

Edit MyTestDataSource

Figure 14.13 A verified data source.

The template file is a collection of CFML code and HTML tags. The CFML tags control sending of data to the data source and control the output of the HTML code.

The following template, when executed through a ColdFusion server, will send a query to our data source and output some of the results:

```
<HTML>
     <TITLE>Testing MySQL ODBC Connection</TITLE>
<BODY>
     <CFQUERY name=”getusers” datasource=”MyTestDataSource”>
            SELECT *
            FROM user
     </CFQUERY>
     <CFOUTPUT query=”getusers”>
            #USER#@#HOST#<BR>
     </CFOUTPUT>
</BODY>
</HTML>
```

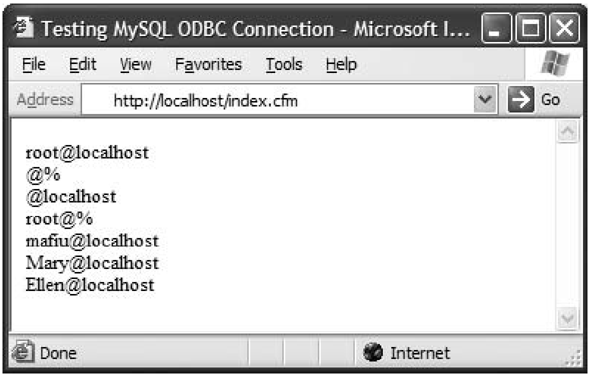
The result of running this template is shown in Figure 14.14.

We will examine the contents of this template to show what is happening. The template begins with standard HTML which is passed straight back to the webserver:

```
<HTML>
       <TITLE>Testing MySQL ODBC Connection</TITLE>
<BODY>
```

This defines the file as HTML and sets the title of the page. Then comes the first CFML tag:

```
     <CFQUERY name=”getusers” datasource=”MyTestDataSource”>
            SELECT *
            FROM user
     </CFQUERY>
```


Testing MySQL ODBC Connection - Microsoft I...
File Edit View Favorites Tools Help
Address http://localhost/index.cfm
Go
Done
Internet

```
root@localhost
@%
@localhost
root@%
mafiu@localhost
Mary@localhost
Ellen@localhost
```

Figure 14.14 Downloading MyODBC.

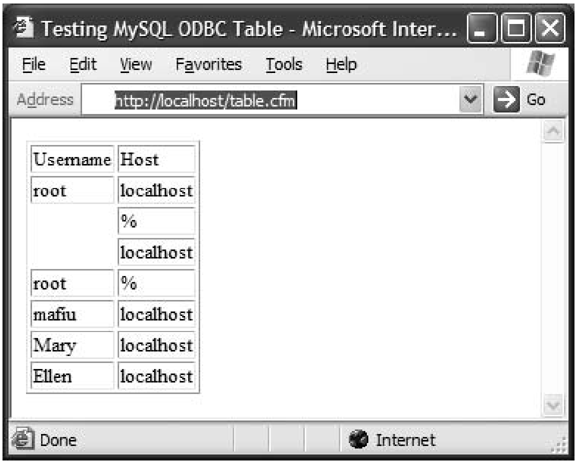
This is the CFQUERY tag. Each query needs a name, as you can use many queries within a template. The query also needs the name of the data source that it is to connect to. The text inside the tag is the actual SQL query. Unlike our ASP example earlier, all of the code for connecting to the data source and sending the query is hidden from the web developer.

To access the results of the query, we use the <CFOUTPUT> tag with a query attribute that refers to the name specified in the <CFQUERY> earlier:

```
      <CFOUTPUT query=”getusers”>
```

<CFOUTPUT> in this form will loop through the code inside it for every row of the query results. If there are no rows returned, processing continues from after the </CFOUTPUT> tag. Inside our tag we use the following:

```
            #USER#@#HOST#<BR>
```


Testing MySQL ODBC Table - Microsoft Inter...
File Edit View Favorites Tools Help
Address http://localhost/table.cfm
Go
Done
Internet

Username	Host
root	localhost
	%
	localhost
root	%
mafiu	localhost
Mary	localhost
Ellen	localhost

Figure 14.15 Downloading MyODBC.

ColdFusion identifies column names and variables within its tags by surrounding them with hashes. So for each row in the query, the line above will output the user column value, an @ sign, the host column value and a line break HTML tag. We then close the <CFOUTPUT> tag and complete the rest of the HTML page:

```
      </CFOUTPUT>
</BODY>
</HTML>
```

As this is a lot easier to follow than the ASP example, due to the simplicity of the tags, we will re-write the template so that the output is tabulated better. This template will then read as follows:

```
<HTML>
     <TITLE>Testing MySQL ODBC Table</TITLE>
<BODY>
     <CFQUERY name=”getusers” datasource=”MyTestDataSource”>
            SELECT *
            FROM user
     </CFQUERY>

     <TABLE>
            <TR><TD>Username</TD><TD>Host</TD></TR>
            <CFOUTPUT query=”getusers”>
             <TR><TD>#USER#</TD><TD>#HOST#</TD></TR>
            </CFOUTPUT>
     <TABLE>
</BODY>
</HTML>
```

The results of the above are shown in Figure 14.15. You will notice how, as when we looked at ASP, we created the HTML outside of the loop, and then looped through each row of the table using the CFOUTPUT associated with the query.

In summary, to connect and use MyODBC data within ColdFusion you:

- Send the SQL query to the data source by wrapping it in a <CFQUERY> tag which you name.
- Use the results by wrapping the column names you wish to use in hashes, and using these names and other HTML tags within a <CFOUTPUT> tag that refers to the query name.

As with ASP, this demonstration can be used to connect ColdFusion to a MyODBC data source on its server, or to any other ODBC data source that is available. It is not MyODBC specific.

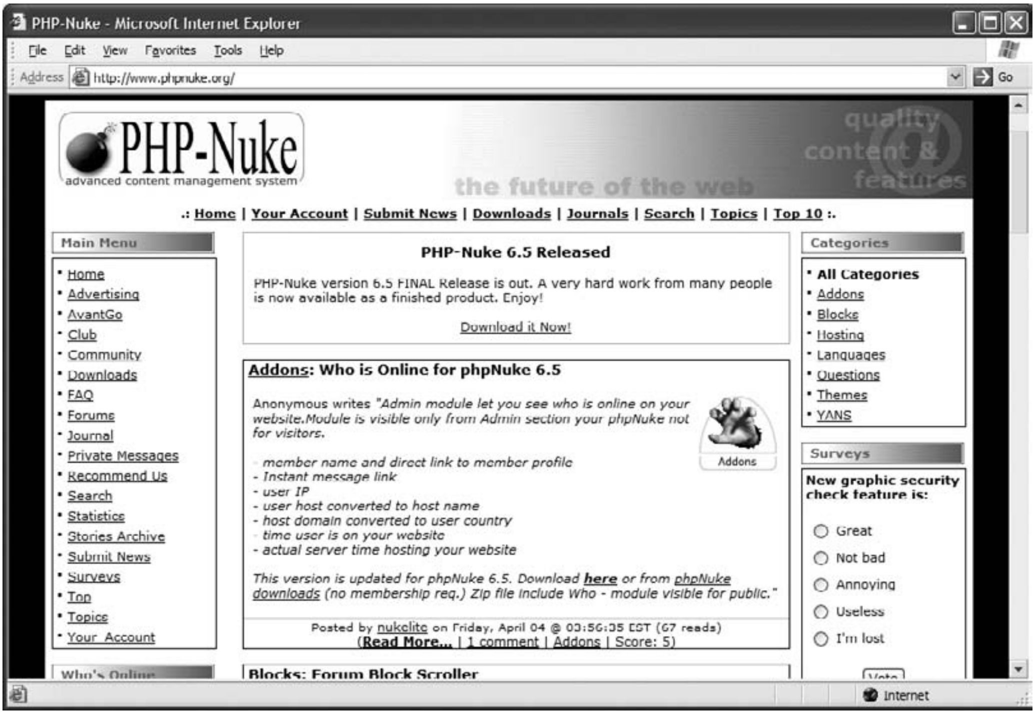
This page intentionally left blank

MySQL with PHP, Apache and Perl 15

IndigoPerl

As you probably realize by now, MySQL is a powerful product which will take a long time to explain and master. At this stage in the book I hope you will be some way to understanding how to use SQL and MySQL. MySQL shows its true power when combined with other programs such as PHP and the Apache webserver. One very common use of this combination is PHP-Nuke, which is shown in Figure 15.1.

