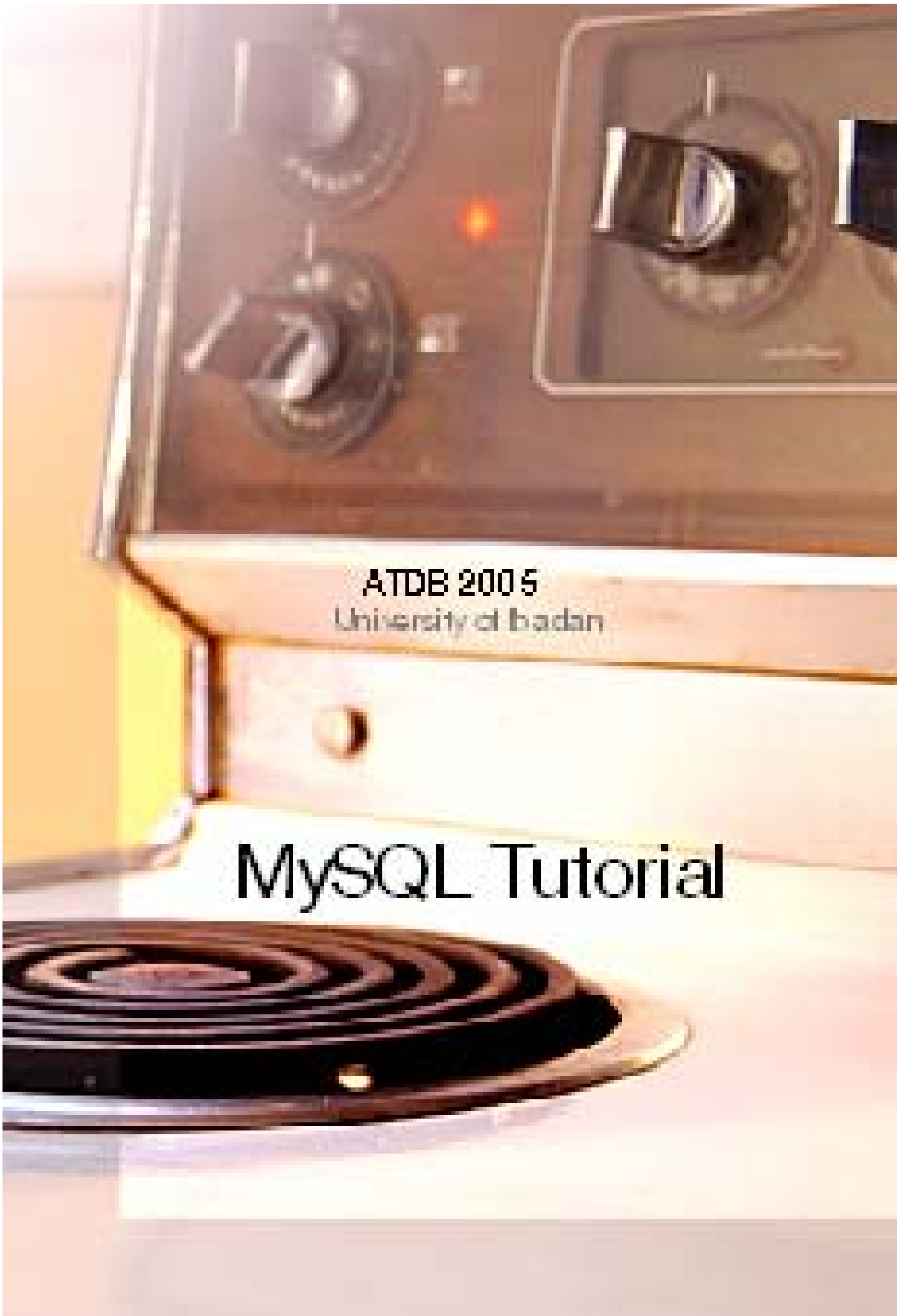

ATDB 2005
University of Ibadan
MySQL Tutorial

This tutorial is inspired by the examples of this book :

MySQL™ The definitive guide to using, programming, and administering MySQL 4.1 and 5.0 Third Edition
By Paul DuBois – SAMS March 2005

Design & Conception: ***P.K helped by A.D, L.W & S.S.***

A MySQL Tutorial

Basic Database Terminology

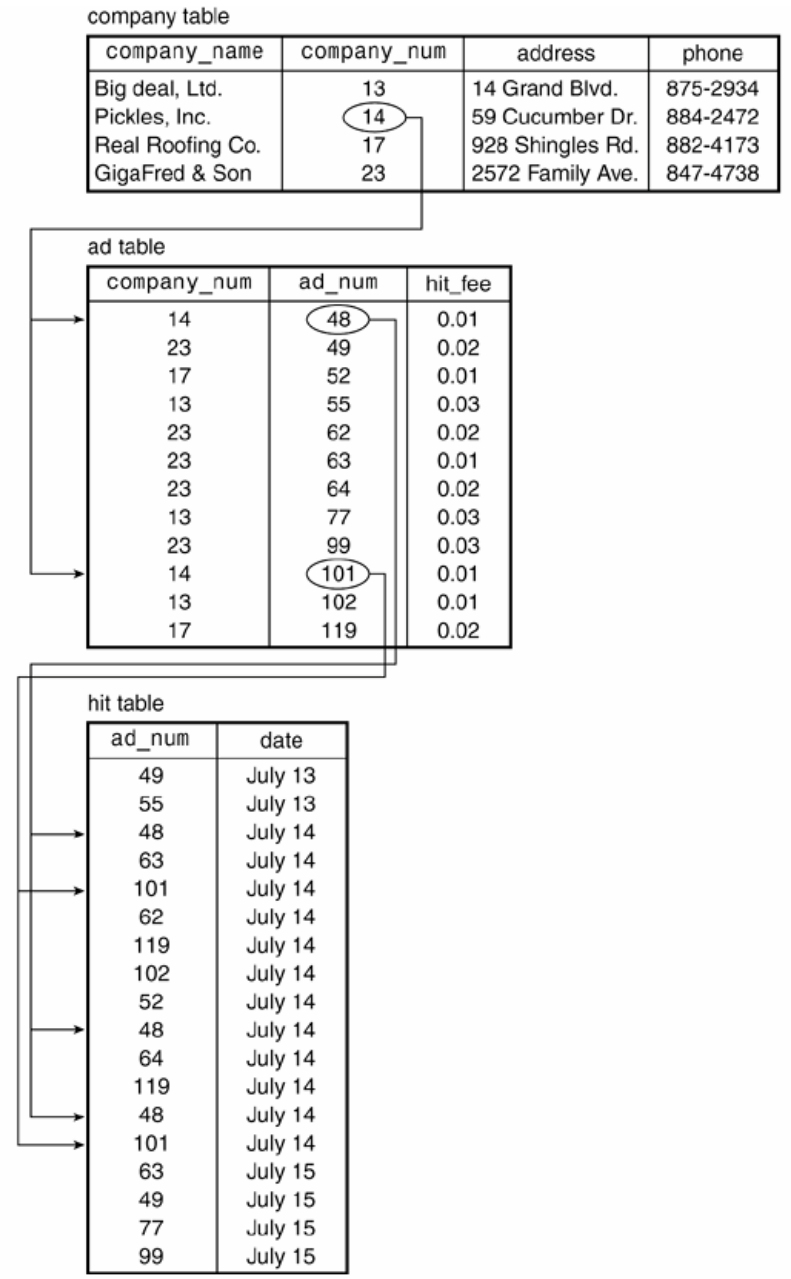
Many relational database concepts are really quite simple. In fact, much of the appeal of relational databases stems from the simplicity of their foundational concepts.

Structural Terminology

Within the database world, MySQL is classified as a relational database management system (RDBMS). That phrase breaks down as follows:

- The database (the "DB" in RDBMS) is the repository for the information you want to store, structured in a simple, regular fashion:
  - The collection of data in a database is organized into tables.
  - Each table is organized into rows and columns.
  - Each row in a table is a record.
  - Records can contain several pieces of information; each column in a table corresponds to one of those pieces.
- The management system (the "MS") is the software that lets you use your data by allowing you to insert, retrieve, modify, or delete records.
- The word "relational" (the "R") indicates a particular kind of DBMS, one that is very good at relating (that is, matching up) information stored in one table to information stored in another by looking for elements common to each of them. The power of a relational DBMS lies in its capability to pull data from those tables conveniently and to join information from related tables to produce answers to questions that can't be answered from individual tables alone.

Here's an example that shows how a relational database organizes data into tables and relates the information from one table to another. Suppose that you run a Web site that includes a banner-advertisement service. You contract with companies that want their ads displayed when people visit the pages on your site. Each time a visitor hits one of your pages, you serve an ad embedded in the page that is sent to the visitor's browser and assess the company a small fee. To represent this information, you maintain three tables. One table, `company`, has columns for company name, number, address, and telephone number. Another table, `ad`, lists ad numbers, the number for the company that "owns" the ad, and the amount you charge per hit. The third table, `hit`, logs each ad hit by ad number and the date on which the ad was served. (*see example below*)



company table

company_name	company_num	address	phone
Big deal, Ltd.	13	14 Grand Blvd.	875-2934
Pickles, Inc.	14	59 Cucumber Dr.	884-2472
Real Roofing Co.	17	928 Shingles Rd.	882-4173
GigaFred & Son	23	2572 Family Ave.	847-4738

ad table

company_num	ad_num	hit_fee
14	48	0.01
23	49	0.02
17	52	0.01
13	55	0.03
23	62	0.02
23	63	0.01
23	64	0.02
13	77	0.03
23	99	0.03
14	101	0.01
13	102	0.01
17	119	0.02

hit table

ad_num	date
49	July 13
55	July 13
48	July 14
63	July 14
101	July 14
62	July 14
119	July 14
102	July 14
52	July 14
48	July 14
64	July 14
119	July 14
48	July 14
101	July 14
63	July 15
49	July 15
77	July 15
99	July 15

Let’s Started…

You have all the background you need now. It's time to put MySQL to work!

This part will help you familiarize yourself with MySQL by providing a tutorial for you to try. As you work through it, you will create a sample database and some tables, and then interact with the database by adding, retrieving, deleting, and modifying information in the tables. During the process of working with the sample database, you will learn the following things:

- The basics of the SQL language that MySQL understands. (If you already know SQL from having used some other RDBMS, it would be a good idea to skim through this tutorial to see whether MySQL's version of SQL differs from the version with which you are familiar.)
- How to communicate with a MySQL server using a few of the standard MySQL client programs. As noted in the previous section, MySQL operates using a client/server architecture in which the server runs on the machine containing the databases and clients connect to the server over a network. This tutorial is based largely on the `mysql` client program, which reads SQL queries from you, sends them to the server to be executed, and displays the results so that you can see what happened. `mysql` runs on all

platforms supported by MySQL and provides the most direct means of interacting with the server, so it's the logical client to begin with.

- This tutorial uses `my_super_db` as the sample database name, but you might need to use a different name as you work through the material. For example, someone else on your system already might be using the name `my_super_db` for their own database, or your MySQL administrator might assign you a different database name. In either case, substitute the actual name of your database for `my_super_db` whenever you see the latter in examples.

Table names can be used exactly as shown in the examples, even if multiple users on your system have their own sample databases. In MySQL, it doesn't matter if other people use the same table names, as long as each of you uses your own database. MySQL prevents you from interfering with each other by keeping the tables in each database separate.

Preliminary Requirements

To try the examples in this tutorial, a few preliminary requirements must be satisfied:

- You need to have the MySQL software installed. (with the WAMP or EasyPHP, it’s easy !!)
- You need a MySQL account so that you can connect to the server.
- You need a database to work with.

The required software includes the MySQL clients and a MySQL server. The client programs must be located on the machine where you'll be working. The server can be located on your machine, although that is not required. As long as you have permission to connect to it, the server can be located anywhere. In addition to the MySQL software, you'll need a MySQL account so that the server will allow you to connect and create your sample database and its tables. (If you already have a MySQL account, you can use that, but you might want to set up a separate account for use with the material in this book.) ..

Note: *At the Computer Science Department (University of Ibadan), Client and Server are installed in each machine with the WAMP distribution.*

At this point, we run into something of a chicken-and-egg problem: In order to set up a MySQL account to use for connecting to the server, it's necessary to connect to the server. Typically, you do this by connecting as the MySQL `root` user on the host where the server is running and issuing a `GRANT` statement to create a new MySQL account. If you've installed MySQL on your own machine and the server is running, you can connect to it and set up a new sample database administrator account with a username of `sampadm` and a password of `secret` as follows (change the name and password to those you want to use, both here and throughout the ‘tuto’):

```
% mysql -p -u root
Enter password:******
mysql> GRANT ALL ON my_super_db.* TO 'sampadm'@'localhost' IDENTIFIED BY 'secret';
```

The `mysql` command includes a `-p` option to cause `mysql` to prompt for the `root` user's MySQL password. Enter the password where you see `******` in the example. I assume that you have already set up a password for the MySQL `root` user and that you know what it is. If you haven't yet assigned a password, just press Enter at the `Enter password:` prompt. However, having no `root` password is insecure and you should assign one as soon as possible.

The `GRANT` statement just shown is appropriate if you'll be connecting to MySQL from the same machine where the server is running. It allows you to connect to the server using the name `sampadm` and the password `secret`, and gives you complete access to the `my_super_db` database. However, `GRANT` doesn't create the database; we'll get to that a bit later.

If you plan to connect to the MySQL server from a different host than the one where the server is running, change `localhost` to the name of the machine where you'll be working. For example, if you will connect to the server from the host `asp.snake.net`, the `GRANT` statement should look like this:

```
mysql> GRANT ALL ON my_super_db.* TO 'sampadm'@'mysite.net' IDENTIFIED BY 'secret';
```

If you don't have control over the server and cannot create an account, ask your MySQL administrator to set up an account for you. Then substitute the MySQL username, password, and database name that the administrator assigns you for `sampadm`, `secret`, and `my_super_db` throughout the examples in this book.

Establishing and Terminating Connections to the MySQL Server

To connect to your server, invoke the `mysql` program from your command prompt (that is, from a console window prompt under Windows). The command looks like this:

```
% mysql options
```

we use `%` throughout this tuto to indicate the command prompt. That's one of the standard Unix prompts; another is `$`. Under Windows, you will see a prompt that looks something like `C:\>`. Note that when entering commands shown in examples, you do not type the prompt itself.

The `options` part of the `mysql` command line might be empty, but more probably you'll have to issue a command that looks something like this:

```
% mysql -h host_name -p -u user_name
```

You might not need to supply all those options when you invoke `mysql`, but it's likely that you'll have to specify at least a name and password. Here's what the options mean:

- `-h host_name` (alternative form: `--host=host_name`)

  The host where the MySQL server is running. If this is the same machine where you are running `mysql`, this option normally can be omitted.

- `-u user_name` (alternative form: `--user=user_name`)

  Your MySQL username. If you're using Unix and your MySQL username is the same as your login name, you can omit this option; `mysql` will use your login name as your MySQL username.

  Under Windows, the default username is `ODBC`, which is unlikely to be a useful default for you. Either specify a `-u` option on the command line, or add a default to your environment by setting the `USER` variable. For example, you can use the following `set` command to specify a username of `sampadm`:

  ```
  C:\> set USER=sampadm
  ```

  If you place this command in your `AUTOEXEC.BAT` file, it takes effect whenever you start up Windows and you won't have to issue it at the prompt.

- `-p` (alternative form: `--password`)

  This option tells `mysql` to prompt you for your MySQL password. For example:

```
% mysql -h host_name -p -u user_name
Enter password:
```

When you see the `Enter password:` prompt, type in your password. (It won't be echoed to the screen, in case someone's looking over your shoulder.) Note that your MySQL password is not necessarily the same as your Unix or Windows password.

If you omit the `-p` option, `mysql` assumes that you don't need one and doesn't prompt for it.

An alternative form of this option is to specify the password value directly on the command line by typing the option as `-pyour_pass` (alternative form: `--password=your_pass`). However, for security reasons, it's best not to do that: The password becomes visible to others that way.

If you do decide to specify the password on the command line, note particularly that there is no space between the `-p` option and the following password value. This behavior of `-p` is a common point of confusion, because it differs from the `-h` and `-u` options, which are associated with the word that follows them regardless of whether there is a space between the option and the word.

Suppose that your MySQL username and password are `sampadm` and `secret`. If the MySQL server is running on the same host where you are going to run `mysql`, you can leave out the `-h` option, and the `mysql` command to connect to the server looks like this:

```
% mysql -p -u sampadm
Enter password:******
```

After you enter the command, `mysql` prints `Enter password:` to prompt for your password, and you type it in (the `******` indicates where you type `secret`).

If all goes well, `mysql` prints a greeting and a `mysql>` prompt indicating that it is waiting for you to issue queries. The full startup sequence looks something like this:

```
% mysql -p -u sampadm
Enter password:******
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 7575 to server version: 4.1.9-log

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

mysql>
```

To connect to a server running on some other machine, it's necessary to specify the hostname using an `-h` option. If that host is `ibadan.supercomputer.net`, the command looks like this:

```
% mysql -h ibadan.supercomputer.net -p -u sampadm
```

In most of the examples that follow that show a `mysql` command line, I'm going to leave out the `-h`, `-u`, and `-p` options for brevity and assume that you'll supply whatever options are necessary. You'll also need to use the same options when you run other MySQL programs, such as `mysqlshow`.

