Chapter 1

Quickstart to MySQL

So, you've obtained a copy of this book. Some of you may be competent with MySQL already and want to dive into the murky waters of database replication or optimizing the server variables. If you're an advanced user, feel free to jump ahead. This chapter is not for you. But beginners need not worry. This book also contains everything you need to get started and eventually become an advanced user.

MySQL is the world's most popular open-source database. *Open source* means that the *source code*, the programming code that makes up MySQL, is freely available to anyone. People all over the world can add to MySQL, fix bugs, make improvements, or suggest optimizations. And they do. MySQL has developed from being a "toy" database a few years ago into a grown-up version 4, having overtaken many commercial databases along the way—and terrified most of the other database manufacturers. It's grown so quickly because of the countless dedicated people who have contributed to the project in some way, as well as the dedication of the MySQL team.

Unlike proprietary projects, where the source code is written by a few people and carefully guarded, open-source projects exclude no one who is interested in contributing if they are competent enough. In 2000 when MySQL was a young upstart of four years old, Michael "Monty" Widenius, the founder of MySQL, predicted big things for MySQL as he attended the first-ever open-source database convention. Many established database vendors scoffed at the time. Some of those vendors are no longer around.

With version 3, MySQL dominated the low end of the Internet market. And with MySQL's release of version 4, the product is now appealing to a much wider range of customers. With the open-source Apache dominating the web server market and various open-source operating systems (such as Linux and FreeBSD) performing strongly in the server market, MySQL's time has come in the database market.

Featured in this chapter:

- Essential database concepts and terminology
- Connecting to and disconnecting from MySQL server
- Creating and dropping databases

- Creating, updating, and dropping tables
- Adding data into a table
- Returning data and deleting data from a table
- Understanding basic statistical and date functions
- Joining more than one table

Understanding MySQL Basics

MySQL is a *relational database management system* (RDBMS). It's a program capable of storing an enormous amount and a wide variety of data and serving it up to meet the needs of any type of organization, from mom-and-pop retail establishments to the largest commercial enterprises and governmental bodies. MySQL competes with well-known proprietary RDBMSs, such as Oracle, SQL Server, and DB2.

MySQL comes with everything needed to install the program, set up differing levels of user access, administer the system, and secure and back up the data. You can develop database applications in most programming languages used today and run them on most operating systems, including some of which you've probably never heard. MySQL utilizes Structured Query Language (SQL), the language used by all relational databases, which you'll meet later in this chapter (see the section titled "Creating and Using Your First Database"). SQL enables you to create databases, as well as add, manipulate, and retrieve data according to specific criteria.

But I'm getting ahead of myself. This chapter provides a brief introduction to relational database concepts. You'll learn exactly what a relational database is and how it works, as well as key terminology. Armed with this information, you'll be ready to jump into creating a simple database and manipulating its data.

What Is a Database?

The easiest way to understand a database is as a collection of related files. Imagine a file (either paper or electronic) of sales orders in a shop. Then there's another file of products, containing stock records. To fulfill an order, you'd need to look up the product in the order file and then look up the stock levels for that particular product in the product file. A database and the software that controls the database, called a *database management system* (DBMS), helps with this kind of task. Most databases today are *relational* databases, named such because they deal with tables of data related by a common field. For example, Table 1.1 shows the Product table, and Table 1.2 shows the Invoice table. As you can see, the relation between the two tables is based on the common field `stock_code`. Any two tables can relate to each other simply by having a field in common.

TABLE 1.1: THE PRODUCT TABLE

STOCK_CODE	DESCRIPTION	PRICE
A416	Nails, box	$0.14
C923	Drawing pins, box	$0.08

TABLE 1.2: THE INVOICE TABLE

INVOICE_CODE	INVOICE_LINE	STOCK_CODE	QUANTITY
3804	1	A416	10
3804	2	C923	15

Database Terminology

Let's take a closer look at the previous two tables to see how they are organized:

- Each table consists of many *rows* and *columns*.
- Each row contains data about one single *entity* (such as one product or one order). This is called a *record*. For example, the first row in Table 1.1 is a record; it describes the A416 product, which is a box of nails that costs 14 cents. To reiterate, the terms *row* and *record* are interchangeable.
- Each column (also called a *tuple*) contains one piece of data that relates to the record, called an *attribute*. Examples of attributes are the quantity of an item sold or the price of a product. An attribute, when referring to a database table, is called a *field*. For example, the data in the Description column in Table 1.1 are fields. To reiterate, the terms *attribute* and *field* are interchangeable.

Given this kind of structure, the database gives you a way to manipulate this data: SQL. SQL is a powerful way to search for records or make changes. Almost all DBMSs use SQL, although many have added their own enhancements to it. This means that when you learn about SQL in this chapter and in more detail in later chapters, you aren't learning something specific to MySQL. Most of what you learn can be used on any other relational database, such as PostgreSQL, Oracle, Sybase, or SQL Server. But after tasting the benefits of MySQL, you probably won't want to change!

Connecting to the MySQL Server

The machine where MySQL runs and stores the data is called the *MySQL server*. To connect to the server, you have several setup options. First, you can have the MySQL client and MySQL server on your desktop, as shown in Figure 1.1. Second, you can have the MySQL client set up on your desktop while the MySQL server is on another machine that you connect to, as shown in Figure 1.2. Finally, your desktop can be any machine connecting to another machine with a MySQL client, which in turn connects to a MySQL server, either on the same machine or another, as shown in Figure 1.3.

FIGURE 1.1
Your machine has both the MySQL client and the MySQL server on it.

FIGURE 1.2
Your machine has the MySQL client. The MySQL server exists on another machine to which you connect.

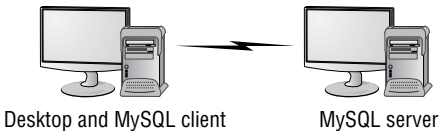

FIGURE 1.3
In this case, your terminal can be any machine capable of connecting to another, as it doesn't even run the lightweight MySQL client on it.

If the MySQL client is not on your desktop and you need to connect to a second machine to use the MySQL client, you'll probably use something such as Telnet or a Secure Shell (SSH) client to do so. Using one of these is a matter of opening the Telnet program, entering the hostname, username, and password. If you're unsure about this, ask your system administrator for help.

Once you've logged into a machine on which the MySQL client program is installed, connecting to the server is easy:

- On a Unix machine (for example, Linux or FreeBSD), run the following command from the command line within your shell:

  ```
  % mysql -h hostname -u username -ppassword databasename
  ```

- On a Windows machine, run the same command from the command prompt:

  ```
  % mysql -h hostname -u username -ppassword databasename
  ```

The `%` refers to the shell prompt. It'll probably look different on your machine—for example, `c:\>` on some Windows setups or `$` on some Unix shells. The `-h` and the `-u` can be followed by a space (you can also leave out the space), but the `-p` must be followed by the password immediately, with no intermediate spaces.

Once you've connected, you'll encounter the `mysql>` prompt, which appears in most distributions on the command line once you've connected. You don't need to type it in; it will appear automatically. Even if a slightly different prompt appears, don't worry, just enter the text in bold. This is the convention used throughout the book.

TIP *There is a more secure way of entering the password, which I recommend in a multiuser environment. Just enter the* `-p` *and omit the password. You'll be prompted for it when MySQL starts; then you can enter the password without it appearing on the screen. This avoids anyone seeing your password entered in plain text.*

The hostname would be the machine hosting the server (perhaps something such as `www.sybex.com` or an IP such as 196.30.168.20). You don't need a hostname if you're already logged into the server (in other words, the MySQL client and server are on the same machine). The administrator assigns you the username and password (this is your MySQL password and username, which is not the same as your login to the client machine). Some insecure systems don't require any username or password.

TIP *Sometimes the system administrator makes your life a little harder by not putting MySQL into the default path. So, when you type the* `mysql` *command, you may get a* `command not found` *error (Unix) or a* `bad command or file name` *error (Windows) even though you know you have MySQL installed. If this happens, you'll need to enter the full path to the MySQL client (for example,* `/usr/local/build/mysql/bin/mysql`*, or on Windows, something such as* `C:\mysql\bin\mysql`*). Ask your administrator for the correct path if this is a problem in your setup.*

To disconnect, simply type **`QUIT`** as shown:

```
mysql> QUIT
Bye
```

You can also type **`EXIT`** or press **`Ctrl+D`**.

NOTE *MySQL does not distinguish between uppercase and lowercase here. You could have typed* **`QUIT`**, **`quit`**, *or even* **`qUIt`** *if you'd wanted.*

Creating and Using Your First Database

The following sections describe how to create a database and perform queries on it. It assumes you have connected to the MySQL server and that you have permissions to use a database. If not, ask your administrator for permission. You will call your database *firstdb*, so ask your administrator to create and give you full permission to that database only. That way you won't run into permission problems later, but you also won't give your administrator a heart attack by causing havoc with existing databases. If your administrator looks flustered or if you have just installed MySQL yourself, then you or your administrator will need to perform one of the following two sets of commands to get started. Remember, just enter the text in bold.