PHP-Nuke is a system utilizing PHP and MySQL that enables you to set up a powerful, multi-user, content managed website very easily. Of course, if you wish to customize these



PHP-Nuke 6.5 Released

PHP-Nuke version 6.5 FINAL Release is out. A very hard work from many people is now available as a finished product. Enjoy!

Download it Now!

Addons: Who is Online for phpNuke 6.5

Anonymous writes *"Admin module let you see who is online on your website.Module is visible only from Admin section your phpNuke not for visitors.*

- member name and direct link to member profile
- Instant message link
- user IP
- user host converted to host name
- host domain converted to user country
- time user is on your website
- actual server time hosting your website

*This version is updated for phpNuke 6.5. Download **here** or from phpNuke downloads (no membership req.) Zip file include Who - module visible for public."*

Posted by nukelite on Friday, April 04 @ 03:56:35 EST (67 reads)
(**Read More...** | 1 comment | Addons | Score: 5)

Blocks: Forum Block Scroller

New graphic security check feature is:

○ Great
○ Not bad
○ Annoying
○ Useless
○ I'm lost

Figure 15.1 PHP-Nuke in use on their own website.

open source programs you need to know a little about the software that is running behind the scenes.

The PHP scripting language, Perl and the Apache webserver are similarly complex products, and I could devote a book to each of them. Even detailing their setup would be too much for a single chapter of this book. To enable me to demonstrate how to use MySQL with these products, I was happy to discover a system called IndigoPerl from IndigoStar software. This can be downloaded from:

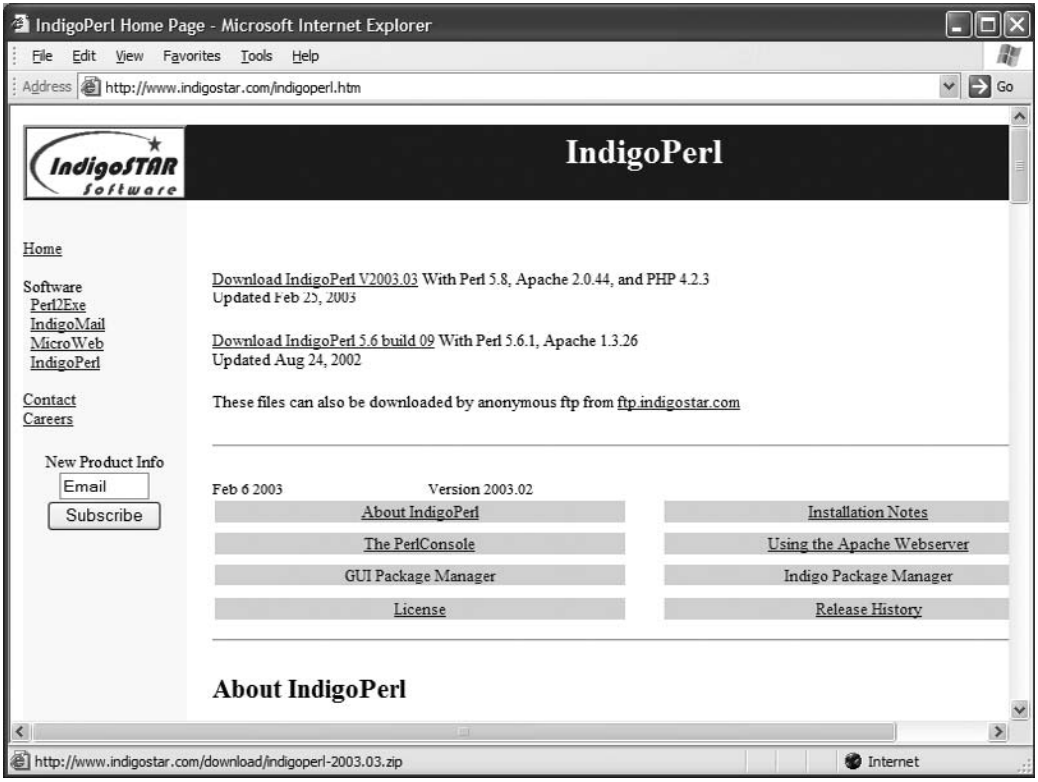
```
http://www.indigostar.com/indigoperl.htm
```

IndigoPerl combines PHP, Apache and their version of Perl as a single preconfigured product for windows.

Figure 15.2 shows the download page for IndigoPerl. You will need to download the zip file to use the examples in this chapter.

Unzip your downloaded file to a temporary directory, then run the setup.bat file which will run in a dos box as shown in Figure 15.3. After selecting the default option for the program location, the software will take several minutes to install itself.

After installation you will be prompted to reboot. After reboot you will see the new telltale, shown in Figure 15.4, in the system tray to let you know that the Apache webserver is installed.



http://www.indigostar.com/indigoperl.htm

IndigoPerl

Download IndigoPerl V2003.03 With Perl 5.8, Apache 2.0.44, and PHP 4.2.3
Updated Feb 25, 2003

Download IndigoPerl 5.6 build 09 With Perl 5.6.1, Apache 1.3.26
Updated Aug 24, 2002

These files can also be downloaded by anonymous ftp from ftp.indigostar.com

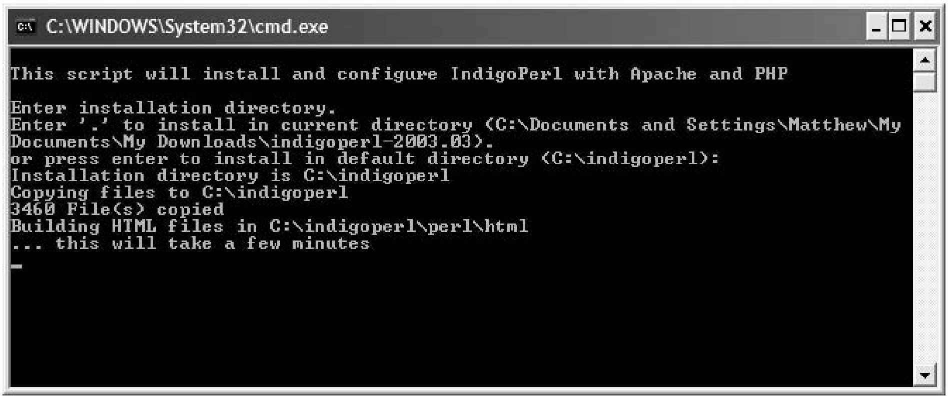
Feb 6 2003 Version 2003.02

About IndigoPerl	Installation Notes
The PerlConsole	Using the Apache Webserver
GUI Package Manager	Indigo Package Manager
License	Release History

About IndigoPerl

http://www.indigostar.com/download/indigoperl-2003.03.zip

Figure 15.2 Obtaining IndigoPerl.



```
This script will install and configure IndigoPerl with Apache and PHP

Enter installation directory.
Enter '.' to install in current directory (C:\Documents and Settings\Matthew\My
Documents\My Downloads\indigoperl-2003.03).
or press enter to install in default directory (C:\indigoperl):
Installation directory is C:\indigoperl
Copying files to C:\indigoperl
3460 File(s) copied
Building HTML files in C:\indigoperl\perl\html
... this will take a few minutes
_
```

Figure 15.3 IndigoPerl setup running.

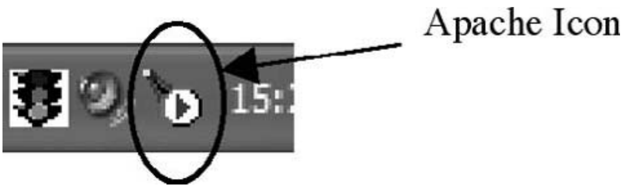


Figure 15.4 Apache webserver icon.