After you've established a connection to the server, you can terminate your session any time by typing `quit`:

```
mysql> quit
Bye
```

You can also quit by typing `exit` or `\q`. On Unix, Ctrl-D also quits.

When you're just starting to learn MySQL, you'll probably consider its security system to be an annoyance because it makes it harder to do what you want. (You must have permission to create and access a database, and you must specify your name and password whenever you connect to the server.) However, after you've moved beyond the sample database used in this book to entering and using your own records, your perspective will change radically. Then you'll appreciate the way that MySQL keeps other people from snooping through (or worse, destroying!) your information.

There are ways to set up your working environment so that you don't have to specify connection parameters on the command line each time you run `mysql`. (see the mysql web site)

Executing SQL Statements

After you're connected to the server, you're ready to issue SQL statements for the server to execute. This section describes some general things you should know about interacting with `mysql`.

To enter a statement in `mysql`, just type it in. At the end of the statement, type a semicolon character (';') and press Enter. The semicolon tells `mysql` that the statement is complete. After you enter a statement, `mysql` sends it to the server to be executed. The server processes the statement and sends the result back to `mysql`, which displays it.

The following example shows a simple statement that asks for the current date and time:

```
mysql> SELECT NOW();
+---------------------+
| NOW()               |
+---------------------+
| 2004-12-29 15:16:31 |
+---------------------+
1 row in set (0.00 sec)
```

`mysql` displays the statement result and a line that shows the number of rows the result consists of and the time elapsed during statement processing. In subsequent examples, I usually will not show the row-count line.

Another way to terminate a statement is to use `\g` rather than a semicolon:

```
mysql> SELECT NOW()\g
+---------------------+
| NOW()               |
+---------------------+
| 2004-12-29 15:16:40 |
+---------------------+
```

Or you can use `\G`, which displays the results in vertical format:

```
mysql> SELECT NOW(), USER(), VERSION()\G
***************************1. row***************************
    NOW(): 2004-12-29 15:16:49
   USER(): sampadm@localhost
VERSION(): 4.1.9-log
```

For a statement that generates short output lines, `\G` is not so useful, but if the lines are so long that they wrap around on your screen, `\G` can make the output much easier to read.

Because `mysql` waits for the statement terminator, you need not enter a statement all on a single line. You can spread it over several lines if you want:

```
mysql> SELECT NOW(),
    -> USER(),
    -> VERSION()
    -> ;
+---------------------+-------------------+-----------+
| NOW()               | USER()            | VERSION() |
+---------------------+-------------------+-----------+
| 2004-12-29 15:16:55 | sampadm@localhost | 4.1.9-log |
+---------------------+-------------------+-----------+
```

Note how the prompt changes from `mysql>` to `->` after you enter the first line of the statement. That tells you that `mysql` thinks you're still entering the statement, which is important feedback: If you forget the semicolon at the end of a statement, the changed prompt helps you realize that `mysql` is still waiting for something. Otherwise, you'll be waiting, wondering why it's taking MySQL so long to execute your statement, and `mysql` will be waiting patiently for you to finish entering your statement! (`mysql` has several other prompts as well)

If you've begun entering a multiple-line statement and decide that you don't want to execute it, type `\c` to clear (cancel) it:

```
mysql> SELECT NOW(),
    -> VERSION(),
    -> \c
mysql>
```

Notice how the prompt changes back to `mysql>` to indicate that `mysql` is ready for a new statement.

The converse of being able to enter a statement over several lines is that you can enter multiple statements on a single line, separated by terminators:

```
mysql> SELECT NOW();SELECT USER();SELECT VERSION();
+---------------------+
| NOW()               |
+---------------------+
| 2004-12-29 15:17:41 |
+---------------------+
+-------------------+
| USER()            |
+-------------------+
| sampadm@localhost |
+-------------------+
+-----------+
| VERSION() |
+-----------+
| 4.1.9-log |
+-----------+
```

For the most part, it doesn't matter whether you enter statements using uppercase, lowercase, or mixed case. These statements are all equivalent:

```
SELECT USER();
select user();
SeLeCt UsEr();
```

The examples in this tuto use uppercase for SQL keywords and function names, and lowercase for database, table, and column names.

When you invoke a function in a statement, there must be no space between the function name and the following parenthesis:

```
mysql> SELECT NOW ();
ERROR 1064 (42000): You have an error in your SQL syntax near '()' at line 1
mysql> SELECT NOW();
+---------------------+
| NOW()               |
+---------------------+
| 2004-12-29 15:18:04 |
+---------------------+
```

These two statements look similar, but the first one fails because the parenthesis doesn't immediately follow the function name.

You can store statements in a file and tell `mysql` to read statements from the file rather than from the keyboard. Use your shell's input redirection facilities for this. For example, if I have statements stored in a file named `myfile.sql`, I can execute its contents with this command:

```
% mysql < myfile.sql
```

You can call the file whatever you want. I use the `.sql` suffix as a convention to indicate that a file contains SQL statements.

Invoking `mysql` this way to execute statements in a file is something that comes up again in "Adding New Records," when we enter data into the `my_super_db` database. It's a lot more convenient to load a table by having `mysql` read `INSERT` statements from a file than to type in each statement manually.

The remainder of this tutorial shows many statements that you can try for yourself. These are indicated by the `mysql>` prompt before the statement, and such examples are usually accompanied by the output of the statement. You should be able to type in these statements as shown, and the resulting output should be the same. Statements that are shown without a prompt are intended simply to illustrate a point, and you need not execute them. (You can try them if you like. If you use `mysql` to do so, remember to include a terminator such as a semicolon at the end of each statement.)

Creating a Database

We'll begin by creating the `my_super_db` sample database and the tables within it, populating its tables, and performing some simple queries on the data contained in those tables. Using a database involves several steps:

1. Creating (initializing) the database
2. Creating the tables within the database
3. Interacting with the tables by inserting, retrieving, modifying, or deleting data

Retrieving existing data is easily the most common operation performed on a database. The next most common operations are inserting new data and updating or deleting existing data. Less frequent are table creation

operations, and least frequent of all is database creation. However, we're beginning from scratch, so we must begin with database creation, the least common thing, and work our way through table creation and insertion of our initial data before we get to where we can do the really common thingretrieving data.

To create a new database, connect to the server using `mysql`. Then issue a `CREATE DATABASE` statement that specifies the database name:

```
mysql> CREATE DATABASE my_super_db;
```

You'll need to create the `my_super_db` database before you can create any of the tables that will go in it or do anything with the contents of those tables.

You might think that creating the database would also make it the default (or current) database, but it doesn't. You can see this by executing the following statement to check what the default database is:

```
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| NULL       |
+------------+
```

`NULL` means "no database is selected." To make `my_super_db` the default database, issue a `USE` statement:

```
mysql> USE my_super_db;
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| my_super_db     |
+------------+
```

Another way to select a default database is to name it on the command line when you invoke `mysql`:

```
% mysql my_super_db
```

That is, in fact, the usual way to select the database you want to use. If you need any connection parameters, specify them on the command line. For example, the following command allows the `sampadm` user to connect to the `my_super_db` database on the local host (the default when no host is named):

```
% mysql -p -u sampadm my_super_db
```

This command connects to the MySQL server running on the host `cobra.snake.net`:

```
% mysql -h cobra.snake.net -p -u sampadm my_super_db
```

Unless specified otherwise, all following examples assume that when you invoke `mysql`, you name the `my_super_db` database on the command line to make it the current database. If you invoke `mysql` but forget to name the database on the command line, just issue a `USE my_super_db` statement at the `mysql>` prompt.

Creating Tables

In this section, we'll build the tables that are needed for the `my_super_db` sample database. First, we'll consider the tables needed for the Historical League, and then those for the grade-keeping project. This is the part where some database books start talking about Analysis and Design, Entity-Relationship Diagrams, Normalization Procedures, and other such stuff. There's a place for all that, but I prefer just to say we need to think a bit about what our database will look like: what tables it should contain, what the contents of each table should be, and some of the issues involved in deciding how to represent the data.

The choices made here about data representation are not absolute. In other situations, you might well elect to represent similar data in a different way, depending on the requirements of your applications and the ways you intend to use your data.

Tables for the U.S. Historical League

Table layout for the Historical League is pretty simple:

- A `president` table. This contains a descriptive record for each U.S. president. We'll need this for the online quiz on the League Web site (the interactive analog to the printed quiz that appears in the children's section of the League's newsletter).
- A `member` table. This is used to maintain current information about each member of the League. It'll be used for creating printed and online versions of the member directory, sending automated membership renewal reminders, and so forth.

The `president` Table

The `president` table is simpler, so let's discuss it first. This table will contain some basic biographical information about each United States president:

- Name. Names can be represented in a table several ways. For example, we could have a single column containing the entire name, or separate columns for the first and last name. It's certainly simpler to use a single column, but that limits you in some ways:
    - If you enter the names with the first name first, you can't sort on last name.
    - If you enter the names with the last name first, you can't display them with the first name first.
    - It's harder to search for names. For example, to search for a particular last name, you must use a pattern and look for names that match the pattern. This is less efficient and slower than looking for an exact last name.

  To avoid these limitations, our `president` table will use separate columns for the first and last names.

  The first name column will also hold the middle name or initial. This shouldn't break any sorting we might do because it's not likely we'll want to sort on middle name (or even first name). Name display should work properly, too, because the middle name immediately follows the first name regardless of whether a name is printed in "Bush, George W." or in "George W. Bush" format.

  There is another slight complication. One president (Jimmy Carter) has a "Jr." at the end of his name. Where does that go? Depending on the format in which names are printed, this president's name is displayed as "James E. Carter, Jr.," or "Carter, James E., Jr." The "Jr." doesn't associate with either first or last name, so we'll create another column to hold a name suffix. This illustrates how even a single value can cause problems when you're trying to determine how to represent your data. It also shows why it's a good idea to know as much as possible about the data values you'll be working with before you put them in a database. If you have incomplete knowledge of what your values look like, you might have to change your table structure after you've already begun to use it. That's not necessarily a disaster, but in general it's something you want to avoid.

- Birthplace (city and state). Like the name, this too can be represented using a single column or multiple columns. It's simpler to use a single column, but as with the name, separate columns allow you to do

some things you can't do easily otherwise. For example, it's easier to find records for presidents born in a particular state if city and state are listed separately. We'll use separate columns for the two values.
- Birth date and death date. The only special problem here is that we can't require the death date to be filled in because some presidents are still living. The special value `NULL` means "no value," so we can use that in the death date column to signify "still alive."

The `member` Table

The `member` table for the Historical League membership list is similar to the `president` table in the sense that each record contains basic descriptive information for a single person. But each `member` record contains more columns:

- Name. We'll use the same three-column representation as for the `president` table: last name, first name, and suffix.
- ID number. This is a unique value assigned to each member when membership first begins. The League hasn't ever used ID numbers before, but now that the records are being made more systematic, it's a good time to start. (I am anticipating that you'll find MySQL beneficial and that you'll think of other ways to apply it to the League's records. When that happens, it'll be easier to associate records in the `member` table with other member-related tables that you create if you use numbers rather than names.)
- Expiration date. Members must renew their memberships periodically to avoid having them lapse. For some applications, you might store the start date of the most recent renewal, but this is not suitable for the League's purposes. Memberships can be renewed for a variable number of years (typically one, two, three, or five years), and a date for the most recent renewal wouldn't tell you when the next renewal must take place. Therefore, we will store the end date of the membership. In addition, the League allows lifetime memberships. We could represent these with a date far in the future, but `NULL` seems more appropriate because "no value" logically corresponds to "never expires."
- Email address. Publishing these addresses will make it easier for those members that have them to communicate with each other more easily. For your purposes as League secretary, these addresses will allow you to send out membership renewal notices electronically rather than by postal mail. This should be easier than going to the post office, and less expensive as well. You'll also be able to use email to send members the current contents of their directory entries and ask them to update the information as necessary.
- Postal address. This is needed for contacting members that don't have email (or who don't respond to it). We'll use columns for street address, city, state, and ZIP code.

  I'm assuming that all League members live in the United States. For organizations with a membership that is international in scope, that assumption is an oversimplification. If you want to deal with addresses from multiple countries, you'll run into some sticky issues having to do with the different address formats used for different countries. For example, ZIP code is not an international standard, and some countries have provinces rather than states.

- Phone number. Like the address columns, this is useful for contacting members.
- Special interest keywords. Every member is assumed to have a general interest in U.S. history, but members probably also have some special areas of interest. This column records those interests. Members can use it to find other members with similar interests.

Creating the Historical League Tables

Now we're ready to create the Historical League tables. For this we use the `CREATE TABLE` statement, which has the following general form:

```
CREATE TABLE tbl_name ( column_specs );
```

`tbl_name` indicates the name you want to give the table. `column_specs` provides the specifications for the columns in the table. It also includes definitions for indexes, if there are any.

For the president table, write the CREATE TABLE statement as follows:

```
CREATE TABLE president
(
    last_name   VARCHAR(15) NOT NULL,
    first_name  VARCHAR(15) NOT NULL,
    suffix      VARCHAR(5) NULL,
    city        VARCHAR(20) NOT NULL,
    state       VARCHAR(2) NOT NULL,
    birth       DATE NOT NULL,
    death       DATE NULL
);
```

You can execute this statement a couple of ways. Either enter it manually by typing it in, or use the prewritten statement that is contained in the create_president.sql file of the my_super_db distribution.

If you want to type in the statement yourself, invoke mysql, making my_super_db the current database:

```
% mysql my_super_db
```

Then enter the CREATE TABLE statement as just shown, including the trailing semicolon so that mysql can tell where the statement ends. Indentation doesn't matter, and you need not put the line breaks in the same places. For example, you can enter the statement as one long line if you want.

To create the president table using a prewritten description, use the create_president.sql file from the my_super_db distribution. This file is located in the my_super_db directory that is created when you unpack the distribution. Change location into that directory, and then run the following command:

```
% mysql my_super_db < create_president.sql
```

Whichever way you invoke mysql, specify any connection parameters you might need (hostname, username, or password) on the command line preceding the database name.

Now let's look more closely at the CREATE TABLE statement. Each column specification in the statement consists of the column name, the data type (the kind of values the column will hold), and possibly some column attributes.

The two data types used in the president table are VARCHAR and DATE. VARCHAR(n) means the column contains variable-length character values, with a maximum length of n characters each. That is, they contain strings that might vary in size, but with an upper bound on their length. You choose the value of n according to how long you expect your values to be. state is defined as VARCHAR(2); that's all we need for entering states by their two-character abbreviations. The other string-valued columns need to be wider to accommodate longer values.

The other data type we've used is DATE. This type indicates, not surprisingly, that the column holds date values. However, what might surprise you is the format in which dates are represented. MySQL expects dates to be written in 'CCYY-MM-DD' format, where CC, YY, MM, and DD represent the century, year within the century, month, and date. This is the SQL standard for date representation (also known as "ISO 8601 format"). For example, to specify a date of "July 23, 2005" in MySQL, you use '2005-07-23', not '07-23-2005' or '23-07-2005'.

The only attributes we're using for the columns in the president table are NULL (values can be missing) and NOT NULL (values must be filled in). Most columns are NOT NULL, because we'll always have a value for them. The two columns that can have NULL values are suffix (most names don't have one), and death (some presidents are still alive, so there is no date of death).

For the member table, the CREATE TABLE statement looks like this:

```
CREATE TABLE member
(
    member_id   INT UNSIGNED NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (member_id),
    last_name   VARCHAR(20) NOT NULL,
    first_name  VARCHAR(20) NOT NULL,
    suffix      VARCHAR(5) NULL,
    expiration  DATE NULL DEFAULT '0000-00-00',
    email       VARCHAR(100) NULL,
    street      VARCHAR(50) NULL,
    city        VARCHAR(50) NULL,
    state       VARCHAR(2) NULL,
    zip         VARCHAR(10) NULL,
    phone       VARCHAR(20) NULL,
    interests   VARCHAR(255) NULL
);
```

As before, you can either type that statement manually into mysql or you can use a prewritten file. The file from the my_super_db distribution that contains the CREATE TABLE statement for the member table is create_member.sql. To use it, execute this command:

```
% mysql my_super_db < create_member.sql
```

In terms of data types, most columns of the member table except two are not very interesting because they are created as variable-length strings. The exceptions are member_id and expiration, which exist to hold sequence numbers and dates, respectively.

The primary consideration for the member_id membership number column is that each of its values should be unique to avoid confusion between members. An AUTO_INCREMENT column is useful here because then we can let MySQL generate unique numbers for us automatically when we add new members. Even though it just contains numbers, the definition for member_id has several parts:

- INT signifies that the column holds integers (numeric values with no fractional part).
- UNSIGNED disallows negative values.
- NOT NULL requires that the column value must be filled in. This prevents members from being created without an ID number.
- AUTO_INCREMENT is a special attribute in MySQL. It indicates that the column holds sequence numbers. The AUTO_INCREMENT mechanism works like this: If you provide no value for the member_id column when you create a new member table record, MySQL automatically generates the next sequence number and assigns it to the column. This special behavior also occurs if you explicitly assign the value NULL to the column. The AUTO_INCREMENT feature makes it easy to assign a unique ID to each new member, because MySQL generates the values for us.

