

MySQL Tutorial

This chapter provides a tutorial introduction to **MySQL** by showing how to use the `mysql` client program to create and use a simple database. `mysql` (sometimes referred to as the "terminal monitor" or just "monitor") is an interactive program that allows you to connect to a **MySQL** server, run queries and view the results. `mysql` may also be used in batch mode: you place your queries in a file beforehand, then tell `mysql` to execute the contents of the file. Both ways of using `mysql` are covered here.

To see a list of options provided by `mysql`, invoke it with the `--help` option:

```
shell> mysql --help
```

This chapter assumes that `mysql` is installed on your machine, and that a **MySQL** server is available to which you can connect. If this is not true, contact your **MySQL** administrator. (If *you* are the administrator, you will need to consult other sections of this manual.)

The chapter describes the entire process of setting up and using a database. If you are interested only in accessing an already-existing database, you may want to skip over the sections that describe how to create the database and the tables it contains.

Since this chapter is tutorial in nature, many details are necessarily left out. Consult the relevant sections of the manual for more information on the topics covered here.

Connecting to and disconnecting from the server

To connect to the server, you'll 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'll also need to specify a hostname. Contact your administrator to find out what connection parameters you should use to connect (i.e., 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: *****
```

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 459 to server version: 3.22.20a-log

Type 'help' for help.

mysql>
```

The prompt tells you that `mysql` is ready for you to enter commands.

Some **MySQL** installations allow 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` at the `mysql>` prompt:

```
mysql> QUIT
Bye
```

You can also disconnect by typing control-D.

Most examples in the following sections assume you are connected to the server. They indicate this by the `mysql>` prompt.

Entering queries

Make sure you are connected to the server, as discussed in the previous section. Doing so will not in itself select any database to work with, but that's okay. At this point, it's 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 commands, using several queries you can try out to familiarize yourself with how `mysql` works.

Here's a simple command that asks the server to tell you its version number and the current date. Type it in as shown below following the `mysql>` prompt and hit the RETURN key:

```
mysql> SELECT VERSION(), CURRENT_DATE;
+-----+-----+
| version() | CURRENT_DATE |
+-----+-----+
| 3.22.20a-log | 1999-03-19 |
+-----+-----+
1 row in set (0.01 sec)
mysql>
```

This query illustrates several things about `mysql`:

- A command normally consists of a SQL statement followed by a semicolon. (There are some exceptions where a semicolon is not needed. `QUIT`, mentioned earlier, is one of them. We'll get to others later.)
- When you issue a command, `mysql` sends it to the server for execution and displays the results, then prints another `mysql>` to indicate that it is ready for another command.
- `mysql` displays query output as a table (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 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's 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 |
+-----+-----+
| 0.707107 | 25 |
+-----+-----+
```

The commands 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() |
+-----+
| 3.22.20a-log |
+-----+

+-----+
| NOW() |
+-----+
| 1999-03-19 00:15:33 |
+-----+
```

A command need not be given all on a single line, so lengthy commands 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's a simple multiple-line statement:

```
mysql> SELECT
-> USER()
-> ,
-> CURRENT_DATE;
+-----+-----+
| USER() | CURRENT_DATE |
+-----+-----+
| joesmith@localhost | 1999-03-18 |
+-----+-----+
```

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 hasn't 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 will always be aware of what `mysql` is waiting for.

If you decide you don't want to execute a command 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 command.

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
<code>mysql></code>	Ready for new command
<code>-></code>	Waiting for next line of multiple-line command
<code>'></code>	Waiting for next line, collecting a string that begins with a single quote (<code>`''</code>)
<code>"></code>	Waiting for next line, collecting a string that begins with a double quote (<code>`""</code>)

Multiple-line statements commonly occur "by accident" when you intend to issue a command 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` will execute it:

```
mysql> SELECT USER()
-> ;
+-----+
| USER() |
+-----+
| joesmith@localhost |
+-----+
```

The `'>` and `">` prompts occur during string collection. 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've 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. That's fine if you really are entering a multiple-line string, but how likely is that? Not very. More often, the `'>` and `">` prompts indicate that you've 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 hit RETURN and wait for the result, nothing will happen. Instead of wondering, "why does this query take 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 quote.)

At this point, what do you do? The simplest thing is to cancel the command. 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 command.