If the icon shows a green arrow then Apache is running. If it shows a red square then the webserver is currently stopped. To start the webserver, double click on the icon and click the start button in the dialogue box that appears.

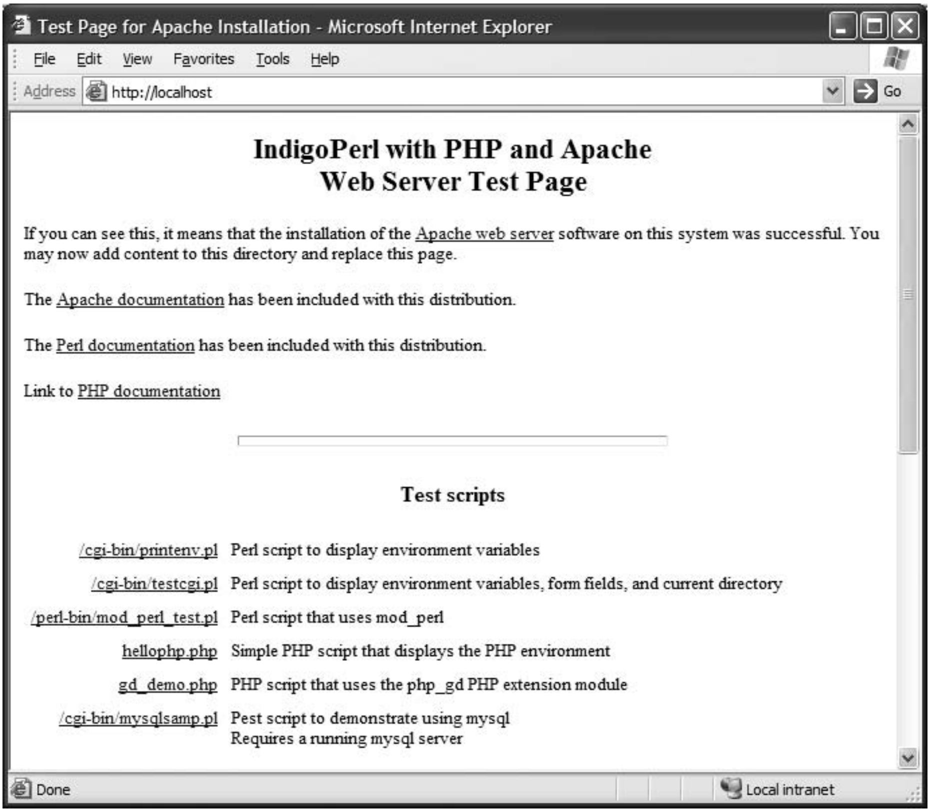
Finally, to verify that your Apache server is set up correctly, open up a web browser and go to the following location:

```
http://localhost
```

Figure 15.5 shows the default webpage that Apache installs to let you know that all is well.

The nice thing about IndigoPerl is that it will seamlessly integrate with the MySQL installation that you have used in the rest of this book. To verify this, you will notice that at the bottom of the page are several test scripts. The last script allows you to test a connection to MySQL, so click on this and you should be rewarded with the screen in Figure 15.6.

This script creates a test MySQL table and inserts a couple of rows, which it then retrieves. Although this proves that it works, it does not really show how to use MySQL from PHP so we will now write our own PHP script to access our database.


Test Page for Apache Installation - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost
Go

IndigoPerl with PHP and Apache Web Server Test Page

If you can see this, it means that the installation of the Apache web server software on this system was successful. You may now add content to this directory and replace this page.

The Apache documentation has been included with this distribution.

The Perl documentation has been included with this distribution.

Link to PHP documentation

Test scripts

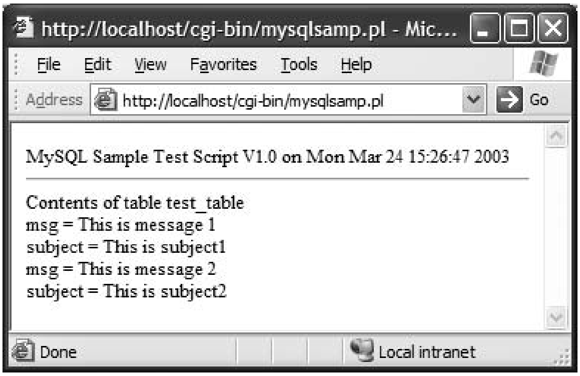
/cgi-bin/printenv.pl	Perl script to display environment variables
/cgi-bin/testcgi.pl	Perl script to display environment variables, form fields, and current directory
/perl-bin/mod_perl_test.pl	Perl script that uses mod_perl
hellophp.php	Simple PHP script that displays the PHP environment
gd_demo.php	PHP script that uses the php_gd PHP extension module
/cgi-bin/mysqlsamp.pl	Pest script to demonstrate using mysql Requires a running mysql server

Figure 15.5 Apache server default webpage.

A Quick PHP Script

PHP is a scripting language that you can embed into HTML documents. The HTML document is saved with the extension .php to indicate to the webserver that it needs to pass the file to the PHP server. Save the following to a text file:

```
<HTML>
<TITLE>Test PHP script</TITLE>
<BODY>
<?PHP
     print “It works.” ;
?>
</BODY>
</HTML>
```


http://localhost/cgi-bin/mysqlsamp.pl - Mic...
File Edit View Favorites Tools Help
Address http://localhost/cgi-bin/mysqlsamp.pl
Go
Done
Local intranet

```
MySQL Sample Test Script V1.0 on Mon Mar 24 15:26:47 2003
Contents of table test_table
msg = This is message 1
subject = This is subject1
msg = This is message 2
subject = This is subject2
```

Figure 15.6 Result of testing connection to MySQL.

Copy the above file into the Apache root directory, which in this installation will be:

```
c:\indigoperl\apache\htdocs
```

Once it is there rename it to the following:

```
test.php
```

You can now view this in a browser, which will need to be pointed at:

```
http://localhost/test.php
```

You should see a webpage with the words *It works* on it, and nothing else. If you see the PHP code and the previous tests worked, then you may not be looking at the file through the webserver, or you may have incorrectly named the file.

Inside a PHP document, all HTML tags are processed as normal, but anything surrounded by <?PHP … ?> is classed as PHP code and passed to the PHP server. The print command simply outputs the string that you provide. All PHP commands are terminated with a semicolon.

Accessing MySQL

MySQL is accessed in PHP through a series of functions whose results are stored in variables. You need to go through the following stages to access your data:

- Connect to the database
- Run the query
- Deal with the results
- Clear the results
- Close the connection to the database.

Connecting to the Database

To connect to the database you use the **mysql_connect** function as follows:

```
$connectionname = mysql_connect(“server”,“username”,“password”);
```

$connectionname is a variable that holds the MySQL connection. When you wish to access this database later on in the script, you will use this variable to reference it.

server is the host name of the server.

username and **password** are the credentials for accessing the MySQL server. You can see why it was previously suggested that you have a separate user defined for accessing your data via the web. You would not want to have to embed your administrator credentials in every webpage that you created. To quickly set up a user to gain access to MySQL from PHP on your local machine, open up the MySQLGUI and run the following:

```
GRANT SELECT       ON mysqlfast.*
                   TO phpuser@localhost
                   IDENTIFIED BY “abominable”
```

This will create a specific user to use PHP and give them access from localhost to the mysqlfast database. They will have SELECT access only, so will not be able to change anything. This is by far the safest way of granting web access to a database; only give the user the minimum access to complete the task that they need to perform. You can follow this command with the **or** keyword that then allows you to execute code if the command fails. We won’t demonstrate this here but on production pages it is always best to check for error conditions when you are scripting to make your webpages more resilient.

In our PHP script we can then use the following to connect to the MySQL server:

```
$myConnection = mysql_connect(“localhost”,“phpuser”,“abominable”);
```

Selecting the Database

Once you have connected to the server, PHP gives you a command that allows you to select the data, **mysql_select_db**. This command is used as follows:

```
mysql_select_db(“databasename”,connection);
```

databasename is the name of the database that we will be using. Your user must have rights to access the database.

connection is the variable that you assigned the connection to in the previous task. If you are only using one connection you can omit this attribute as PHP will default to the last connection made.

