**4.1 Connecting to and Disconnecting from the Server**

To connect to the server, you will usually need to provide a MySQL user name when you invoke mysql

and, most likely, a password. If the server runs on a machine other than the one where you log in, you will

also need to specify a host name. Contact your administrator to find out what connection parameters you

should use to connect (that is, what host, user name, and password to use). Once you know the proper

parameters, you should be able to connect like this:

shell> **mysql -h *host* -u *user* -p**

Enter password: **\*\*\*\*\*\*\*\***

*host* and *user* represent the host name where your MySQL server is running and the user name of your

MySQL account. Substitute appropriate values for your setup. The \*\*\*\*\*\*\*\* represents your password;

enter it when mysql displays the Enter password: prompt.

If that works, you should see some introductory information followed by a mysql> prompt:

shell> **mysql -h *host* -u *user* -p**

Enter password: **\*\*\*\*\*\*\*\***

Welcome to the MySQL monitor. Commands end with ; or \g.

Your MySQL connection id is 25338 to server version: 5.7.18-standard

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

mysql>

The mysql> prompt tells you that mysql is ready for you to enter SQL statements.

If you are logging in on the same machine that MySQL is running on, you can omit the host, and simply

use the following:

shell> **mysql -u *user* -p**

If, when you attempt to log in, you get an error message such as ERROR 2002 (HY000): Can't

connect to local MySQL server through socket '/tmp/mysql.sock' (2), it means that

the MySQL server daemon (Unix) or service (Windows) is not running. Consult the administrator or see the

section of Chapter 2, *Installing and Upgrading MySQL* that is appropriate to your operating system.

For help with other problems often encountered when trying to log in, see Section B.5.2, “Common Errors

When Using MySQL Programs”.

Some MySQL installations permit users to connect as the anonymous (unnamed) user to the server

running on the local host. If this is the case on your machine, you should be able to connect to that server

by invoking mysql without any options:

shell> **mysql**

After you have connected successfully, you can disconnect any time by typing QUIT (or \q) at the mysql>

prompt:

mysql> **QUIT**

Bye

On Unix, you can also disconnect by pressing Control+D.

Most examples in the following sections assume that you are connected to the server. They indicate this by

the mysql> prompt.

**From XAMPP**cd c:\xampp\mysql\bin

mysql.exe -u root --password

**4.2 Entering Queries**

Make sure that you are connected to the server, as discussed in the previous section. Doing so does not in

itself select any database to work with, but that is okay. At this point, it is more important to find out a little

about how to issue queries than to jump right in creating tables, loading data into them, and retrieving data

from them. This section describes the basic principles of entering queries, using several queries you can

try out to familiarize yourself with how mysql works.

Here is a simple query that asks the server to tell you its version number and the current date. Type it in as

shown here following the mysql> prompt and press Enter:

mysql> **SELECT VERSION(), CURRENT\_DATE;**

+--------------+--------------+

| VERSION() | CURRENT\_DATE |

+--------------+--------------+

| 5.7.1-m4-log | 2012-12-25 |

+--------------+--------------+

1 row in set (0.01 sec)

mysql>

This query illustrates several things about mysql:

• A query normally consists of an SQL statement followed by a semicolon. (There are some exceptions

where a semicolon may be omitted. QUIT, mentioned earlier, is one of them. We'll get to others later.)

• When you issue a query, mysql sends it to the server for execution and displays the results, then prints

another mysql> prompt to indicate that it is ready for another query.

• mysql displays query output in tabular form (rows and columns). The first row contains labels for

the columns. The rows following are the query results. Normally, column labels are the names of the

columns you fetch from database tables. If you're retrieving the value of an expression rather than a

table column (as in the example just shown), mysql labels the column using the expression itself.

• mysql shows how many rows were returned and how long the query took to execute, which gives you

a rough idea of server performance. These values are imprecise because they represent wall clock time

(not CPU or machine time), and because they are affected by factors such as server load and network

latency. (For brevity, the “rows in set” line is sometimes not shown in the remaining examples in this

chapter.)

Keywords may be entered in any lettercase. The following queries are equivalent:

mysql> **SELECT VERSION(), CURRENT\_DATE;**

mysql> **select version(), current\_date;**

mysql> **SeLeCt vErSiOn(), current\_DATE;**

Here is another query. It demonstrates that you can use mysql as a simple calculator:

mysql> **SELECT SIN(PI()/4), (4+1)\*5;**

+------------------+---------+

| SIN(PI()/4) | (4+1)\*5 |

+------------------+---------+

| 0.70710678118655 | 25 |

+------------------+---------+

1 row in set (0.02 sec)

The queries shown thus far have been relatively short, single-line statements. You can even enter multiple

statements on a single line. Just end each one with a semicolon:

mysql> **SELECT VERSION(); SELECT NOW();**

+------------------+

| VERSION() |

+------------------+

| 5.7.10-ndb-7.5.1 |

+------------------+

1 row in set (0.00 sec)

+---------------------+

| NOW() |

+---------------------+

| 2016-01-29 18:02:55 |

+---------------------+

1 row in set (0.00 sec)

A query need not be given all on a single line, so lengthy queries that require several lines are not a

problem. mysql determines where your statement ends by looking for the terminating semicolon, not by

looking for the end of the input line. (In other words, mysql accepts free-format input: it collects input lines

but does not execute them until it sees the semicolon.)

Here is a simple multiple-line statement:

mysql> **SELECT**

-> **USER()**

-> **,**

-> **CURRENT\_DATE;**

+---------------+--------------+

| USER() | CURRENT\_DATE |

+---------------+--------------+

| jon@localhost | 2010-08-06 |

+---------------+--------------+

In this example, notice how the prompt changes from mysql> to -> after you enter the first line of a

multiple-line query. This is how mysql indicates that it has not yet seen a complete statement and is

waiting for the rest. The prompt is your friend, because it provides valuable feedback. If you use that

feedback, you can always be aware of what mysql is waiting for.