It's important to know what the `'>` and `">` prompts signify, because if you mistakenly enter an unterminated string, any further lines you type will appear to be ignored by `mysql` -- including a line containing `QUIT!` This can be quite confusing, especially if you don't know that you need to supply the terminating quote before you can cancel the current command.

Creating and using a database

Now that you know how to enter commands, it's time to access a database.

Suppose you have several pets in your home (your "menagerie") and you'd 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 how to do all that:

- How to create a database
- How to create a table
- How to load data into the table
- How to retrieve data from the table in various ways
- How to use multiple tables

The menagerie database will be 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.

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

```
mysql> SHOW DATABASES;  
+-----+  
| Database |  
+-----+  
| mysql    |  
| test     |  
| tmp      |  
+-----+
```

The list of databases is probably different on your machine, but the `mysql` and `test` databases are likely to be among them. The `mysql` database is required because it describes user access privileges. The `test` database is often provided as a workspace for users to try things out.

If the `test` database exists, try to access it:

```
mysql> USE test
Database changed
```

Note that `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 you want to call yours `menagerie`. The administrator needs to execute a command like this:

```
mysql> GRANT ALL ON menagerie.* TO your_mysql_name;
```

where `your_mysql_name` is the **MySQL** user name assigned to you.

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.)

Creating a database does not select it for use, you must do that explicitly. To make `menagerie` the current database, use this command:

```
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 above. 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: *****
```

Note that `menagerie` is not your password on the command just shown. If you want to supply your password on the command line after the `-p` option, you must do so with no intervening space (e.g., 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.

Creating a table

Creating the database is the easy part, but at this point it's empty, as `SHOW TABLES` will tell 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 will need, and what columns will be in each of them.

You'll 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. Since 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's 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's 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'll soon need to send out birthday greetings, 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 for now: 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 since the column values will vary in length. The lengths of those columns need not all be the same, and need not be 20. You can pick any length from 1 to 255, 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.)

Animal sex can be represented in a variety of ways, for example, "m" and "f", or perhaps "male" and "female". It's 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.

Now that 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 are.

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 your pet records can be described as shown below. (Observe that **MySQL** expects dates in `YYYY-MM-DD` format; this may be different than 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	1998-08-31	1995-07-29
Chirpy	Gwen	bird	f	1998-09-11	
Whistler	Gwen	bird		1997-12-09	
Slim	Benny	snake	m	1996-04-29	

Since 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`. 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 command:

```
mysql> LOAD DATA LOCAL INFILE "pet.txt" INTO TABLE pet;
```

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.

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 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);
```

Note that 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.

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's present, `conditions_to_satisfy` specifies conditions that rows must satisfy to qualify for retrieval.

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	1998-08-31	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 instance, after you've just loaded it with your initial dataset. As it happens, the output just shown reveals an error in your data file: Bowser appears to have been born after he died! Consulting your original pedigree papers, you find that the correct birth year is 1989, not 1998.

There are at least a couple of 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 "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";
```

As shown above, it is easy to retrieve an entire table. But typically you don't want to do that, particularly when the table 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.

Selecting particular rows

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 now as 1989, not 1998.

String comparisons are normally case-insensitive, so you can specify the name as "bowser", "BOWSER", etc. The query result will be the same.

You can specify conditions on any column, not just `name`. For example, if you want to know which animals were born 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. If you do that, it's a good idea to use parentheses to indicate 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

Selecting particular columns

If you don't want to see entire rows from your table, just name the columns in which you're 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

However, notice that the query simply retrieves the `owner` field 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

Sorting rows

You may have noticed in the preceding examples that the result rows are displayed in no particular order. However, it's 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

To sort in reverse order, add the `DESC` (descending) 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. For example, to sort by type of animal, then by birth date within animal type with 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

Note that the `DESC` keyword applies only to the column name immediately preceding it (`birth`); `species` values are still sorted in ascending order.

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, compute age as the difference between the birth date and the current date. Do this by converting the two dates to days, take the difference, and divide by 365 (the number of days in a year):

```
mysql> SELECT name, (TO_DAYS(NOW())-TO_DAYS(birth))/365 FROM pet;
```

name	(TO_DAYS(NOW())-TO_DAYS(birth))/365
Fluffy	6.15
Claws	5.04
Buffy	9.88
Fang	8.59
Bowser	9.58
Chirpy	0.55
Whistler	1.30
Slim	2.92
Puffball	0.00

Although the query works, there are some things about it that could be improved. First, the result could be scanned more easily if the rows were presented in some order. Second, the heading for the age column isn't very meaningful.