Sending a Query

With the connection ready, we can now send the query to the connection. PHP uses the function **mysql_query** as follows:

```
$results=mysql_query("query");
```

query is a string containing the query we wish to execute.

$results is a variable that stores the results set that the function returns.

So to send a query that selects everything from our log table, we would issue the command:

```
$myResults=mysql_query("SELECT webpageid,browser,ipnumber FROM LOG");
```

The above works fine if you are using a static query, such as the one above, because it is easy to insert the query string into the function. However, if you were executing a dynamic query, such as searching for a word that was stored in a variable that had been typed in from a webpage, it is best to store the query in a string, which you can build up as required first, and then insert this query string into the function as follows:

```
$searchterm = "Mozilla";


$query = "SELECT * FROM Log WHERE Browser LIKE '$searchterm%'";
$myResults=mysql_query($query);
```

Now the results of the query have successfully been stored in a results set, we can begin to process these results. If you were running an INSERT query you would send it using **mysql_query** and not have to worry about processing the results.

Accessing the Records

We have now stored the records in a record set variable. There are several ways in which you can obtain the results from this set. We will use **mysql_fetch_row** as follows:

```
$variable=mysql_fetch_row($resultsSet)
```

The variable that this function returns is an indexed array, starting at 0, for every cell in the current row of the results set. If we were to run the following on our results:

```
$row=mysql_fetch_row($resultsSet);
```

The following would output all of the columns in the current row:

```
print "Row: $row[0], $row[1], $row[3]<BR>";
```

The index order in the array corresponds to the order of the columns as specified in the query, or the creation order of the columns in the CREATE TABLE command if * was used.

If **mysql_fetch_row** is used a second time on the same results set, it will return the next row in the set, or FALSE if there are no more rows. Ideally, in use you will want to program some control structure around the output to handle more than one row. The following will output all of the results in the record set in a table using a **do...while** loop:

```
print “<TABLE border=1>
       <TR><TD>WebPageID</TD><TD>Browser</TD><TD>IPNumber</TD></TR>”;

       do {
       $row = mysql_fetch_row($myResults);
       print “<TR><TD>$row[0]</TD><TD>$row[1]</TD><TD>$row[2]</TD></TR>”;
       } while ($row);
```

```
print “</TABLE>”;
```

Clearing the Record Set

After we have finished with the record set we need to do some housekeeping. The record set may contain hundreds of rows if accessing a large database, which may hang around tying up system resources if you leave them there. This could have major implications if lots of people were looking at the webpage at one time. When you have finished processing your record set you can use **mysql_free_result** as follows:

```
mysql_free_result($resultsSet);
```

Closing the Connection

Finally, we have to close the connection to the database. Although this happens automatically once the script finishes processing, closing it earlier will free the resources that it holds for use by other processes. To close the connection use:

```
mysql_close($myConnection);
```

Testing the Script

We now have all that we need to connect to the database, execute the query and process the results. Here are all of the parts put together:

```
<HTML>
<TITLE>Connecting PHP to MySQL</TITLE>
<BODY>

<?PHP
     $myConnection = mysql_connect(“localhost”, “phpuser”, “abominable”)
                                  or die(“Could not connect”);
     mysql_select_db(“mysqlfast”) or die(“Could not select database”);
     $query = “SELECT webpageid,browser,ipnumber FROM Log”;
     $myResults=mysql_query(“$query”)or die(“Invalid query: “ .
     mysql_error());
     print “<TABLE border=1>
     <TR><TD>WebPageID</TD><TD>Browser</TD><TD>IPNumber</TD></TR>”;
     do {
     $row = mysql_fetch_row($myResults);
     print “<TR><TD>$row[0]</TD><TD>$row[1]</TD><TD>$row[2]</TD></TR>”;}
             while ($row);
     print “</TABLE>”;
     mysql_free_result($resultsSet);
     mysql_close($myConnection);
?>
</BODY>
</HTML>
```

I have put a little error trapping in the script so that if you want to try a few different queries you can do so and get some feedback if there are any problems. This script can be seen running in Figure 15.7.

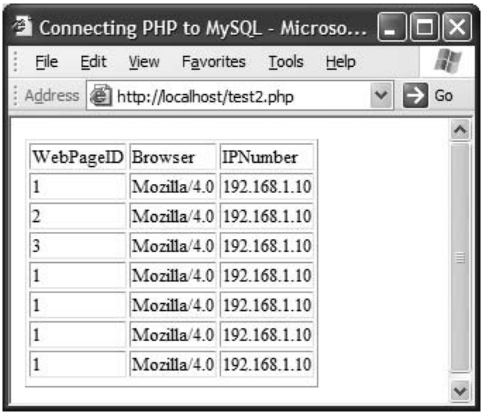
Working with Perl

Perl is an interpreted programming language that you can use to write CGI scripts. Like some of the other applications we have just discussed, Perl accesses databases through a set of driver functions that are known as DBI (Database Interface). Perl scripts reside in a cgi-bin directory of your webserver, so in our example installation ours are stored at:

```
C:\indigoperl\apache\cgi-bin
```

Figure 15.6 has already shown one of the scripts in this directory running when we pointed the browser at:

```
http://localhost/cgi-bin/mysqlsamp2.pl
```


Connecting PHP to MySQL - Microso...
File
Edit
View
Favorites
Tools
Help
Address
http://localhost/test2.php
Go

WebPageID	Browser	IPNumber
1	Mozilla/4.0	192.168.1.10
2	Mozilla/4.0	192.168.1.10
3	Mozilla/4.0	192.168.1.10
1	Mozilla/4.0	192.168.1.10
1	Mozilla/4.0	192.168.1.10
1	Mozilla/4.0	192.168.1.10
1	Mozilla/4.0	192.168.1.10

Figure 15.7 The working PHP script.

You would normally need to install the DBI drivers for MySQL to get Perl working with MySQL, but the IndigoPerl installation has already done this for us.

As we have done previously, we will now create a Perl file in the cgi-bin directory above and show the various steps needed to connect to your database in Perl. Perl files are identified with the .pl extension. Create a blank text file in the cgi directory as follows:

```
C:\indigoperl\apache\cgi-bin\mytest.pl
```

You can then drop this file into Notepad and begin to build up the Perl script.

Defining a Perl and HTML Document

Although the .pl extension and the position of the file in the web root indicates to Apache that the file is to be handled by the Perl and CGI system, we have to let the CGI processor know that the script it is using is Perl. This is indicated by the following line at the start of the file:

```
#!perl
```

The hash symbol in Perl can be used to insert comments into a script, or to temporarily disable code for testing. In this occasion, however, this command will work even though it looks as if it is commented out.

As already mentioned, Perl uses a DBI to connect to a database. We need to let Perl know that we will be using this interface with the command:

```
use DBI;
```

Some texts say that this command can be omitted, and the DBI will be invoked automatically as it is used later in the script. However, in this particular implementation, you do have to let ActivePerl know that it will be using the DBI in advance. You will also notice that this command is followed by a semicolon. This is the standard termination for a command in Perl. We can now begin to get Perl to set up output to an HTML page, and begin building this page. We do this with the following:

```
print “Content-type: text/html\n\n”;
print “<html><TITLE>mytest.pl</TITLE><body>\n”;
```

You will notice that we have used the command **print**. This command will output what follows it to an output stream. As this is being called by a web browser the stream should be back to Apache. In other applications we have used in this book we have just started the webpage with “<HTML><TITLE>...”, however the CGI processor needs to know the MIME type of the document we are creating, so the Content-type line is necessary. If you omit this line you will generate a *500 Internal Server Error* on your page view. You will get a similar error if you omit a semicolon from the end of a command in your script.