The PRIMARY KEY clause indicates that the member_id column is indexed to allow fast lookups. It also sets up the constraint that each value in the column must be unique. The latter property is desirable for member ID values, because it prevents us from using the same ID twice by mistake. (Besides, MySQL requires every AUTO_INCREMENT column to have some kind of index, so the table definition would be illegal without one.)

If you don't understand that stuff about AUTO_INCREMENT and PRIMARY KEY, just think of them as giving us a magic way of generating ID numbers. It doesn't particularly matter what the values are, as long as they're unique for each member

The expiration column is a DATE. It has a default value of '0000-00-00', which is a non-NULL value that means no legal date has been entered. The reason for this is that, as mentioned earlier, we're using the

convention that `expiration` can be `NULL` to indicate which members have a lifetime membership. If we don't specify otherwise, a column that can contain `NULL` also has `NULL` as its default value. That's not desirable in this case: If you create a new member record but forget to specify the expiration date, MySQL fills in the `expiration` column with `NULL` automatically thus making the member appear to have a lifetime membership! By specifying that the column has a default value of `'0000-00-00'` instead, we avoid this problem. That default also gives us a value we can search for periodically to find records for which the expiration date was mistakenly omitted.

Now that you've told MySQL to create a couple of tables, check to make sure that it did so as you expect. In `mysql`, issue the following statement to see the structure of the `president` table:

```
mysql> DESCRIBE president;
+------------+-------------+------+-----+------------+-------+
| Field      | Type        | Null | Key | Default    | Extra |
+------------+-------------+------+-----+------------+-------+
| last_name  | varchar(15) |      |     |            |       |
| first_name | varchar(15) |      |     |            |       |
| suffix     | varchar(5)  | YES  |     | NULL       |       |
| city       | varchar(20) |      |     |            |       |
| state      | char(2)     |      |     |            |       |
| birth      | date        |      |     | 0000-00-00 |       |
| death      | date        | YES  |     | NULL       |       |
+------------+-------------+------+-----+------------+-------+
```

The output looks pretty much as we'd expect, except that the information for the `state` column says its type is `CHAR(2)`. That's odd; wasn't it defined as `VARCHAR(2)`? Yes, it was, but MySQL has silently changed the type from `VARCHAR` to `CHAR`. The reason for this has to do with efficiency of storage space for short character columns, which I won't go into here. For our purposes here, there is no difference between the two types. The important thing is that the column stores two-character values.

If you issue a `DESCRIBE member` statement, `mysql` will show you similar information for the `member` table.

`DESCRIBE` is useful when you forget the name of a column in a table, or need to know its type or how wide it is, and so forth. It's also useful for finding out the order in which MySQL stores columns in table rows. That order is important when you issue `INSERT` or `LOAD DATA` statements that expect column values to be listed in the default column order.

The information produced by `DESCRIBE` can be obtained in different ways. It may be abbreviated as `DESC`, or written as an `EXPLAIN` or `SHOW` statement. The following statements all are synonymous:

```
DESCRIBE president;
DESC president;
EXPLAIN president;
SHOW COLUMNS FROM president;
SHOW FIELDS FROM president;
```

These statements also allow you to restrict the output to particular columns. For example, you can add a `LIKE` clause at the end of a `SHOW` statement to display information only for column names that match a given pattern:

```
mysql> SHOW COLUMNS FROM president LIKE '%name';
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| last_name  | varchar(15) |      |     |         |       |
| first_name | varchar(15) |      |     |         |       |
+------------+-------------+------+-----+---------+-------+
```

`DESCRIBE president '%name'` is equivalent. The '%' character used here is a special wildcard character.

The `SHOW` statement has other forms that are useful for obtaining different types of information from MySQL. `SHOW TABLES` lists the tables in the current database, so with the two tables we've created so far in the `my_super_db` database, the output looks like this:

```
mysql> SHOW TABLES;
+----------------------------+
| Tables_in_my_super_db      |
+----------------------------+
| member                     |
| president                  |
+----------------------------+
```

`SHOW DATABASES` lists the databases that are managed by the server to which you're connected:

```
mysql> SHOW DATABASES;
+-----------+
| Database  |
+-----------+
| menagerie |
| mysql     |
| my_super_db|
| test      |
+-----------+
```

The list of databases varies from server to server, but you should see at least `my_super_db` and `test`. You created `my_super_db` yourself, and the database named `test` is created as part of the MySQL installation procedure. Depending on your access rights, you might see the database named `mysql`, which holds the grant tables that control who can do what.

The `mysqlshow` client program provides a command-line interface to the same kinds of information that the `SHOW` statement displays. With no arguments, `mysqlshow` displays a list of databases:

```
% mysqlshow
+-------------+
|  Databases  |
+-------------+
| menagerie   |
| mysql       |
| my_super_db |
| test        |
+-------------+
```

With a database name, `mysqlshow` displays the tables in the given database:

```
% mysqlshow my_super_db
Database: my_super_db
+-----------+
|  Tables   |
+-----------+
| member    |
| president |
+-----------+
```

With a database and table name, `mysqlshow` displays information about the columns in the table, much like the `SHOW COLUMNS` statement.

Remember that when you run `mysqlshow`, you might need to provide appropriate command-line options for username, password, and hostname. These options are the same as when you run `mysql`.

Tables for the Grade-Keeping Project

To determine what tables are required for the grade-keeping project, let's consider how you might write down scores when you use a paper-based gradebook. The main body of this page is a matrix for recording scores. There is also other information needed for making sense of the scores. Student names and ID numbers are listed down the side of the matrix. (For simplicity, only four students are shown.) Along the top of the matrix, you put down the dates when you give quizzes and tests. The figure shows that you've given quizzes on September 3, 6, 16, and 23, and tests on September 9 and October 1.

students		scores						
		Q	Q	T	Q	Q	T	
ID	name	9/3	9/6	9/9	9/16	9/23	10/1	...
1	Billy	14	10	73	14	15	67	...
2	Missy	17	10	68	17	14	73	...
3	Johnny	15	10	78	12	17	82	...
4	Jenny	14	13	85	13	19	79	...
...	...	...	...	...	...	...	...	...

To keep track of this kind of information using a database, we need a `score` table. What should records in this table contain? That's easy. For each row, we need the student name, the date of the quiz or test, and the score.

`score` table

name	date	score
Billy	2004-09-23	15
Missy	2004-09-23	14
Johnny	2004-09-23	17
Jenny	2004-09-23	19
Billy	2004-10-01	67
Missy	2004-10-01	73
Johnny	2004-10-01	82
Jenny	2004-10-01	79

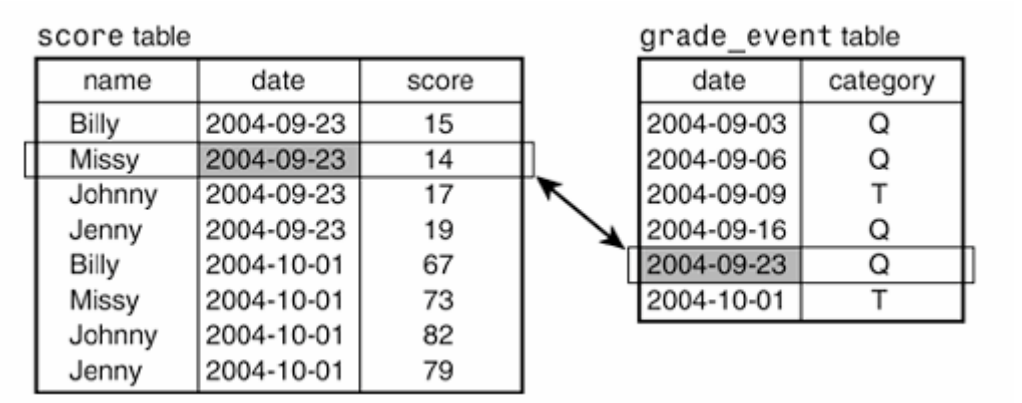
Unfortunately, there is a problem with setting up the table in this way, because it leaves out some information. For example, looking at the records in table below, we can't tell whether scores are for a quiz or a test. It could be important to know score categories when determining final grades if quizzes and tests are weighted differently. We might try to infer the category from the range of scores on a given date (quizzes usually are worth fewer points than a test), but that's problematic because it relies on inference and not something explicit in the data.

It's possible to distinguish scores by recording the category in each record, for example, by adding a column to the `score` table that contains '`T`' or '`Q`' for each row to indicate "test" or "quiz". This has the advantage of making the score category explicit in the data. The disadvantage is redundancy. Observe that for all records with a given date, the score category column always has the same value. The scores for September 23 all have a category of '`Q`', and those for October 1 all have a category of '`T`'. This is unappealing. If we record a set of scores for a quiz or test this way, not only will we be putting in the same date for each new record in the set, we'll be putting in the same score category over and over again. Ugh. Who wants to enter all that redundant information?

score table

name	date	score	category
Billy	2004-09-23	15	Q
Missy	2004-09-23	14	Q
Johnny	2004-09-23	17	Q
Jenny	2004-09-23	19	Q
Billy	2004-10-01	67	T
Missy	2004-10-01	73	T
Johnny	2004-10-01	82	T
Jenny	2004-10-01	79	T

Let's try an alternative representation. Instead of recording score categories in the `score` table, we'll figure them out from the dates. We can keep a list of dates and use it to keep track of what kind of "grade event" (quiz or test) occurred on each date. Then we can determine whether any given score was from a quiz or a test by combining it with the information in our event list: Match the date in the `score` table record with the date in the `grade_event` table to get the event category. The next figure shows this table layout and demonstrates how the association works for a `score` table record with a date of September 23. By matching the record with the corresponding record in the `grade_event` table, we see that the score is from a quiz.



This is much better than trying to infer the score category based on some guess. Instead, we're deriving the category directly from data recorded explicitly in the database. It's also preferable to recording score categories in the `score` table; now we need to record each category only one time, rather than once per score record.

However, now we're combining information from multiple tables. If you're like me, when you first hear about this kind of thing, you think, "Yeah, that's a cute idea, but isn't it a lot of work to do all that looking up all the time; doesn't it just make things more complicated?"

In a way, that's correct; it is more work. Keeping two lists of records is more complicated than keeping one list. But take another look at your gradebook. Aren't you already keeping two sets of records? Consider these facts:

- You keep track of scores using the cells in the score matrix, where each cell is indexed by student name and date (down the side and along the top of the matrix). This represents one set of records; it's analogous to the contents of the `score` table.
- How do you know what kind of event each date represents? You've written a little '`T`' or '`Q`' above the date! Thus, you're also keeping track of the association between date and score category along the top of the matrix. This represents a second set of records; it's analogous to the `grade_event` table contents.

In other words, even though you may not think about it as such, you're really not doing anything different with the gradebook than what I'm proposing to do by keeping information in two tables. The only real difference is that the two kinds of information aren't so explicitly separated in the paper-based gradebook.

The page in the gradebook illustrates something about the way we think of information, and also something about the difficulty of figuring out how to put information in a database: We tend to integrate different kinds of information and interpret them as a whole. Databases don't work like that, which is one reason they sometimes seem artificial and unnatural. Our natural tendency to unify information makes it quite difficult sometimes even to realize when we have multiple types of data instead of just one. Because of this, you may find it a challenge to "think as a database thinks" about how best to represent your data.

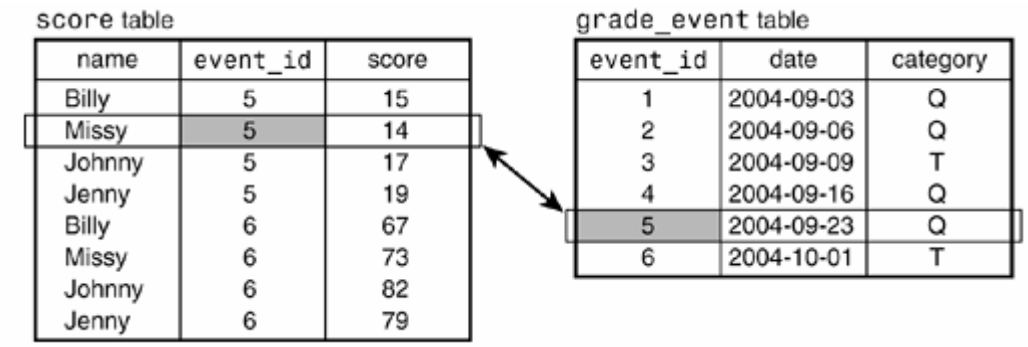
One requirement imposed on the `grade_event` table is that the dates must be unique because each date is used to link together records from the `score` and `grade_event` tables. In other words, you cannot give two quizzes on the same day, or a quiz and a test. If you do, you'll have two sets of records in the `score` table and two records in the `grade_event` table, all with the same date, and you won't be able to tell how to match `score` records with `grade_event` records.

That problem will never come up if there is never more than one grade event per day. But is it valid to assume that will never happen? It might seem so; after all, you don't consider yourself sadistic enough to give a quiz and a test on the same day. But I hope you'll pardon me if I'm skeptical. I've often heard people claim about their data, "That odd case will never occur." Then it turns out the odd case does occur on occasion, and usually you have to redesign your tables to fix problems that the odd case causes.

It's better to think about the possible problems in advance and anticipate how to handle them. So, let's suppose that you might need to record two sets of scores for the same day sometimes. How can we handle that? As it turns out, this problem isn't so difficult to solve. With a minor change to the way we lay out our data, multiple events on a given date won't cause trouble:

1. Add a column to the `grade_event` table and use it to assign a unique number to each record in the table. In effect, this gives each event its own ID number, so we'll call this the `event_id` column.
2. When you put scores in the `score` table, record the event ID rather than the date.

The result of these changes is shown in the next figure. Now you link together the `score` and `grade_event` tables using the event ID rather than the date, and you use the `grade_event` table to determine not just the category of each score, but also the date on which it occurred. Also, it's no longer the date that must be unique in the `grade_event` table, it's the event ID. This means you can have a dozen tests and quizzes on the same day, and you'll be able to keep them straight in your records. (No doubt your students will be thrilled to hear this.)



Unfortunately, from a human standpoint, this table layout seems less satisfactory than the previous ones. The `score` table is more abstract because it contains fewer columns that have a readily apparent meaning. The table layout shown earlier was easy to look at and understand because the `score` table had columns for both dates and score categories. The current `score` table has columns for neither. This seems highly removed from anything we can think about easily. Who wants to look at a `score` table that has "event IDs" in it? That just doesn't mean much to us.

At this point you reach a crossroads. You're intrigued by the possibility of being able to perform grade-keeping electronically and not having to do all kinds of tedious manual calculations when assigning grades. But after considering how you actually would represent score information in a database, you're put off by how abstract and disconnected the representation seems to make that information.

This leads naturally to a question: "Would it be better not to use a database at all? Maybe MySQL isn't for me." As you might guess, we will answer that question in the negative, because otherwise this book will come to a quick end. But when you're thinking about how to do a job, it's not a bad idea to consider various alternatives and to ask whether you're better off using a database system such as MySQL, or something else such as a spreadsheet program:

- The gradebook has rows and columns, and so does a spreadsheet. This makes the gradebook and a spreadsheet conceptually and visually very similar.
- A spreadsheet program can perform calculations, so you could total up each student's scores using a calculation field. It might be a little tricky to weight quizzes and tests differently, but you could do it.

On the other hand, if you want to look at just part of your data (quizzes only or tests only, for example), perform comparisons such as boys versus girls, or display summary information in a flexible way, it's a different story. A spreadsheet doesn't work so well, whereas relational database systems perform those operations easily.

Another point to consider is that the abstract and disconnected nature of your data as represented in a relational database is not really a big deal, anyway. It's necessary to think about that representation when setting up the database so that you don't lay out your data in a way that doesn't make sense for what you want to do with it. However, after you determine the representation, you're going to rely on the database engine to pull together and present your data in a way that is meaningful to you. You're not going to look at it as a bunch of disconnected pieces.

For example, when you retrieve scores from the `score` table, you don't want to see event IDs; you want to see dates. That's not a problem. The database can look up dates from the `grade_event` table based on the event ID and show them to you. You may also want to see whether the scores are for tests or quizzes. That's not a problem, either. The database can look up score categories the same wayusing event ID. Remember, that's what a relational database system like MySQL is good at: relating one thing to another to pull out information from multiple sources to present you with what you really want to see. In the case of our grade-keeping data, MySQL does the thinking about pulling information together using event IDs so that you don't have to.

