

# SQL Basics

If you think about it, you'll realize that you can perform two basic tasks with a database: you can put data in and pull data out. And most often, your primary tool to accomplish these two functions is a language known as SQL, or Structured Query Language.

As a standards-compliant relational database management system (RDBMS), MySQL understands SQL fairly well, and it even offers up some interesting extensions to the SQL standard. To use MySQL effectively, you'll need to be able to speak SQL fluently—it's your primary means of interacting with the database server, and it plays a very important role in helping you get to the data you need rapidly and efficiently.

Over the course of this chapter, we'll be explaining some of the basic SQL commands to create and enter information into a database, together with examples that should make things clearer. In case you've never used a database, or the thought of learning another language scares you, don't worry, because SQL is considerably simpler than most programming languages, and you should have no trouble picking it up.

---

## A Brief History of SQL

Before we get into the nitty-gritty of SQL command syntax, let's spend a few moments understanding how SQL came into existence.

SQL began life as SEQUEL<sup>1</sup>, the Structured English Query Language, a component of an IBM research project called System/R. System/R was a prototype of the first relational database system; it was created at IBM's San Jose laboratories in 1974, and SEQUEL was the first query language to support multiple tables and multiple users.

In the late 1970s, SQL made its first appearance in a commercial role as the query language used by the Oracle RDBMS. This was quickly followed by the Ingres RDBMS, which also used SQL, and by the 1980s, SQL had become the de facto standard for the rapidly growing RDBMS industry. In 1989, SQL became an ANSI standard commonly referred to as SQL89; this was later updated in 1992 to become SQL92 or SQL2, the standard in use on most of today's commercial RDBMSs (including MySQL).

---

<sup>1</sup> The name was later changed to SQL for legal reasons.

**Breaking the Rules**

Although most of today's commercial RDBMSs do support the SQL92 standard, many of them also take liberties with the specification, extending SQL with proprietary extensions and enhancements. (MySQL is an example of such.) Most often, these enhancements are designed to improve performance or add extra functionality to the system; however, they can cause substantial difficulties when migrating from one DBMS to another.

A complete list of MySQL's deviations from the SQL specification is available at <http://www.mysql.com/doc/en/Compatibility.html>.

**An Overview of SQL**

As a language, SQL was designed to be "human-friendly"; most of its commands resemble spoken English, making it easy to read, understand, and learn. Commands are formulated as statements, and every statement begins with an "action word." The following examples demonstrate this:

```
CREATE DATABASE toys;
USE toys;
SELECT id FROM toys WHERE targetAge > 3;
DELETE FROM catalog WHERE productionStatus = "Revoked";
```

As you can see, it's pretty easy to understand what each statement does. This simplicity is one of the reasons SQL is so popular, and also so easy to learn.

SQL statements can be divided into three broad categories, each concerned with a different aspect of database management:

- **Statements used to define the structure of a database** These statements define the relationships among different pieces of data, definitions for database, table and column types, and database indices. In the SQL specification, this component is referred to as Data Definition Language (DDL), and it is discussed in detail in Chapter 8 of this book.
- **Statements used to manipulate data** These statements control adding and removing records, querying and joining tables, and verifying data integrity. In the SQL specification, this component is referred to as Data Manipulation Language (DML), and it is discussed in detail in Chapter 9 of this book.
- **Statements used to control the permissions and access level to different pieces of data** These statements define the access levels and security privileges for databases, tables and fields, which may be specified on a per-user and/or per-host basis. In the SQL specification, this component is referred to as Data Control Language (DCL), and it is discussed in detail in Chapter 14.

Typically, every SQL statement ends in a semicolon, and white space, tabs, and carriage returns are ignored by the SQL processor. The following two statements

are equivalent, even though the first is on a single line and the second is split over multiple lines.

```
DELETE FROM catalog WHERE productionStatus = "Revoked";

DELETE FROM
    catalog
    WHERE productionStatus =

"Revoked";
```

## A (My)SQL Tutorial

With the language basics out of the way, let's run through a quick tutorial to get you up to speed on a few more SQL basics. In the following section, we'll design a set of relational tables, create a database to store them, re-create the table design in MySQL, insert records into the database, and query the system to obtain answers to several burning questions.