Connecting to MySQL with DBI

After defining our parameters for Perl and the webpage, we can now connect to the database, which we do with the following:

```
DBI->connect(‘DBI:mysqlPP:mysqlfast:host=localhost’,phpuser,abominable);
```

This function connects to the database, using the mysqlPP driver in the DBI. You will notice that the first string within the query contains the host details of where the MySQL server is. The second and third values in the function are the username and password of the MySQL account that we used in our PHP test earlier. The whole command above will return a database handle to the connected data source if successful, and this needs to be stored for later use, so we cannot just issue the code above on its own. Instead we insert the output into a variable as follows:

```
my $dbh =DBI->connect(
            ‘DBI:mysqlPP:mysqlfast:host=localhost’, phpuser, abominable );
```

Before we use a variable in Perl we usually should define it with the **my** command as follows:

```
my $myvariable;
```

We can then assign a value to this variable as required. However, we can cut corners a little by defining it at the same time as assigning the value by following the **my** statement with equals and providing a value to assign as above.

Generating the Query

Finally, we are at the point where we can use a query. As we have used simple queries with previous systems, our final example will complicate matters slightly with a join query as follows:

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

With the Perl DBI, we need to get the query ready before actually running it. We do this as follows:

```
my $sth = $dbh->prepare(  ‘SELECT title, datecreated, ipnumber
                                 FROM webpage, log
                                 WHERE log.webpageid=webpage.id’);
```

This time we send the prepare function with the query to the database handler, and store the results in a set handler variable. If there are any errors in the query, this function will terminate before the query is actually executed

Finally, we can execute the query by sending the execute function to the set handler:

```
$sth->execute() || quit();
```

Processing the Query Results

After successful execution of the query, the set handler variable will contain a set of records that we can process. As with PHP there are many ways in which you can do this with Perl. We will use a function similar to our PHP example, which returns the current row of the record set as an indexed array. This is called fetchrow_array() and is sent to the set handler as follows:

```
 myarray = $sth->fetchrow_array()
```

When we have a row inside the array, we use an index number to retrieve the value of that index. It is useful for us to store this value in a more meaningful variable that we can use while printing, so we can use the following to do this:

```
 my $pagetitle = $myarray[0];
```

The two commands above on their own will only get the information from one row, so to use this on a multiple-row query we need to wrap it up in a control structure as follows:

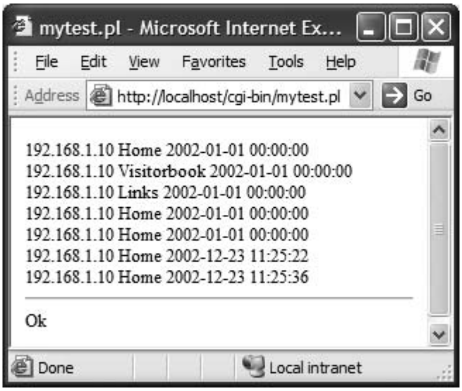
```
 while (@myarray = $sth->fetchrow_array()) {
 my $pagetitle = $myarray[0];
 my $ipnumber = $myarray[1];
 my $datecreated = $myarray[2];
 print “$datecreated $pagetitle $ipnumber<BR>\n”;
 }
```

This will loop through each row of the record set, retrieving, processing and printing the row each time. It will stop once the last row has been processed.

All that remains is to finish off the HTML file as follows:

```
print”<HR>Ok”;
print”</BODY></HTML>\n”;
```

I have included the OK line to indicate that the script has got to the end without error. If an error occurs anywhere on the script, you can often get just a blank document or a half-completed webpage returned. If you are writing a Perl script incrementally and testing each part, it is often useful to put an indicator like this on the last line so that you can see whether your last piece of code has executed successfully or not.


mytest.pl - Microsoft Internet Ex...
File Edit View Favorites Tools Help
Address http://localhost/cgi-bin/mytest.pl
Go
Done
Local intranet

```
192.168.1.10 Home 2002-01-01 00:00:00
192.168.1.10 Visitorbook 2002-01-01 00:00:00
192.168.1.10 Links 2002-01-01 00:00:00
192.168.1.10 Home 2002-01-01 00:00:00
192.168.1.10 Home 2002-01-01 00:00:00
192.168.1.10 Home 2002-12-23 11:25:22
192.168.1.10 Home 2002-12-23 11:25:36
________________________________________
Ok
```

Figure 15.8 Perl has successfully retrieved MySQL data.

So here is the complete script which can be seen running in Figure 15.8:

```
#!perl
use DBI;

print “Content-type: text/html\n\n”;
print “<HTML><TITLE>mytest.pl</TITLE><body>\n”;

my $dbh = DBI->connect(‘DBI:mysqlPP:mysqlfast:host=localhost’,
                                     phpuser,abominable);

my $sth = $dbh->prepare(‘SELECT title, datecreated, ipnumber from webpage, log
                              WHERE log.webpageid=webpage.id’);

$sth->execute() || quit();
while (@myarray = $sth->fetchrow_array()) {
     my $pagetitle = $myarray[0];
     my $ipnumber = $myarray[1];
     my $datecreated = $myarray[2];
     print “$datecreated $pagetitle $ipnumber<BR>\n”;
     }
print”<HR>Ok”;
print”</BODY></HTML>\n”;
```

Coping with Errors

If you have typed the script in and tested it, you will hopefully see the screen shown in Figure 15.8. If you have made a typing error, you may have a screen as shown in Figure 15.9. If your code is working correctly but you want to try and break the code, try changing the password in the connect string and trying again.

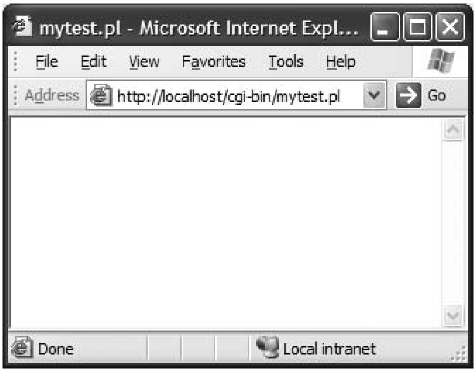

mytest.pl - Microsoft Internet Expl...
File Edit View Favorites Tools Help
Address http://localhost/cgi-bin/mytest.pl
Go
Done
Local intranet

Figure 15.9 An error has occurred.

All that you can tell from the screen in Figure 15.9 is that something has gone wrong. Perl itself is not as easy to use when errors occur as some of the other systems we have used in this book, and so to get some feedback we have to do some work as a programmer. First we have to redirect any errors that may be generated back to the standard output stream as follows:

```
BEGIN {
 $| = 1;
 open (STDERR, “>&STDOUT”);
}
```

Next we have to output an error after a command that may contain an error has executed. The DBI has an option to do this by using the following code after the connect() function before the semicolon:

```
or die “\n\nConnect Error: “ . $dbh->errstr;
```

This forces the processing of the script to stop if an error is encountered. You can send a message to the error stream at this time and include the actual error message using the **errstr** command.

Reformatting the Output

While we are modifying this script we will tidy up the output of the query slightly by putting the results in a table as follows:

```
print “<TABLE border=1>”;
print “<TR><TD>Date Accessed</TD><TD>Page Title:</TD><TD>IP
Number</TD></TR>”;
while (@myarray = $sth->fetchrow_array()) {
     my $pagetitle = $myarray[0];
     my $ipnumber = $myarray[1];
     my $datecreated = $myarray[2];
     print"<TR><TD>$datecreated</TD><TD>$pagetitle</TD>
                                   <TD>$ipnumber</TD></TR>";
     }
print "</TABLE>";
```

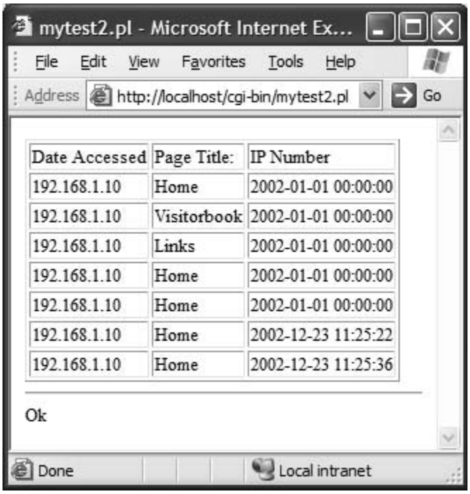
The above also puts the column names in a header row. Notice that the row definition and column names are outside of the loop, and the recurring rows are on the inside.