If you decide you do not want to execute a query that you are in the process of entering, cancel it by typing

\c:

mysql> **SELECT**

-> **USER()**

-> **\c**

mysql>

Here, too, notice the prompt. It switches back to mysql> after you type \c, providing feedback to indicate

that mysql is ready for a new query.

The following table shows each of the prompts you may see and summarizes what they mean about the

state that mysql is in.

**Prompt Meaning**

mysql> Ready for new query

-> Waiting for next line of multiple-line query

'> Waiting for next line, waiting for completion of a string that began with a single quote (')

"> Waiting for next line, waiting for completion of a string that began with a double quote (")

`> Waiting for next line, waiting for completion of an identifier that began with a backtick (`)

/\*> Waiting for next line, waiting for completion of a comment that began with /\*

Multiple-line statements commonly occur by accident when you intend to issue a query on a single line, but

forget the terminating semicolon. In this case, mysql waits for more input:

mysql> **SELECT USER()**

->

If this happens to you (you think you've entered a statement but the only response is a -> prompt), most

likely mysql is waiting for the semicolon. If you don't notice what the prompt is telling you, you might sit

there for a while before realizing what you need to do. Enter a semicolon to complete the statement, and

mysql executes it:

mysql> **SELECT USER()**

-> **;**

+---------------+

| USER() |

+---------------+

| jon@localhost |

+---------------+

The '> and "> prompts occur during string collection (another way of saying that MySQL is waiting

for completion of a string). In MySQL, you can write strings surrounded by either ' or " characters (for

example, 'hello' or "goodbye"), and mysql lets you enter strings that span multiple lines. When

you see a '> or "> prompt, it means that you have entered a line containing a string that begins with a '  
or " quote character, but have not yet entered the matching quote that terminates the string. This often

indicates that you have inadvertently left out a quote character. For example:

mysql> **SELECT \* FROM my\_table WHERE name = 'Smith AND age < 30;**

'>

If you enter this SELECT statement, then press **Enter** and wait for the result, nothing happens. Instead

of wondering why this query takes so long, notice the clue provided by the '> prompt. It tells you that

mysql expects to see the rest of an unterminated string. (Do you see the error in the statement? The string

'Smith is missing the second single quotation mark.)

At this point, what do you do? The simplest thing is to cancel the query. However, you cannot just type \c

in this case, because mysql interprets it as part of the string that it is collecting. Instead, enter the closing

quote character (so mysql knows you've finished the string), then type \c:

mysql> **SELECT \* FROM my\_table WHERE name = 'Smith AND age < 30;**

'> **'\c**

mysql>

The prompt changes back to mysql>, indicating that mysql is ready for a new query.

The `> prompt is similar to the '> and "> prompts, but indicates that you have begun but not completed a

backtick-quoted identifier.

It is important to know what the '>, ">, and `> prompts signify, because if you mistakenly enter an

unterminated string, any further lines you type appear to be ignored by mysql—including a line containing

QUIT. This can be quite confusing, especially if you do not know that you need to supply the terminating

quote before you can cancel the current query.

**4.3 Creating and Using a Database**

Once you know how to enter SQL statements, you are ready to access a database.

Suppose that you have several pets in your home (your menagerie) and you would like to keep track of

various types of information about them. You can do so by creating tables to hold your data and loading

them with the desired information. Then you can answer different sorts of questions about your animals by

retrieving data from the tables. This section shows you how to perform the following operations:

**• Create a database**

**• Create a table**

**• Load data into the table**

**• Retrieve data from the table in various ways**

**• Use multiple tables**

The menagerie database is simple (deliberately), but it is not difficult to think of real-world situations

in which a similar type of database might be used. For example, a database like this could be used by

a farmer to keep track of livestock, or by a veterinarian to keep track of patient records. A menagerie

distribution containing some of the queries and sample data used in the following sections can be

obtained from the MySQL Web site. It is available in both compressed tar file and Zip formats at http://

dev.mysql.com/doc/.

Use the SHOW statement to find out what databases currently exist on the server:

mysql> **SHOW DATABASES;**

+----------+

| Database |

+----------+

| mysql |

| test |

| tmp |

+----------+

The mysql database describes user access privileges. The test database often is available as a

workspace for users to try things out.

The list of databases displayed by the statement may be different on your machine; SHOW DATABASES

does not show databases that you have no privileges for if you do not have the SHOW DATABASES

privilege. See **Section 14.7.5.14, “SHOW DATABASES Syntax”.**

If the test database exists, try to access it:

mysql> **USE test**

Database changed

USE, like QUIT, does not require a semicolon. (You can terminate such statements with a semicolon if you

like; it does no harm.) The USE statement is special in another way, too: it must be given on a single line.

You can use the test database (if you have access to it) for the examples that follow, but anything you

create in that database can be removed by anyone else with access to it. For this reason, you should

probably ask your MySQL administrator for permission to use a database of your own. Suppose that you

want to call yours menagerie. The administrator needs to execute a statement like this:

mysql> **GRANT ALL ON menagerie.\* TO 'your\_mysql\_name'@'your\_client\_host';**

where your\_mysql\_name is the MySQL user name assigned to you and your\_client\_host is the host

from which you connect to the server.

**4.3.1 Creating and Selecting a Database**

If the administrator creates your database for you when setting up your permissions, you can begin using

it. Otherwise, you need to create it yourself:

mysql> **CREATE DATABASE menagerie;**

Under Unix, database names are case sensitive (unlike SQL keywords), so you must always refer to

your database as menagerie, not as Menagerie, MENAGERIE, or some other variant. This is also true

for table names. (Under Windows, this restriction does not apply, although you must refer to databases

and tables using the same lettercase throughout a given query. However, for a variety of reasons, the

recommended best practice is always to use the same lettercase that was used when the database was

created.)