Now, just to provide a little advance preview of how you'd tell MySQL to do this relating of one thing to another, suppose that you want to see the scores for September 23, 2004. The query to pull out scores for an event given on a particular date looks like this:

```
SELECT score.name, grade_event.date, score.score, grade_event.category
FROM score, grade_event
WHERE grade_event.date = '2004-09-23'
AND score.event_id = grade_event.event_id;
```

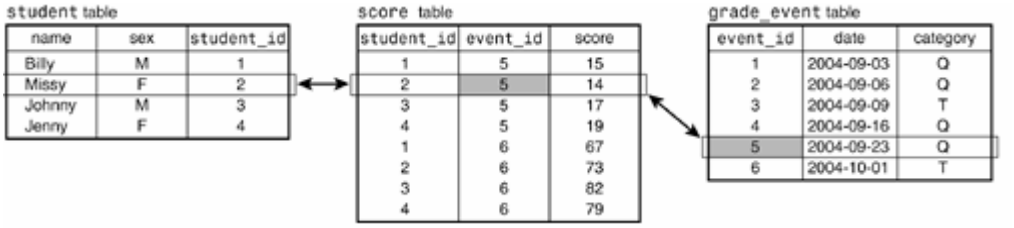
Pretty scary, huh? This query retrieves the student name, the date, score, and the score category by joining (relating) `score` table records to `grade_event` table records. The result looks like this:

```
+--------+------------+-------+----------+
| name   | date       | score | category |
+--------+----------- +-------+----------+
| Billy  | 2004-09-23 |    15 | Q        |
| Missy  | 2004-09-23 |    14 | Q        |
| Johnny | 2004-09-23 |    17 | Q        |
| Jenny  | 2004-09-23 |    19 | Q        |
+--------+------------+-------+----------+
```

Notice anything familiar about the format of that information? You should; it's the same as the before last table layout! And you don't need to know the event ID to get this result. You specify the date you're interested in and let MySQL figure out which score records go with that date. So, if you've been wondering whether all the abstraction and disconnectedness loses us anything when it comes to getting information out of the database in a form that's meaningful to us, it doesn't.

Of course, after looking at that query, you might be wondering something else, too. Namely, it looks kind of long and complicated; isn't writing something like that a lot of work to go to just to find the scores for a given date? Yes, it is. However, there are ways to avoid typing several lines of SQL each time you want to issue a query. Generally, you figure out once how to perform a query such as that one and then you store it so that you can repeat it easily as necessary.

we've actually jumped the gun a little bit in showing you that query. It is, believe it or not, a little simpler than the one we're really going to use to pull out scores. The reason for this is that we need to make one more change to our table layout. Instead of recording student name in the `score` table, we'll use a unique student ID. (That is, we'll use the value from the "ID" column of your gradebook rather than from the "Name" column.) Then we create another table called `student` that contains name and `student_id` columns.



`student` table

name	sex	student_id
Billy	M	1
Missy	F	2
Johnny	M	3
Jenny	F	4

`score` table

student_id	event_id	score
1	5	15
2	5	14
3	5	17
4	5	19
1	6	67
2	6	73
3	6	82
4	6	79

`grade_event` table

event_id	date	category
1	2004-09-03	Q
2	2004-09-06	Q
3	2004-09-09	T
4	2004-09-16	Q
5	2004-09-23	Q
6	2004-10-01	T

Why make this modification? For one thing, there might be two students with the same name. Using a unique student ID number helps you tell their scores apart. (This is exactly analogous to the way you can tell scores apart for a test and quiz given on the same day by using a unique event ID rather than the date.) After making this change to the table layout, the query we'll actually use to pull out scores for a given date becomes a little more complex:

```
SELECT student.name, grade_event.date, score.score, grade_event.category
FROM grade_event, score, student
WHERE grade_event.date = '2004-09-23'
AND grade_event.event_id = score.event_id
AND score.student_id = student.student_id;
```

If you're concerned because you don't find the meaning of that query immediately obvious, don't be. Most people wouldn't. We'll see the query again after we get further along into this tutorial, but the difference between now and later is that later you'll understand it. And, no, I'm not kidding.

You'll note that we added something to the `student` table that wasn't in your gradebook: It contains a column for recording sex. This will allow for simple things such as counting the number of boys and girls in the class or more complex things like comparing scores for boys and girls.

We're almost done designing the tables for the grade-keeping project. We need just one more table to record absences for attendance purposes. Its contents are relatively straightforward: a student ID number and a date Each row in the table indicates that the given student was absent on the given date. At the end of the grading period, we'll call on MySQL's counting abilities to summarize the table's contents to tell us how many days each student was absent.

absence table

student_id	date
2	2004-09-02
4	2004-09-15
2	2004-09-20

Now that we know what our grade-keeping tables should look like, we're ready to create them. The `CREATE TABLE` statement for the `student` table looks like this:

```
CREATE TABLE student
(
    name        VARCHAR(20) NOT NULL,
    sex         ENUM('F','M') NOT NULL,
    student_id  INT UNSIGNED NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (student_id)
) ENGINE = InnoDB;
```

Observe that I've added something new to the `CREATE TABLE` statement (the `ENGINE` clause at the end). I will explain its purpose shortly. If you happen to be using an older version of MySQL that does not understand `ENGINE`, substitute the keyword `TYPE` instead.

Type the `CREATE TABLE` statement into `mysql` or execute the following command:

```
% mysql my_super_db < create_student.sql
```

The `CREATE TABLE` statement creates a table named `student` with three columns: `name`, `sex`, and `student_id`.

`name` is a variable-length string column that can hold up to 20 characters. This name representation is simpler than the one used for the Historical League tables; it uses a single column rather than separate first name and last name columns. That's because I know in advance that no grade-keeping query examples will need to do anything that would work better with separate columns. (Yes, that's cheating. I admit it.)

`sex` indicates whether a student is a boy or a girl. It's an `ENUM` (enumeration) column, which means it can take only one of the values listed in the column specification: `'F'` for female or `'M'` for male. `ENUM` is useful when you have a restricted set of values that a column can hold. We could have used `CHAR(1)` instead, but `ENUM` makes it more explicit what the column values can be. If you forget what the possible values are, issue a `DESCRIBE` statement. For an `ENUM` column, MySQL displays the list of legal enumeration values:

```
mysql> DESCRIBE student 'sex';
+-------+---------------+------+-----+---------+-------+
| Field | Type          | Null | Key | Default | Extra |
+-------+---------------+------+-----+---------+-------+
| sex   | enum('F','M') |      |     | F       |       |
+-------+---------------+------+-----+---------+-------+
```

By the way, values in an `ENUM` column need not be just a single character. The `sex` column could have been defined as something like `ENUM('female','male')` instead.

`student_id` is an integer column that will contain unique student ID numbers. Normally, you'd probably get ID numbers for your students from a central source, such as the school office. We'll just make them up, using an

AUTO_INCREMENT column that is defined in much the same way as the member_id column that is part of the member table created earlier.

If you really were going to get student ID numbers from the office rather than generating them automatically, you would define the student_id column without the AUTO_INCREMENT attribute. But leave in the PRIMARY KEY clause, to disallow duplicate IDs.

Now, what about the ENGINE clause at the end of the CREATE TABLE statement? This clause, if present, names the storage engine that MySQL should use for creating the table. A "storage engine" is a handler that manages a certain kind of table. MySQL has several storage engines, each with its own characteristics.

If you omit the ENGINE clause, MySQL picks a default engine, which usually is MyISAM. "ISAM" is short for "indexed sequential access method," and the MyISAM engine is based on that access method with some MySQL-specific stuff added. Earlier, we provided no ENGINE clause when creating the Historical League tables (president and member), so they'll probably be MyISAM tables. For the grade-keeping project, we're explicitly using the InnoDB storage engine instead. InnoDB offers something called "referential integrity" through the use of foreign keys. That means we can use MySQL to enforce certain constraints on the interrelationships between tables. This is important for the grade-keeping project tables:

- Score records are tied to grade events and to students: We don't want to allow entry of records into the score table unless the student ID and grade event ID are known in the student and grade_event tables.
- Similarly, absence records are tied to students: We don't want to allow entry of records into the absence table unless the student ID is known in the student table.

To enforce these constraints, we'll set up foreign key relationships. "Foreign" means "in another table," and "foreign key" indicates a key value that must match a key value in that other table. These concepts will become clearer as we create the rest of the grade-keeping project tables.

The grade_event table definition is as follows:

```
CREATE TABLE grade_event
(
    date        DATE NOT NULL,
    category    ENUM('T','Q') NOT NULL,
    event_id    INT UNSIGNED NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (event_id)
) ENGINE = InnoDB;
```

To create the grade_event table, type that CREATE TABLE statement into mysql or execute the following command:

```
% mysql my_super_db < create_grade_event.sql
```

The date column holds a standard MySQL DATE value, in 'CCYY-MM-DD' (year-first) format.

category represents score category. Like sex in the student table, category is an enumeration column. The allowable values are 'T' and 'Q', representing "test" and "quiz."

event_id is an AUTO_INCREMENT column that is defined as a PRIMARY KEY, similar to student_id in the student table. Using AUTO_INCREMENT allows us to generate unique event ID values easily. As with the student_id column in the student table, the particular values are less important than that they be unique.

All the columns are defined as NOT NULL because none of them can be missing.

The score table looks like this:

```
CREATE TABLE score
(
    student_id  INT UNSIGNED NOT NULL,
    event_id    INT UNSIGNED NOT NULL,
    score       INT NOT NULL,
    PRIMARY KEY (event_id, student_id),
    INDEX (student_id),
    FOREIGN KEY (event_id) REFERENCES grade_event (event_id),
    FOREIGN KEY (student_id) REFERENCES student (student_id)
) ENGINE = InnoDB;
```

Here again the table definition contains something new: the FOREIGN KEY construct. We'll get to this in just a bit.

Create the table by typing the statement into mysql or by executing the following command:

```
% mysql my_super_db < create_score.sql
```

The score column is an INT to hold integer score values. If you wanted to allow scores such as 58.5 that have a fractional part, you'd use one of the data types that can represent them, such as FLOAT or DECIMAL.

The student_id and event_id columns are integer columns that indicate the student and event to which each score applies. By using them to link to the corresponding ID value columns in the student and grade_event tables, we'll be able to look up the student name and event date. There are several important points to note about the student_id and event_id columns:

- We've made the combination of the two columns a PRIMARY KEY. This ensures that we won't have duplicate scores for a student for a given quiz or test. Note that it's the combination of event_id and student_id that is unique. In the score table, neither value is unique by itself. There will be multiple score records for each event_id value (one per student), and multiple records for each student_id value (one for each quiz and test) taken by the student.
- For each ID column, a FOREIGN KEY clause defines a constraint. The REFERENCES part of the clause indicates which table and column the score column refers to. The constraint on event_id is that each value in the column must match some event_id value in the grade_event table. Similarly, each student_id value in the score table must match some student_id value in the student table.

The PRIMARY KEY definition ensures that we won't create duplicate score records. The FOREIGN KEY definitions ensure that we won't have records with bogus ID values that don't exist in the grade_event or student tables.

Why is there an index on student_id? The reason is that, for any columns in a FOREIGN KEY definition, there should be an index on them, or they should be the columns that are listed first in a multiple-column index. For the FOREIGN KEY on event_id, that column is listed first in the PRIMARY KEY. For the FOREIGN KEY on student_id, the PRIMARY KEY cannot be used because student_id is not listed first. So, instead, we create a separate index on student_id.

The absence table for recording lapses in attendance looks like this:

```
CREATE TABLE absence
(
    student_id  INT UNSIGNED NOT NULL,
    date        DATE NOT NULL,
    PRIMARY KEY (student_id, date),
    FOREIGN KEY (student_id) REFERENCES student (student_id)
) ENGINE = InnoDB;
```

Type that statement into `mysql` or execute the following command:

```
% mysql my_super_db < create_absence.sql
```

The `student_id` and `date` columns both are defined as `NOT NULL` to disallow missing values. We make the combination of the two columns a primary key so that we don't accidentally create duplicate records. It wouldn't be fair to count a student absent twice on the same day, would it?

The `absence` table also includes a foreign key relationship, defined to ensure that each `student_id` value matches a `student_id` value in the `student` table.

The inclusion of foreign key relationships in the grade-keeping tables is meant to enact constraints at data entry time: We want to insert only those records that contain legal grade event and student ID values. However, the foreign key relationships have another effect as well. They set up dependencies that constrain the order in which you create and drop tables:

- The `score` table refers to the `grade_event` and `student` tables, so they must be created first before you can create the `score` table. Similarly, `absence` refers to `student`, so `student` must exist before you can create `absence`.
- If you drop (remove) tables, the reverse is true. You cannot drop the `grade_event` table if you have not dropped the `score` table first, and `student` cannot be dropped unless you have first dropped `score` and `absence`.

> **Note**
>
> If for some reason your MySQL server does not include InnoDB support, you can create the grade-keeping project tables as MyISAM tables instead. Substitute `MyISAM` for `InnoDB` in each `CREATE TABLE` statement or just omit the `ENGINE` clause. However, if you use MyISAM tables, the demonstrations later in this book that show how foreign keys work using these tables will not work.

Adding New Records

At this point, our database and its tables have been created. Now we need to put some records into the tables. However, it's useful to know how to check what's in a table after you put something into it, so although retrieval is not covered in any detail until later in "Retrieving Information," you should know at least that the following statement will show you the complete contents of any table `tbl_name`:

```
SELECT * FROM tbl_name;
```

For example:

```
mysql> SELECT * FROM student;
Empty set (0.00 sec)
```

Right now, `mysql` indicates that the table is empty, but you'll see a different result after trying the examples in this section.

There are several ways to add data to a database. You can insert records into a table manually by issuing `INSERT` statements. You can also add records by reading them from a file, either in the form of prewritten

INSERT statements that you feed to mysql, or as raw data values that you load using the LOAD DATA statement or the mysqlimport client program.

Adding Records with INSERT

Let's start adding records by using INSERT, an SQL statement for which you specify the table into which you want to insert a row of data and the values to put in the row. The INSERT statement has several forms:

- You can specify values for all the columns:
- INSERT INTO tbl_name VALUES(value1,value2,...);

For example:

```
mysql> INSERT INTO student VALUES('Kyle','M',NULL);
mysql> INSERT INTO grade_event VALUES('2004-9-3','Q',NULL);
```

With this syntax, the VALUES list must contain a value for each column in the table, in the order that the columns are stored in the table. (Normally, this is the order in which the columns are specified in the table's CREATE TABLE statement.) If you're not sure what the column order is, issue a DESCRIBE tbl_name statement to find out.

You can quote string and date values in MySQL using either single or double quotes, but single quotes are more standard. The NULL values are for the AUTO_INCREMENT columns in the student and grade_event tables. Inserting a "missing value" into an AUTO_INCREMENT column causes MySQL to generate the next sequence number for the column.

MySQL allows you to insert several rows into a table with a single INSERT statement by specifying multiple value lists:

```
INSERT INTO tbl_name VALUES(...),(...),... ;
```

For example:

```
mysql> INSERT INTO student VALUES('Avery','F',NULL),('Nathan','M',NULL);
```

This involves less typing than multiple INSERT statements, and also is more efficient for the server to execute. Note that parentheses enclose the set of column values for each row. The following statement is illegal because it does not contain the correct number of values within parentheses:

```
mysql> INSERT INTO student VALUES('Avery','F',NULL,'Nathan','M',NULL);
ERROR 1136 (21S01): Column count doesn't match value count at row 1
```

- You can name the columns to which you want to assign values, and then list the values. This is useful when you want to create a record for which only a few columns need to be set up initially.
- INSERT INTO tbl_name (col_name1,col_name2,...) VALUES(value1,value2,...);

For example:

```
mysql> INSERT INTO member (last_name,first_name) VALUES('Stein','Waldo');
```

This form of `INSERT` allows multiple value lists, too:

```
mysql> INSERT INTO student (name,sex) VALUES('Abby','F'),('Joseph','M');
```

For any column not named in the column list, MySQL assigns its default value. For example, the preceding statements contain no values for the `member_id` or `student_id` columns, so MySQL assigns the default value of `NULL`. (`member_id` and `student_id` are `AUTO_INCREMENT` columns, so the net effect in each case is to generate and assign the next sequence number, just as if you had assigned `NULL` explicitly.)

- You can name columns and values in `col_name=value` form.
- `INSERT INTO tbl_name SET col_name1=value1, col_name2=value2, ... ;`

For example:

```
mysql> INSERT INTO member SET last_name='Stein',first_name='Waldo';
```

For any column not named in the `SET` clause, MySQL assigns its default value. This form of `INSERT` cannot be used to insert multiple rows with a single statement.

Now that you know how `INSERT` works, you can use it to see whether the foreign key relationships we set up really prevent entry of bad records in the `score` and `absence` tables. Try inserting records that contain ID values that are not present in the `grade_event` or `student` tables:

```
mysql> INSERT INTO score (event_id,student_id,score) VALUES(9999,9999,0);
ERROR 1216 (23000): Cannot add or update a child row: a foreign key
constraint fails
mysql> INSERT INTO absence SET student_id=9999, date='2004-09-16';
ERROR 1216 (23000): Cannot add or update a child row: a foreign key
constraint fails
```

It appears that the constraints are working.

Adding New Records from a File

Another method for loading records into a table is to read them directly from a file. The file can contain `INSERT` statements or it can contain raw data. For example, the `my_super_db` distribution contains a file named `insert_president.sql` that contains `INSERT` statements for adding new records to the `president` table. Assuming that you are in the same directory where that file is located, you can execute those statements like this:

```
% mysql my_super_db < insert_president.sql
```

If you're already running mysql, you can use a source command to read the file:

```
mysql> source insert_president.sql;
```

If you have the records stored in a file as raw data values rather than as INSERT statements, you can load them with the LOAD DATA statement or with the mysqlimport client program.