If You've Just Installed MySQL

First, connect to the MySQL database as *root*. Because you're just starting, you won't have a root password yet, and the first thing you should do is assign a password to the root user (See Chapter 14, "Database Security").

```
% mysql -u root mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 15 to server version: 4.0.2-alpha-Max

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> SET PASSWORD=PASSWORD('g00r002b');
Query OK, 0 rows affected (0.00 sec)
```

For ease of use, you're going to use the same password for the root user, *g00r002b*, as you will for the user you're creating, *guru2b*.

Next, you will have to create the *firstdb* database that you're going to be working with throughout:

```
mysql> CREATE DATABASE firstdb;
Query OK, 1 row affected (0.01 sec)
```

Finally, the user you're going to be working as, *guru2b*, with a password of *g00r002b*, needs to be created and given full access to the *firstdb* database:

```
mysql> GRANT ALL ON firstdb.* to guru2b@localhost
 IDENTIFIED BY 'g00r002b';
Query OK, 0 rows affected (0.01 sec)
mysql> exit
Bye
```

If an Administrator Needs to Give You Permission

First the administrator will have to connect to the MySQL database as *root* (or any other user that has permissions to grant a new user their own permissions):

```
% mysql -u root -p mysql
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 15 to server version: 4.0.2-alpha-Max
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

Next, your administrator will have to create the *firstdb* database that you're going to be working with throughout:

```
mysql> CREATE DATABASE firstdb;
Query OK, 1 row affected (0.01 sec)
```

Finally, the user you're going to be working as, *guru2b*, with a password of *g00r002b*, needs to be created and given full access to the *firstdb* database. Note that this assumes you will be connecting to the database from *localhost* (i.e., the database client and database server are on the same machine). If this is not the case, your administrator will have to replace *localhost* with the appropriate host name:

```
mysql> GRANT ALL ON firstdb.* to guru2b@localhost
 IDENTIFIED BY 'g00r002b';
Query OK, 0 rows affected (0.01 sec)
mysql> exit
Bye
```

guru2b is your MySQL username, and I'll use it throughout the book, and *g00r002b* is your password. You may choose, or be assigned, another username. You'll learn how to grant permissions in Chapter 14.

Using Your Database

If you're new to SQL or MySQL, this is your chance to get your hands dirty. I suggest you do the examples in the next few sections in the order they're presented. But you only really learn when you go beyond the book and start to write your own queries. So, experiment as you go along. Try variations you think may work. Don't be afraid to make mistakes at this stage! Mistakes are the way you learn. None of the data is valuable. It'd be better to delete your sample database by accident now than the vital million records you're processing in a year's time.

You'll start by creating a table inside your sample database, and then populating this table with data. Once you've got some tables with data in them, you'll learn how to perform queries on these tables. First, connect to your newly created table with the following command:

```
% mysql -u guru2b -pg00r002b firstdb
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 15 to server version: 4.0.2-alpha-Max

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

If the permissions have not been set correctly, you'll get an error message, as follows:

```
ERROR 1044: Access denied for user: 'guru2be@localhost' to database 'firstdb'
```

You or your administrator will need to retrace the steps in the previous two sections if this is the case.

All these permission hassles may seem troublesome now, but they're a useful feature. At some stage you'll want to make sure that not just anybody can access your data, and permissions are the way you'll ensure this.

You can also connect without specifying a database, as follows:

```
% mysql -u guru2b -pg00r002b guru2b
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 15 to server version: 4.0.2-alpha-Max

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

Then to make sure you're using the right database, you'll need to let MySQL know which database to use. To make sure that any further work you do affects the right database, enter the following bold statement:

```
mysql> USE firstdb
Database changed
```

You can connect to your database either way for now, specifying the database when you connect or later once you are connected. In the future, when you have more than one database to use on the system, you'll find it much easier to change between databases with the `USE` statement.

Creating a Table

Now that you've connected to your database, you'll want to put something in it. To get going, you're going to create a database that could track a sales team. As you learned, a database consists of many tables, and to start with, you'll create a table that can contains data about sales representatives. You will store their names, employee numbers, and commissions. To create a table, you're also going to use the `CREATE` command, but you need to specify `TABLE` rather than `DATABASE`, as well as a few extras. Enter the following `CREATE` statement:

```
mysql> CREATE TABLE sales_rep(
  employee_number INT,
  surname VARCHAR(40),
```

```
  first_name VARCHAR(30),
  commission TINYINT
);
Query OK, 0 rows affected (0.00 sec)
```

WARNING *Don't forget the semicolon at the end of the line. All MySQL commands must end with a semicolon. Forgetting it is one of the prime reasons for beginner frustration. Also be aware that if you've forgotten the semicolon and press Enter, you just need to add the semicolon before pressing Enter again. MySQL accepts commands over multiple lines.*

You don't need to enter the statement exactly as printed; I've used multiple lines to make it easier to follow, but entering it on one line will make it easier for you. Also, if you use a different case, it will still work. Throughout this book, I use uppercase to represent MySQL keywords and lowercase to represent names you can choose yourself. For example, you could have entered the following:

```
mysql> create table SALES_REPRESENTATIVE(
  EMPLOYEE_NO int,
  SURNAME varchar(40),
  FIRST_NAME varchar(30),
  COMMISSION tinyint
);
```

without any problems. However, entering the following:

```
mysql> CREATE TABLES sales_rep(
  employee_number INT,
  surname VARCHAR(40),
  first_name VARCHAR(30),
  commission TINYINT
);
```

would give you this error:

```
ERROR 1064: You have an error in your SQL syntax near
 'TABLES sales_reps(employee_number INT,surname
 VARCHAR(40),first_name VARCHAR(30)' at line 1
```

because you've misspelled `TABLE`. So take care to enter the capitalized text exactly; you can rename the text appearing in lowercase without any problems (as long as you are consistent and use the same names throughout).

You may be wondering about the `INT`, `VARCHAR`, and `TINYINT` terms that appear after the fieldnames. They're what are called *data types* or *column types.* `INT` stands for *integer*, or a whole number usually ranging from –2,147,483,648 to 2,147,483,647. That's about a third of the world's population, so it should cover the sales team no matter how large it grows. `VARCHAR` stands for *variable length character*. The number in brackets is the maximum length of the character string. An amount of 30 and 40 characters should suffice for the first name and surname, respectively. And `TINYINT` stands for *tiny integer*, usually a whole number from –128 to 127. The commission field refers to a percentage value, and because no one can earn more than 100 percent, a tiny integer is sufficient. Chapter 2, "Data Types and Table Types," goes into more detail about the various column types and when to use them.

Listing Existing Tables in a Database with *SHOW TABLES*

Now that you have a table, you can confirm its existence with `SHOW TABLES`:

```
mysql> SHOW TABLES;
+-------------------+
| Tables_in_firstdb |
+-------------------+
| sales_rep         |
+-------------------+
1 row in set (0.00 sec)
```

`SHOW TABLES` lists all the existing tables in the current database. In the case of your newly created *firstdb* there's only one: *sales_rep*. So unless your short-term memory is really shot, this command probably wasn't much use. But in big databases with many tables, the name of that obscure table you created two months ago has slipped your mind. Or perhaps you're encountering a new database for the first time. That's when `SHOW TABLES` is invaluable.

Examining the Table Structure with *DESCRIBE*

`DESCRIBE` is the command that shows you the structure of a table. To see that MySQL has created your table correctly, type the following:

```
mysql> DESCRIBE sales_rep;
+-----------------+-------------+------+-----+---------+-------+
| Field           | Type        | Null | Key | Default | Extra |
+-----------------+-------------+------+-----+---------+-------+
| employee_number | int(11)     | YES  |     | NULL    |       |
| surname         | varchar(40) | YES  |     | NULL    |       |
| first_name      | varchar(30) | YES  |     | NULL    |       |
| commission      | tinyint(4)  | YES  |     | NULL    |       |
+-----------------+-------------+------+-----+---------+-------+
4 rows in set (0.01 sec)
```

There are all kinds of extra columns in this table with which you're not yet familiar. For now, you should be interested in the field and the type. You'll learn about the other headings in Chapter 2. The fieldnames are exactly as you entered them, and the two `VARCHAR` fields have the size you allocated them. Notice that the `INT` and `TINYINT` fields have been allocated a size, too, even though you didn't specify one when you created them. Remember that a `TINYINT` by default ranges from –128 to 127 (four characters including the minus sign), and an `INT` ranges from –2,147,483,648 to 2,147,483,647 (11 characters including the minus sign), so the seemingly mysterious size allocation refers to the display width.

Inserting New Records into a Table

Now that you have a table, you'll want to put some data into it. Let's assume there are three sales reps (as shown in Table 1.3).

TABLE 1.3: SALES REPS

EMPLOYEE NUMBER	SURNAME	FIRST NAME	COMMISSION PERCENT
1	Rive	Sol	10
2	Gordimer	Charlene	15
3	Serote	Mike	10

To enter this data into the table, you use the SQL statement `INSERT` to create a record, as follows:

```
mysql> INSERT INTO sales_rep(employee_number,surname,first_name,commission)
   VALUES(1,'Rive','Sol',10);
mysql> INSERT INTO sales_rep(employee_number,surname,first_name,commission)
   VALUES(2,'Gordimer','Charlene',15);
mysql> INSERT INTO sales_rep(employee_number,surname,first_name,commission)
   VALUES(3,'Serote','Mike',10);
```

NOTE *The string field (a* `VARCHAR` *character field) needs a single quote around its value, but the numeric fields (*`commission`*,* `employee_number`*) don't. Make sure you have enclosed the right field values in quotes and that you have matched quotes correctly (whatever gets an open quote must get a close quote), as this often entraps newcomers to SQL.*

There is also a shortcut `INSERT` statement to enter the data. You could have used the following:

```
mysql> INSERT INTO sales_rep VALUES(1,'Rive','Sol',10);
mysql> INSERT INTO sales_rep VALUES(2,'Gordimer','Charlene',15);
mysql> INSERT INTO sales_rep VALUES(3,'Serote','Mike',10);
```

When entering commands in this way, you *must* enter the fields in the same order as they are defined in the database. You could not use the following:

```
mysql> INSERT INTO sales_rep VALUES(1,'Sol','Rive',10);
Query OK, 1 row affected (0.00 sec)
```

Although this seems to work, the data would have been entered in the wrong order, with *Sol* as the surname and *Rive* the first name.

TIP *I suggest getting into the habit now of using the full* `INSERT` *statement, especially if you are planning to run queries through a programming language. First, it reduces the chances of error (you could not see just from the statement whether* first_name *and* surname *were in the wrong order), and second, it makes your programs more flexible. Chapter 5, "Programming with MySQL," discusses this topic in more detail.*

INSERTING DATA WITHIN ONE *INSERT* STATEMENT

Another shortcut you could have used would have been to enter all the data within one `INSERT` statement, with each record separated with a comma, as follows:

```
mysql> INSERT INTO sales_rep (employee_number,surname,first_name,commission)
VALUES
```

```
(1,'Rive','Sol',10),
(2,'Gordimer','Charlene',15),
(3,'Serote','Mike',10);
Query OK, 3 rows affected (0.05 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

This method involves less typing of course, and the server also processes it more quickly.

Inserting High Volumes of Data from a Text File with *LOAD DATA*

A final way for inserting data—and the best way if you're inserting high volumes of data all at once—is to use the `LOAD DATA` statement, as follows:

```
mysql> LOAD DATA LOCAL INFILE "sales_rep.sql" INTO TABLE sales_rep;
```

The format of the data file must be correct, with no exceptions. In this case, where you're using the defaults, the text file has each record on a new line, and each field separated by a tab. Assuming the `\t` character represents a tab, and each line ends with a newline character, the text file would have to look exactly as follows:

```
1\tRive\tSol\t10
2\tGordimer\tCharlene\t15
3\tSerote\tMike\t10
```

A variation of this is used for restoring backups, discussed in Chapter 11, "Database Backups." This is even more efficient than a multirecord `INSERT` statement. The `LOCAL` keyword tells the server that the file is found on the client machine (the machine from which you connect). Omitting it means MySQL looks for the file on the database server. By default, `LOAD DATA` assumes that the values are in the same order as in the table definition, that each field is separated by a tab, and that each record is separated with a newline.

Retrieving Information from a Table

Getting information out of a table in MySQL is easy. You can use the powerful `SELECT` statement, for example:

```
mysql> SELECT commission FROM sales_rep WHERE surname='Gordimer';
+------------+
| commission |
+------------+
|         15 |
+------------+
1 row in set (0.01 sec)
```

The `SELECT` statement has several parts. The first part, immediately after the `SELECT`, is the list of fields. You could have returned a number of other fields, instead of just commission, as follows:

```
mysql> SELECT commission,employee_number FROM
 sales_rep WHERE surname='Gordimer';
```

```
+------------+-----------------+
| commission | employee_number |
+------------+-----------------+
|         15 |               2 |
+------------+-----------------+
1 row in set (0.00 sec)
```

You could also use a wildcard (*) to return all the fields, as follows:

```
mysql> SELECT * FROM sales_rep WHERE surname='Gordimer';
+-----------------+----------+------------+------------+
| employee_number | surname  | first_name | commission |
+-----------------+----------+------------+------------+
|               2 | Gordimer | Charlene   |         15 |
+-----------------+----------+------------+------------+
1 row in set (0.00 sec)
```

The * wildcard means all fields in the table. So in the previous example, all four fields are returned, in the same order they exist in the table structure.

The part of the SELECT statement after the WHERE is called the *WHERE clause*. The clause is highly flexible and can contain many conditions, of varying kinds. Take a look at the following example:

```
mysql> SELECT * FROM sales_rep WHERE commission>10
 OR surname='Rive' AND first_name='Sol';
+-----------------+----------+------------+------------+
| employee_number | surname  | first_name | commission |
+-----------------+----------+------------+------------+
|               1 | Rive     | Sol        |         10 |
|               2 | Gordimer | Charlene   |         15 |
+-----------------+----------+------------+------------+
2 rows in set (0.00 sec)
```

Understanding AND and OR is fundamental to using SQL properly. The conditions on both sides of an AND keyword must be true for the whole to be true. Only one of the conditions in an OR statement needs to be true. Table 1.4 shows the AND/OR truth table.

TABLE 1.4: AND/OR TRUTH TABLE

KEYWORD	HOT	HUMID	OVERALL
AND	True	True	True, hot AND humid
AND	True	False	False, not hot AND humid as it's not humid
AND	False	True	False, not hot AND humid as it's not hot
AND	False	False	False, not hot AND humid as neither are true
OR	True	True	True, hot OR humid as both are true

Continued on next page

TABLE 1.4: AND/OR TRUTH TABLE *(continued)*

KEYWORD	HOT	HUMID	OVERALL
OR	True	False	True, hot OR humid as it's hot
OR	False	True	True, hot OR humid as it's humid
OR	False	False	False, not hot OR humid as neither are true

THE ORDER IN WHICH MYSQL PROCESSES CONDITIONS

The following example demonstrates a trap into which many have fallen. Let's assume a manager asks you for a list of all employees with the surname Rive and the first name Sol or a commission greater than 10 percent. You may well try the following query:

```
mysql> SELECT * FROM sales_rep WHERE surname='Rive'
 AND first_name='Sol' OR commission>10;
+-----------------+----------+------------+------------+
| employee_number | surname  | first_name | commission |
+-----------------+----------+------------+------------+
|               1 | Rive     | Sol        |         10 |
|               2 | Gordimer | Charlene   |         15 |
+-----------------+----------+------------+------------+
2 rows in set (0.00 sec)
```

This result may be exactly what you want. But what if the manager had meant something slightly different? The employee must have a surname of Rive and then can either have a first name of Sol or have a commission of greater than 10 percent. The second result would then not apply, as although her commission is greater than 10 percent, her first name is not Sol. The `AND` construct means that both clauses must be true. You would have to give the query as follows:

```
mysql> SELECT * FROM sales_rep WHERE surname='Rive'
 AND (first_name='Sol' OR commission>10);
+-----------------+---------+------------+------------+
| employee_number | surname | first_name | commission |
+-----------------+---------+------------+------------+
|               1 | Rive    | Sol        |         10 |
+-----------------+---------+------------+------------+
1 row in set (0.00 sec)
```

Note the parentheses in this query. When you have multiple conditions, it becomes critical to know the order in which the conditions must be processed. Does the `OR` part or the `AND` part come first? Often you will be given unclear verbal instructions, but this example shows you the importance of finding out getting clarity before you implement the query. Sometimes errors like these are never discovered! It's what's often meant by *computer error*, but it all comes down eventually to a person, usually someone who devised the wrong query.

Appendix B, "MySQL Function and Operator Reference," contains a list of operators and their precedence. I suggest that you use parentheses to determine precedence in your queries. Some books and people swear by learning the built-in precedence. For example, you may have learned at school

that 1 + 1 * 3 = 4, not 6, because you know that multiplication is performed before addition. The same applies to AND, which is performed before OR. But not everyone may know this, so use the parentheses to make it clear to everyone what you mean 1 + (1 * 3). Even after many years of programming, I still don't know the full list of operator precedence, and probably never will.