**Note**

If you get an error such as ERROR 1044 (42000): Access denied for user

'micah'@'localhost' to database 'menagerie' when attempting to

create a database, this means that your user account does not have the necessary

privileges to do so. Discuss this with the administrator or see **Section 7.2, “The**

**MySQL Access Privilege System”.**

Creating a database does not select it for use; you must do that explicitly. To make menagerie the current

database, use this statement:

mysql> **USE menagerie**

Database changed

Your database needs to be created only once, but you must select it for use each time you begin a mysql

session. You can do this by issuing a USE statement as shown in the example. Alternatively, you can select

the database on the command line when you invoke mysql. Just specify its name after any connection

parameters that you might need to provide. For example:

shell> **mysql -h *host* -u *user* -p menagerie**

Enter password: **\*\*\*\*\*\*\*\***

**Important**

menagerie in the command just shown is **not** your password. If you want to supply

your password on the command line after the -p option, you must do so with no

intervening space (for example, as -pmypassword, not as -p mypassword).

However, putting your password on the command line is not recommended,

because doing so exposes it to snooping by other users logged in on your machine.

**Note**

You can see at any time which database is currently selected using SELECT

DATABASE().

**4.3.2 Creating a Table**

Creating the database is the easy part, but at this point it is empty, as SHOW TABLES tells you:

mysql> **SHOW TABLES;**

Empty set (0.00 sec)

The harder part is deciding what the structure of your database should be: what tables you need and what

columns should be in each of them.

You want a table that contains a record for each of your pets. This can be called the pet table, and

it should contain, as a bare minimum, each animal's name. Because the name by itself is not very

interesting, the table should contain other information. For example, if more than one person in your

family keeps pets, you might want to list each animal's owner. You might also want to record some basic

descriptive information such as species and sex.

How about age? That might be of interest, but it is not a good thing to store in a database. Age changes

as time passes, which means you'd have to update your records often. Instead, it is better to store a fixed

value such as date of birth. Then, whenever you need age, you can calculate it as the difference between

the current date and the birth date. MySQL provides functions for doing date arithmetic, so this is not

difficult. Storing birth date rather than age has other advantages, too:

• You can use the database for tasks such as generating reminders for upcoming pet birthdays. (If you

think this type of query is somewhat silly, note that it is the same question you might ask in the context

of a business database to identify clients to whom you need to send out birthday greetings in the current

week or month, for that computer-assisted personal touch.)

• You can calculate age in relation to dates other than the current date. For example, if you store death

date in the database, you can easily calculate how old a pet was when it died.

You can probably think of other types of information that would be useful in the pet table, but the ones

identified so far are sufficient: name, owner, species, sex, birth, and death.

Use a CREATE TABLE statement to specify the layout of your table:

mysql> **CREATE TABLE pet (name VARCHAR(20), owner VARCHAR(20),**

**species VARCHAR(20), sex CHAR(1), birth DATE, death DATE);**

VARCHAR is a good choice for the name, owner, and species columns because the column values vary

in length. The lengths in those column definitions need not all be the same, and need not be 20. You can

normally pick any length from 1 to 65535, whatever seems most reasonable to you. If you make a poor

choice and it turns out later that you need a longer field, MySQL provides an ALTER TABLE statement.

Several types of values can be chosen to represent sex in animal records, such as 'm' and 'f', or

perhaps 'male' and 'female'. It is simplest to use the single characters 'm' and 'f'.

The use of the DATE data type for the birth and death columns is a fairly obvious choice.

Once you have created a table, SHOW TABLES should produce some output:

mysql> **SHOW TABLES;**

+---------------------+

| Tables in menagerie |

+---------------------+

| pet |

+---------------------+

To verify that your table was created the way you expected, use a DESCRIBE statement:

mysql> **DESCRIBE pet;**

+---------+-------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+---------+-------------+------+-----+---------+-------+

| name | varchar(20) | YES | | NULL | |

owner | varchar(20) | YES | | NULL | |

| species | varchar(20) | YES | | NULL | |

| sex | char(1) | YES | | NULL | |

| birth | date | YES | | NULL | |

| death | date | YES | | NULL | |

+---------+-------------+------+-----+---------+-------+

You can use DESCRIBE any time, for example, if you forget the names of the columns in your table or what types they have.

**4.3.3 Loading Data into a Table**

After creating your table, you need to populate it. The LOAD DATA and INSERT statements are useful for this. Suppose that your pet records can be described as shown here. (Observe that MySQL expects dates in 'YYYY-MM-DD' format; this may be different from what you are used to.)

**name owner species sex birth death**

Fluffy Harold cat f 1993-02-04

Claws Gwen cat m 1994-03-17

Buffy Harold dog f 1989-05-13

Fang Benny dog m 1990-08-27

Bowser Diane dog m 1979-08-31 1995-07-29

Chirpy Gwen bird f 1998-09-11

Whistler Gwen bird 1997-12-09

Slim Benny snake m 1996-04-29

Because you are beginning with an empty table, an easy way to populate it is to create a text file

containing a row for each of your animals, then load the contents of the file into the table with a single

statement.

You could create a text file pet.txt containing one record per line, with values separated by tabs, and

given in the order in which the columns were listed in the CREATE TABLE statement. For missing values

(such as unknown sexes or death dates for animals that are still living), you can use NULL values. To

represent these in your text file, use \N (backslash, capital-N). For example, the record for Whistler the bird

would look like this (where the whitespace between values is a single tab character):

Whistler Gwen bird \N 1997-12-09 \N

To load the text file pet.txt into the pet table, use this statement:

mysql> **LOAD DATA LOCAL INFILE '/path/pet.txt' INTO TABLE pet;**

If you created the file on Windows with an editor that uses \r\n as a line terminator, you should use this

statement instead:

mysql> **LOAD DATA LOCAL INFILE '/path/pet.txt' INTO TABLE pet**

-> **LINES TERMINATED BY '\r\n';**

You can specify the column value separator and end of line marker explicitly in the LOAD DATA statement

if you wish, but the defaults are tab and linefeed. These are sufficient for the statement to read the file

pet.txt properly.

If the statement fails, it is likely that your MySQL installation does not have local file capability enabled by

default. See Section 7.1.6, “Security Issues with LOAD DATA LOCAL”, for information on how to change

this.