Here is the whole modified script with the error handling and reformatting included:

```
#!perl
BEGIN {
 $| = 1;
 open (STDERR, ">&STDOUT");
}

use DBI;
print "Content-type: text/html\n\n";
print "<html><TITLE>mytest2.pl</TITLE><body>\n";
my $dbh =
DBI->connect('DBI:mysqlPP:mysqlfast:host=localhost',phpuser,abominable) or
die "\n\nConnect Error: " . $dbh->errstr;
my $sth = $dbh->prepare('SELECT title, datecreated, ipnumber from webpage,
log
WHERE log.webpageid=webpage.id') or die "\n\nPrepare Error: " . $dbh-
>errstr;
$sth->execute() || quit();

print "<TABLE border=1>";
print "<TR><TD>Date Accessed</TD><TD>Page Title:</TD><TD>IP
Number</TD></TR>";
while (@myarray = $sth->fetchrow_array()) {
     my $pagetitle = $myarray[0];
     my $ipnumber = $myarray[1];
     my $datecreated = $myarray[2];
     print"<TR><TD>$datecreated</TD><TD>$pagetitle</TD>
                                   <TD>$ipnumber</TD></TR>";
     }
print "</TABLE>";
print"<HR>Ok";
print"</BODY></HTML>\n";
```

If you run this you will see the results shown in Figure 15.10.

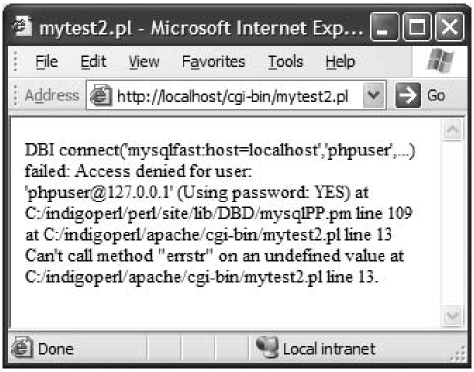
However, if you made a typing error while entering your script, you may have got different results. Hopefully, the results will give you an indication of what the error was.


mytest2.pl - Microsoft Internet Ex...
File Edit View Favorites Tools Help
Address http://localhost/cgi-bin/mytest2.pl
Go
Done
Local intranet

Date Accessed	Page Title:	IP Number
192.168.1.10	Home	2002-01-01 00:00:00
192.168.1.10	Visitorbook	2002-01-01 00:00:00
192.168.1.10	Links	2002-01-01 00:00:00
192.168.1.10	Home	2002-01-01 00:00:00
192.168.1.10	Home	2002-01-01 00:00:00
192.168.1.10	Home	2002-12-23 11:25:22
192.168.1.10	Home	2002-12-23 11:25:36

Ok

Figure 15.10 More error checking and neater formatting.


mytest2.pl - Microsoft Internet Exp...
File Edit View Favorites Tools Help
Address http://localhost/cgi-bin/mytest2.pl
Go
Done
Local intranet

```
DBI connect('mysqlfast:host=localhost','phpuser',...)
failed: Access denied for user:
'phpuser@127.0.0.1' (Using password: YES) at
C:/indigoperl/perl/site/lib/DBD/mysqlPP.pm line 109
at C:/indigoperl/apache/cgi-bin/mytest2.pl line 13
Can't call method "errstr" on an undefined value at
C:/indigoperl/apache/cgi-bin/mytest2.pl line 13.
```

Figure 15.11 Error checking at work.

Figure 15.11 shows how our error-handling code has given us a meaningful error when the password was incorrect on the connect string. Although this does not exactly pinpoint the problem, the diagnostics given are close enough for you to establish whereabouts the error occurred and some clue as to correcting it.

Making life easier 16

Other Tools

In the final chapter of this book I thought I would introduce you to two more tools that can make using MySQL easier. I did not mention these earlier because by using these tools from the start you would not have been able to learn as much of the basic SQL that is required to really use MySQL well. These programs are:

- MySQL Control Centre, the new graphical console for MySQL from the developers.
- PHPMyAdmin, a web-based interface to Administer MySQL functions.

MySQL Control Centre

MySQL Control Centre can be downloaded free from the Graphical clients section of the MySQL.com downloads area:

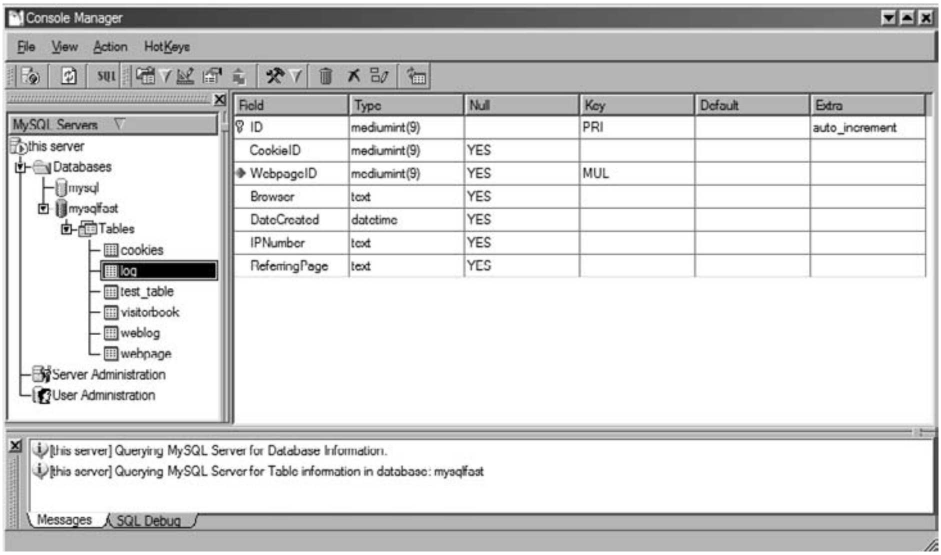
```
http://www.mysql.com/downloads/
```

The control centre is distributed as a zip file which when uncompressed gives you a setup program to run. The setup is reasonably standard but also installs various language packs that you may wish to deselect to save space if not needed.

Once installed and running, you will see the screen shown in Figure 16.1. The left side has a standard menu tree where you can select any of the MySQL objects. The currently selected object in the figure is the Log table. When you select a table, the panel on the right-hand side of the screen will give you some information about that table, similar to if you had executed a DESCRIBE LOG command. This screen gives you a quick overview of the whole system.

The control centre shows its power when you right click on an object. Figure 16.2 shows the context menu that appears when you right click on the Log table. Here you are given a menu which allows you to use the Log table to great extent.

Selecting the **Edit Table** option from the context menu reveals the screen shown in Figure 16.3. Here you can change the design of your table with the graphical interface. You will note that in the figure the **Insert Row** icon is just about to be selected. This allows you to add a new column to the existing table without having to remember the ALTER TABLE syntax. In this case the new column would be added before the **Browser** column.



Field	Type	Null	Key	Default	Extra
ID	mediumint(9)		PRI		auto_increment
CookieID	mediumint(9)	YES			
WebpageID	mediumint(9)	YES	MUL		
Browser	text	YES			
DateCreated	datetime	YES			
IPNumber	text	YES			
ReferringPage	text	YES			

[this server] Querying MySQL Server for Database Information.
[this server] Querying MySQL Server for Table information in database: mysqlfast

Figure 16.1 The main MySQL Control Centre screen.