PATTERN MATCHING: *LIKE* AND %

Let's look at some more additions to the SELECT statement. What if you want to return the details for Mike Serote? Simple, you say—you'd use the following query:

```
mysql> SELECT * FROM sales_rep WHERE surname='Serote' and first_name='Mike';
+-----------------+---------+------------+------------+
| employee_number | surname | first_name | commission |
+-----------------+---------+------------+------------+
|               3 | Serote  | Mike       |         10 |
+-----------------+---------+------------+------------+
1 row in set (0.00 sec)
```

But what if you've forgotten how to spell *Serote*? Was it *Serotte* or maybe *Serota*? You may have to try a number of queries before you succeed, and if you don't happen to remember the exact spelling you will never succeed. You may want to try just using *Mike*, but remember that many databases consist of hundreds of thousands of records. Luckily, there is a better way. MySQL allows the LIKE statement. If you remember the surname begins with Sero, you can use the following:

```
mysql> SELECT * FROM sales_rep WHERE surname LIKE 'Sero%';
+-----------------+---------+------------+------------+
| employee_number | surname | first_name | commission |
+-----------------+---------+------------+------------+
|               3 | Serote  | Mike       |         10 |
+-----------------+---------+------------+------------+
1 row in set (0.00 sec)
```

Take note of the %. It's a wildcard, similar to *, but specifically for use inside a SELECT condition. It means *0 or more characters*. So all of the earlier permutations on the spelling you were considering would have been returned. You can use the wildcard any number of times, which allows queries such as this:

```
mysql> SELECT * FROM sales_rep WHERE surname LIKE '%e%';
+-----------------+----------+------------+------------+
| employee_number | surname  | first_name | commission |
+-----------------+----------+------------+------------+
|               1 | Rive     | Sol        |         10 |
|               2 | Gordimer | Charlene   |         15 |
|               3 | Serote   | Mike       |         10 |
+-----------------+----------+------------+------------+
3 rows in set (0.00 sec)
```

This returns all the records, as it looks for any surname with an *e* in it. This is different from the following query, which only looks for surnames that start with an *e*:

```
mysql> SELECT * FROM sales_rep WHERE surname LIKE 'e%';
Empty set (0.00 sec)
```

You could also use a query such as the following, which searches for surnames that have an *e* anywhere in the name and then end with an *e*:

```
mysql> SELECT * FROM sales_rep WHERE surname LIKE '%e%e';
+-----------------+---------+------------+------------+
| employee_number | surname | first_name | commission |
+-----------------+---------+------------+------------+
|               3 | Serote  | Mike       |         10 |
+-----------------+---------+------------+------------+
1 row in set (0.00 sec)
```

Let's add a few more records to the table so that you can try some more complex queries. Add the following two records:

```
mysql> INSERT INTO sales_rep values(4,'Rive','Mongane',10);
mysql> INSERT INTO sales_rep values(5,'Smith','Mike',12);
```

Sorting

Another useful and commonly used clause allows sorting of the results. An alphabetical list of employees would be useful, and you can use the `ORDER BY` clause to help you generate it:

```
mysql> SELECT * FROM sales_rep ORDER BY surname;
+-----------------+----------+------------+------------+
| employee_number | surname  | first_name | commission |
+-----------------+----------+------------+------------+
|               2 | Gordimer | Charlene   |         15 |
|               1 | Rive     | Sol        |         10 |
|               4 | Rive     | Mongane    |         10 |
|               3 | Serote   | Mike       |         10 |
|               5 | Smith    | Mike       |         12 |
+-----------------+----------+------------+------------+
5 rows in set (0.00 sec)
```

You may have noticed that this list is not quite correct if you want to sort by name because Sol Rive appears before Mongane Rive. To correct this, you need to sort on the first name as well when the surnames are the same. To achieve this, use the following:

```
mysql> SELECT * FROM sales_rep ORDER BY surname,first_name;
+-----------------+----------+------------+------------+
| employee_number | surname  | first_name | commission |
+-----------------+----------+------------+------------+
|               2 | Gordimer | Charlene   |         15 |
|               4 | Rive     | Mongane    |         10 |
|               1 | Rive     | Sol        |         10 |
|               3 | Serote   | Mike       |         10 |
|               5 | Smith    | Mike       |         12 |
+-----------------+----------+------------+------------+
5 rows in set (0.00 sec)
```

Now the order is correct. To sort in reverse order (descending order), you use the `DESC` keyword. The following query returns all records in order of commission earned, from high to low:

```
mysql> SELECT * FROM sales_rep ORDER BY commission DESC;
+-----------------+----------+------------+------------+
| employee_number | surname  | first_name | commission |
+-----------------+----------+------------+------------+
|               2 | Gordimer | Charlene   |         15 |
|               5 | Smith    | Mike       |         12 |
|               1 | Rive     | Sol        |         10 |
|               4 | Rive     | Mongane    |         10 |
|               3 | Serote   | Mike       |         10 |
+-----------------+----------+------------+------------+
5 rows in set (0.00 sec)
```

Again, you may want to further sort the three employees earning 10-percent commission. To do so, you can use the `ASC` keyword. Although not strictly necessary because it is the default sort order, the keyword helps add clarity:

```
mysql> SELECT * FROM sales_rep ORDER BY commission DESC,
 surname ASC,first_name ASC;
+-----------------+----------+------------+------------+
| employee_number | surname  | first_name | commission |
+-----------------+----------+------------+------------+
|               2 | Gordimer | Charlene   |         15 |
|               5 | Smith    | Mike       |         12 |
|               4 | Rive     | Mongane    |         10 |
|               1 | Rive     | Sol        |         10 |
|               3 | Serote   | Mike       |         10 |
+-----------------+----------+------------+------------+
5 rows in set (0.01 sec)
```

LIMITING THE NUMBER OF RESULTS

So far you have always gotten the full number of results that satisfy the condition returned to you. In a real-world database, however, there may be many thousands of records, and you do not want to see them all at once. MySQL, therefore, allows you to use the `LIMIT` clause. `LIMIT` is non-standard SQL, which means you won't be able to use it in the same way with all other databases, but it is a powerful and useful MySQL enhancement. If you only wanted to find the employee with the highest commission (assuming there is only one, as with this dataset), you could run this query:

```
mysql> SELECT first_name,surname,commission FROM sales_rep
 ORDER BY commission DESC;
```

```
+------------+----------+------------+
| first_name | surname  | commission |
+------------+----------+------------+
| Charlene   | Gordimer |         15 |
| Mike       | Smith    |         12 |
| Sol        | Rive     |         10 |
| Mike       | Serote   |         10 |
| Mongane    | Rive     |         10 |
+------------+----------+------------+
5 rows in set (0.00 sec)
```

Clearly the employee you're after is Charlene Gordimer. But `LIMIT` allows you to return only that record, as follows:

```
mysql> SELECT first_name,surname,commission FROM sales_rep
 ORDER BY commission DESC LIMIT 1;
+------------+----------+------------+
| first_name | surname  | commission |
+------------+----------+------------+
| Charlene   | Gordimer |         15 |
+------------+----------+------------+
1 row in set (0.00 sec)
```

If there's only one number after the `LIMIT` clause, it determines the number of rows returned.

NOTE `LIMIT 0` *returns no records. This may seem useless, but it's a useful way to test your query on large databases without actually running it.*

The `LIMIT` clause does not only allow you to return a limited number of records starting from the beginning or end of the dataset. You can also tell MySQL what *offset* to use—in other words, from which result to start limiting. If there are two numbers after the `LIMIT` clause, the first is the offset and the second is the row limit. The next example returns the second record, in descending order:

```
mysql> SELECT first_name,surname,commission FROM
 sales_rep ORDER BY commission DESC LIMIT 1,1;
+------------+---------+------------+
| first_name | surname | commission |
+------------+---------+------------+
| Mike       | Smith   |         12 |
+------------+---------+------------+
1 row in set (0.00 sec)
```

The default offset is 0 (computers always start counting at 0), so by specifying an offset of 1, you're taking the second record. You can check this by running this query:

```
mysql> SELECT first_name,surname,commission FROM sales_rep
 ORDER BY commission DESC LIMIT 0,1;
```

```
+------------+----------+------------+
| first_name | surname  | commission |
+------------+----------+------------+
| Charlene   | Gordimer |         15 |
+------------+----------+------------+
1 row in set (0.00 sec)
```

`LIMIT 1` is the same as `LIMIT 0,1` as the default 0 is assumed if it is not specified.

Now let's try another example. How would you return the third, fourth, and fifth records sorted in descending order on the commission field?

```
mysql> SELECT first_name,surname,commission FROM sales_rep
 ORDER BY commission DESC LIMIT 2,3;
+------------+---------+------------+
| first_name | surname | commission |
+------------+---------+------------+
| Sol        | Rive    |         10 |
| Mike       | Serote  |         10 |
| Mongane    | Rive    |         10 |
+------------+---------+------------+
3 rows in set (0.00 sec)
```

The 2 is the offset (remember the offset starts at 0, so 2 is actually the third record), and the 3 is the number of records to return.