At this point, we'll encourage you to try out the examples that follow as you're reading along. This process will not only give you some insight into how MySQL works, but it will also teach you the fundamentals of SQL in preparation for the chapters ahead.

### Understanding an RDBMS

Let's start at the beginning. Every database is composed of one or more *tables*. These tables, which structure data into rows and columns, are what lend organization to the data.

Here's an example of what a typical table looks like:

member_id	fname	lname	tel	email
1	John	Doe	1234567	jdoe@somewhere.com
2	Jane	Doe	8373728	jane@site.com
3	Steve	Klingon	7449373	steve@alien-race.com
4	Santa	Claus	9999999	santa@the-north-pole.com

As you can see, a table divides data into rows, with a new entry (or *record*) on every row. If you flip back to my original database-as-filing-cabinet analogy in Chapter 1, you'll see that every file in the cabinet corresponds to one row in the table.

The data in each row is further broken down into cells (or *fields*), each of which contains a value for a particular attribute of the data. For example, if you consider the record for the user Steve Klingon, you'll see that the record is clearly divided into separate fields for member ID, first name, last name, phone number, and e-mail address.

The rows within a table are not arranged in any particular order; they can be sorted alphabetically, by ID, by member name, or by any other criteria you choose to specify. Therefore, it becomes necessary that you have some method of identifying a specific record in a table. In our example, each record is identified by a member ID, which is a number unique to each row or record; this unique field is referred to as the *primary key* for that table.

You should note at this point that MySQL is a *relational database management system*, or RDBMS. A relational database is typically composed of multiple tables that contain interrelated pieces of information. SQL allows you to combine the data from these tables in a variety of ways, thereby allowing you to create and analyze new relationships among your data.

What we have in our first example is a single table. While this is fine by itself, it's when you add more tables and relate the information among them that you truly start to see the power inherent in this system. Consider the following example, which adds two more tables; the second contains a list of movies available for rent, while the third links the movies with the members via their primary keys.

member_id	fname	lname	tel	email
1	John	Doe	1234567	jdoe@somewhere.com
2	Jane	Doe	8373728	jane@site.com
3	Steve	Klingon	7449373	steve@alien-race.com
4	Santa	Claus	9999999	santa@the-north-pole.com

video_id	title	director
1	Star Wars: The Phantom Menace	George Lucas
2	ET	Steven Spielberg
3	Charlie's Angels	McG
4	Any Given Sunday	Oliver Stone
5	Hollow Man	Paul Verhoeven
6	Woman On Top	Fina Torres

member_id	video_id
2	6
4	2
1	1
1	2
1	3

If you take a close look at the third table, you'll see that it links each member with the video(s) he or she has rented. Thus we see that Jane Doe (member #2) has rented *Woman On Top* (video #6), while John Doe (member #1) has decided to spend the weekend on the couch with *Star Wars* (video #1), *ET* (video #2), and *Charlie's Angels* (video #3).

In other words, the third table has set up a relationship between the first and second table; this is the fundamental concept behind a RDBMS. After one or more relationships are set up, it is possible for you to extract a subset of the data (a *data slice*) to answer specific questions.