The first problem can be handled by adding an `ORDER BY name` clause to sort the output by name. To deal with the column heading, provide a name for the column so that a different label appears in the output (this is called a column alias):

```
mysql> SELECT name, (TO_DAYS(NOW())-TO_DAYS(birth))/365 AS age
-> FROM pet ORDER BY name;
```

name	age
Bowser	9.58
Buffy	9.88
Chirpy	0.55
Claws	5.04
Fang	8.59
Fluffy	6.15
Puffball	0.00
Slim	2.92
Whistler	1.30

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

```
mysql> SELECT name, (TO_DAYS(NOW())-TO_DAYS(birth))/365 AS age
-> FROM pet ORDER BY age;
```

name	age
Puffball	0.00
Chirpy	0.55
Whistler	1.30
Slim	2.92
Claws	5.04
Fluffy	6.15
Fang	8.59
Bowser	9.58
Buffy	9.88

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 or not 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, (TO_DAYS(death)-TO_DAYS(birth))/365 AS age
-> FROM pet WHERE death IS NOT NULL ORDER BY age;
```

name	birth	death	age
Bowser	1989-08-31	1995-07-29	5.91

The query uses `death IS NOT NULL` rather than `death != NULL` because `NULL` is a special value. This is explained later. See section: [Working with NULL values](#).

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 date-part extraction functions, such as `YEAR()`, `MONTH()` and `DAY()`. `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 easy, too. Suppose the current month is April. Then the month value is 4 and you 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, of course. You don't just 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 even write the query so that it works no matter what the current month is. That way you don't have to use a particular month number in the query. `DATE_ADD()` allows you to add a time interval to a given date. If you add a month to the value of `NOW()`, 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(NOW(), 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 around" the month value to 0 if it is currently 12):

```
mysql> SELECT name, birth FROM pet
-> WHERE MONTH(birth) = MOD(MONTH(NOW()),12) + 1;
```

Working with NULL values

The `NULL` value can be surprising until you get used to it. Conceptually, `NULL` means "missing value" or "unknown value" and it is treated somewhat differently than other values. To test for `NULL`, you cannot use the arithmetic comparison operators such as `=`, `<` or `!=`. 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    |
+-----+-----+-----+-----+
```

Clearly you get no meaningful results from these comparisons. Use the `IS NULL` and `IS NOT NULL` operators instead:

```
mysql> SELECT 1 IS NULL, 1 IS NOT NULL;
+-----+-----+
| 1 IS NULL | 1 IS NOT NULL |
+-----+-----+
|          0 |                1 |
+-----+-----+
```

In **MySQL**, 0 means false and 1 means true.

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`.

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 allows you to use ``_`` to match any single character, and ``%`` to match an arbitrary number of characters (including zero characters). SQL patterns are case insensitive. Some examples are shown below. Note that 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 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).

Some characteristics of extended regular expressions are:

- `'.'` 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 lowercase 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.
- Regular expressions are case sensitive, but you can use a character class to match both lettercases if you wish. For example, `'[aA]'` matches lowercase or uppercase `'a'` and `'[a-zA-Z]'` matches any letter in either case.
- The pattern matches if it occurs anywhere in the value being tested (SQL patterns match only if they match 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 above are rewritten below to use `REGEXP`:

To find names beginning with `'b'`, use `'^'` to match the beginning of the name and `'[bB]'` to match either lowercase or uppercase `'b'`:

```
mysql> SELECT * FROM pet WHERE name REGEXP "^[bB]";
```

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'`, 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 '[wW]' to match either lowercase or uppercase 'w':

```
mysql> SELECT * FROM pet WHERE name REGEXP "[wW]";
```

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

Since 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

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 censuses on your animals.