NOTE `LIMIT` *is often used in search engines running MySQL, for example, to display 10 results per page. The first page results would then use* `LIMIT 0,10`*, the second would use* `LIMIT 10,10`*, and so on.*

RETURNING THE MAXIMUM VALUE WITH *MAX()*

MySQL has a vast number of functions that allow you to fine-tune your queries. If you happen to have glanced at Appendix B already, don't be put off by thinking you have to learn pages of terms. No one remembers them all, but once you know they exist, you can look up to see if there is one to do what you want it to do. And once you use them a lot, you'll find you've memorized the more common functions. This is much more effortless than memorizing them upfront! The first function you're going to look at is the `MAX()` function. To return the highest commission, you would use this query:

```
mysql> SELECT MAX(commission) from sales_rep;
+-----------------+
| MAX(commission) |
+-----------------+
|              15 |
+-----------------+
1 row in set (0.00 sec)
```

Notice the parentheses when you use functions. The function is applied to whatever is inside the parentheses. Throughout this book, I write functions with the parentheses whenever I refer to them to remind you that it is a function and how to use it—for example, `MAX()`.

WARNING *Be careful about spaces in your queries. In most cases they make no difference, but when you're dealing with functions (functions can usually be identified by the fact that they need to enclose something in parentheses), you need to be especially careful. If you'd put a space after the word COUNT, you'd have gotten a MySQL syntax error.*

Returning Distinct Records

Sometimes, you don't want to return duplicate results. Take a look at the following query:

```
mysql> SELECT surname FROM sales_rep ORDER BY surname;
+----------+
| surname  |
+----------+
| Gordimer |
| Rive     |
| Rive     |
| Serote   |
| Smith    |
+----------+
5 rows in set (0.00 sec)
```

This query is all well and good, but what if you just wanted to return a list of surnames, with each surname appearing only once? You don't want Rive to appear for both Mongane and Sol, but just once. The solution is to use `DISTINCT`, as follows:

```
mysql> SELECT DISTINCT surname FROM sales_rep ORDER BY surname;
+----------+
| surname  |
+----------+
| Gordimer |
| Rive     |
| Serote   |
| Smith    |
+----------+
4 rows in set (0.00 sec)
```

Counting

As you can see beneath the results so far, MySQL displays the number of rows returned, such as `4 rows in set`. Sometimes, all you really want to return is the number of results, not the contents of the records themselves. In this case, you'd use the `COUNT()` function:

```
mysql> SELECT COUNT(surname) FROM sales_rep;
+----------------+
| COUNT(surname) |
+----------------+
|              5 |
+----------------+
1 row in set (0.01 sec)
```

It doesn't really make much difference which field you counted in the previous example, as there are as many surnames as there are first names in the table. You would have gotten the same results if you used the following query:

```
mysql> SELECT COUNT(*) FROM sales_rep;
+----------+
| COUNT(*) |
+----------+
|        5 |
+----------+
1 row in set (0.00 sec)
```

To count the number of distinct surnames, you combine `COUNT()` with `DISTINCT`, as follows:

```
mysql> SELECT COUNT(DISTINCT surname) FROM sales_rep;
+-------------------------+
| COUNT(DISTINCT surname) |
+-------------------------+
|                       4 |
+-------------------------+
1 row in set (0.00 sec)
```

RETURNING THE AVERAGE, MINIMUM, AND TOTAL VALUES WITH *AVG()*, *MIN()*, AND *SUM()*

These functions work the same way as `MAX()`. You put whatever you want to work on inside the parentheses. So, to return the average commission, use the following:

```
mysql> SELECT AVG(commission) FROM sales_rep;
+-----------------+
| AVG(commission) |
+-----------------+
|         11.4000 |
+-----------------+
1 row in set (0.00 sec)
```

And to find the lowest commission that any of the sales staff is earning, use this:

```
mysql> SELECT MIN(commission) FROM sales_rep;
+-----------------+
| MIN(commission) |
+-----------------+
|              10 |
+-----------------+
1 row in set (0.00 sec)
```

`SUM()` works in the same way. It's unlikely you would find much use for totaling the commissions as shown here, but you can get an idea of how it works:

```
mysql> SELECT SUM(commission) from sales_rep;
```

```
+-----------------+
| SUM(commission) |
+-----------------+
|              57 |
+-----------------+
1 row in set (0.00 sec)
```

Performing Calculations in a Query

SQL allows you to perform calculations in your query, as well. As a simple example, try the following:

```
mysql> SELECT 1+1;
+-----+
| 1+1 |
+-----+
|   2 |
+-----+
1 row in set (0.00 sec)
```

Obviously this is not the reason you use MySQL. There's no danger of schools embracing MySQL for students to use in mathematics examinations. But it's a useful feature inside a query. For example, if you want to see what commissions the sales reps would be earning if you increased everyone's commission by 1 percent, use this:

```
mysql> SELECT first_name,surname,commission + 1 FROM sales_rep;
+------------+----------+----------------+
| first_name | surname  | commission + 1 |
+------------+----------+----------------+
| Sol        | Rive     |             11 |
| Charlene   | Gordimer |             16 |
| Mike       | Serote   |             11 |
| Mongane    | Rive     |             11 |
| Mike       | Smith    |             12 |
+------------+----------+----------------+
5 rows in set (0.00 sec)
```

Deleting Records

To delete a record, MySQL uses the `DELETE` statement. It's similar to `SELECT`, except that as the entire record is deleted, there is no need to specify any columns. You just need the table name and the condition. For example, Mike Smith resigns, so to remove him, you use this:

```
mysql> DELETE FROM sales_rep WHERE employee_number = 5;
Query OK, 1 row affected (0.00 sec)
```

You could also have used the first name and surname as a condition to delete, and in this case it would have worked as well. But for real-world databases, you'll want to use a unique field to identify the right person. Chapter 3 will introduce the topic of indexes, but for now keep in mind that *employee_number*

is a unique field, and it is best to use it (Mike Smith is not that uncommon a name!). You will formalize the fact that *employee_number* is unique in your database structure in the section on indexes.

WARNING *Be careful to use a condition with your* `DELETE` *statement. Simply entering* `DELETE FROM sales_rep;` *would have deleted all records in the table. There is no undo option, and as you haven't yet learned to back up your data, you'd be in trouble!*

Changing Records in a Table

You've learned to add records using `INSERT`, remove them using `DELETE`, and return results using `SELECT`. All that's missing is learning how to change data in existing records. Let's assume that Sol Rive has just sold a huge shipment of sand to the desert dwellers in Namibia, and he is rewarded with an increase to a 12-percent commission. To correctly reflect this fact, you use the `UPDATE` statement, as follows:

```
mysql> UPDATE sales_rep SET commission = 12 WHERE
 employee_number=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

WARNING *Again, be careful to apply a condition. Without the* `WHERE` *clause, you would have changed everybody's commission to 12 percent!*

`INSERT`, `SELECT`, `UPDATE`, and `DELETE` make up the four standard statements for manipulating data. They are part of SQL's Data Manipulation Language (DML). With them, you have all the ammunition you need to make changes to the data in your records. You'll come across more advanced queries in Chapter 2.

Dropping Tables and Databases

There are also statements to define the data structure, and these are said to be part of SQL's Data Definition Language (DDL). You've already seen one—the `CREATE` statement—which you used to create your database and then the tables and structures within that database. As with data manipulation, you may want to remove or change these tables. Let's create a table and then remove (or drop) it again:

```
mysql> CREATE TABLE commission (id INT);
Query OK, 0 rows affected (0.01 sec)
mysql> DROP TABLE commission;
Query OK, 0 rows affected (0.00 sec)d
```

WARNING *No warnings, no notification—the table, and all data in it, has been dropped! Be careful with this statement.*

You can do the same with a database:

```
mysql> CREATE DATABASE shortlived;
Query OK, 1 row affected (0.01 sec)
mysql> DROP DATABASE shortlived;
Query OK, 0 rows affected (0.00 sec)
```

Now you get some idea why permissions become so important! Allowing anyone who can connect to MySQL such power would be disastrous. You'll learn how to prevent catastrophes like this with permissions in Chapter 14.

Changing Table Structure

The final DDL statement, `ALTER`, allows you to change the structure of tables. You can add columns, change column definitions, rename tables, and drop columns.

Adding a Column

Let's say that you realize you need to create a column in your *sales_reps* table to store the date the person joined. `UPDATE` won't do, as this only changes the data, not the structure. To make this change, you use the `ALTER` statement:

```
mysql> ALTER TABLE sales_rep ADD date_joined DATE;
Query OK, 4 rows affected (0.01 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

TIP `DATE` *is a column type that stores data in the format year-month-day (YYYY-MM-DD). If you're used to entering dates in other ways, such as the U.S. format (MM/DD/YYYY), you'll need to make some adjustments.*

Your manager gives you another requirement for your database. (Although most changes are easy to perform, it's always better to get the design right from the beginning, as some changes will have unhappy consequences. Chapter 9, "Database Design," introduces the topic of database design.) You need to store the year that the sales rep was born, in order to perform analysis on the age distribution of the staff. MySQL has a `YEAR` column type, which you can use. Add the following column:

```
mysql> ALTER TABLE sales_rep ADD year_born YEAR;
Query OK, 4 rows affected (0.02 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

Changing a Column Definition

But seconds after adding the year, your manager comes up with a better idea. Why not store the sales rep's date of birth instead? That way the year is available as before, but the company can also surprise the reps with a birthday present. To change the column definition, use the following:

```
mysql> ALTER TABLE sales_rep CHANGE year_born birthday
 DATE;
Query OK, 4 rows affected (0.03 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

After the `CHANGE` clause comes the old column name, then the new column name, followed by its definition. To change the definition, but not the name of the column, you would simply keep the name the same as before, as shown here:

```
mysql> ALTER TABLE tablename CHANGE oldname oldname new_column_definition;
```

Alternatively, use the `MODIFY` clause, which doesn't require the repetition of the name, as follows:

```
mysql> ALTER TABLE tablename MODIFY oldname new_column_definition;
```

Renaming a Table

One morning your manager barges in demanding that the term *sales rep* no longer be used. From henceforth, the employees are *cash-flow enhancers*, and records need to be kept of their *enhancement value*. Noting the wild look in your manager's eye, you decide to comply, first by adding the new field:

```
mysql> ALTER TABLE sales_rep ADD enhancement_value int;
Query OK, 4 rows affected (0.05 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

and then renaming the table. To do so, use `RENAME` in the `ALTER` statement as follows:

```
mysql> ALTER TABLE sales_rep RENAME cash_flow_specialist;
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

The next day your manager is looking a little sheepish and red-eyed, and is mumbling something about doctored punch. You decide to change the table name back and drop the new column, before anyone notices.

```
mysql> ALTER TABLE cash_flow_specialist RENAME TO
 sales_rep;
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

NOTE *Note the difference between the two* `ALTER RENAME` *statements:* `TO` *has been included after the second* `RENAME`*. Both statements are identical in function, however. There are quite a few cases like this where MySQL has more than one way of doing something. In fact, there's even a third way to rename a table; you could use* `RENAME old tablename TO new_tablename`*. These sorts of things are often to provide compliance with other databases or to the ANSI SQL standard.*

Dropping a Column

To remove the unwanted *enhancement_value* (what made us think `INT` was right for this mysterious field anyway?), use `ALTER … DROP`, as follows:

```
mysql> ALTER TABLE sales_rep DROP enhancement_value;
Query OK, 4 rows affected (0.06 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

Using Date Functions

Now that you've added a couple of date columns, let's look at some of MySQL's date functions. The table structure currently looks as follows:

```
mysql> DESCRIBE sales_rep;
```

```
+-----------------+-------------+------+-----+---------+-------+
| Field           | Type        | Null | Key | Default | Extra |
+-----------------+-------------+------+-----+---------+-------+
| employee_number | int(11)     | YES  |     | NULL    |       |
| surname         | varchar(40) | YES  |     | NULL    |       |
| first_name      | varchar(30) | YES  |     | NULL    |       |
| commission      | tinyint(4)  | YES  |     | NULL    |       |
| date_joined     | date        | YES  |     | NULL    |       |
| birthday        | date        | YES  |     | NULL    |       |
+-----------------+-------------+------+-----+---------+-------+
6 rows in set (0.00 sec)
```

If you do a query, returning *date_joined* and *birthday*, you get the following result:

```
mysql> SELECT date_joined,birthday FROM sales_rep;
+-------------+----------+
| date_joined | birthday |
+-------------+----------+
| NULL        | NULL     |
| NULL        | NULL     |
| NULL        | NULL     |
| NULL        | NULL     |
+-------------+----------+
4 rows in set (0.00 sec)
```

The `NULL` values indicate that you never entered anything into these fields. You'll have noticed the `Null` heading returned when you `DESCRIBE` a table. `YES` is the default for this; it means the field is allowed to have nothing in it. Sometimes you want to specify that a field can never contain a `NULL` value. You'll learn how to do this in Chapter 3. `NULL` values often affect the results of queries and have their own complexities, which are also examined in chapters 2 and 3. To make sure you have no `NULL` values, update the sales rep records as follows:

```
mysql> UPDATE sales_rep SET date_joined =
 '2000-02-15', birthday='1976-03-18'
 WHERE employee_number=1;
mysql> UPDATE sales_rep SET date_joined =
 '1998-07-09', birthday='1958-11-30'
 WHERE employee_number=2;
mysql> UPDATE sales_rep SET date_joined =
 '2001-05-14', birthday='1971-06-18'
 WHERE employee_number=3;
mysql> UPDATE sales_rep SET date_joined =
 '2002-11-23', birthday='1982-01-04'
 WHERE employee_number=4;
```

There are a host of useful date functions. These examples show just a few; see Appendix A, "MySQL Syntax Reference," and Appendix B for more information.

Specifying the Date Format

All is not lost if you want to return dates in a format of your choice, rather than the standard YYYY-MM-DD format. To return the birthdays of all staff in the format MM/DD/YYYY, use the `DATE_FORMAT()` function, as follows:

```
mysql> SELECT DATE_FORMAT(date_joined,'%m/%d/%Y')
 FROM sales_rep WHERE employee_number=1;
+-------------------------------------+
| date_format(date_joined,'%m/%d/%Y') |
+-------------------------------------+
| 02/15/2000                          |
+-------------------------------------+
```

The part in single quotes after the *date_joined* column is called the *format string*. Inside the format string, you use a *specifier* to specify exactly what format to return. `%m` returns the month (01–12), `%d` returns the day (01–31), and `%Y` returns the year in four digits. There are a huge range of specifiers; see Appendix B for a full listing. In the meantime, some examples follow:

```
mysql> SELECT DATE_FORMAT(date_joined,'%W %M %e %y')
 FROM sales_rep WHERE employee_number=1;
+---------------------------------------+
| DATE_FORMAT(date_joined,'%W %M %e %y') |
+---------------------------------------+
| Tuesday February 15 00                |
+---------------------------------------+
```

`%W` returns the weekday name, `%M` returns the month name, `%e` returns the day (1–31), and `%y` returns the year in a two-digit format. Note that `%d` also returns the day (01-31), but is different to `%e` because the leading zeros are included.

In the following query, `%a` is the abbreviated weekday name, `%D` is the day of month with the suffix attached, `%b` is the abbreviated month name, and `%Y` is the four-digit year:

```
mysql> SELECT DATE_FORMAT(date_joined,'%a %D %b, %Y')
 FROM sales_rep WHERE employee_number=1;
+---------------------------------------+
| DATE_FORMAT(date_joined,'%a %D %b, %Y') |
+---------------------------------------+
| Tue 15th Feb, 2000                    |
+---------------------------------------+
```

NOTE *You can add any of your own characters in the format string. The previous examples have used a slash (/) and a comma (,). You can add any text you want to format the date as you want it.*

Returning the Current Date and Time

To find out what date it is, according to the server, you can use the `CURRENT_DATE()` function. There is another function, `NOW()`, which returns the time, as well:

```
mysql> SELECT NOW(),CURRENT_DATE();
```

```
+---------------------+----------------+
| NOW()               | CURRENT_DATE() |
+---------------------+----------------+
| 2002-04-07 18:32:31 | 2002-04-07     |
+---------------------+----------------+
1 row in set (0.00 sec)
```

NOTE `NOW()` *returns the date and time. There is a column type called* `DATETIME` *that allows you to store data in the same format (YYYY-MM-DD HH:MM:SS) in your tables.*

You can do other conversions on the *birthday* field when you return it. Just in case you were worried about not being able to return the year, because you've replaced the year field with a date of birth, you can use the `YEAR()` function, as follows:

```
mysql> SELECT YEAR(birthday) FROM  sales_rep;
+----------------+
| YEAR(birthday) |
+----------------+
|           1976 |
|           1958 |
|           1982 |
|           1971 |
+----------------+
4 rows in set (0.00 sec)
```

MySQL has other functions for returning just a part of a date: `MONTH()` and `DAYOFMONTH()`:

```
mysql> SELECT MONTH(birthday),DAYOFMONTH(birthday) FROM sales_rep;
+-----------------+----------------------+
| MONTH(birthday) | DAYOFMONTH(birthday) |
+-----------------+----------------------+
|               3 |                   18 |
|              11 |                   30 |
|               1 |                    4 |
|               6 |                   18 |
+-----------------+----------------------+
4 rows in set (0.00 sec)
```

Creating More Advanced Queries

By now you should be comfortable with basic queries. The good news is that in the real world, most queries are fairly simple, like what you've done so far. And the better designed your databases (see Part II, "Designing a Database"), the more simple your queries. But there are situations where you'll need more—the most common being joining two or more tables together (this kind of query is called a *join*).