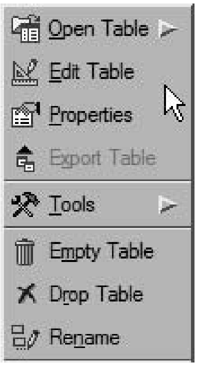
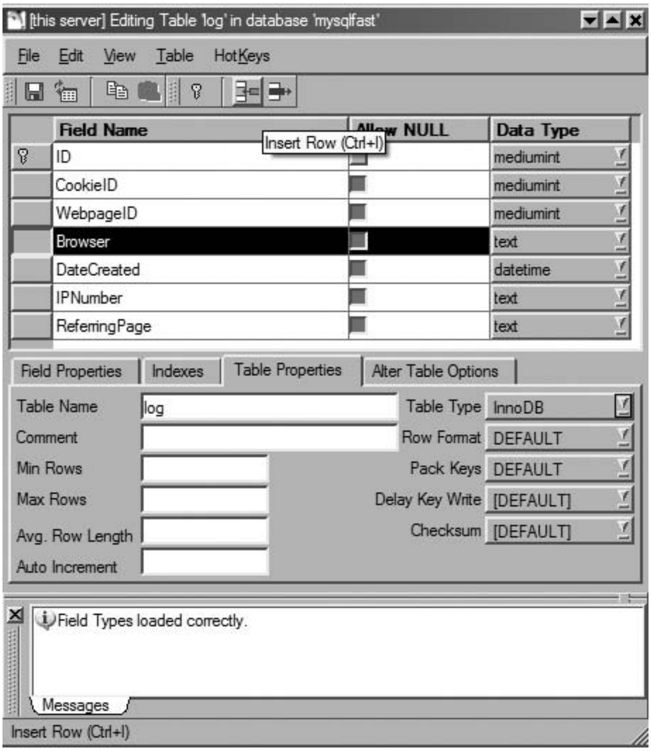


Figure 16.2 A table's context menu.

Below the list of column names are the properties for the object selected. In the figure we have just selected the **Table Properties.** This outlines the attributes of the table and even allows you to change table types from a dropdown list. Each individual column has properties that you can also change if required.

Under these options is a status window that gives you any messages that the MySQL server sends while you are navigating your database.

Figure 16.4 shows the screen you are taken to if you select one of the **Open Table** options from the context menu. This time you have a screen that shows the results of the query. In the figure, we have just pressed the **SQL** button which has opened a window that contains the select query that the Control Centre used to generate the results set. A WHERE clause was then added to the select statement and the Execute Query button (the exclamation



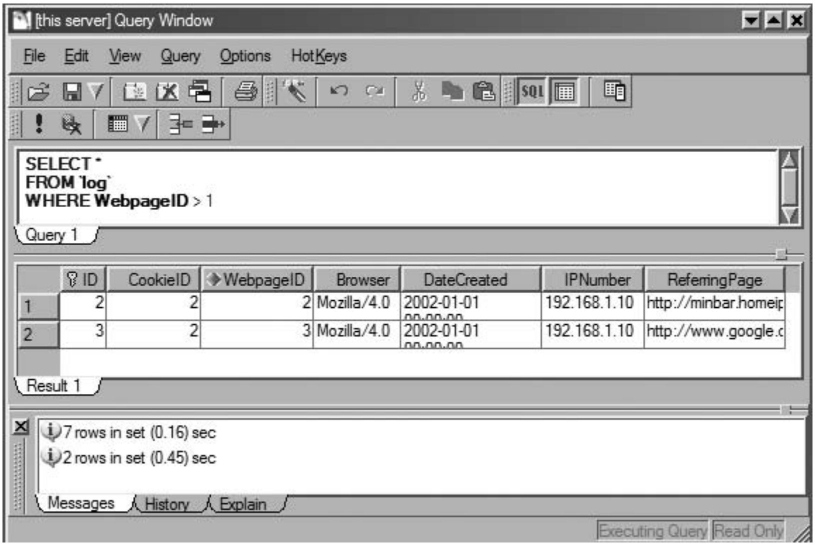
[this server] Editing Table 'log' in database 'mysqlfast'

	Field Name	Allow NULL	Data Type
🔑	ID	□	mediumint
	CookieID	■	mediumint
	WebpageID	■	mediumint
	Browser	□	text
	DateCreated	■	datetime
	IPNumber	■	text
	RefeningPage	■	text

Table Name: log
Comment:
Min Rows:
Max Rows:
Avg. Row Length:
Auto Increment:
Table Type: InnoDB
Row Format: DEFAULT
Pack Keys: DEFAULT
Delay Key Write: [DEFAULT]
Checksum: [DEFAULT]

Field Types loaded correctly.

Figure 16.3 Altering the design of your table.



[this server] Query Window

```
SELECT *
FROM `log`
WHERE WebpageID > 1
```

	ID	CookieID	WebpageID	Browser	DateCreated	IPNumber	RefeningPage
1	2	2	2	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http://minbar.homeip
2	3	2	3	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http://www.google.c

7 rows in set (0.16) sec
2 rows in set (0.45) sec

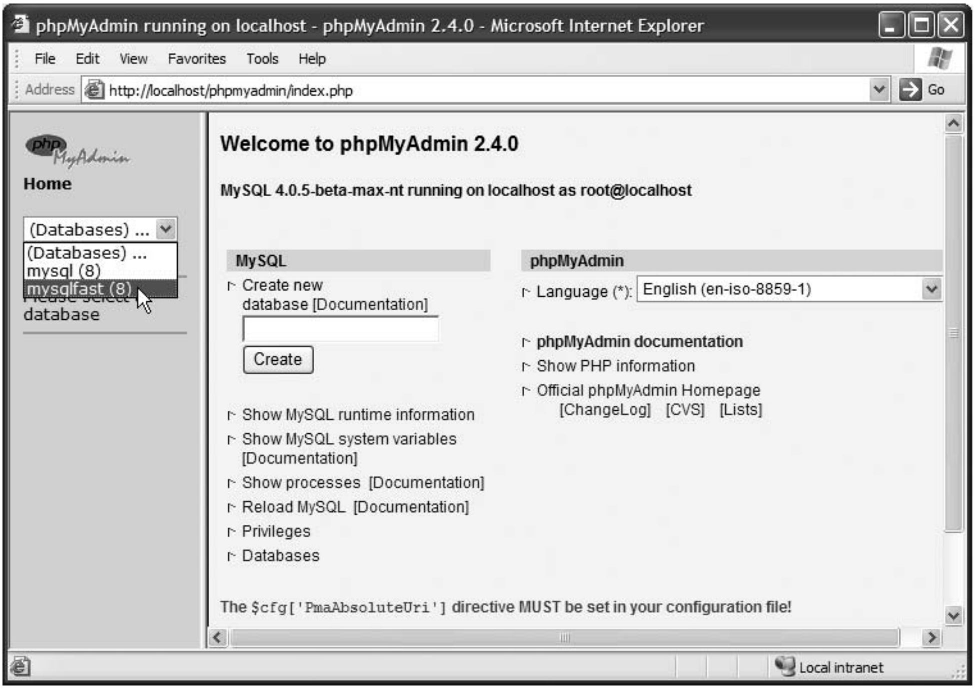
Figure 16.4 Lots of information about your running query.

mark) clicked. The query then runs and the results pane is updated. You can see in the figure that the status window contains the rows returned for the first and second queries that were run, and other tabs on this window allow you to return to previous queries and even EXPLAIN the query as it runs.

We used the MySQLGUI in this book because its table output was clear and uncluttered. Although the Control Centre screen presents us with a lot more information, it also has one hidden function that the GUI did not have. If you click on one of the cells in the results pane, and the cell contains a single column value, you can edit the contents of the cell. This allows you to change your data without using INSERT or UPDATE queries, although this is what the MySQL server does in the background. It is quite common to use a select statement to match a primary key in the Control Centre interface, and then edit some of the other columns by clicking in the results pane. If the cell to be changed is a large column type, like a text type, you will get the value appearing in another window to edit.

You will also see why introducing you to this earlier may have stopped you from learning SQL, as this tool simplifies database access considerably. However, you still need to use your knowledge of SQL to get the most from this sort of tool. You will also realize that you still need SQL skills to embed it correctly inside scripting languages.

The MySQL Control Centre is a much superior product to the GUI, which explains why development of the latter has ceased. The Control Centre is also similar to the Graphical tools that are provided with Oracle, Microsoft SQL server and others, and really proves that MySQL has now come of age and can compete with these products.



phpMyAdmin running on localhost - phpMyAdmin 2.4.0 - Microsoft Internet Explorer

Address http://localhost/phpmyadmin/index.php

Welcome to phpMyAdmin 2.4.0

MySQL 4.0.5-beta-max-nt running on localhost as root@localhost

Language (*): English (en-iso-8859-1)

The $cfg['PmaAbsoluteUri'] directive MUST be set in your configuration file!