## Creating a Database

If you've understood the concept so far, it's now time for you to get down to brass tacks. Start up your MySQL client. (Note in the following code listings that anything you type appears in boldface.)

```
[user@host]# mysql -u root -p
Password: *****
```

Assuming everything is set up properly and you entered the correct password, you should see a prompt that looks something like this:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 80 to server version: 4.0.9-gamma-standard
Type 'help;' or '\h' for help.
mysql>
```

This is the MySQL command prompt; you'll be using this to enter all your SQL statements. Note that all MySQL commands end with a semicolon or the `\g` signal and can be entered in either uppercase or lowercase type.

Since all tables are stored in a database, the first command you need to know is the `CREATE DATABASE` command, which looks like this:

```
CREATE DATABASE database-name
```

Go on and try it out by creating a database called `library`:

```
mysql> CREATE DATABASE library;
Query OK, 1 row affected (0.05 sec)
```

You can view all available databases with the `SHOW DATABASES` command:

```
mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| library  |
| mysql    |
| test     |
+-----+
3 rows in set (0.00 sec)
```

Once you have obtained a list of databases, you can select the database you wish to use with the `USE` command, which looks like this:

```
USE database-name
```

For the moment, we'll restrict our activities to the database you just created:

```
mysql> USE library;
Database changed
```

After you've selected a database, you can view available tables in it with the `SHOW TABLES` command.

```
mysql> SHOW TABLES;
Empty set (0.00 sec)
```

Because this is a new database, no tables appear yet. Let's fix that.

You can read more about manipulating databases in Chapter 8.

## Adding Tables

The SQL command used to create a new table in a database typically looks like this:

```
CREATE TABLE table-name (field-name-1 field-type-1 modifiers,  
field-name-2 field-type-2 modifiers, ... , field-name-n  
field-type-n modifiers)
```

The table name cannot contain spaces, slashes, or periods; other than this, any character is fair game. Each table (and the data it contains) is stored as a set of three files in your MySQL data directory.

Here's a sample command to create the `members` table in the example you saw a couple sections back:

```
mysql> CREATE TABLE members (member_id int(11) NOT NULL auto_increment,  
fname varchar(50) NOT NULL, lname varchar(50) NOT NULL, tel varchar(15),  
email varchar(50) NOT NULL, PRIMARY KEY (member_id));  
Query OK, 0 rows affected (0.05 sec)
```

Note that each field name is followed by a “type,” which identifies the type of data that will be allowed in that field, and (sometimes) a length value indicating the maximum length of that field. For example, in the first line, the field named `member_id` is followed by an `int` type of maximum length 11. MySQL offers a number of different data types to handle different data requirements. Some of the more important ones are summarized in the sidebar “Not My Type.”

### Not My Type

Following are some of the important data types you’ll find when using MySQL:

- **INT** A numeric type that can accept values in the range of `-2147483648` to `2147483647`
- **DECIMAL** A numeric type with support for floating-point or decimal numbers
- **DOUBLE** A numeric type for double-precision floating-point numbers
- **DATE** A date field in the `YYYY-MM-DD` format
- **TIME** A time field in the `HH:MM:SS` format
- **DATETIME** A combined date/time type in the `YYYY-MM-DD HH:MM:SS` format
- **YEAR** A field specifically for year displays in the range 1901 to 2155, in either `YYYY` or `YY` format
- **TIMESTAMP** A timestamp type, in `YYYYMMDDHHMMSS` format
- **CHAR** A string type with a maximum size of 255 characters and a fixed length
- **VARCHAR** A string type with a maximum size of 255 characters and a variable length
- **TEXT** A string type with a maximum size of 65,535 characters
- **BLOB** A binary type for variable data
- **ENUM** A string type that can accept one value from a list of previously defined possible values
- **SET** A string type that can accept zero or more values from a set of previously defined possible values

You can put a few additional constraints (*modifiers*) on your table, to increase the consistency of the data that will be entered into it:

- You can specify whether the field is allowed to be empty or must necessarily be filled with data by placing the `NULL` and `NOT NULL` modifiers after each field definition.
- You can specify a primary key for the table with the `PRIMARY KEY` modifier, which is followed by the name of the column designated as the primary key.
- You can specify that values entered into a field must be “unique”—that is, not duplicated—with the `UNIQUE` modifier.
- The `AUTO_INCREMENT` modifier, which is available only for numeric fields, indicates that MySQL should automatically generate a number for that field (by incrementing the previous value by 1).

Now go ahead and create the other two tables using the following SQL statements:

```
mysql> CREATE TABLE videos (video_id int(11) NOT NULL auto_increment,
title varchar(255) NOT NULL, director varchar(255) NOT NULL,
PRIMARY KEY (video_id));
Query OK, 0 rows affected (0.05 sec)
mysql> CREATE TABLE status (member_id int(11) NOT NULL,
video_id tinyint(11) NOT NULL);
Query OK, 0 rows affected (0.05 sec)
```

In case you make a mistake, note that you can alter a table definition with the `ALTER TABLE` command, which looks like this:

```
ALTER TABLE table-name ADD new-field-name new-field-type
```

On the other hand, if you simply want to modify an existing column, use this:

```
ALTER TABLE table-name MODIFY old-field-name
new-field-type modifiers
```

Just as you can create a table, you can delete a table with the `DROP TABLE` command, which looks like this:

```
DROP TABLE table-name
```

Here’s an example:

```
mysql> DROP TABLE members;
Query OK, 0 rows affected (0.00 sec)
```

This will immediately wipe out the specified table, together with all the data it contains—so use it with care!

You can read more about manipulating tables in Chapter 8.

## Adding Records

Once you've created a table, it's time to begin entering data into it, and the SQL command to accomplish this is the `INSERT` command. The syntax of the `INSERT` command is as follows:

```
INSERT into table-name (field-name-1, field-name2, field-name-n)
VALUES (value-1, value-2, value-n)
```

Here's an example:

```
mysql> INSERT INTO members (member_id, fname, lname, tel, email)
VALUES (NULL, 'John', 'Doe', '1234567', 'jdoe@somewhere.com');
Query OK, 1 row affected (0.06 sec)
```

You could also use the abbreviated form of the `INSERT` statement, in which field names are left unspecified:

```
mysql> INSERT INTO members VALUES (NULL, 'John', 'Doe', '1234567',
'jdoe@somewhere.com');
Query OK, 1 row affected (0.06 sec)
```

Here's the flip side: by specifying field names in the `INSERT` statement, I have the flexibility of inserting values in any order I please. Because of this, the following statements are equivalent:

```
mysql> INSERT INTO members (member_id, fname, lname, tel, email)
VALUES (NULL, 'John', 'Doe', '1234567', 'jdoe@somewhere.com');
Query OK, 1 row affected (0.06 sec)
mysql> INSERT INTO members (fname, lname, email, tel, member_id)
VALUES ('John', 'Doe', 'jdoe@somewhere.com', '1234567', NULL);
Query OK, 1 row affected (0.00 sec)
```

Fields that are not specified will automatically be set to their default values.

Now that you know how to insert records, try inserting some sample records for the three tables, using the sample data in the section titled “Understanding an RDBMS” as reference. (You can also find the SQL commands to build these tables on this book's accompanying website <http://www.mysql-tcr.com/>.)

## Removing and Modifying Records

Just as you insert records into a table, you can also delete records with the `DELETE` command, which looks like this:

```
DELETE FROM table-name
```

For example, the command

```
mysql> DELETE FROM members;
Query OK, 0 rows affected (0.06 sec)
```

would delete all the records from the `members` table.

You can select a specific subset of rows to be deleted by adding the `WHERE` clause to the `DELETE` statement. The following example would delete only those records that had a member ID of 16:

```
mysql> DELETE FROM members WHERE member_id = 16;
Query OK, 1 row affected (0.06 sec)
```

And, finally, there's an `UPDATE` command designed to help you change existing values in a table; it looks like this:

```
UPDATE table-name SET field-name = new-value
```

This command would act on all values in the field *field-name*, changing them all to `<new_value>`. If you'd like to alter the value in a single field only, you can use the `WHERE` clause, as with the `DELETE` command.

Using this knowledge, I could update John Doe's e-mail address in the table:

```
mysql> UPDATE members SET email = 'john@somewhere.com' WHERE member_id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