GIVING COLUMNS A NEW HEADING WITH *AS*

The previous queries aren't too easy to read or understand. Let's tidy up the previous query, by sorting on the month and returning the sales rep names. You also introduce aliases with the `AS` keyword, where you give a column another name:

```
mysql> SELECT surname,first_name,MONTH(birthday)
 AS month,DAYOFMONTH(birthday) AS day FROM sales_rep
 ORDER BY month;
+----------+------------+-------+------+
| surname  | first_name | month | day  |
+----------+------------+-------+------+
| Rive     | Mongane    |     1 |    4 |
| Rive     | Sol        |     3 |   18 |
| Serote   | Mike       |     6 |   18 |
| Gordimer | Charlene   |    11 |   30 |
+----------+------------+-------+------+
4 rows in set (0.01 sec)
```

JOINING COLUMNS WITH *CONCAT*

Sometimes you may prefer to display the person's name as one result field, rather than separate the first name and surname fields. You can join the results of columns together, using the `CONCAT()` function (which stands for *concatenate*), as follows:

```
mysql> SELECT CONCAT(first_name, ' ',surname)
 AS name,MONTH(birthday) AS month,DAYOFMONTH(birthday)
 AS day FROM sales_rep ORDER BY month;
+-------------------+-------+------+
| name              | month | day  |
+-------------------+-------+------+
| Mongane Rive      |     1 |    4 |
| Sol Rive          |     3 |   18 |
| Mike Serote       |     6 |   18 |
| Charlene Gordimer |    11 |   30 |
+-------------------+-------+------+
4 rows in set (0.00 sec)
```

NOTE *Note the space used inside* `CONCAT()`*. Just as with date specifiers, you can use any characters to format your* `CONCAT()`*.*

FINDING THE DAY IN A YEAR

To find out what day in the year (from 1–366) Sol Rive joined the company, use the following:

```
mysql> SELECT DAYOFYEAR(date_joined) FROM sales_rep
 WHERE employee_number=1;
+------------------------+
| DAYOFYEAR(date_joined) |
+------------------------+
|                     46 |
+------------------------+
```

Working with Multiple Tables

The real power of relational databases comes from the fact that there are relations between tables. So far you've only been working on one table to get familiar with SQL syntax. But most real-world applications have more than one table, so you need to know how to work with these situations. First, let's add two new tables to the database. Table 1.5 will contain customer data (a customer ID, first name, and surname), and Table 1.6 will contain sales data (a customer ID, a sales rep ID, a sales value in dollars, and a unique code for the sale).

Table 1.5: The Customer Table

ID	First_name	Surname
1	Yvonne	Clegg
2	Johnny	Chaka-Chaka
3	Winston	Powers
4	Patricia	Mankunku

Table 1.6: The Sales Table

Code	Sales_rep	Customer	Value
1	1	1	2000
2	4	3	250
3	2	3	500
4	1	4	450
5	3	1	3800
6	1	2	500

Can you create these tables? Here are the statements I used:

```
mysql> CREATE TABLE customer(
  id int,
  first_name varchar(30),
  surname varchar(40)
);
Query OK, 0 rows affected (0.00 sec)
mysql> CREATE TABLE sales(
  code int,
  sales_rep int,
  customer int,
  value int
);
```

```
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO customer(id,first_name,surname) VALUES
(1,'Yvonne','Clegg'),
(2,'Johnny','Chaka-Chaka'),
(3,'Winston','Powers'),
(4,'Patricia','Mankunku');
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0
mysql> INSERT INTO sales(code,sales_rep,customer,value) VALUES
(1,1,1,2000),
(2,4,3,250),
(3,2,3,500),
(4,1,4,450),
(5,3,1,3800),
(6,1,2,500);
Query OK, 6 rows affected (0.00 sec)
Records: 6  Duplicates: 0  Warnings: 0
```

JOINING TWO OR MORE TABLES

As you can see, this uses the sales rep's employee number and the customer's ID in the sales table. If you examine the first sales record, you can see that it was made of sales_rep 1, which, looking up on the *sales_rep* table, is Sol Rive. The process you go through manually, of looking at the relation between the two tables, is the same one MySQL does, as long as you tell it what relation to use. Let's write a query that returns all the information from the first sales record, as well as the sales rep's name:

```
mysql> SELECT sales_rep,customer,value,first_name,surname
 FROM sales,sales_rep WHERE code=1 AND
 sales_rep.employee_number=sales.sales_rep;
+-----------+----------+-------+------------+---------+
| sales_rep | customer | value | first_name | surname |
+-----------+----------+-------+------------+---------+
|         1 |        1 |  2000 | Sol        | Rive    |
+-----------+----------+-------+------------+---------+
1 row in set (0.00 sec)
```

The first part of the query, after the `SELECT`, lists the fields you want to return. Easy enough—you just list the fields you want from both tables.

The second part, after the `FROM`, tells MySQL which tables to use. In this case it's two tables: the *sales* and *sales_rep* tables.

The third part, after the `WHERE`, contains the condition: `code=1`, which returns the first record from the sales table. The next part is what makes this query a join. This is where you tell MySQL which fields to join on or which fields the relation between the tables exists on. The relation between the *sales* table and the *sales_rep* table is between the *employee_number* field in *sales_rep*, and the *sales_rep* field in

the *sales* table. So, because you find a 1 in the *sales_rep* field, you must look for employee_number 1 in the *sales_rep* table.

Let's try another one. This time you want to return all the sales that Sol Rive (employee_number 1) has made. Let's look at the thought process behind building this query:

- Which tables do you need? Clearly *sales_rep,* and *sales.* That's already part of the query: `FROM sales_rep,sales`.
- What fields do you want? You want all the sales information. So the field list becomes `SELECT code,customer,value`.
- And finally, what are your conditions? The first is that you only want Sol Rive's results, and the second is to specify the relation, which is between the *sales_rep* field in the *sales* table and the *employee_number* field in the *sales_rep* table. So, the conditions are as follows: `WHERE first_name='Sol' and surname='Rive' AND sales.sales_rep = sales_rep.employee_number`.

The final query is as follows:

```
mysql> SELECT code,customer,value FROM sales_rep,sales
 WHERE first_name='Sol' AND surname='Rive' AND
 sales.sales_rep = sales_rep.employee_number;
+------+----------+-------+
| code | customer | value |
+------+----------+-------+
|    1 |        1 |  2000 |
|    4 |        4 |   450 |
|    6 |        2 |   500 |
+------+----------+-------+
3 rows in set (0.00 sec)
```

Notice the notation in the relation's condition: `sales.sales_rep` or `sales_rep.employee_number`. Specifying the table name, then a dot, then the fieldname helps to make the queries clearer in the previous examples and is mandatory when different tables have fieldnames that are the same. You can also use this notation in the field list. For example, you could have written the previous query as follows:

```
mysql> SELECT code,customer,value FROM sales,
 sales_rep WHERE first_name='Sol' AND surname='Rive'
 AND sales_rep = employee_number;
+------+----------+-------+
| code | customer | value |
+------+----------+-------+
|    1 |        1 |  2000 |
|    4 |        4 |   450 |
|    6 |        2 |   500 |
+------+----------+-------+
3 rows in set (0.00 sec)
```

without table names before any fieldnames because there are no fields that have the same names in the different tables. Or you could have written it like this:

```
mysql> SELECT sales.code,sales.customer,sales.value
 FROM sales,sales_rep WHERE sales_rep.first_name='Sol'
 AND sales_rep.surname='Rive' AND sales.sales_rep =
 sales_rep.employee_number;
+------+----------+-------+
| code | customer | value |
+------+----------+-------+
|    1 |        1 |  2000 |
|    4 |        4 |   450 |
|    6 |        2 |   500 |
+------+----------+-------+
3 rows in set (0.00 sec)
```

Both queries give identical results.

To demonstrate what happens when you have fieldnames that are identical, let's change the *sales_rep* field in the sales table to *employee_number*. Don't worry; you'll change it back before anyone notices:

```
mysql> ALTER TABLE sales CHANGE sales_rep
 employee_number int;
Query OK, 6 rows affected (0.00 sec)
Records: 6  Duplicates: 0  Warnings: 0
```

Now let's try the join again, with the name corrected, but without using the dot notation to specify the table names:

```
mysql> SELECT code,customer,value FROM sales_rep,sales
 WHERE first_name='Sol' AND surname='Rive'AND employee_number = employee_number;
ERROR 1052: Column: 'employee_number' in where clause is ambiguous
```

Just from reading the query you can probably see it is not clear. So now you have to use the table names every time you reference one of the *employee_number* fields:

```
mysql> SELECT code,customer,value FROM sales_rep,sales
 WHERE sales_rep.employee_number=1 AND sales_rep.employee_number =
 sales.employee_number;
+------+----------+-------+
| code | customer | value |
+------+----------+-------+
|    1 |        1 |  2000 |
|    4 |        4 |   450 |
|    6 |        2 |   500 |
+------+----------+-------+
3 rows in set (0.00 sec)
```

TIP *You could have used* `sales.employee_number` *instead of* `sales_rep.employee_number` *in the* `WHERE` *clause, but it's best to use the smaller table because this gives MySQL less work to do to return the query. You will learn more about optimizing queries in Chapter 4.*

Let's change the fieldname back to what it was before going any further:

```
mysql> ALTER TABLE sales CHANGE employee_number sales_rep INT;
Query OK, 6 rows affected (0.00 sec)
Records: 6  Duplicates: 0  Warnings: 0
```

Performing Date Calculations

Performing date calculations is relatively easy. We're going to work out someone's age based on their birthday in the next section, but first you'll try a more simple calculation. To find the number of years between the current date and the birthday, you use two functions, `YEAR()` and `NOW()`:

```
mysql> SELECT YEAR(NOW()) - YEAR(birthday) FROM sales_rep;
+------------------------------+
| YEAR(NOW()) - YEAR(birthday) |
+------------------------------+
|                           26 |
|                           44 |
|                           31 |
|                           20 |
+------------------------------+
4 rows in set (0.00 sec)
```

NOTE *You can also use* `CURRENT_DATE()` *instead of* `NOW()`*, which will give you the same result.*

The previous query result does not return the age, only the difference in years. It does not take into account days and months. This section describes age calculation; it may be tricky for novice users. Don't be put off. Once you've practiced your basic queries, this type of query will be old hat!