When you want to add new records one at a time, the INSERT statement is useful. In its simplest form,

you supply values for each column, in the order in which the columns were listed in the CREATE TABLE

statement. Suppose that Diane gets a new hamster named “Puffball.” You could add a new record using

an INSERT statement like this:

mysql> **INSERT INTO pet**

**VALUES ('Puffball','Diane','hamster','f','1999-03-30',NULL);**

String and date values are specified as quoted strings here. Also, with INSERT, you can insert NULL

directly to represent a missing value. You do not use \N like you do with LOAD DATA.

From this example, you should be able to see that there would be a lot more typing involved to load your

records initially using several INSERT statements rather than a single LOAD DATA statement.

**4.3.4 Retrieving Information from a Table**

The SELECT statement is used to pull information from a table. The general form of the statement is:

SELECT *what\_to\_select*

FROM *which\_table*

WHERE *conditions\_to\_satisfy*;

*what\_to\_select* indicates what you want to see. This can be a list of columns, or \* to indicate “all

columns.” *which\_table* indicates the table from which you want to retrieve data. The WHERE clause

is optional. If it is present, *conditions\_to\_satisfy* specifies one or more conditions that rows must

satisfy to qualify for retrieval.

**4.3.4.1 Selecting All Data**

The simplest form of SELECT retrieves everything from a table:

mysql> **SELECT \* FROM pet;**

+----------+--------+---------+------+------------+------------+

| name | owner | species | sex | birth | death |

+----------+--------+---------+------+------------+------------+

| Fluffy | Harold | cat | f | 1993-02-04 | NULL |

| Claws | Gwen | cat | m | 1994-03-17 | NULL |

| Buffy | Harold | dog | f | 1989-05-13 | NULL |

| Fang | Benny | dog | m | 1990-08-27 | NULL |

| Bowser | Diane | dog | m | 1979-08-31 | 1995-07-29 |

| Chirpy | Gwen | bird | f | 1998-09-11 | NULL |

| Whistler | Gwen | bird | NULL | 1997-12-09 | NULL |

| Slim | Benny | snake | m | 1996-04-29 | NULL |

| Puffball | Diane | hamster | f | 1999-03-30 | NULL |

+----------+--------+---------+------+------------+------------+

This form of SELECT is useful if you want to review your entire table, for example, after you've just loaded it

with your initial data set. For example, you may happen to think that the birth date for Bowser doesn't seem quite right. Consulting your original pedigree papers, you find that the correct birth year should be 1989,not 1979.

There are at least two ways to fix this:

• Edit the file pet.txt to correct the error, then empty the table and reload it using DELETE and LOAD

DATA:

mysql> **DELETE FROM pet;**

mysql> **LOAD DATA LOCAL INFILE '/path/pet.txt' INTO TABLE pet;**

However, if you do this, you must also re-enter the record for Puffball.

• Fix only the erroneous record with an UPDATE statement:

mysql> **UPDATE pet SET birth = '1989-08-31' WHERE name = 'Bowser';**

The UPDATE changes only the record in question and does not require you to reload the table.

**4.3.4.2 Selecting Particular Rows**

As shown in the preceding section, it is easy to retrieve an entire table. Just omit the WHERE clause from

the SELECT statement. But typically you don't want to see the entire table, particularly when it becomes

large. Instead, you're usually more interested in answering a particular question, in which case you specify some constraints on the information you want. Let's look at some selection queries in terms of questions about your pets that they answer.

You can select only particular rows from your table. For example, if you want to verify the change that you

made to Bowser's birth date, select Bowser's record like this:

mysql> **SELECT \* FROM pet WHERE name = 'Bowser';**

+--------+-------+---------+------+------------+------------+

| name | owner | species | sex | birth | death |

+--------+-------+---------+------+------------+------------+

| Bowser | Diane | dog | m | 1989-08-31 | 1995-07-29 |

+--------+-------+---------+------+------------+------------+

The output confirms that the year is correctly recorded as 1989, not 1979. String comparisons normally are case-insensitive, so you can specify the name as 'bowser', 'BOWSER', and so forth. The query result is the same.

You can specify conditions on any column, not just name. For example, if you want to know which animals were born during or after 1998, test the birth column:

mysql> **SELECT \* FROM pet WHERE birth >= '1998-1-1';**

+----------+-------+---------+------+------------+-------+

| name | owner | species | sex | birth | death |

+----------+-------+---------+------+------------+-------+

| Chirpy | Gwen | bird | f | 1998-09-11 | NULL |

| Puffball | Diane | hamster | f | 1999-03-30 | NULL |

+----------+-------+---------+------+------------+-------+

You can combine conditions, for example, to locate female dogs:

mysql> **SELECT \* FROM pet WHERE species = 'dog' AND sex = 'f';**

+-------+--------+---------+------+------------+-------+

| name | owner | species | sex | birth | death |

+-------+--------+---------+------+------------+-------+

| Buffy | Harold | dog | f | 1989-05-13 | NULL |

+-------+--------+---------+------+------------+-------+

The preceding query uses the AND logical operator. There is also an OR operator:

mysql> **SELECT \* FROM pet WHERE species = 'snake' OR species = 'bird';**

+----------+-------+---------+------+------------+-------+

| name | owner | species | sex | birth | death |

+----------+-------+---------+------+------------+-------+

| Chirpy | Gwen | bird | f | 1998-09-11 | NULL |

| Whistler | Gwen | bird | NULL | 1997-12-09 | NULL |

| Slim | Benny | snake | m | 1996-04-29 | NULL |

+----------+-------+---------+------+------------+-------+

AND and OR may be intermixed, although AND has higher precedence than OR. If you use both operators, it is a good idea to use parentheses to indicate explicitly how conditions should be grouped:

mysql> **SELECT \* FROM pet WHERE (species = 'cat' AND sex = 'm')**

**OR (species = 'dog' AND sex = 'f');**

+-------+--------+---------+------+------------+-------+

| name | owner | species | sex | birth | death |

+-------+--------+---------+------+------------+-------+

| Claws | Gwen | cat | m | 1994-03-17 | NULL |

| Buffy | Harold | dog | f | 1989-05-13 | NULL |

+-------+--------+---------+------+------------+-------+

**4.3.4.3 Selecting Particular Columns**

If you do not want to see entire rows from your table, just name the columns in which you are interested,