You can also alter multiple fields by separating them with commas:

```
mysql> UPDATE members SET email = 'john@somewhere.com',
lname = 'Doe The First' WHERE member_id = 2;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

Notice how MySQL provides you with feedback on the number of records matching your query and the number of rows changed by it.

## Executing Queries

Once the data's in the database, it's time to do something with it. MySQL allows you to extract specific "slices" of data from your database using a variety of `SELECT` statements.

The simplest form of the `SELECT` query is the "catch-all" query, which returns all the records in a specific table. It looks like this:

```
mysql> SELECT * FROM members;
+-----+-----+-----+-----+-----+
| member_id | fname | lname  | tel      | email                               |
+-----+-----+-----+-----+-----+
|          1 | John  | Doe    | 1234567  | jdoe@somewhere.com                 |
|          2 | Jane  | Doe    | 8373728  | jane@site.com                       |
|          3 | Steve | Klingon | 7449373  | steve@alien-race.com               |
|          4 | Santa | Claus  | 9999999  | santa@the-north-pole.com           |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

The asterisk (\*) indicates that you'd like to see all the columns present in the table. If, instead, you'd prefer to see only one or two specific columns in the result set, you can specify the column name(s) in the `SELECT` statement, like this:

```
mysql> SELECT lname FROM members;
+-----+
| lname |
+-----+
| Doe   |
| Doe   |
| Klingon |
| Claus |
+-----+
4 rows in set (0.00 sec)
```

In most cases, it is preferable to name the explicit fields that you would like to see in the result set. This allows the application to survive structural changes in its table(s), and it is also usually more efficient because MySQL selects only the fields that it needs.

You can eliminate duplicate entries using the `DISTINCT` keyword; the following query will not display members with the last name "Doe" more than once.

```
mysql> SELECT DISTINCT lname FROM members;
+-----+
| lname |
+-----+
| Doe   |
| Klingon |
| Claus |
+-----+
3 rows in set (0.05 sec)
```

Of course, the whole idea of structuring data into rows and columns is to make it easier to get a focused result set. And a great part of that focus comes from the `WHERE` clause (you may remember this from the `UPDATE` and `DELETE` statements you learned in the preceding sections) to the `SELECT` statement, which allows you to define specific criteria for the result set. Records that do not meet the specified criteria will not appear in the result set.

For example, let's suppose that you want to see a list of all members with the last name "Doe":

```
mysql> SELECT * FROM members WHERE lname = "Doe";
+-----+-----+-----+-----+-----+
| member_id | fname | lname | tel      | email                               |
+-----+-----+-----+-----+-----+
|          1 | John  | Doe   | 1234567  | jdoe@somewhere.com                |
|          2 | Jane  | Doe   | 8373728  | jane@site.com                     |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

Or let's suppose that you want Santa Claus's e-mail address:

```
mysql> SELECT email FROM members WHERE fname = "Santa";
+-----+-----+
| email                               |
+-----+-----+
| santa@the-north-pole.com           |
+-----+-----+
1 row in set (0.06 sec)
```

Or suppose that you want to see a list of all movies by George Lucas:

```
mysql> SELECT title, director FROM videos WHERE director = "George
Lucas";
+-----+-----+-----+
| title                               | director |
+-----+-----+-----+
| Star Wars: The Phantom Menace       | George Lucas |
+-----+-----+-----+
1 row in set (0.06 sec)
```

(Yes, I know the collection is incomplete. Maybe I should write to Santa for the rest...)

### Using Comparison and Logical Operators

You can also use comparison and logical operators to modify your SQL query further. This comes in handy if your table contains a large amount of numeric data, as illustrated here:

name	math	physics	literature
john	68	37	45
jim	96	89	92
bill	65	12	57
harry	69	25	82

The six comparison operators available to use in MySQL are displayed in Table 4-1.

You can also use the logical operators **AND**, **OR**, and **NOT** to create more complex queries. Table 4-2 explains what each one does.

Now, looking at the table of grades, if you wanted to create a list of all students who scored over 90 on their math papers, you could formulate a query that looked like this:

```
mysql> SELECT * FROM grades WHERE math > 90;
+-----+-----+-----+-----+
| name | math | physics | literature |
+-----+-----+-----+-----+
| jim  |  96 |      89 |          92 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Suppose you wanted to identify the smartest kid in class (you know this guy—he always sits in the front row, answers every question perfectly, and usually has wires on his teeth).

```
mysql> SELECT name FROM grades WHERE math > 85
AND physics > 85 AND literature > 85;
+-----+
| name |
+-----+
| jim  |
+-----+
1 row in set (0.00 sec)
```

**TABLE 4-1**  
MySQL  
Comparison  
Operators

Operator	What It Means
=	Is equal to
!=	Is not equal to
>	Is greater than
<	Is less than
>=	Is greater than/equal to
<=	Is less than/equal to

**TABLE 4-2**  
MySQL Logical  
Operators

Operator	What It Means
AND	All of the specified conditions must match.
OR	Any of the specified conditions must match.
NOT	Invert the condition.

What if you needed to identify the students who flunked at least one subject?

```
mysql> SELECT * FROM grades WHERE math <= 25
OR physics <= 25 OR literature <= 25;
+-----+-----+-----+-----+
| name | math | physics | literature |
+-----+-----+-----+-----+
| bill | 65 | 12 | 57 |
| harry | 69 | 25 | 82 |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

And finally, you can also perform basic mathematical operations within your query; the next example demonstrates how the three grades can be added together to create a total grade:

```
mysql> SELECT name, math+physics+literature FROM grades;
+-----+-----+
| name | math+physics+literature |
+-----+-----+
| john | 150 |
| jim | 277 |
| bill | 134 |
| harry | 176 |
+-----+-----+
4 rows in set (0.05 sec)
```

Obviously, such an operation should be attempted only on fields of the same type.

### Using Built-In Functions

MySQL also offers a bunch of built-in functions that come in handy when you're trying to obtain numeric totals and averages of specific fields. The first of these is the useful `COUNT()` function, which counts the number of records in the result set and displays this total.

Consider the following example, which displays the total number of records in the `videos` table:

```
mysql> SELECT COUNT(*) FROM videos;
+-----+
| COUNT(*) |
+-----+
|         6 |
+-----+
1 row in set (0.00 sec)
```

This comes in handy when you quickly need to calculate the total number of records in a table.

The `SUM()` function calculates the sum of the values in the result set, while the `AVG()` function calculates the average. For example, if you wanted to calculate the average grade in math, physics, and literature, you could use a query like this:

```
mysql> SELECT AVG(math), AVG(physics), AVG(literature) FROM grades;
+-----+-----+-----+
| AVG(math) | AVG(physics) | AVG(literature) |
+-----+-----+-----+
|   74.5000 |    40.7500 |    69.0000 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

You can identify the smallest and largest value in a specific column with the `MIN()` and `MAX()` functions. The following queries display the lowest and highest grade in math, respectively:

```
mysql> SELECT MIN(math) FROM grades;
+-----+
| MIN(math) |
+-----+
|         65 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT MAX(math) FROM grades;
+-----+
| MAX(math) |
+-----+
|         96 |
+-----+
1 row in set (0.00 sec)
```

You can read more about MySQL's built-in functions in Chapter 7.

## Ordering and Limiting Result Sets

If you'd like to see the data from your table ordered by a specific field, MySQL offers the `ORDER BY` construct. This construct allows you to specify both the column name and the direction (ascending or descending) in which you would like to see data displayed.