You need to subtract the years as you've done previously but then subtract a further year if a full year has not passed. Someone born on the 10th of December in 2001 is not one year old in January 2002, but only after the 10th of December in 2002. A good way of doing this is to take the MM-DD components of the two date fields (the current date and the birthdate) and compare them. If the current one is larger, a full year has passed, and the year calculation can be left. If the current MM-DD part is less than the birth date one, less than a full year has passed, and you must subtract one from the calculation. This may sound tricky, and there are some quite complex ways of doing the calculation floating around, but MySQL makes it easier because it evaluates a true expression to 1 and a false expression to 0:

```
mysql> SELECT YEAR(NOW()) > YEAR(birthday) FROM
 sales_rep WHERE employee_number=1;
+------------------------------+
| YEAR(NOW()) > YEAR(birthday) |
+------------------------------+
|                            1 |
+------------------------------+
1 row in set (0.00 sec)
mysql> SELECT YEAR(NOW()) < YEAR(birthday) FROM
```

```
 sales_rep WHERE employee_number=1;
+------------------------------+
| YEAR(NOW()) < YEAR(birthday) |
+------------------------------+
|                            0 |
+------------------------------+
1 row in set (0.00 sec)
```

The current year is greater than the birthday year of employee 1. That is true and evaluates to 1. The current year is not less than the birthday year. That is false and evaluates to 0.

Now you need a quick way to return just the MM-DD component of the date. And a shortcut way of doing this is to use the `RIGHT()` string function:

```
mysql> SELECT RIGHT(CURRENT_DATE,5),RIGHT(birthday,5) FROM sales_rep;
+-----------------------+-------------------+
| RIGHT(CURRENT_DATE,5) | RIGHT(birthday,5) |
+-----------------------+-------------------+
| 04-06                 | 03-18             |
| 04-06                 | 11-30             |
| 04-06                 | 01-04             |
| 04-06                 | 06-18             |
+-----------------------+-------------------+
4 rows in set (0.00 sec)
```

The `5` inside the `RIGHT()` function refers to the number of characters from the right side of the string that the function returns. The full string is `2002-04-06`, and the five rightmost characters are `04-06` (the dash is included). So, now you have all the components to do the date calculation:

```
mysql> SELECT surname, first_name, (YEAR(CURRENT_DATE) -
 YEAR(birthday)) -(RIGHT(CURRENT_DATE,5)<RIGHT(birthday,5))
 AS age FROM sales_rep;
+----------+------------+------+
| surname  | first_name | age  |
+----------+------------+------+
| Rive     | Sol        |   26 |
| Gordimer | Charlene   |   43 |
| Rive     | Mongane    |   20 |
| Serote   | Mike       |   30 |
+----------+------------+------+
4 rows in set (0.00 sec)
```

Your results may not match these results exactly because time marches on, and you'll be running the query at a later date than I did.

WARNING *Be careful to match parentheses carefully when doing such a complex calculation. For every opening parenthesis, you need a closing parenthesis, and in the correct place!*

Can you spot a case when the previous age query will not work? When the current year is the same as the birth year, you'll end up with –1 as the answer. Once you've had a good look at the appendixes,

try and work out your own way of calculating age. There are many possibilities and just as many plaintive cries to MySQL to develop an `AGE()` function.

Grouping in a Query

Now that you have a sales table, let's put the `SUM()` function to better use than you did earlier, by returning the total value of sales:

```
mysql> SELECT SUM(value) FROM sales;
+------------+
| SUM(value) |
+------------+
|       7500 |
+------------+
1 row in set (0.00 sec)
```

Now you want to find out the total sales of each sales rep. To do this manually, you would need to group the sales table according to sales_rep. You would place all of the sales made by sales_rep 1 together, total the value, and then repeat with sales rep 2. SQL has the `GROUP BY` clause, which MySQL uses in the same way:

```
mysql> SELECT sales_rep,SUM(value) FROM sales GROUP BY
 sales_rep;
+-----------+------------+
| sales_rep | SUM(value) |
+-----------+------------+
|         1 |       2950 |
|         2 |        500 |
|         3 |       3800 |
|         4 |        250 |
+-----------+------------+
4 rows in set (0.00 sec)
```

If you had tried the same query without grouping, you'd have gotten an error:

```
mysql> SELECT sales_rep,SUM(value) FROM sales;
ERROR 1140: Mixing of GROUP columns
 (MIN(),MAX(),COUNT()...) with no GROUP columns
 is illegal if there is no GROUP BY clause
```

This query makes no sense, as it tries to mix a summary field, `SUM()`, with an ordinary field. What are you expecting? The sum of all values repeated next to each *sales_rep*?

You can sort the output of a grouped query as well. To return the total sales of each sales rep from highest to lowest, you just add an `ORDER BY` clause:

```
mysql> SELECT sales_rep,SUM(value) AS sum FROM sales
 GROUP BY sales_rep ORDER BY sum desc;
```

```
+-----------+------+
| sales_rep | sum  |
+-----------+------+
|         3 | 3800 |
|         1 | 2950 |
|         2 |  500 |
|         4 |  250 |
+-----------+------+
```

Now try a more complex query that uses a number of the concepts you have learned. Let's return the name of a sales rep with the fewest number of sales. First, you would have to return an employee number. You may get back a different number when you run the query, as there are three people who've made only one sale. It doesn't matter which one you return for now. You would do this as follows:

```
mysql> SELECT sales_rep,COUNT(*) as count from sales
 GROUP BY sales_rep ORDER BY count LIMIT 1;
+-----------+-------+
| sales_rep | count |
+-----------+-------+
|         4 |     1 |
+-----------+-------+
1 row in set (0.00 sec)
```

Can you take it even further? Can you perform a join as well to return the name of sales rep 4? If you can do this having started the book as a novice, you're well on the way to earning your username, "guru to be"! Here's the query to do so:

```
mysql> SELECT first_name,surname,sales_rep,COUNT(*) AS
 count from sales,sales_rep WHERE sales_rep=employee_number
 GROUP BY sales_rep,first_name,surname ORDER BY count
 LIMIT 1;
+------------+---------+-----------+-------+
| first_name | surname | sales_rep | count |
+------------+---------+-----------+-------+
| Mongane    | Rive    |         4 |     1 |
+------------+---------+-----------+-------+
1 row in set (0.00 sec)
```

Summary

MySQL is a relational database management system. Logically, data is structured into tables, which are related to each other by means of a common field. Tables consist of rows (or records), and records consist of columns (or fields). Fields can be of differing types: a numeric type, a string type, or a date type. (This chapter merely introduces SQL. You will build your skills in the language throughout this book.)

The MySQL server is what stores the data and runs queries on it. To connect to the MySQL server, you need the MySQL client. This can be on the same machine as the server or a remote machine.

The power of a database management system comes from the capability to structure data and retrieve it for a wide variety of highly specific requirements. The industry standard for manipulating and defining data is SQL. Its most important commands are the following:

- The `CREATE` statement creates databases and the tables within the database.
- The `INSERT` statement places records into a table.
- The `SELECT` statement returns the results of a query.
- The `DELETE` statement removes records from a table.
- The `UPDATE` statement modifies the data in a table.
- The `ALTER` statement changes the structure of a table, utilizing clauses such as `ADD` to add a new column, `CHANGE` to change the name or definition of an existing column, `RENAME` to rename a table, or `DROP` to remove a table.

Functions add to the power of MySQL. You can recognize a function by the parentheses immediately following it. There are many MySQL functions—mathematical ones such as `SUM()` to calculate the total of a result set, date and time functions such as `YEAR()` to extract the year portion of a date, and string functions such as `RIGHT()` to extract part of a string starting on the right side of that string.

Armed with this basic information, you are now ready to learn the crucial intricacies of structuring data, move on to more advanced SQL, and encounter the various types of tables MySQL uses for different kinds of solutions.