separated by commas. For example, if you want to know when your animals were born, select the name

and birth columns:

mysql> **SELECT name, birth FROM pet;**

+----------+------------+

| name | birth |

+----------+------------+

| Fluffy | 1993-02-04 |

| Claws | 1994-03-17 |

| Buffy | 1989-05-13 |

| Fang | 1990-08-27 |

| Bowser | 1989-08-31 |

| Chirpy | 1998-09-11 |

| Whistler | 1997-12-09 |

| Slim | 1996-04-29 |

| Puffball | 1999-03-30 |

+----------+------------+

To find out who owns pets, use this query:

mysql> **SELECT owner FROM pet;**

+--------+

| owner |

+--------+

| Harold |

| Gwen |

| Harold |

| Benny |

| Diane |

| Gwen |

| Gwen |

| Benny |

| Diane |

+--------+

Notice that the query simply retrieves the owner column from each record, and some of them appear more

than once. To minimize the output, retrieve each unique output record just once by adding the keyword

DISTINCT:

mysql> **SELECT DISTINCT owner FROM pet;**

+--------+

| owner |

+--------+

| Benny |

| Diane |

| Gwen |

| Harold |

+--------+

You can use a WHERE clause to combine row selection with column selection. For example, to get birth

dates for dogs and cats only, use this query:

mysql> **SELECT name, species, birth FROM pet**

**WHERE species = 'dog' OR species = 'cat';**

+--------+---------+------------+

| name | species | birth |

+--------+---------+------------+

| Fluffy | cat | 1993-02-04 |

| Claws | cat | 1994-03-17 |

| Buffy | dog | 1989-05-13 |

| Fang | dog | 1990-08-27 |

| Bowser | dog | 1989-08-31 |

+--------+---------+------------+

**4.3.4.4 Sorting Rows**

You may have noticed in the preceding examples that the result rows are displayed in no particular order. It is often easier to examine query output when the rows are sorted in some meaningful way. To sort a result, use an ORDER BY clause.

Here are animal birthdays, sorted by date:

mysql> **SELECT name, birth FROM pet ORDER BY birth;**

+----------+------------+

| name | birth |

+----------+------------+

| Buffy | 1989-05-13 |

| Bowser | 1989-08-31 |

| Fang | 1990-08-27 |

| Fluffy | 1993-02-04 |

| Claws | 1994-03-17 |

| Slim | 1996-04-29 |

| Whistler | 1997-12-09 |

| Chirpy | 1998-09-11 |

| Puffball | 1999-03-30 |

+----------+------------+

On character type columns, sorting—like all other comparison operations—is normally performed in a

case-insensitive fashion. This means that the order is undefined for columns that are identical except for

their case. You can force a case-sensitive sort for a column by using BINARY like so: ORDER BY BINARY *col\_name*.

The default sort order is ascending, with smallest values first. To sort in reverse (descending) order, add

the DESC keyword to the name of the column you are sorting by:

mysql> **SELECT name, birth FROM pet ORDER BY birth DESC;**

+----------+------------+

| name | birth |

+----------+------------+

| Puffball | 1999-03-30 |

| Chirpy | 1998-09-11 |

| Whistler | 1997-12-09 |

| Slim | 1996-04-29 |

| Claws | 1994-03-17 |

| Fluffy | 1993-02-04 |

| Fang | 1990-08-27 |

| Bowser | 1989-08-31 |

| Buffy | 1989-05-13 |

+----------+------------+

You can sort on multiple columns, and you can sort different columns in different directions. For example,

to sort by type of animal in ascending order, then by birth date within animal type in descending order

(youngest animals first), use the following query:

mysql> **SELECT name, species, birth FROM pet**

**ORDER BY species, birth DESC;**

+----------+---------+------------+

| name | species | birth |

+----------+---------+------------+

| Chirpy | bird | 1998-09-11 |

| Whistler | bird | 1997-12-09 |

| Claws | cat | 1994-03-17 |

| Fluffy | cat | 1993-02-04 |

| Fang | dog | 1990-08-27 |

| Bowser | dog | 1989-08-31 |

| Buffy | dog | 1989-05-13 |

| Puffball | hamster | 1999-03-30 |

| Slim | snake | 1996-04-29 |

+----------+---------+------------+

The DESC keyword applies only to the column name immediately preceding it (birth); it does not affect

the species column sort order.

**4.3.4.5 Date Calculations**

MySQL provides several functions that you can use to perform calculations on dates, for example, to

calculate ages or extract parts of dates.

To determine how many years old each of your pets is, use the TIMESTAMPDIFF() function. Its

arguments are the unit in which you want the result expressed, and the two date for which to take the

difference. The following query shows, for each pet, the birth date, the current date, and the age in years.

An *alias* (age) is used to make the final output column label more meaningful.

mysql> **SELECT name, birth, CURDATE(),**

**TIMESTAMPDIFF(YEAR,birth,CURDATE()) AS age**

**FROM pet;**

+----------+------------+------------+------+

| name | birth | CURDATE() | age |

+----------+------------+------------+------+

| Fluffy | 1993-02-04 | 2003-08-19 | 10 |

| Claws | 1994-03-17 | 2003-08-19 | 9 |

| Buffy | 1989-05-13 | 2003-08-19 | 14 |

| Fang | 1990-08-27 | 2003-08-19 | 12 |  
| Bowser | 1989-08-31 | 2003-08-19 | 13 |

| Chirpy | 1998-09-11 | 2003-08-19 | 4 |

| Whistler | 1997-12-09 | 2003-08-19 | 5 |

| Slim | 1996-04-29 | 2003-08-19 | 7 |

| Puffball | 1999-03-30 | 2003-08-19 | 4 |

+----------+------------+------------+------+

The query works, but the result could be scanned more easily if the rows were presented in some order.