The LOAD DATA statement acts as a bulk loader that reads data from a file. Use it from within mysql:

```
mysql> LOAD DATA LOCAL INFILE 'member.txt' INTO TABLE member;
```

Assuming that the member.txt data file is located in your current directory on the client host, this statement reads it and sends its contents to the server to be loaded into the member table. (The member.txt file can be found in the my_super_db distribution.)

By default, the LOAD DATA statement assumes that column values are separated by tabs and that lines end with newlines (also known as "linefeeds"). It also assumes that the values are present in the order that columns are stored in the table. It's possible to read files in other formats or to specify a different column order.

The keyword LOCAL in the LOAD DATA statement causes the data file to be read by the client program (in this case mysql) and sent to the server to be loaded. It is possible to omit LOCAL, but then the file must be located on the server host and you need the FILE server access privilege that most MySQL users don't have. You should also specify the full pathname to the file so that the server can find it.

If you get the following error with LOAD DATA LOCAL, the LOCAL capability might be disabled by default:

```
ERROR 1148 (42000): The used command is not allowed with this MySQL version
```

Try again after invoking mysql with the --local-infile option. For example:

```
% mysql --local-infile my_super_db
mysql> LOAD DATA LOCAL INFILE 'member.txt' INTO TABLE member;
```

If that doesn't work, either, the server also needs to be told to allow LOCAL.

Another way to load a data file is to use the mysqlimport client program. You invoke mysqlimport from the command prompt, and it generates a LOAD DATA statement for you:

```
% mysqlimport --local my_super_db member.txt
```

As with the mysql program, if you need to specify connection parameters, indicate them on the command line preceding the database name.

For the command just shown, mysqlimport generates a LOAD DATA statement to load member.txt into the member table. That's because mysqlimport determines the table name from the name of the data file, using everything up to the first period of the filename as the table name. For example, mysqlimport would load files named member.txt and president.txt into the member and president tables. This means you should choose your filenames carefully or mysqlimport won't use the correct table name. If you wanted to load files named member1.txt and member2.txt, mysqlimport would think it should load them into tables named member1 and

member2. If what you really want is to load both files into the member table, you could use names like member.1.txt and member.2.txt, or member.txt1 and member.txt2.

Resetting the my_super_db Database to a Known State

After you have tried the record-adding methods just described in the preceding discussion, you should re-create and load the my_super_db database tables to reset the database so that its contents are the same as what the next sections assume. Using the mysql program in the directory containing the my_super_db distribution files, issue these statements:

```
% mysql my_super_db
mysql> source create_member.sql;
mysql> source create_president.sql;
mysql> source insert_member.sql;
mysql> source insert_president.sql;
mysql> DROP TABLE IF EXISTS absence, score, grade_event, student;
mysql> source create_student.sql;
mysql> source create_grade_event.sql;
mysql> source create_score.sql;
mysql> source create_absence.sql;
mysql> source insert_student.sql;
mysql> source insert_grade_event.sql;
mysql> source insert_score.sql;
mysql> source insert_absence.sql;
```

If you don't want to type those statements individually (which is not unlikely), try this command on Unix:

```
% sh init_all_tables.sh my_super_db
```

On Windows, use this command instead:

```
C:\> init_all_tables.bat my_super_db
```

Whichever command you use, if you need to specify connection parameters, list them on the command line before the database name.

Retrieving Information

Our tables have been created and loaded with data now, so let's see what we can do with that data. To retrieve and display information from your tables, use the SELECT statement. It allows you to retrieve information in as general or specific a manner as you like. You can display the entire contents of a table:

```
SELECT * FROM president;
```

Or you can select as little as a single column of a single row:

```
SELECT birth FROM president WHERE last_name = 'Eisenhower';
```

The SELECT statement has several clauses that you combine as necessary to retrieve the information in which you're interested. Each of these clauses can be simple or complex, so SELECT statements as a whole can be simple or complex. However, rest assured that you won't find any page-long queries that take an hour to figure out in this book. (When I see arm-length queries in something that I'm reading, I generally skip right over them, and I'm guessing that you do the same.)

A simplified syntax of the SELECT statement is:

```
SELECT what to retrieve
FROM table or tables
WHERE conditions that data must satisfy;
```

To write a SELECT statement, specify what you want to retrieve and then some optional clauses. The clauses just shown (FROM and WHERE) are the most common ones, although others can be specified as well, such as GROUP BY, ORDER BY, and LIMIT. Remember that SQL is a free-format language, so when you write your own SELECT statements, you need not put line breaks in the same places I do.

The FROM clause is usually present, but it need not be if you don't need to name any tables. For example, the following query simply displays the values of some expressions. These can be calculated without referring to the contents of any table, so no FROM clause is necessary:

```
mysql> SELECT 2+2, 'Hello, world', VERSION();
+-----+--------------+-----------+
| 2+2 | Hello, world | VERSION() |
+-----+--------------+-----------+
|   4 | Hello, world | 4.1.9-log |
+-----+--------------+-----------+
```

When you do use a FROM clause to specify a table from which to retrieve data, you'll also indicate which columns you want to see. The most "generic" form of SELECT uses * as a column specifier, which is shorthand for "all columns." The following query retrieves all columns from the student table and displays them:

```
mysql> SELECT * FROM student;
+-----------+-----+------------+
| name      | sex | student_id |
+-----------+-----+------------+
| Megan     | F   |          1 |
| Joseph    | M   |          2 |
| Kyle      | M   |          3 |
| Katie     | F   |          4 |
...
```

The columns are displayed in the order that MySQL stores them in the table. This is the same order in which the columns are listed when you issue a DESCRIBE student statement. (The "…" shown at the end of the example indicates that the query returns more rows than are shown.)

You can explicitly name the column or columns you want to see. To select just student names, do this:

```
mysql> SELECT name FROM student;
+-----------+
| name      |
+-----------+
| Megan     |
| Joseph    |
| Kyle      |
| Katie     |
...
```

If you name more than one column, separate them by commas. The following statement is equivalent to SELECT * FROM student, but names each column explicitly:

```
mysql> SELECT name, sex, student_id FROM student;
```

```
+-----------+-----+------------+
| name      | sex | student_id |
+-----------+-----+------------+
| Megan     | F   |          1 |
| Joseph    | M   |          2 |
| Kyle      | M   |          3 |
| Katie     | F   |          4 |
...
```

You can name columns in any order:

```
SELECT name, student_id FROM student;
SELECT student_id, name FROM student;
```

You can even name a column more than once if you like, although generally that's kind of pointless.

It's also possible to select columns from more than one table at a time. This is called a "join" between tables. We'll get to joins in "Retrieving Information from Multiple Tables."

Column names are not case sensitive in MySQL, so the following queries are equivalent:

```
SELECT name, student_id FROM student;
SELECT NAME, STUDENT_ID FROM student;
SELECT nAmE, sTuDeNt_Id FROM student;
```

On the other hand, database and table names might be case sensitive. It depends on the filesystem used on the server host and on how MySQL is configured. Windows filenames are not case sensitive, so a server running on Windows does not treat database and table names as case sensitive. On Unix systems, filenames generally are case sensitive, so a server would treat database and table names as case sensitive. An exception to this occurs under Mac OS X, which offers both HFS+ and UFS filesystems: HFS+ is the default, and it uses case-sensitive filenames.

Specifying Retrieval Criteria

To restrict the set of records retrieved by the `SELECT` statement, use a `WHERE` clause that specifies criteria for selecting rows. You can select rows by looking for column values that satisfy various criteria, and you can look for different types of values. For example, you can search for certain numeric values:

```
mysql> SELECT * FROM score WHERE score > 95;
+------------+----------+-------+
| student_id | event_id | score |
+------------+----------+-------+
|          5 |        3 |    97 |
|         18 |        3 |    96 |
|          1 |        6 |   100 |
|          5 |        6 |    97 |
|         11 |        6 |    98 |
|         16 |        6 |    98 |
+------------+----------+-------+
```

Or you can look for string values containing character data. String comparisons normally are not case sensitive:

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name='ROOSEVELT';
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
```

```
| Roosevelt | Theodore    |
| Roosevelt | Franklin D. |
+-----------+-------------+
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name='roosevelt';
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Roosevelt | Theodore    |
| Roosevelt | Franklin D. |
+-----------+-------------+
```

Or you can look for dates:

```
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth < '1750-1-1';
+------------+------------+------------+
| last_name  | first_name | birth      |
+------------+------------+------------+
| Washington | George     | 1732-02-22 |
| Adams      | John       | 1735-10-30 |
| Jefferson  | Thomas     | 1743-04-13 |
+------------+------------+------------+
```

It's also possible to search for combinations of values:

```
mysql> SELECT last_name, first_name, birth, state FROM president
    -> WHERE birth < '1750-1-1' AND (state='VA' OR state='MA');
+------------+------------+------------+-------+
| last_name  | first_name | birth      | state |
+------------+------------+------------+-------+
| Washington | George     | 1732-02-22 | VA    |
| Adams      | John       | 1735-10-30 | MA    |
| Jefferson  | Thomas     | 1743-04-13 | VA    |
+------------+------------+------------+-------+
```

Expressions in `WHERE` clauses can use arithmetic operators, comparison operators, and logical operators. You can also use parentheses to group parts of an expression. Operations can be performed using constants, table columns, and function calls.

Arithmetic Operators

Operator	Meaning
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`DIV`	Integer division
`%`	Modulo (remainder after division)

. Comparison Operators

Operator	Meaning
`<`	Less than

. Comparison Operators

Operator	Meaning
`<=`	Less than or equal to
`=`	Equal to
`<=>`	Equal to (works even for `NULL` values)
`<>` or `!=`	Not equal to
`>=`	Greater than or equal to
`>`	Greater than

Logical Operators

Operator	Meaning
`AND`	Logical AND
`OR`	Logical OR
`XOR`	Logical exclusive-OR
`NOT`	Logical negation

When you're formulating a statement that requires logical operators, take care not to confuse the meaning of the logical `AND` operator with the way we use "and" in everyday speech. Suppose that you want to find "presidents born in Virginia and presidents born in Massachusetts." That condition is phrased using "and," which seems to imply that you'd write the statement as follows:

```
mysql> SELECT last_name, first_name, state FROM president
    -> WHERE state='VA' AND state='MA';
Empty set (0.36 sec)
```

It's clear from the empty result that the statement doesn't work. Why not? Because what the statement really means is "Select presidents who were born both in Virginia and in Massachusetts," which makes no sense. In English, you might express the statement using "and," but in SQL, you connect the two conditions with `OR`:

```
mysql> SELECT last_name, first_name, state FROM president
    -> WHERE state='VA' OR state='MA';
+------------+-------------+-------+
| last_name  | first_name  | state |
+------------+-------------+-------+
| Washington | George      | VA    |
| Adams      | John        | MA    |
| Jefferson  | Thomas      | VA    |
| Madison    | James       | VA    |
| Monroe     | James       | VA    |
| Adams      | John Quincy | MA    |
| Harrison   | William H.  | VA    |
| Tyler      | John        | VA    |
| Taylor     | Zachary     | VA    |
| Wilson     | Woodrow     | VA    |
| Kennedy    | John F      | MA    |
| Bush       | George H.W. | MA    |
+------------+-------------+-------+
```

This disjunction between natural language and SQL is something to be aware of, not just when formulating your own queries, but also when you write queries for other people. It's best to listen carefully as they describe what they want to retrieve, but you don't necessarily want to transcribe their descriptions into SQL using the same logical operators. For the example just described, the proper English equivalent for the query is "Select presidents who were born either in Virginia or in Massachusetts."

You might find it easier to use the `IN()` operator when formulating queries like this, where you're looking for any of several values. The preceding query can be rewritten using `IN()` like this:

```
SELECT last_name, first_name, state FROM president
WHERE state IN('VA','MA');
```

`IN()` is especially convenient when you're comparing a column to a large number of values.

The `NULL` Value

The `NULL` value is special. It means "no value," so you can't compare it to known values the way you can compare two known values to each other. If you attempt to use `NULL` with the usual arithmetic comparison operators, the result is undefined:

```
mysql> SELECT NULL < 0, NULL = 0, NULL <> 0, NULL > 0;
+----------+----------+-----------+----------+
| NULL < 0 | NULL = 0 | NULL <> 0 | NULL > 0 |
+----------+----------+-----------+----------+
|     NULL |     NULL |      NULL |     NULL |
+----------+----------+-----------+----------+
```

In fact, you can't even compare `NULL` to itself because the result of comparing two unknown values cannot be determined:

```
mysql> SELECT NULL = NULL, NULL <> NULL;
+-------------+--------------+
| NULL = NULL | NULL <> NULL |
+-------------+--------------+
|        NULL |         NULL |
+-------------+--------------+
```

To perform searches for `NULL` values, you must use a special syntax. Instead of using `=`, `<>`, or `!=` to test for equality or inequality, use `IS NULL` or `IS NOT NULL`. For example, presidents who are still living have their death dates represented as `NULL` in the `president` table. To find them, use the following query:

```
mysql> SELECT last_name, first_name FROM president WHERE death IS NULL;
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Ford      | Gerald R    |
| Carter    | James E.    |
| Bush      | George H.W. |
| Clinton   | William J.  |
| Bush      | George W.   |
+-----------+-------------+
```

To find non-`NULL` values, use `IS NOT NULL`. This query finds names that have a suffix part:

```
mysql> SELECT last_name, first_name, suffix
    -> FROM president WHERE suffix IS NOT NULL;
```

```
+-----------+------------+--------+
| last_name | first_name | suffix |
+-----------+------------+--------+
| Carter    | James E.   | Jr.    |
+-----------+------------+--------+
```

The MySQL-specific <=> comparison operator is true even for NULL-to-NULL comparisons. The preceding two queries can be rewritten to use this operator as follows:

```
SELECT last_name, first_name FROM president WHERE death <=> NULL;

SELECT last_name, first_name, suffix
FROM president WHERE NOT (suffix <=> NULL);
```

Sorting Query Results

Every MySQL user notices sooner or later that if you create a table, load some records into it, and then issue a SELECT * FROM tbl_name statement, the records tend to be retrieved in the same order in which they were inserted. That makes a certain intuitive sense, so it's natural to assume that records are retrieved in insertion order by default. But that is not the case. If you delete and insert rows after loading the table initially, those actions likely will change the order in which the server returns the table's rows. (Deleting records puts "holes" in the table, which MySQL tries to fill later when you insert new records.)

What you should remember about record retrieval order is this: There is no guarantee about the order in which the server returns rows, unless you specify that order yourself. To do so, add an ORDER BY clause to the statement that defines the sort order you want. The following query returns president names, sorted lexically (alphabetically) by last name:

```
mysql> SELECT last_name, first_name FROM president
    -> ORDER BY last_name;
+------------+---------------+
| last_name  | first_name    |
+------------+---------------+
| Adams      | John          |
| Adams      | John Quincy   |
| Arthur     | Chester A.    |
| Buchanan   | James         |
...
```

Ascending order is the default sort order in an ORDER BY clause. You can specify explicitly whether to sort a column in ascending or descending order by using the ASC or DESC keywords after column names in the ORDER BY clause. For example, to sort president names in reverse (descending) name order, use DESC like this:

```
mysql> SELECT last_name, first_name FROM president
    -> ORDER BY last_name DESC;
+------------+---------------+
| last_name  | first_name    |
+------------+---------------+
| Wilson     | Woodrow       |
| Washington | George        |
| Van Buren  | Martin        |
| Tyler      | John          |
...
```

You can sort using multiple columns, and each column can be sorted independently in ascending or descending order. The following query retrieves rows from the president table, sorts them by reverse state of birth, and by ascending last name within each state:

```
mysql> SELECT last_name, first_name, state FROM president
    -> ORDER BY state DESC, last_name ASC;
+------------+--------------+-------+
| last_name  | first_name   | state |
+------------+--------------+-------+
| Arthur     | Chester A.   | VT    |
| Coolidge   | Calvin       | VT    |
| Harrison   | William H.   | VA    |
| Jefferson  | Thomas       | VA    |
| Madison    | James        | VA    |
| Monroe     | James        | VA    |
| Taylor     | Zachary      | VA    |
| Tyler      | John         | VA    |
| Washington | George       | VA    |
| Wilson     | Woodrow      | VA    |
| Eisenhower | Dwight D.    | TX    |
| Johnson    | Lyndon B.    | TX    |
...
```

`NULL` values in a column sort at the beginning for ascending sorts and at the end for descending sorts. If you want to ensure that `NULL` values will appear at a given end of the sort order, add an extra sort column that distinguishes `NULL` from non-`NULL` values. For example, if you sort presidents by reverse death date, living presidents (those with `NULL` death dates) will appear at the end of the sort order. To put them at the beginning instead, use this query:

```
mysql> SELECT last_name, first_name, death FROM president
    -> ORDER BY IF(death IS NULL,0,1), death DESC;
+------------+--------------+------------+
| last_name  | first_name   | death      |
+------------+--------------+------------+
| Clinton    | William J.   | NULL       |
| Bush       | George H.W.  | NULL       |
| Carter     | James E.     | NULL       |
| Ford       | Gerald R     | NULL       |
| Bush       | George W.    | NULL       |
| Reagan     | Ronald W.    | 2004-06-05 |
| Nixon      | Richard M    | 1994-04-22 |
| Johnson    | Lyndon B.    | 1973-01-22 |
...
| Jefferson  | Thomas       | 1826-07-04 |
| Adams      | John         | 1826-07-04 |
| Washington | George       | 1799-12-14 |
+------------+--------------+------------+
```

The `IF()` function evaluates the expression given by its first argument and returns the value of its second or third argument, depending on whether the expression is true or false. For the query shown, `IF()` evaluates to 0 for `NULL` values and 1 for non-`NULL` values. This places all `NULL` values ahead of all non-`NULL` values.

Limiting Query Results

When a query returns many rows, but you want to see only a few of them, add a `LIMIT` clause. `LIMIT` is especially useful in conjunction with `ORDER BY`. MySQL allows you to limit the output of a query to the first `n` rows of the result that would otherwise be returned. The following query selects the five presidents who were born first:

```
mysql> SELECT last_name, first_name, birth FROM president
    -> ORDER BY birth LIMIT 5;
+------------+------------+------------+
| last_name  | first_name | birth      |
+------------+------------+------------+
| Washington | George     | 1732-02-22 |
```

```
| Adams      | John       | 1735-10-30 |
| Jefferson  | Thomas     | 1743-04-13 |
| Madison    | James      | 1751-03-16 |
| Monroe     | James      | 1758-04-28 |
+------------+------------+------------+
```

If you sort in reverse order, using `ORDER BY` birth `DESC`, you get the five most recently born presidents instead:

```
mysql> SELECT last_name, first_name, birth FROM president
    -> ORDER BY birth DESC LIMIT 5;
+-----------+-------------+------------+
| last_name | first_name  | birth      |
+-----------+-------------+------------+
| Clinton   | William J.  | 1946-08-19 |
| Bush      | George W.   | 1946-07-06 |
| Carter    | James E.    | 1924-10-01 |
| Bush      | George H.W. | 1924-06-12 |
| Kennedy   | John F      | 1917-05-29 |
+-----------+-------------+------------+
```

`LIMIT` also allows you to pull a section of records out of the middle of a result set. To do this, you must specify two values. The first value is the number of records to skip at the beginning of the result set, and the second is the number of records to return. The following query is similar to the previous one but returns 5 records after skipping the first 10:

```
mysql> SELECT last_name, first_name, birth FROM president
    -> ORDER BY birth DESC LIMIT 10, 5;
+-----------+-------------+------------+
| last_name | first_name  | birth      |
+-----------+-------------+------------+
| Truman    | Harry S     | 1884-05-08 |
| Roosevelt | Franklin D. | 1882-01-30 |
| Hoover    | Herbert C.  | 1874-08-10 |
| Coolidge  | Calvin      | 1872-07-04 |
| Harding   | Warren G.   | 1865-11-02 |
+-----------+-------------+------------+
```

To pull a randomly selected record or records from a table, use `ORDER BY RAND()` in conjunction with `LIMIT`:

```
mysql> SELECT last_name, first_name FROM president
    -> ORDER BY RAND() LIMIT 1;
+-----------+------------+
| last_name | first_name |
+-----------+------------+
| Johnson   | Lyndon B.  |
+-----------+------------+
mysql> SELECT last_name, first_name FROM president
    -> ORDER BY RAND() LIMIT 3;
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Harding   | Warren G.   |
| Bush      | George H.W. |
| Jefferson | Thomas      |
+-----------+-------------+
```

Calculating and Naming Output Column Values

Most of the queries shown so far produce output by retrieving values from tables. MySQL also allows you to calculate output values from the results of expressions. Expressions can be simple or complex. The following query evaluates a simple expression (a constant) and a more complex expression involving several arithmetic operations and a couple of function calls:

```
mysql> SELECT 17, FORMAT(SQRT(3*3+4*4),0);
+----+------------------------+
| 17 | FORMAT(SQRT(3*3+4*4),0) |
+----+------------------------+
| 17 | 5                      |
+----+------------------------+
```

Expressions also can refer to table columns:

```
mysql> SELECT CONCAT(first_name,' ',last_name),CONCAT(city,', ',state)
    -> FROM president;
+---------------------------------+-------------------------+
| CONCAT(first_name,' ',last_name) | CONCAT(city,', ',state) |
+---------------------------------+-------------------------+
| George Washington                | Wakefield, VA           |
| John Adams                       | Braintree, MA           |
| Thomas Jefferson                 | Albemarle County, VA    |
| James Madison                    | Port Conway, VA         |
...
```

That query formats president names as a single string by concatenating first and last names separated by a space. It also formats birthplaces as the birth cities and states separated by a comma and a space.

When you use an expression to calculate a column value, the expression becomes the column's name and is used for its heading. That can lead to a very wide column if the expression is long, as the preceding query illustrates. To deal with this, you can assign the column a different name using the `AS name` construct. Such names are called "column aliases." The output from the previous query can be made more meaningful like this:

```
mysql> SELECT CONCAT(first_name,' ',last_name) AS Name,
    -> CONCAT(city,', ',state) AS Birthplace
    -> FROM president;
+----------------------+-------------------------+
| Name                 | Birthplace              |
+----------------------+-------------------------+
| George Washington    | Wakefield, VA           |
| John Adams           | Braintree, MA           |
| Thomas Jefferson     | Albemarle County, VA    |
| James Madison        | Port Conway, VA         |
...
```

If the column alias contains spaces, you'll need to put it in quotes:

```
mysql> SELECT CONCAT(first_name,' ',last_name) AS 'President Name',
    -> CONCAT(city,', ',state) AS 'Place of Birth'
    -> FROM president;
+----------------------+-------------------------+
| President Name       | Place of Birth          |
+----------------------+-------------------------+
| George Washington    | Wakefield, VA           |
| John Adams           | Braintree, MA           |
| Thomas Jefferson     | Albemarle County, VA    |
| James Madison        | Port Conway, VA         |
...
```

The keyword AS is optional when you provide a column alias:

```
mysql> SELECT 1, 1 AS one, 1 one;
+---+-----+-----+
| 1 | one | one |
+---+-----+-----+
| 1 |   1 |   1 |
+---+-----+-----+
```

I prefer to include the AS. Without it, it's easier to inadvertently write a query that is legal but does not produce the intended result. For example, you might write a query to select president names like this, forgetting the comma between the first_name and last_name columns:

```
mysql> SELECT first_name last_name FROM president;
+---------------+
| last_name     |
+---------------+
| George        |
| John          |
| Thomas        |
| James         |
...
```

As a result, the query does not display two columns. Instead, it displays only the first_name column and treats last_name as the column alias, which becomes its label. If a query does not retrieve the number of columns you expect and uses column names other than you expect, be on the lookout for a missing comma somewhere between columns.

Working with Dates

The principal thing to keep in mind when using dates in MySQL is that it always expects dates with the year first. To write July 27, 2005, use '2005-07-27'. Do not use '07-27-2005' or '27-07-2005', as you might be more accustomed to writing.

You can perform many kinds of operations on dates:

- Sort by date. (We've seen this several times already.)
- Look for particular dates or a range of dates.
- Extract parts of a date value, such as the year, month, or day.
- Calculate the difference between dates.
- Compute a date by adding an interval to or subtracting an interval from another date.

Some examples of these operations follow.

To look for particular dates, either by exact value or in relation to another value, compare a DATE column to the value in which you're interested:

```
mysql> SELECT * FROM grade_event WHERE date = '2004-10-01';
+------------+----------+----------+
| date       | category | event_id |
+------------+----------+----------+
| 2004-10-01 | T        |        6 |
+------------+----------+----------+
mysql> SELECT last_name, first_name, death
    -> FROM president
    -> WHERE death >= '1970-01-01' AND death < '1980-01-01';
+-----------+------------+------------+
| last_name | first_name | death      |
```

```
+----------+-----------+------------+
| Truman   | Harry S   | 1972-12-26 |
| Johnson  | Lyndon B. | 1973-01-22 |
+----------+-----------+------------+
```

To test or retrieve parts of dates, use functions such as `YEAR()`, `MONTH()`, or `DAYOFMONTH()`. For example, to find presidents who were born in March, look for dates with a month value of 3:

```
mysql> SELECT last_name, first_name, birth
    -> FROM president WHERE MONTH(birth) = 3;
+-----------+------------+------------+
| last_name | first_name | birth      |
+-----------+------------+------------+
| Madison   | James      | 1751-03-16 |
| Jackson   | Andrew     | 1767-03-15 |
| Tyler     | John       | 1790-03-29 |
| Cleveland | Grover     | 1837-03-18 |
+-----------+------------+------------+
```

The query also can be written in terms of the month name:

```
mysql> SELECT last_name, first_name, birth
    -> FROM president WHERE MONTHNAME(birth) = 'March';
+-----------+------------+------------+
| last_name | first_name | birth      |
+-----------+------------+------------+
| Madison   | James      | 1751-03-16 |
| Jackson   | Andrew     | 1767-03-15 |
| Tyler     | John       | 1790-03-29 |
| Cleveland | Grover     | 1837-03-18 |
+-----------+------------+------------+
```

To be more specific, you can combine tests for `MONTH()` and `DAYOFMONTH()` to find presidents born on a particular day in March:

```
mysql> SELECT last_name, first_name, birth
    -> FROM president WHERE MONTH(birth) = 3 AND DAYOFMONTH(birth) = 29;
+-----------+------------+------------+
| last_name | first_name | birth      |
+-----------+------------+------------+
| Tyler     | John       | 1790-03-29 |
+-----------+------------+------------+
```

This is the kind of query you'd use for generating one of those "these celebrities have birthdays today" lists such as you see in the Entertainment section of your newspaper. However, if you want to select records that match month and day for "the current date," you don't have to plug in literal values the way the previous query did. To check for presidents born today, no matter what day of the year today is, compare their birthdays to the month and day parts of `CURDATE()`, which always returns the current date:

```
SELECT last_name, first_name, birth
FROM president WHERE MONTH(birth) = MONTH(CURDATE())
AND DAYOFMONTH(birth) = DAYOFMONTH(CURDATE());
```

You can subtract one date from another, which allows you to find the interval between dates. For example, to determine which presidents lived the longest, subtract the birth date from the death date. To do this, convert birth and death to days using the `TO_DAYS()` function and take the difference:

```
mysql> SELECT last_name, first_name, birth, death,
    -> TO_DAYS(death) - TO_DAYS(birth) AS age
    -> FROM president WHERE death IS NOT NULL
    -> ORDER BY age DESC LIMIT 5;
+-----------+------------+------------+------------+-------+
| last_name | first_name | birth      | death      | age   |
+-----------+------------+------------+------------+-------+
| Reagan    | Ronald W.  | 1911-02-06 | 2004-06-05 | 34088 |
| Adams     | John       | 1735-10-30 | 1826-07-04 | 33119 |
| Hoover    | Herbert C. | 1874-08-10 | 1964-10-20 | 32943 |
| Truman    | Harry S    | 1884-05-08 | 1972-12-26 | 32373 |
| Madison   | James      | 1751-03-16 | 1836-06-28 | 31150 |
+-----------+------------+------------+------------+-------+
```

To convert age in days to approximate age in years, divide by 365 (the `FLOOR()` function used here chops off any fractional part from the result to produce an integer):

```
mysql> SELECT last_name, first_name, birth, death,
    -> FLOOR((TO_DAYS(death) - TO_DAYS(birth))/365) AS age
    -> FROM president WHERE death IS NOT NULL
    -> ORDER BY age DESC LIMIT 5;
+-----------+------------+------------+------------+------+
| last_name | first_name | birth      | death      | age  |
+-----------+------------+------------+------------+------+
| Reagan    | Ronald W.  | 1911-02-06 | 2004-06-05 |   93 |
| Adams     | John       | 1735-10-30 | 1826-07-04 |   90 |
| Hoover    | Herbert C. | 1874-08-10 | 1964-10-20 |   90 |
| Truman    | Harry S    | 1884-05-08 | 1972-12-26 |   88 |
| Madison   | James      | 1751-03-16 | 1836-06-28 |   85 |
+-----------+------------+------------+------------+------+
```

In this particular case, the age values happen to correspond to true age at death. But the formula used in the query might not always do so, because years are not always exactly 365 days long. To calculate ages as we normally think of them, take the difference between the year parts of the dates, and then subtract one if the calendar day of the death date occurs earlier than that of the birth date:

```
mysql> SELECT last_name, first_name, birth, death,
    -> (YEAR(death) - YEAR(birth)) - IF(RIGHT(death,5) < RIGHT(birth,5),1,0)
    -> AS age
    -> FROM president WHERE death IS NOT NULL
    -> ORDER BY age DESC LIMIT 5;
+-----------+------------+------------+------------+------+
| last_name | first_name | birth      | death      | age  |
+-----------+------------+------------+------------+------+
| Reagan    | Ronald W.  | 1911-02-06 | 2004-06-05 |   93 |
| Adams     | John       | 1735-10-30 | 1826-07-04 |   90 |
| Hoover    | Herbert C. | 1874-08-10 | 1964-10-20 |   90 |
| Truman    | Harry S    | 1884-05-08 | 1972-12-26 |   88 |
| Madison   | James      | 1751-03-16 | 1836-06-28 |   85 |
+-----------+------------+------------+------------+------+
```

The `IF()` expression used here performs the calendar day test based on a simple substring comparison of the last five characters of the dates. This works for two reasons. First, MySQL treats dates as strings if you pass them to a string functionin this case, `RIGHT()`, which returns the rightmost `n` characters of a string. Second, MySQL produces dates with a fixed number of digits in each of their subparts. The comparison would not work if leading zeros were not present for month and day values less than ten.

Taking a difference between dates also is useful for determining how far dates are from some reference date. For example, that's how you can tell which Historical League members need to renew their memberships soon. Compute the difference between each member's expiration date and the current date, and if it's less than some

threshold value, a renewal will soon be needed. The following query finds memberships that have already expired or that will be due for renewal within 60 days:

```
SELECT last_name, first_name, expiration FROM member
WHERE (TO_DAYS(expiration) - TO_DAYS(CURDATE())) < 60;
```

To calculate one date from another, you can use `DATE_ADD()` or `DATE_SUB()`. These functions take a date and an interval and produce a new date. For example:

```
mysql> SELECT DATE_ADD('1970-1-1', INTERVAL 10 YEAR);
+---------------------------------------+
| DATE_ADD('1970-1-1', INTERVAL 10 YEAR) |
+---------------------------------------+
| 1980-01-01                            |
+---------------------------------------+
mysql> SELECT DATE_SUB('1970-1-1', INTERVAL 10 YEAR);
+---------------------------------------+
| DATE_SUB('1970-1-1', INTERVAL 10 YEAR) |
+---------------------------------------+
| 1960-01-01                            |
+---------------------------------------+
```

A query shown earlier in this section selected presidents who died during the 1970s, using literal dates for the endpoints of the selection range. That query can be rewritten to use a literal starting date and an ending date calculated from the starting date and an interval:

```
mysql> SELECT last_name, first_name, death
    -> FROM president
    -> WHERE death >= '1970-1-1'
    -> AND death < DATE_ADD('1970-1-1', INTERVAL 10 YEAR);
+-----------+------------+------------+
| last_name | first_name | death      |
+-----------+------------+------------+
| Truman    | Harry S    | 1972-12-26 |
| Johnson   | Lyndon B.  | 1973-01-22 |
+-----------+------------+------------+
```

The membership-renewal query can be written in terms of `DATE_ADD()`:

```
SELECT last_name, first_name, expiration FROM member
WHERE expiration < DATE_ADD(CURDATE(), INTERVAL 60 DAY);
```

Near the beginning of this chapter, you saw the following query for determining which of a dentist's patients haven't come in for their checkup in a while:

```
SELECT last_name, first_name, last_visit FROM patient
WHERE last_visit < DATE_SUB(CURDATE(), INTERVAL 6 MONTH);
```

That query may not have meant much to you then. Is it more meaningful now?

Pattern Matching

MySQL supports pattern matching operations that allow you to select records without supplying an exact comparison value. To perform a pattern match, you use special operators (`LIKE` and `NOT LIKE`), and you specify a string containing wildcard characters. The character '_' matches any single character, and '%' matches any

sequence of characters (including an empty sequence). Pattern matches using `LIKE` or `NOT LIKE` are not case sensitive.

This pattern matches last names that begin with a 'W' or 'w' character:

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name LIKE 'W%';
+------------+------------+
| last_name  | first_name |
+------------+------------+
| Washington | George     |
| Wilson     | Woodrow    |
+------------+------------+
```

The following query demonstrates a common error. The pattern match is erroneous because it does not use `LIKE`, it uses a pattern with an arithmetic comparison operator:

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name = 'W%';
Empty set (0.00 sec)
```

The only way for such a comparison to succeed is for the column to contain exactly the string `'W%'` or `'w%'`.

This pattern matches last names that contain 'W' or 'w' anywhere in the name, not just at the beginning:

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name LIKE '%W%';
+------------+------------+
| last_name  | first_name |
+------------+------------+
| Washington | George     |
| Wilson     | Woodrow    |
| Eisenhower | Dwight D.  |
+------------+------------+
```

This pattern matches last names that contain exactly four characters:

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name LIKE '____';
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Polk      | James K.    |
| Taft      | William H.  |
| Ford      | Gerald R    |
| Bush      | George H.W. |
| Bush      | George W.   |
+-----------+-------------+
```

Setting and Using User-Defined Variables

MySQL allows you to define your own variables. These can be set using query results, which provides a convenient way to save values for use in later queries. Suppose that you want to find out which presidents were born before Andrew Jackson. To determine that, you can retrieve his birth date into a variable and then select other presidents with a birth date earlier than the value of the variable

(This problem could be solved in a single query using a join or a subquery, but we're not to the point of writing those yet. Besides, sometimes it's just easier to use a variable.)