For example, if you'd like to see data from the `members` table arranged by ID, you could try this:

```
mysql> SELECT * FROM members ORDER BY member_id;
+-----+-----+-----+-----+-----+
| member_id | fname | lname | tel | email |
+-----+-----+-----+-----+-----+
|          1 | John  | Doe   | 1234567 | jdoe@somewhere.com |
|          2 | Jane  | Doe   | 8373728 | jane@site.com |
|          3 | Steve | Klingon | 7449373 | steve@alien-race.com |
|          4 | Santa | Claus | 9999999 | santa@the-north-pole.com |
+-----+-----+-----+-----+-----+
4 rows in set (0.06 sec)
```

You could reverse the order with the additional `DESC` modifier:

```
mysql> SELECT * FROM members ORDER BY member_id DESC;
+-----+-----+-----+-----+-----+
| member_id | fname | lname | tel | email |
+-----+-----+-----+-----+-----+
|          4 | Santa | Claus | 9999999 | santa@the-north-pole.com |
|          3 | Steve | Klingon | 7449373 | steve@alien-race.com |
|          2 | Jane  | Doe   | 8373728 | jane@site.com |
|          1 | John  | Doe   | 1234567 | jdoe@somewhere.com |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

You can limit the number of records in the result set with the `LIMIT` keyword. This keyword takes two parameters, which specify the row to start with and the number of rows to display. So the query

```
SELECT * FROM videos LIMIT 2,2;
```

would return rows 3 and 4 from the result set.

```
mysql> SELECT * FROM videos LIMIT 2,2;
+-----+-----+-----+
| video_id | title | director |
+-----+-----+-----+
|          3 | Charlie's Angels | McG |
|          4 | Any Given Sunday | Oliver Stone |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

You can combine the `ORDER BY` and `LIMIT` constructs to get the four newest records in the table quickly, as the following example demonstrates:

```
mysql> SELECT * FROM videos ORDER BY video_id DESC LIMIT 0, 4;
```

video_id	title	director
6	Woman On Top	Fina Torres
5	Hollow Man	Paul Verhoeven
4	Any Given Sunday	Oliver Stone
3	Charlie's Angels	McG

```
4 rows in set (0.00 sec)
```

### Using Wildcards

MySQL also supports the `LIKE` keyword, which is used to return results from a wildcard search and comes in handy when you're not sure what you're looking for. Two types of wildcards are allowed in a `LIKE` construct: the `%` (percent) character, which is used to signify zero or more occurrences of a character, and the `_` (underscore) character, which is used to signify exactly one occurrence of a character.

Let's suppose I wanted a list of all members whose first names contained the letter *e*. My query would look like this:

```
mysql> SELECT * FROM members WHERE fname LIKE '%e%';
```

member_id	fname	lname	tel	email
2	Jane	Doe	8373728	jane@site.com
3	Steve	Klingon	7449373	steve@alien-race.com

```
2 rows in set (0.16 sec)
```

I could also use this technique to search through my `videos` collection for movies containing the word segment *man* in their title.

```
mysql> SELECT title, director FROM videos WHERE title LIKE '%man%';
```

title	director
Hollow Man	Paul Verhoeven
Woman On Top	Fina Torres

```
2 rows in set (0.05 sec)
```

You can read more about executing queries and manipulating table data in Chapter 9.

**Like, You Know, Man...**

It should be noted that the `...LIKE %string%...` construct is generally considered an inefficient and suboptimal way of performing a full-text search, as MySQL is not able to use keys for lookup in this case. The recommended approach in this case is to use full-text indices and a `MATCH AGAINST` command instead (see Chapter 8 for more on this).

**Joining Tables**

So far, all the queries you've seen have been concentrated on a single table. But SQL also allows you to query two or more tables at a time and display a combined result set. This is technically referred to as a *join*, since it involves "joining" different tables at specific points to create new views of the data.

When using a join, it's recommended that you prefix each column name with the name of the table to which it belongs. (I haven't done this in any of the examples you've seen so far because all the columns have been localized to a single table.) For example, you would use `members.fname` to refer to the column named `fname` in the table `members`, and you'd use `status.video_id` to refer to the `video_id` column in the `status` table.