This can be done by adding an ORDER BY name clause to sort the output by name:

mysql> **SELECT name, birth, CURDATE(),**

**TIMESTAMPDIFF(YEAR,birth,CURDATE()) AS age**

**FROM pet ORDER BY name;**

+----------+------------+------------+------+

| name | birth | CURDATE() | age |

+----------+------------+------------+------+

| Bowser | 1989-08-31 | 2003-08-19 | 13 |

| Buffy | 1989-05-13 | 2003-08-19 | 14 |

| Chirpy | 1998-09-11 | 2003-08-19 | 4 |

| Claws | 1994-03-17 | 2003-08-19 | 9 |

| Fang | 1990-08-27 | 2003-08-19 | 12 |

| Fluffy | 1993-02-04 | 2003-08-19 | 10 |

| Puffball | 1999-03-30 | 2003-08-19 | 4 |

| Slim | 1996-04-29 | 2003-08-19 | 7 |

| Whistler | 1997-12-09 | 2003-08-19 | 5 |

+----------+------------+------------+------+

To sort the output by age rather than name, just use a different ORDER BY clause:

mysql> **SELECT name, birth, CURDATE(),**

-> **TIMESTAMPDIFF(YEAR,birth,CURDATE()) AS age**

-> **FROM pet ORDER BY age;**

+----------+------------+------------+------+

| name | birth | CURDATE() | age |

+----------+------------+------------+------+

| Chirpy | 1998-09-11 | 2003-08-19 | 4 |

| Puffball | 1999-03-30 | 2003-08-19 | 4 |

| Whistler | 1997-12-09 | 2003-08-19 | 5 |

| Slim | 1996-04-29 | 2003-08-19 | 7 |

| Claws | 1994-03-17 | 2003-08-19 | 9 |

| Fluffy | 1993-02-04 | 2003-08-19 | 10 |

| Fang | 1990-08-27 | 2003-08-19 | 12 |

| Bowser | 1989-08-31 | 2003-08-19 | 13 |

| Buffy | 1989-05-13 | 2003-08-19 | 14 |

+----------+------------+------------+------+

A similar query can be used to determine age at death for animals that have died. You determine which

animals these are by checking whether the death value is NULL. Then, for those with non-NULL values,

compute the difference between the death and birth values:

mysql> **SELECT name, birth, death,**

**TIMESTAMPDIFF(YEAR,birth,death) AS age**

**FROM pet WHERE death IS NOT NULL ORDER BY age;**

+--------+------------+------------+------+

| name | birth | death | age |

+--------+------------+------------+------+

| Bowser | 1989-08-31 | 1995-07-29 | 5 |

+--------+------------+------------+------+

The query uses death IS NOT NULL rather than death <> NULL because NULL is a special value that cannot be compared using the usual comparison operators. This is discussed later.

What if you want to know which animals have birthdays next month? For this type of calculation, year

and day are irrelevant; you simply want to extract the month part of the birth column. MySQL provides

several functions for extracting parts of dates, such as YEAR(), MONTH(), and DAYOFMONTH(). MONTH()is the appropriate function here. To see how it works, run a simple query that displays the value of both birth and MONTH(birth):

mysql> **SELECT name, birth, MONTH(birth) FROM pet;**

+----------+------------+--------------+

| name | birth | MONTH(birth) |

+----------+------------+--------------+

| Fluffy | 1993-02-04 | 2 |

| Claws | 1994-03-17 | 3 |

| Buffy | 1989-05-13 | 5 |

| Fang | 1990-08-27 | 8 |

| Bowser | 1989-08-31 | 8 |

| Chirpy | 1998-09-11 | 9 |

| Whistler | 1997-12-09 | 12 |

| Slim | 1996-04-29 | 4 |

| Puffball | 1999-03-30 | 3 |

+----------+------------+--------------+

Finding animals with birthdays in the upcoming month is also simple. Suppose that the current month is

April. Then the month value is 4 and you can look for animals born in May (month 5) like this:

mysql> **SELECT name, birth FROM pet WHERE MONTH(birth) = 5;**

+-------+------------+

| name | birth |

+-------+------------+

| Buffy | 1989-05-13 |

+-------+------------+

There is a small complication if the current month is December. You cannot merely add one to the month

number (12) and look for animals born in month 13, because there is no such month. Instead, you look for animals born in January (month 1).

You can write the query so that it works no matter what the current month is, so that you do not have to

use the number for a particular month. DATE\_ADD() enables you to add a time interval to a given date.

If you add a month to the value of CURDATE(), then extract the month part with MONTH(), the result

produces the month in which to look for birthdays:

mysql> **SELECT name, birth FROM pet**

-> **WHERE MONTH(birth) = MONTH(DATE\_ADD(CURDATE(),INTERVAL 1 MONTH));**

A different way to accomplish the same task is to add 1 to get the next month after the current one after

using the modulo function (MOD) to wrap the month value to 0 if it is currently 12:

mysql> **SELECT name, birth FROM pet**

-> **WHERE MONTH(birth) = MOD(MONTH(CURDATE()), 12) + 1;**

MONTH() returns a number between 1 and 12. And MOD(something,12)returns a number between 0

and 11. So the addition has to be after the MOD(), otherwise we would go from November (11) to January (1).

**4.3.4.6 Working with NULL Values**

The NULL value can be surprising until you get used to it. Conceptually, NULL means “a missing unknown value” and it is treated somewhat differently from other values.

To test for NULL, use the IS NULL and IS NOT NULL operators, as shown here:

mysql> **SELECT 1 IS NULL, 1 IS NOT NULL;**

+-----------+---------------+

| 1 IS NULL | 1 IS NOT NULL |

+-----------+---------------+

| 0 | 1 |

+-----------+---------------+

You cannot use arithmetic comparison operators such as =, <, or <> to test for NULL. To demonstrate this for yourself, try the following query:

mysql> **SELECT 1 = NULL, 1 <> NULL, 1 < NULL, 1 > NULL;**

+----------+-----------+----------+----------+

| 1 = NULL | 1 <> NULL | 1 < NULL | 1 > NULL |

+----------+-----------+----------+----------+

| NULL | NULL | NULL | NULL |

+----------+-----------+----------+----------+

Because the result of any arithmetic comparison with NULL is also NULL, you cannot obtain any meaningful results from such comparisons. In MySQL, 0 or NULL means false and anything else means true. The default truth value from a Boolean operation is 1.