Figure 16.5 PHPMyAdmin main page.

PHPMyAdmin

PHPMyAdmin is a series of scripts that you can use in conjunction with Apache and PHP to administer your webserver. As we have already set these programs up with IndigoPerl, the PHPMyAdmin setup is very simple. This can currently be obtained as a download from:

```
http://www.phpmyadmin.net
```

You may also find the collection by doing a Google search.

Once downloaded and unzipped, you need to copy the files to the following directory in your web root directory:

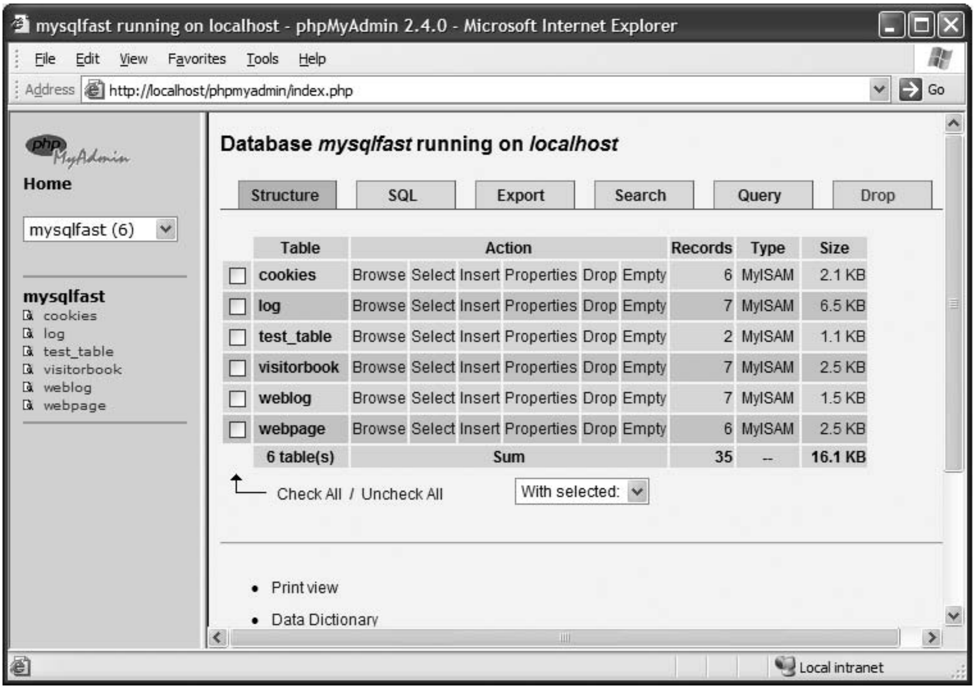
```
c:\indigoperl\apache\htdocs\phpmyadmin
```

You can then immediately access this tool by pointing your web browser at the following address:

```
http://localhost/phpmyadmin/index.php
```

Figure 16.5 shows the front page of the admin website.

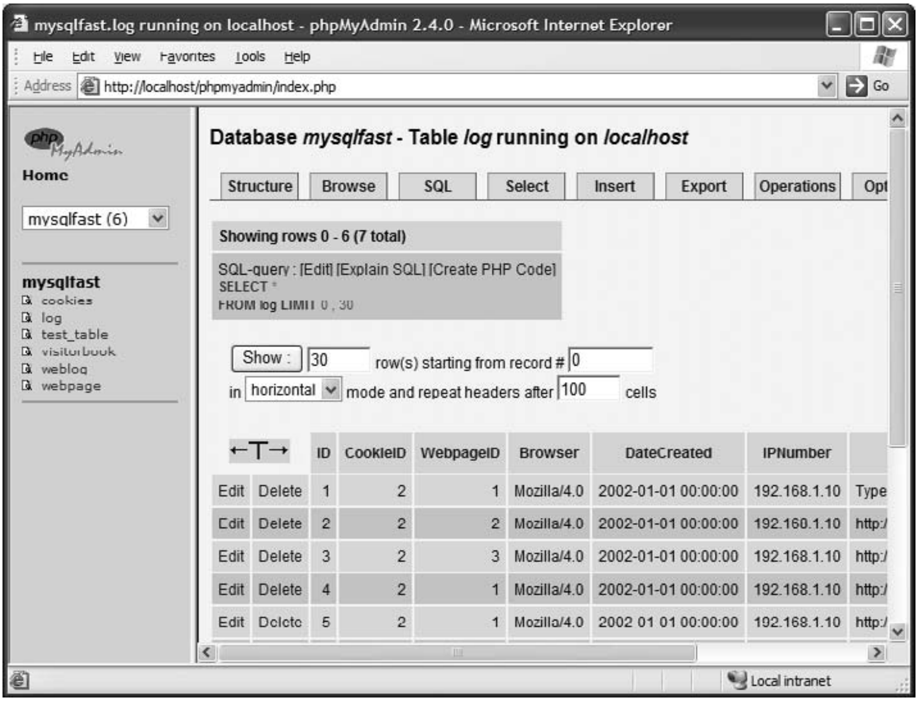
Because this tool is web based, it means that if your MySQL installation is web enabled, you can access it from any machine with a web browser and internet connection. You do not



Database *mysqlfast* running on *localhost*

	Table	Action						Records	Type	Size
☐	cookies	Browse	Select	Insert	Properties	Drop	Empty	6	MyISAM	2.1 KB
☐	log	Browse	Select	Insert	Properties	Drop	Empty	7	MyISAM	6.5 KB
☐	test_table	Browse	Select	Insert	Properties	Drop	Empty	2	MyISAM	1.1 KB
☐	visitorbook	Browse	Select	Insert	Properties	Drop	Empty	7	MyISAM	2.5 KB
☐	weblog	Browse	Select	Insert	Properties	Drop	Empty	7	MyISAM	1.5 KB
☐	webpage	Browse	Select	Insert	Properties	Drop	Empty	6	MyISAM	2.5 KB
	6 table(s)	Sum						35	--	16.1 KB

Figure 16.6 Listing the tables in a database.


mysqlfast.log running on localhost - phpMyAdmin 2.4.0 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/phpmyadmin/index.php
Go
phpMyAdmin
Home
mysqlfast (6)
mysqlfast
cookies
log
test_table
visitorbook
weblog
webpage
Structure
Browse
SQL
Select
Insert
Export
Operations
Opt
Show :
←T→
Local intranet

Database *mysqlfast* - Table *log* running on *localhost*

Showing rows 0 - 6 (7 total)

SQL-query : [Edit] [Explain SQL] [Create PHP Code]
SELECT *
FROM log LIMIT 0 , 30

Show : 30 row(s) starting from record # 0
in horizontal mode and repeat headers after 100 cells

		ID	CookieID	WebpageID	Browser	DateCreated	IPNumber	
Edit	Delete	1	2	1	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	Type
Edit	Delete	2	2	2	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http:/
Edit	Delete	3	2	3	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http:/
Edit	Delete	4	2	1	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http:/
Edit	Delete	5	2	1	Mozilla/4.0	2002-01-01 00:00:00	192.168.1.10	http:/

Figure 16.7 Query results via the web.

need a specific client running to enable you to administer your databases. This freedom obviously comes with some problems such as security issues. The README file included with the package explains how to password protect PHPMyAdmin to begin to secure your installation.

You can see from the screen shown in Figure 16.5 that you can select a database to work with or create a new one. You can also see that our mysqlfast database that we created in this book is in the list of databases. Selecting this database gives the complex screen shown in Figure 16.6.

This screen lists all of the tables that are in my mysqlfast database. You can access all parts of your MySQL data from this interface. One useful part is the SQL tab. Clicking on this will allow you to enter a SQL query directly into the web interface. Figure 16.7 shows the results of the following query:

```
SELECT *
FROM Log;
```

You will see that Figure 16.7 shows the results of the query, as well as giving you the options to go and edit some of the data that has been returned.

PHPMyAdmin is a great tool that I was very pleased to discover. As long as you consider the possible security implications of using it, you will find it another invaluable aid in your use of MySQL.

Index