Here's an example of a simple join:

```
mysql> SELECT member_id, video_id, fname FROM status, members WHERE
status.member_id = members.member_id;
+-----+-----+-----+
| member_id | video_id | fname |
+-----+-----+-----+
|          1 |          1 | John  |
|          1 |          2 | John  |
|          1 |          3 | John  |
|          2 |          6 | Jane  |
|          4 |          2 | Santa |
+-----+-----+-----+
5 rows in set (0.00 sec)
```

In this case, the `status` and `members` tables have been joined together through the common column `member_id`.

You can specify the columns you'd like to see from the joined tables, as with any `SELECT` statement:

```
mysql> SELECT fname, lname, video_id FROM members, status WHERE
members.member_id = status.member_id;
```

```
+-----+-----+-----+
| fname | lname | video_id |
+-----+-----+-----+
| Jane  | Doe   |         6 |
| Santa | Claus |         2 |
| John  | Doe   |         1 |
| John  | Doe   |         2 |
| John  | Doe   |         3 |
+-----+-----+-----+
```

```
5 rows in set (0.16 sec)
```

You can also join three tables together. The following example uses the `status` table, combined with member information and video details, to create a composite table that displays which members have which videos.

```
mysql> SELECT fname, lname, title FROM members, videos, status WHERE
status.member_id = members.member_id AND status.video_id =
videos.video_id;
```

```
+-----+-----+-----+
| fname | lname | title
+-----+-----+-----+
| Jane  | Doe   | Woman On Top
| Santa | Claus | ET
| John  | Doe   | Star Wars: The Phantom Menace
| John  | Doe   | ET
| John  | Doe   | Charlie's Angels
+-----+-----+-----+
```

```
5 rows in set (0.17 sec)
```

You can read about more advanced aspects of data retrieval and manipulation, such as joins, subqueries and transactions, in Chapters 10, 11 and 12.

### Joined at the Hip

Note that, when joining tables, it is important to ensure that each join has an associated constraint that permits the use of a key. Otherwise, performance will degrade exponentially as tables grow in size.

**Linking Out**

Interested in learning more about SQL? Here are a few resources, both online and offline, to help you get started:

- *SQL Tutorial* <http://www.w3schools.com/sql/default.asp>
- *A Gentle Introduction to SQL* <http://www.w3schools.com/sql/default.asp>
- *An Interactive Online SQL Course* <http://www.sqlcourse.com/>
- *The SQL.org Portal* <http://www.sql.org/>
- *SQL A Beginner's Guide, Second Edition* by Robert Sheldon (ISBN: 0072228857), McGraw-Hill\Osborne
- *SQL The Complete Reference, Second Edition* by James Groff and Paul Weinberg (ISBN: 0072225599), McGraw-Hill\Osborne
- *Introduction to Relational Databases and SQL Programming* by Christopher Allen, Simon Chatwin and Catherine Creary (ISBN: 0072229241), McGraw-Hill Technology Education

**Aliasing Table Names**

If the thought of writing long table names over and over again doesn't appeal to you, you can assign simple aliases to each table and use these instead. The following example assigns the aliases *m*, *s*, and *v* to the *members*, *status*, and *videos* tables, respectively.

```
mysql> SELECT m.fname, m.lname, v.title FROM members m, status s,
videos v WHERE s.member_id = m.member_id AND s.video_id = v.video_id;
```

fname	lname	title
Jane	Doe	Woman On Top
Santa	Claus	ET
John	Doe	Star Wars: The Phantom Menace
John	Doe	ET
John	Doe	Charlie's Angels

```
5 rows in set (0.00 sec)
```

**Summary**

Over the course of the last few pages, you were briefly introduced to SQL, its history, features, and syntax. We took you on a whirlwind tour of the language, showing you how to create databases and tables; insert, modify, and delete records; and execute queries. We showed you how to create simple queries that return all the records in a table, and then modify those simple queries with operators, wildcards, joins, and built-in functions to filter down to the precise data you need.

This chapter was intended as an overview of MySQL and a primer for the more detailed material ahead. Over the next few chapters, the introductory material in this chapter is discussed in depth, with specific focus on MySQL's particular dialect of SQL.