This special treatment of NULL is why, in the previous section, it was necessary to determine which

animals are no longer alive using death IS NOT NULL instead of death <> NULL.

Two NULL values are regarded as equal in a GROUP BY.

When doing an ORDER BY, NULL values are presented first if you do ORDER BY ... ASC and last if you do ORDER BY ... DESC.

A common error when working with NULL is to assume that it is not possible to insert a zero or an empty

string into a column defined as NOT NULL, but this is not the case. These are in fact values, whereas NULL means “not having a value.” You can test this easily enough by using IS [NOT] NULL as shown:

mysql> **SELECT 0 IS NULL, 0 IS NOT NULL, '' IS NULL, '' IS NOT NULL;**

+-----------+---------------+------------+----------------+

| 0 IS NULL | 0 IS NOT NULL | '' IS NULL | '' IS NOT NULL |

+-----------+---------------+------------+----------------+

| 0 | 1 | 0 | 1 |

+-----------+---------------+------------+----------------+

Thus it is entirely possible to insert a zero or empty string into a NOT NULL column, as these are in fact

NOT NULL.

**4.3.4.7 Pattern Matching**

MySQL provides standard SQL pattern matching as well as a form of pattern matching based on extended regular expressions similar to those used by Unix utilities such as vi, grep, and sed.

SQL pattern matching enables you to use \_ to match any single character and % to match an arbitrary

number of characters (including zero characters). In MySQL, SQL patterns are case-insensitive by default.

Some examples are shown here. You do not use = or <> when you use SQL patterns; use the LIKE or

NOT LIKE comparison operators instead.

To find names beginning with b:

mysql> **SELECT \* FROM pet WHERE name LIKE 'b%';**

+--------+--------+---------+------+------------+------------+

| name | owner | species | sex | birth | death |

+--------+--------+---------+------+------------+------------+

| Buffy | Harold | dog | f | 1989-05-13 | NULL |

| Bowser | Diane | dog | m | 1989-08-31 | 1995-07-29 |

+--------+--------+---------+------+------------+------------+

To find names ending with fy:

mysql> **SELECT \* FROM pet WHERE name LIKE '%fy';**

+--------+--------+---------+------+------------+-------+

| name | owner | species | sex | birth | death |

+--------+--------+---------+------+------------+-------+

| Fluffy | Harold | cat | f | 1993-02-04 | NULL |

| Buffy | Harold | dog | f | 1989-05-13 | NULL |

+--------+--------+---------+------+------------+-------+

To find names containing a w:

mysql> **SELECT \* FROM pet WHERE name LIKE '%w%';**

+----------+-------+---------+------+------------+------------+

| name | owner | species | sex | birth | death |

+----------+-------+---------+------+------------+------------+

| Claws | Gwen | cat | m | 1994-03-17 | NULL |

| Bowser | Diane | dog | m | 1989-08-31 | 1995-07-29 |

| Whistler | Gwen | bird | NULL | 1997-12-09 | NULL |

+----------+-------+---------+------+------------+------------+

To find names containing exactly five characters, use five instances of the \_ pattern character:

mysql> **SELECT \* FROM pet WHERE name LIKE '\_\_\_\_\_';**

+-------+--------+---------+------+------------+-------+

| name | owner | species | sex | birth | death |

+-------+--------+---------+------+------------+-------+

| Claws | Gwen | cat | m | 1994-03-17 | NULL |

| Buffy | Harold | dog | f | 1989-05-13 | NULL |

+-------+--------+---------+------+------------+-------+

The other type of pattern matching provided by MySQL uses extended regular expressions. When you

test for a match for this type of pattern, use the REGEXP and NOT REGEXP operators (or RLIKE and NOT RLIKE, which are synonyms).

The following list describes some characteristics of extended regular expressions:

• . matches any single character.

• A character class [...] matches any character within the brackets. For example, [abc] matches a, b,or c. To name a range of characters, use a dash. [a-z] matches any letter, whereas [0-9] matches

any digit.

• \* matches zero or more instances of the thing preceding it. For example, x\* matches any number of x

characters, [0-9]\* matches any number of digits, and .\* matches any number of anything.

• A REGEXP pattern match succeeds if the pattern matches anywhere in the value being tested. (This

differs from a LIKE pattern match, which succeeds only if the pattern matches the entire value.)

• To anchor a pattern so that it must match the beginning or end of the value being tested, use ^ at the

beginning or $ at the end of the pattern.

To demonstrate how extended regular expressions work, the LIKE queries shown previously are rewritten here to use REGEXP. To find names beginning with b, use ^ to match the beginning of the name:

mysql> **SELECT \* FROM pet WHERE name REGEXP '^b';**

+--------+--------+---------+------+------------+------------+

| name | owner | species | sex | birth | death |

+--------+--------+---------+------+------------+------------+

| Buffy | Harold | dog | f | 1989-05-13 | NULL |

| Bowser | Diane | dog | m | 1989-08-31 | 1995-07-29 |

+--------+--------+---------+------+------------+------------+

If you really want to force a REGEXP comparison to be case sensitive, use the BINARY keyword to make

one of the strings a binary string. This query matches only lowercase b at the beginning of a name:

mysql> **SELECT \* FROM pet WHERE name REGEXP BINARY '^b';**

To find names ending with fy, use $ to match the end of the name:

mysql> **SELECT \* FROM pet WHERE name REGEXP 'fy$';**

+--------+--------+---------+------+------------+-------+

| name | owner | species | sex | birth | death |

+--------+--------+---------+------+------------+-------+

| Fluffy | Harold | cat | f | 1993-02-04 | NULL |

| Buffy | Harold | dog | f | 1989-05-13 | NULL |

+--------+--------+---------+------+------------+-------+

To find names containing a w, use this query:

mysql> **SELECT \* FROM pet WHERE name REGEXP 'w';**

+----------+-------+---------+------+------------+------------+

| name | owner | species | sex | birth | death |

+----------+-------+---------+------+------------+------------+

| Claws | Gwen | cat | m | 1994-03-17 | NULL |

| Bowser | Diane | dog | m | 1989-08-31 | 1995-07-29 |

| Whistler | Gwen | bird | NULL | 1997-12-09 | NULL |

+----------+-------+---------+------+------------+------------+

Because a regular expression pattern matches if it occurs anywhere in the value, it is not necessary in the

previous query to put a wildcard on either side of the pattern to get it to match the entire value like it would be if you used an SQL pattern.