```
mysql> SELECT @birth := birth FROM president
    -> WHERE last_name = 'Jackson' AND first_name = 'Andrew';
+-----------------+
| @birth := birth |
+-----------------+
| 1767-03-15      |
+-----------------+
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth < @birth ORDER BY birth;
+------------+------------+------------+
| last_name  | first_name | birth      |
+------------+------------+------------+
| Washington | George     | 1732-02-22 |
| Adams      | John       | 1735-10-30 |
| Jefferson  | Thomas     | 1743-04-13 |
| Madison    | James      | 1751-03-16 |
| Monroe     | James      | 1758-04-28 |
+------------+------------+------------+
```

User variables are named using `@var_name` syntax and assigned a value in a `SELECT` statement using an expression of the form `@var_name := value`. The first query therefore looks up the birth date for Andrew Jackson and assigns it to the `@birth` variable. (The result of the `SELECT` still is displayed; assigning a query result to a variable doesn't suppress the query output.) The second query refers to the variable and uses its value to find other `president` records with a lesser `birth` value.

Variables also can be assigned using a `SET` statement. In this case, either `=` or `:=` are allowable as the assignment operator:

```
mysql> SET @today = CURDATE();
mysql> SET @one_week_ago := DATE_SUB(@today, INTERVAL 7 DAY);
mysql> SELECT @today, @one_week_ago;
+------------+---------------+
| @today     | @one_week_ago |
+------------+---------------+
| 2004-12-29 | 2004-12-22    |
+------------+---------------+
```

Generating Summaries

One of the most useful things MySQL can do for you is to boil down lots of raw data and summarize it. MySQL becomes a powerful ally when you learn to use it to generate summaries because that is an especially tedious, time-consuming, error-prone activity when done manually.

One simple form of summarizing is to determine which unique values are present in a set of values. Use the `DISTINCT` keyword to remove duplicate rows from a result. For example, the different states in which presidents have been born can be found like this:

```
mysql> SELECT DISTINCT state FROM president ORDER BY state;
+-------+
| state |
+-------+
| AR    |
| CA    |
| CT    |
| GA    |
| IA    |
| IL    |
```

```
| KY    |
| MA    |
| MO    |
| NC    |
| NE    |
| NH    |
| NJ    |
| NY    |
| OH    |
| PA    |
| SC    |
| TX    |
| VA    |
| VT    |
+-------+
```

Another form of summarizing involves counting, using the `COUNT()` function. If you use `COUNT(*)`, it tells you the number of rows selected by your query. If a query has no `WHERE` clause, it selects all rows, so `COUNT(*)` tells you the number of rows in your table. The following query shows how many membership records the Historical League `member` table contains:

```
mysql> SELECT COUNT(*) FROM member;
+----------+
| COUNT(*) |
+----------+
|      102 |
+----------+
```

If a query does have a `WHERE` clause, `COUNT(*)` tells you how many rows the clause matches. This query shows how many quizzes you have given to your class so far:

```
mysql> SELECT COUNT(*) FROM grade_event WHERE category = 'Q';
+----------+
| COUNT(*) |
+----------+
|        4 |
+----------+
```

`COUNT(*)` counts every row selected. By contrast, `COUNT(col_name)` counts only non-`NULL` values. The following query demonstrates these differences:

```
mysql> SELECT COUNT(*), COUNT(email), COUNT(expiration) FROM member;
+----------+--------------+-------------------+
| COUNT(*) | COUNT(email) | COUNT(expiration) |
+----------+--------------+-------------------+
|      102 |           80 |                96 |
+----------+--------------+-------------------+
```

This shows that although the `member` table has 102 records, only 80 of them have a value in the `email` column. It also shows that six members have a lifetime membership. (A `NULL` value in the `expiration` column indicates a lifetime membership, and since 96 out of 102 records are not `NULL`, that leaves six.)

`COUNT()` combined with `DISTINCT` counts the number of distinct non-`NULL` values in a result. For example, to count the number of different states in which presidents have been born, do this:

```
mysql> SELECT COUNT(DISTINCT state) FROM president;
+----------------------+
```

```
| COUNT(DISTINCT state) |
+-----------------------+
|                    20 |
+-----------------------+
```

You can produce an overall count of values in a column, or break down the counts by categories. For example, you may know the overall number of students in your class as a result of running this query:

```
mysql> SELECT COUNT(*) FROM student;
+----------+
| COUNT(*) |
+----------+
|       31 |
+----------+
```

But how many students are boys and how many are girls? One way to find out is by asking for a count for each sex separately:

```
mysql> SELECT COUNT(*) FROM student WHERE sex='f';
+----------+
| COUNT(*) |
+----------+
|       15 |
+----------+
mysql> SELECT COUNT(*) FROM student WHERE sex='m';
+----------+
| COUNT(*) |
+----------+
|       16 |
+----------+
```

However, although that approach works, it's tedious and not really very well suited for columns that might have several different values. Consider how you'd determine the number of presidents born in each state this way. You'd have to find out which states are represented so as not to miss any (`SELECT DISTINCT state FROM president`), and then run a `SELECT COUNT(*)` query for each state. That is clearly something you don't want to do.

Fortunately, it's possible to use a single query to count how many times each distinct value occurs in a column. For the student list, count boys and girls like this:

```
mysql> SELECT sex, COUNT(*) FROM student GROUP BY sex;
+-----+----------+
| sex | COUNT(*) |
+-----+----------+
| F   |       15 |
| M   |       16 |
+-----+----------+
```

The same form of query tells us how many presidents were born in each state:

```
mysql> SELECT state, COUNT(*) FROM president GROUP BY state;
+-------+----------+
| state | COUNT(*) |
+-------+----------+
| AR    |        1 |
| CA    |        1 |
| CT    |        1 |
| GA    |        1 |
```

```
| IA    |         1 |
| IL    |         1 |
| KY    |         1 |
| MA    |         4 |
| MO    |         1 |
| NC    |         2 |
| NE    |         1 |
| NH    |         1 |
| NJ    |         1 |
| NY    |         4 |
| OH    |         7 |
| PA    |         1 |
| SC    |         1 |
| TX    |         2 |
| VA    |         8 |
| VT    |         2 |
+-------+-----------+
```

When you count values in groups this way, the `GROUP BY` clause is necessary; it tells MySQL how to cluster values before counting them. You'll just get an error if you omit it.

The use of `COUNT(*)` with `GROUP BY` to count values has a number of advantages over counting occurrences of each distinct column value individually:

- You don't have to know in advance what values are present in the column you're summarizing.
- You need only a single query, not several.
- You get all the results with a single query, so you can sort the output.

The first two advantages are important for expressing queries more easily. The third advantage is important because it affords more flexibility in displaying results. By default, MySQL uses the columns named in the `GROUP BY` clause to sort the results, but you can specify an `ORDER BY` clause to sort in a different order. For example, if you want number of presidents grouped by state of birth, but sorted with the most well-represented states first, you can use an `ORDER BY` clause as follows:

```
mysql> SELECT state, COUNT(*) AS count FROM president
    -> GROUP BY state ORDER BY count DESC;
+-------+-------+
| state | count |
+-------+-------+
| VA    |     8 |
| OH    |     7 |
| MA    |     4 |
| NY    |     4 |
| NC    |     2 |
| VT    |     2 |
| TX    |     2 |
| SC    |     1 |
| NH    |     1 |
| PA    |     1 |
| KY    |     1 |
| NJ    |     1 |
| IA    |     1 |
| MO    |     1 |
| CA    |     1 |
| NE    |     1 |
| GA    |     1 |
| IL    |     1 |
| AR    |     1 |
| CT    |     1 |
+-------+-------+
```

When the column you want to use for sorting is produced by a summary function, you cannot refer to the function directly in the `ORDER BY` clause. Instead, give the column an alias and refer to it that way. The preceding query demonstrates this, where the `COUNT(*)` column is aliased as `count`. Another way to refer to such a column in an `ORDER BY` clause is by its position in the output. The previous query could have been written as follows instead:

```
SELECT state, COUNT(*) FROM president
GROUP BY state ORDER BY 2 DESC;
```

Referring to columns by position is allowable in MySQL, but problematic:

- Use of column positions leads to less understandable queries because numbers are less meaningful than names.
- If you add, remove, or reorder output columns, you must remember to check the `ORDER BY` clause and fix the column number if it has changed.
- The syntax of referring to column positions in `ORDER BY` clauses is no longer part of standard SQL and should be considered deprecated.

Aliases have none of those problems.

If you want to group results using `GROUP BY` with a calculated column, you can refer to it using an alias or column position, just as with `ORDER BY`. The following query determines how many presidents were born in each month of the year:

```
mysql> SELECT MONTH(birth) AS Month, MONTHNAME(birth) AS Name,
    -> COUNT(*) AS count
    -> FROM president GROUP BY Name ORDER BY Month;
+-------+-----------+-------+
| Month | Name      | count |
+-------+-----------+-------+
|     1 | January   |     4 |
|     2 | February  |     4 |
|     3 | March     |     4 |
|     4 | April     |     4 |
|     5 | May       |     2 |
|     6 | June      |     1 |
|     7 | July      |     4 |
|     8 | August    |     4 |
|     9 | September |     1 |
|    10 | October   |     6 |
|    11 | November  |     5 |
|    12 | December  |     3 |
+-------+-----------+-------+
```

Using column positions, the query would be written like this:

```
SELECT MONTH (birth), MONTHNAME(birth), COUNT(*)
FROM president GROUP BY 2 ORDER BY 1;
```

`COUNT()` can be combined with `ORDER BY` and `LIMIT`. For example, to find the four most well-represented states in the `president` table, use this statement

```
mysql> SELECT state, COUNT(*) AS count FROM president
    -> GROUP BY state ORDER BY count DESC LIMIT 4;
+-------+-------+
| state | count |
+-------+-------+
| VA    |     8 |
```

```
| OH    |     7 |
| MA    |     4 |
| NY    |     4 |
+-------+-------+
```

If you don't want to limit query output with a `LIMIT` clause, but rather by looking for particular values of `COUNT()`, use a `HAVING` clause. `HAVING` is similar to `WHERE` in that it specifies conditions that must be satisfied by output rows. It differs from `WHERE` in that it can refer to the results of summary functions like `COUNT()`. The following query will tell you which states are represented by two or more presidents:

```
mysql> SELECT state, COUNT(*) AS count FROM president
    -> GROUP BY state HAVING count > 1 ORDER BY count DESC;
+-------+-------+
| state | count |
+-------+-------+
| VA    |     8 |
| OH    |     7 |
| MA    |     4 |
| NY    |     4 |
| NC    |     2 |
| VT    |     2 |
| TX    |     2 |
+-------+-------+
```

More generally, this is the type of query to run when you want to find duplicated values in a column. Or, to find non-duplicated values, use `HAVING count = 1`.

There are several summary functions other than `COUNT()`. The `MIN()`, `MAX()`, `SUM()`, and `AVG()` functions are useful for determining the minimum, maximum, total, and average values in a column. You can even use them all at the same time. The following query shows various numeric characteristics for each quiz and test you've given. It also shows how many scores go into computing each of the values. (Some students may have been absent and are not counted.)

```
mysql> SELECT
    -> event_id,
    -> MIN(score) AS minimum,
    -> MAX(score) AS maximum,
    -> MAX(score)-MIN(score)+1 AS range,
    -> SUM(score) AS total,
    -> AVG(score) AS average,
    -> COUNT(score) AS count
    -> FROM score
    -> GROUP BY event_id;
+----------+---------+---------+-------+-------+---------+-------+
| event_id | minimum | maximum | range | total | average | count |
+----------+---------+---------+-------+-------+---------+-------+
|        1 |       9 |      20 |    12 |   439 | 15.1379 |    29 |
|        2 |       8 |      19 |    12 |   425 | 14.1667 |    30 |
|        3 |      60 |      97 |    38 |  2425 | 78.2258 |    31 |
|        4 |       7 |      20 |    14 |   379 | 14.0370 |    27 |
|        5 |       8 |      20 |    13 |   383 | 14.1852 |    27 |
|        6 |      62 |     100 |    39 |  2325 | 80.1724 |    29 |
+----------+---------+---------+-------+-------+---------+-------+
```

This information might be more meaningful if you knew whether the `event_id` values represented quizzes or tests, of course. However, to produce that information, we need to consult the `grade_event` table as well; we'll revisit this query in "Retrieving Information from Multiple Tables."

If you want to produce extra output lines that give you a "summary of summaries," add a `WITH ROLLUP` clause. This tells MySQL to calculate "super-aggregate" values for the grouped rows. Here's a simple example based on an earlier statement that counts the number of students of each sex. The `WITH ROLLUP` clause produces another line that summarizes the counts for both sexes:

```
mysql> SELECT sex, COUNT(*) FROM student GROUP BY sex WITH ROLLUP;
+-----+----------+
| sex | COUNT(*) |
+-----+----------+
| F   |       15 |
| M   |       16 |
| NULL|       31 |
+-----+----------+
```

The `NULL` in the grouped column indicates that corresponding count is the summary value for the preceding groups.

`WITH ROLLUP` can be used with the other aggregate functions as well. The following statement calculates grade summaries as just shown a few paragraphs earlier, but also produces an extra super-aggregate line:

```
mysql> SELECT
    -> event_id,
    -> MIN(score) AS minimum,
    -> MAX(score) AS maximum,
    -> MAX(score)-MIN(score)+1 AS range,
    -> SUM(score) AS total,
    -> AVG(score) AS average,
    -> COUNT(score) AS count
    -> FROM score
    -> GROUP BY event_id
    -> WITH ROLLUP;
+----------+---------+---------+-------+-------+---------+-------+
| event_id | minimum | maximum | range | total | average | count |
+----------+---------+---------+-------+-------+---------+-------+
|        1 |       9 |      20 |    12 |   439 | 15.1379 |    29 |
|        2 |       8 |      19 |    12 |   425 | 14.1667 |    30 |
|        3 |      60 |      97 |    38 |  2425 | 78.2258 |    31 |
|        4 |       7 |      20 |    14 |   379 | 14.0370 |    27 |
|        5 |       8 |      20 |    13 |   383 | 14.1852 |    27 |
|        6 |      62 |     100 |    39 |  2325 | 80.1724 |    29 |
|     NULL |       7 |     100 |    94 |  6376 | 36.8555 |   173 |
+----------+---------+---------+-------+-------+---------+-------+
```

In this output, the final line displays aggregate values calculated based on all the preceding group summary values.

`WITH ROLLUP` is useful because it provides extra information that you would otherwise have to get by running another query. Using a single query is more efficient because the server need not examine the data twice. If the `GROUP BY` clause names more than one column, `WITH ROLLUP` produces additional super-aggregate lines that contain higher-level summary values.

Summary functions are fun to play with because they're so powerful, but it's easy to get carried away with them. Consider this query:

```
mysql> SELECT
    -> state AS State,
    -> AVG((TO_DAYS(death)-TO_DAYS(birth))/365) AS Age
    -> FROM president WHERE death IS NOT NULL
    -> GROUP BY state ORDER BY Age;
+-------+-----------+
```

```
| State | Age       |
+-------+-----------+
| KY    | 56.208219 |
| VT    | 58.852055 |
| NC    | 60.141096 |
| OH    | 62.866145 |
| NH    | 64.917808 |
| NY    | 69.342466 |
| NJ    | 71.315068 |
| TX    | 71.476712 |
| MA    | 72.642009 |
| VA    | 72.822945 |
| PA    | 77.158904 |
| SC    | 78.284932 |
| CA    | 81.336986 |
| MO    | 88.693151 |
| IA    | 90.254795 |
| IL    | 93.391781 |
+-------+-----------+
```

The query selects presidents who have died, groups them by state of birth, determines their approximate age at time of death, computes the average age (per state), and then sorts the results by average age. In other words, the query determines, for non-living presidents, the average age of death by state of birth.

And what does that demonstrate? It shows only that you can write the query. It certainly doesn't show that the query is worth writing. Not all things you can do with a database are equally meaningful. Nevertheless, people sometimes go query-happy when they find out what they can do with their database. This may account for the rise of increasingly esoteric and bizarre statistics on televised sporting events over the last few years. The sports statisticians can use their databases to figure out everything you'd ever want to know about a team, and also everything you'd never want to know. Do you really care which third-string quarterback holds the record for most interceptions on third down when his team is leading by more than 14 points with the ball inside the 15-yard line in the last two minutes of the second quarter?

Retrieving Information from Multiple Tables

The statements that we've written so far have pulled data from a single table. But MySQL is capable of working much harder for you. I've mentioned before that the power of a relational DBMS lies in its capability to relate one thing to another because that allows you to combine information from multiple tables to answer questions that can't be answered from individual tables alone. This section describes how to write statements that do that.

When you select information from multiple tables, you're performing an operation called a "join." That's because you're producing a result by joining the information in one table to the information in another table. This is done by matching up common values in the tables.

Let's work through an example. Earlier, in "Tables for the Grade-Keeping Project," a query to retrieve quiz or test scores for a given date was presented without explanation. Now it's time for the explanation. The query actually involves a three-way join, so we'll build up to it in two steps. In the first step, we construct a query to select scores for a given date as follows:

```
mysql> SELECT student_id, date, score, category
    -> FROM grade_event, score
    -> WHERE date = '2004-09-23'
    -> AND grade_event.event_id = score.event_id;
+------------+------------+-------+----------+
| student_id | date       | score | category |
+------------+------------+-------+----------+
|          1 | 2004-09-23 |    15 | Q        |
|          2 | 2004-09-23 |    12 | Q        |
|          3 | 2004-09-23 |    11 | Q        |
|          5 | 2004-09-23 |    13 | Q        |
|          6 | 2004-09-23 |    18 | Q        |
```

```
...
```

The query works by finding the `grade_event` record with the given date (`'2004-09-23'`), and then using the event ID in that record to locate scores that have the same event ID. For each matching `grade_event` record and `score` record combination, it displays the student ID, score, date, and event category.

The query differs from others we have written in two important respects:

- The `FROM` clause names more than one table because we're retrieving data from more than one table:
- `FROM grade_event, score`

- The `WHERE` clause specifies that the `grade_event` and `score` tables are joined by matching up the `event_id` values in each table:
- `WHERE ... grade_event.event_id = score.event_id`

Notice how we refer to the `event_id` columns as `grade_event.event_id` and `score.event_id` using `tbl_name.col_name` syntax so that MySQL knows which tables we're referring to. This is because `event_id` occurs in both tables, so it's ambiguous if used without a table name to qualify it. The other columns in the query (`date`, `score`, and `category`) can be used without a table qualifier because they appear in only one of the tables and thus are unambiguous.

I generally prefer to qualify every column in a join to make it clearer (more explicit) which table each column is part of, and that's how I'll write joins from now on. In fully qualified form, the query looks like this:

```
SELECT score.student_id, grade_event.date, score.score, grade_event.category
FROM grade_event, score
WHERE grade_event.date = '2004-09-23'
AND grade_event.event_id = score.event_id;
```

The first-stage query uses the `grade_event` table to map a date to an event ID, and then uses the ID to find the matching scores in the `score` table. Output from the query contains `student_id` values, but names would be more meaningful. By using the `student` table, we can map student IDs onto names, which is the second step. To accomplish name display, use the fact that the `score` and `student` tables both have `student_id` columns allowing the records in them to be linked. The resulting query is as follows:

```
mysql> SELECT
    -> student.name, grade_event.date, score.score, grade_event.category
    -> FROM grade_event, score, student
    -> WHERE grade_event.date = '2004-09-23'
    -> AND grade_event.event_id = score.event_id
    -> AND score.student_id = student.student_id;
+-----------+------------+-------+----------+
| name      | date       | score | category |
+-----------+------------+-------+----------+
| Megan     | 2004-09-23 |    15 | Q        |
| Joseph    | 2004-09-23 |    12 | Q        |
| Kyle      | 2004-09-23 |    11 | Q        |
| Abby      | 2004-09-23 |    13 | Q        |
| Nathan    | 2004-09-23 |    18 | Q        |
...
```

This query differs from the previous one as follows:

- The `FROM` clause now includes the `student` table because the statement uses it in addition to the `grade_event` and `score` tables.
- The `student_id` column was unambiguous before, so it was possible to refer to it in either unqualified (`student_id`) or qualified (`score.student_id`) form. Now it is ambiguous because it is present in both the `score` and `student` tables. Therefore, it must be qualified as `score.student_id` or `student.student_id` to make it clear which table to use.
- The `WHERE` clause has an additional term specifying that `score` table records are matched against `student` table records based on student ID:
- `WHERE ... score.student_id = student.student_id`
- The query displays the student name rather than the student ID. (You could display both if you wanted. Just add `student.student_id` to the list of output columns.)

With this query, you can plug in any date and get back the scores for that date, complete with student names and the score category. You don't have to know anything about student IDs or event IDs. MySQL takes care of figuring out the relevant ID values and using them to match up table rows.

Another task the grade-keeping project involves is summarizing student absences. Absences are recorded by student ID and date in the `absence` table. To get student names (not just IDs), we need to join the `absence` table to the `student` table, based on the `student_id` value. The following query lists student ID number and name along with a count of absences:

```
mysql> SELECT student.student_id, student.name,
    -> COUNT(absence.date) AS absences
    -> FROM student, absence
    -> WHERE student.student_id = absence.student_id
    -> GROUP BY student.student_id;
+------------+-------+----------+
| student_id | name  | absences |
+------------+-------+----------+
|          3 | Kyle  |        1 |
|          5 | Abby  |        1 |
|         10 | Peter |        2 |
|         17 | Will  |        1 |
|         20 | Avery |        1 |
+------------+-------+----------+
```

The output produced by the query is fine if we want to know only which students had absences. But if we turn in this list to the school office, they might say, "What about the other students? We want a value for every student." That's a slightly different question. It means we want to know the number of absences, even for students who had none. Because the question is different, the query that answers it is different as well.

To answer the question, we will use a `LEFT JOIN` rather than a regular join. `LEFT JOIN` tells MySQL to produce a row of output for each row selected from the table named first in the join (that is, the table named to the left of the `LEFT JOIN` keywords). By naming the `student` table first, we'll get output for every student, even those who are not represented in the `absence` table. To write this query, use `LEFT JOIN` between the tables named in the `FROM` clause (rather than separating them by a comma), and add an `ON` clause that says how to match up records in the two tables. The query looks like this:

```
mysql> SELECT student.student_id, student.name,
    -> COUNT(absence.date) AS absences
    -> FROM student LEFT JOIN absence
    -> ON student.student_id = absence.student_id
    -> GROUP BY student.student_id;
+------------+-----------+----------+
| student_id | name      | absences |
+------------+-----------+----------+
|          1 | Megan     |        0 |
|          2 | Joseph    |        0 |
|          3 | Kyle      |        1 |
|          4 | Katie     |        0 |
```

```
|          5 | Abby      |        1 |
|          6 | Nathan    |        0 |
|          7 | Liesl     |        0 |
...
```

Earlier, in "Generating Summaries," we ran a query that produced a numeric characterization of the data in the `score` table. Output from that query listed event ID but did not include event dates or categories, because we didn't know then how to join the `score` table to the `grade_event` table to map the IDs onto dates and categories. Now we do. The following query is similar to one run earlier, but shows the dates and categories rather than simply the numeric event IDs:

```
mysql> SELECT
    -> grade_event.date,grade_event.category,
    -> MIN(score.score) AS minimum,
    -> MAX(score.score) AS maximum,
    -> MAX(score.score)-MIN(score.score)+1 AS range,
    -> SUM(score.score) AS total,
    -> AVG(score.score) AS average,
    -> COUNT(score.score) AS count
    -> FROM score, grade_event
    -> WHERE score.event_id = grade_event.event_id
    -> GROUP BY grade_event.date;
+------------+----------+---------+---------+-------+-------+---------+-------+
| date       | category | minimum | maximum | range | total | average | count |
+------------+----------+---------+---------+-------+-------+---------+-------+
| 2004-09-03 | Q        |       9 |      20 |    12 |   439 | 15.1379 |    29 |
| 2004-09-06 | Q        |       8 |      19 |    12 |   425 | 14.1667 |    30 |
| 2004-09-09 | T        |      60 |      97 |    38 |  2425 | 78.2258 |    31 |
| 2004-09-16 | Q        |       7 |      20 |    14 |   379 | 14.0370 |    27 |
| 2004-09-23 | Q        |       8 |      20 |    13 |   383 | 14.1852 |    27 |
| 2004-10-01 | T        |      62 |     100 |    39 |  2325 | 80.1724 |    29 |
+------------+----------+---------+---------+-------+-------+---------+-------+
```

You can use functions such as `COUNT()` and `AVG()` to produce a summary over multiple columns, even if the columns come from different tables. The following query determines the number of scores and the average score for each combination of event date and student sex:

```
mysql> SELECT grade_event.date, student.sex,
    -> COUNT(score.score) AS count, AVG(score.score) AS average
    -> FROM grade_event, score, student
    -> WHERE grade_event.event_id = score.event_id
    -> AND score.student_id = student.student_id
    -> GROUP BY grade_event.date, student.sex;
+------------+-----+-------+---------+
| date       | sex | count | average |
+------------+-----+-------+---------+
| 2004-09-03 | F   |    14 | 14.6429 |
| 2004-09-03 | M   |    15 | 15.6000 |
| 2004-09-06 | F   |    14 | 14.7143 |
| 2004-09-06 | M   |    16 | 13.6875 |
| 2004-09-09 | F   |    15 | 77.4000 |
| 2004-09-09 | M   |    16 | 79.0000 |
| 2004-09-16 | F   |    13 | 15.3077 |
| 2004-09-16 | M   |    14 | 12.8571 |
| 2004-09-23 | F   |    12 | 14.0833 |
| 2004-09-23 | M   |    15 | 14.2667 |
| 2004-10-01 | F   |    14 | 77.7857 |
| 2004-10-01 | M   |    15 | 82.4000 |
+------------+-----+-------+---------+
```

We can use a similar query to perform one of the grade-keeping project tasks: computing the total score per student at the end of the semester. The query is as follows:

```
SELECT student.student_id, student.name,
SUM(score.score) AS total, COUNT(score.score) AS n
FROM grade_event, score, student
WHERE grade_event.event_id = score.event_id
AND score.student_id = student.student_id
GROUP BY score.student_id
ORDER BY total;
```

There is no requirement that a join be performed between different tables. It might seem odd at first, but you can join a table to itself. For example, you can determine whether any presidents were born in the same city by checking each president's birthplace against every other president's birthplace:

```
mysql> SELECT p1.last_name, p1.first_name, p1.city, p1.state
    -> FROM president AS p1, president AS p2
    -> WHERE p1.city = p2.city AND p1.state = p2.state
    -> AND (p1.last_name <> p2.last_name OR p1.first_name <> p2.first_name)
    -> ORDER BY state, city, last_name;
+-----------+-------------+-----------+-------+
| last_name | first_name  | city      | state |
+-----------+-------------+-----------+-------+
| Adams     | John Quincy | Braintree | MA    |
| Adams     | John        | Braintree | MA    |
+-----------+-------------+-----------+-------+
```

There are two tricky things about this query:

- It's necessary to refer to two instances of the same table, so we create table aliases (`p1`, `p2`) and use them to disambiguate references to the table's columns. As with column aliases, the `AS` keyword is optional when naming table aliases.
- Each president's record matches itself, but we don't want to see that in the output. The second line of the `WHERE` clause disallows matches of a record to itself by making sure that the records being compared are for different presidents.

A similar query finds presidents who were born on the same day. However, birth dates cannot be compared directly because that would miss presidents who were born in different years. Instead, use `MONTH()` and `DAYOFMONTH()` to compare month and day of the birth date:

```
mysql> SELECT p1.last_name, p1.first_name, p1.birth
    -> FROM president AS p1, president AS p2
    -> WHERE MONTH(p1.birth) = MONTH(p2.birth)
    -> AND DAYOFMONTH(p1.birth) = DAYOFMONTH(p2.birth)
    -> AND (p1.last_name <> p2.last_name OR p1.first_name <> p2.first_name)
    -> ORDER BY p1.last_name;
+-----------+------------+------------+
| last_name | first_name | birth      |
+-----------+------------+------------+
| Harding   | Warren G.  | 1865-11-02 |
| Polk      | James K.   | 1795-11-02 |
+-----------+------------+------------+
```

Using `DAYOFYEAR()` rather than the combination of `MONTH()` and `DAYOFMONTH()` would result in a simpler query, but it would produce incorrect results when comparing dates from leap years to dates from non-leap years.

Another kind of multiple-table retrieval uses something called a "subquery," which is one SELECT nested within another. Suppose that you want to identify those students who have perfect attendance. This is equivalent to determining which students are not represented in the absence table, which can be done like this:

```
mysql> SELECT * FROM student
    -> WHERE student_id NOT IN (SELECT student_id FROM absence);
+-----------+-----+------------+
| name      | sex | student_id |
+-----------+-----+------------+
| Megan     | F   |          1 |
| Joseph    | M   |          2 |
| Katie     | F   |          4 |
| Nathan    | M   |          6 |
| Liesl     | F   |          7 |
...
```

The nested SELECT determines the set of student_id values that are present in the absence table, and the outer SELECT retrieves student records that don't match any of those IDs.

A subquery also provides a single-statement solution to the question asked earlier about which presidents were born before Andrew Jackson. The original solution used two statements and a user variable, but it can be done with a subquery as follows:

```
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth < (SELECT birth FROM president
    -> WHERE last_name = 'Jackson' AND first_name = 'Andrew');
+------------+------------+------------+
| last_name  | first_name | birth      |
+------------+------------+------------+
| Washington | George     | 1732-02-22 |
| Adams      | John       | 1735-10-30 |
| Jefferson  | Thomas     | 1743-04-13 |
| Madison    | James      | 1751-03-16 |
| Monroe     | James      | 1758-04-28 |
+------------+------------+------------+
```

The inner SELECT determines Andrew Jackson's birth date, and the outer SELECT retrieves presidents with a birth date earlier than his.

Deleting or Updating Existing Records

Sometimes you want to get rid of records or change their contents. The DELETE and UPDATE statements let you do this. This section discusses how to use them.

The DELETE statement has this form:

```
DELETE FROM tbl_name
WHERE which records to delete;
```

The WHERE clause that specifies which records should be deleted is optional, but if you leave it out, all records in the table are deleted. In other words, the simplest DELETE statement is also the most dangerous:

```
DELETE FROM tbl_name;
```

That statement wipes out the table's contents entirely, so be careful with it! To delete specific records, use the WHERE clause to identify the records in which you're interested. This is similar to using a WHERE clause in a

SELECT statement to avoid selecting the entire table. For example, to specifically delete from the president table only those presidents born in Ohio, use this statement:

```
mysql> DELETE FROM president WHERE state='OH';
Query OK, 7 rows affected (0.00 sec)
```

If you're not really sure which records a DELETE statement will remove, it's often a good idea to test the WHERE clause first by using it with a SELECT statement to find out which records it matches. This can help you ensure that you'll actually delete the records you intend, and only those records. Suppose that you want to delete the record for Teddy Roosevelt. Would the following statement do the job?

```
DELETE FROM president WHERE last_name='Roosevelt';
```

Yes, in the sense that it would delete the record you have in mind. No, in the sense that it also would delete the record for Franklin Roosevelt. It's safer to check the WHERE clause with a SELECT statement first, like this:

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name='Roosevelt';
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Roosevelt | Theodore    |
| Roosevelt | Franklin D. |
+-----------+-------------+
```

From that you can see the need to be more specific by adding a condition for the first name:

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name='Roosevelt' AND first_name='Theodore';
+-----------+------------+
| last_name | first_name |
+-----------+------------+
| Roosevelt | Theodore   |
+-----------+------------+
```

Now you know the WHERE clause that properly identifies only the desired record, and the DELETE statement can be constructed correctly:

```
mysql> DELETE FROM president
    -> WHERE last_name='Roosevelt' AND first_name='Theodore';
```

To modify existing records, use UPDATE, which has this form:

```
UPDATE tbl_name
SET which columns to change
WHERE which records to update;
```

The WHERE clause is just as for DELETE. It's optional, so if you don't specify one, every record in the table will be updated. For example, the following statement changes the name of each student to "George":

```
mysql> UPDATE student SET name='George';
```

Obviously, you must be careful with statements like that, so normally you add a `WHERE` clause to be more specific about which records to update. Suppose that you recently added a new member to the Historical League but filled in only a few columns of his entry:

```
mysql> INSERT INTO member (last_name,first_name)
    -> VALUES('York','Jerome');
```

Then you realize you forgot to set his membership expiration date. You can fix that with an `UPDATE` statement that includes an appropriate `WHERE` clause to identify which record to change:

```
mysql> UPDATE member
    -> SET expiration='2006-7-20'
    -> WHERE last_name='York' AND first_name='Jerome';
```

You can update multiple columns with a single statement. The following `UPDATE` modifies Jerome's email and postal addresses:

```
mysql> UPDATE member
    -> SET email='jeromey@aol.com', street='123 Elm St',
    -> city='Anytown', state='NY', zip='01003'
    -> WHERE last_name='York' AND first_name='Jerome';
```

You can also "unset" a column by setting its value to `NULL` (assuming that the column allows `NULL` values). If at some point in the future Jerome later decides to pay the big membership renewal fee that allows him to become a lifetime member, you can mark his record as "never expires" by setting his expiration date to `NULL`:

```
mysql> UPDATE member
    -> SET expiration=NULL
    -> WHERE last_name='York' AND first_name='Jerome';
```

With `UPDATE`, just as for `DELETE`, it's not a bad idea to test a `WHERE` clause using a `SELECT` statement to make sure that you're choosing the right records to update. Otherwise, if your selection criteria are too narrow or too broad, you'll update too few or too many records.

If you've tried the statements in this section, you'll have deleted and modified records in the `my_super_db` tables. Before proceeding to the next section, you should undo those changes. Do that by reloading the tables using the instructions given earlier, in "Resetting the `my_super_db` Database to a Known State."