To find names containing exactly five characters, use ^ and $ to match the beginning and end of the name,and five instances of .in between:

mysql> **SELECT \* FROM pet WHERE name REGEXP '^.....$';**

+-------+--------+---------+------+------------+-------+

| name | owner | species | sex | birth | death |

+-------+--------+---------+------+------------+-------+

| Claws | Gwen | cat | m | 1994-03-17 | NULL |

| Buffy | Harold | dog | f | 1989-05-13 | NULL |

+-------+--------+---------+------+------------+-------+

You could also write the previous query using the {*n*} (“repeat-*n*-times”) operator:

mysql> **SELECT \* FROM pet WHERE name REGEXP '^.{5}$';**

+-------+--------+---------+------+------------+-------+

| name | owner | species | sex | birth | death |

+-------+--------+---------+------+------------+-------+

| Claws | Gwen | cat | m | 1994-03-17 | NULL |

| Buffy | Harold | dog | f | 1989-05-13 | NULL |

+-------+--------+---------+------+------------+-------+

**4.3.4.8 Counting Rows**

Databases are often used to answer the question, “How often does a certain type of data occur in a table?” For example, you might want to know how many pets you have, or how many pets each owner has, or you might want to perform various kinds of census operations on your animals.

Counting the total number of animals you have is the same question as “How many rows are in the pet

table?” because there is one record per pet. COUNT(\*) counts the number of rows, so the query to count

your animals looks like this:

mysql> **SELECT COUNT(\*) FROM pet;**

+----------+

| COUNT(\*) |

+----------+

| 9 |

+----------+

Earlier, you retrieved the names of the people who owned pets. You can use COUNT() if you want to find

out how many pets each owner has:

mysql> **SELECT owner, COUNT(\*) FROM pet GROUP BY owner;**

+--------+----------+

| owner | COUNT(\*) |

+--------+----------+

| Benny | 2 |

| Diane | 2 |

| Gwen | 3 |

| Harold | 2 |

+--------+----------+

The preceding query uses GROUP BY to group all records for each owner. The use of COUNT() in

conjunction with GROUP BY is useful for characterizing your data under various groupings. The following

examples show different ways to perform animal census operations.

Number of animals per species:

mysql> **SELECT species, COUNT(\*) FROM pet GROUP BY species;**

+---------+----------+

| species | COUNT(\*) |

+---------+----------+

| bird | 2 |

| cat | 2 |

| dog | 3 |

| hamster | 1 |

| snake | 1 |

+---------+----------+

Number of animals per sex:

mysql> **SELECT sex, COUNT(\*) FROM pet GROUP BY sex;**

| sex | COUNT(\*) |

+------+----------+

| NULL | 1 |

| f | 4 |

| m | 4 |

+------+----------+

(In this output, NULL indicates that the sex is unknown.) Number of animals per combination of species and sex:

mysql> **SELECT species, sex, COUNT(\*) FROM pet GROUP BY species, sex;**

+---------+------+----------+

| species | sex | COUNT(\*) |

+---------+------+----------+

| bird | NULL | 1 |

| bird | f | 1 |

| cat | f | 1 |

| cat | m | 1 |

| dog | f | 1 |

| dog | m | 2 |

| hamster | f | 1 |

| snake | m | 1 |

+---------+------+----------+

You need not retrieve an entire table when you use COUNT(). For example, the previous query, when

performed just on dogs and cats, looks like this:

mysql> **SELECT species, sex, COUNT(\*) FROM pet**

**WHERE species = 'dog' OR species = 'cat'**

**GROUP BY species, sex;**

+---------+------+----------+

| species | sex | COUNT(\*) |

+---------+------+----------+

| cat | f | 1 |

| cat | m | 1 |

| dog | f | 1 |

| dog | m | 2 |

+---------+------+----------+

Or, if you wanted the number of animals per sex only for animals whose sex is known:

mysql> **SELECT species, sex, COUNT(\*) FROM pet**

-> **WHERE sex IS NOT NULL**

-> **GROUP BY species, sex;**

+---------+------+----------+

| species | sex | COUNT(\*) |

+---------+------+----------+

| bird | f | 1 |

| cat | f | 1 |

| cat | m | 1 |

| dog | f | 1 |

| dog | m | 2 |

| hamster | f | 1 |

| snake | m | 1 |

+---------+------+----------+

If you name columns to select in addition to the COUNT() value, a GROUP BY clause should be present

that names those same columns. Otherwise, the following occurs:

• If the ONLY\_FULL\_GROUP\_BY SQL mode is enabled, an error occurs:

mysql> **SET sql\_mode = 'ONLY\_FULL\_GROUP\_BY';**

Query OK, 0 rows affected (0.00 sec)

mysql> **SELECT owner, COUNT(\*) FROM pet;**

ERROR 1140 (42000): In aggregated query without GROUP BY, expression

#1 of SELECT list contains nonaggregated column 'menagerie.pet.owner';

this is incompatible with sql\_mode=only\_full\_group\_by

• If ONLY\_FULL\_GROUP\_BY is not enabled, the query is processed by treating all rows as a single group,

but the value selected for each named column is indeterminate. The server is free to select the value

from any row:

mysql> **SET sql\_mode = '';**

Query OK, 0 rows affected (0.00 sec)

mysql> **SELECT owner, COUNT(\*) FROM pet;**

+--------+----------+

| owner | COUNT(\*) |

+--------+----------+

| Harold | 8 |

+--------+----------+

1 row in set (0.00 sec)