Counting the total number of animals you have is the same question as "how many rows are in the pet table?" since there is one record per pet. The COUNT() function counts the number of non-NULL results, 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

Note the use of `GROUP BY` to group together all records for each `owner`. Without it, all you get is an error message:

```
mysql> SELECT owner, COUNT(owner) FROM pet;
ERROR 1140 at line 1: Mixing of GROUP columns (MIN(),MAX(),COUNT()...)
with no GROUP columns is illegal if there is no GROUP BY clause
```

`COUNT()` and `GROUP BY` are useful for characterizing your data in various ways. 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 "sex 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 known-sex animals:

```
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

Using more than one table

The `pet` table keeps track of which pets you have. If you want to record other information about them, such as events in their lives like visits to the vet or when litters are born, you need another table. What should this table look like?

- It needs to contain the pet name so you know which animal each event pertains to.
- It needs a date so you know when the event occurred.
- It needs a field to describe the event.
- If you want to be able to categorize events, it would be useful to have an event type field.

Given these considerations, the `CREATE TABLE` statement for the `event` table might look like this:

```
mysql> CREATE TABLE event (name VARCHAR(20), date DATE,
-> type VARCHAR(15), remark VARCHAR(255));
```

As with the `pet` table, it's easiest to load the initial records by creating a tab-delimited text file containing the information:

Fluffy	1995-05-15	litter	4 kittens, 3 female, 1 male
Buffy	1993-06-23	litter	5 puppies, 2 female, 3 male
Buffy	1994-06-19	litter	3 puppies, 3 female
Chirpy	1999-03-21	vet	needed beak straightened
Slim	1997-08-03	vet	broken rib
Bowser	1991-10-12	kennel	
Fang	1991-10-12	kennel	
Fang	1998-08-28	birthday	Gave him a new chew toy
Claws	1998-03-17	birthday	Gave him a new flea collar
Whistler	1998-12-09	birthday	First birthday

Load the records like this:

```
mysql> LOAD DATA LOCAL INFILE "event.txt" INTO TABLE event;
```

Based on what you've learned from the queries you've run on the `pet` table, you should be able to perform retrievals on the records in the `event` table; the principles are the same. But when is the `event` table by itself insufficient to answer questions you might ask?

Suppose you want to find out the ages of each pet when they had their litters. The `event` table indicates when this occurred, but to calculate age of the mother, you need her birth date. Since that is stored in the `pet` table, you need both tables for the query:

```
mysql> SELECT pet.name, (TO_DAYS(date) - TO_DAYS(birth))/365 AS age, remark
-> FROM pet, event
-> WHERE pet.name = event.name AND type = "litter";
```

name	age	remark
Fluffy	2.27	4 kittens, 3 female, 1 male
Buffy	4.12	5 puppies, 2 female, 3 male
Buffy	5.10	3 puppies, 3 female

There are several things to note about this query:

- The `FROM` clause lists two tables since the query needs to pull information from both of them.
- When combining information from multiple tables, you need to specify how records in one table can be matched to records in the other. This is easy since they both have a `name` column. The query uses `WHERE` clause to match up records in the two tables based on the `name` values.
- Since the `name` column occurs in both tables, you must be specific about which table you mean when referring to the column. This is done by prepending the table name to the column name.

You need not have two different tables to perform a join. Sometimes it is useful to join a table to itself, if you want to compare records in a table to other records in that same table. For example, to find breeding pairs among your pets, you can join the `pet` table with itself to pair up males and females of like species:

```
mysql> SELECT p1.name, p1.sex, p2.name, p2.sex, p1.species
-> FROM pet AS p1, pet AS p2
-> WHERE p1.species = p2.species AND p1.sex = "f" AND p2.sex = "m";
```

name	sex	name	sex	species
Fluffy	f	Claws	m	cat
Buffy	f	Fang	m	dog
Buffy	f	Bowser	m	dog

In this query, we specify aliases for the table name in order to be able to refer to the columns and keep straight which instance of the table each column reference is associated with.

Getting information about databases and tables

What if you forget the name of a database or table, or what the structure of a given table is (e.g., what its columns are called)? **MySQL** addresses this problem through several statements that provide information about the databases and tables it supports.

You have already seen `SHOW DATABASES`, which lists the databases managed by the server. To find out which database is currently selected, use the `DATABASE()` function:

```
mysql> SELECT DATABASE();
```

DATABASE()
menagerie

If you haven't selected any database yet, the result is blank.

To find out what tables the current database contains (for example, when you're not sure about the name of a table), use this command:

```
mysql> SHOW TABLES;
```

Tables in menagerie
event
pet

If you want to find out about the structure of a table, the `DESCRIBE` command is useful; it displays information about each of a table's columns:

```
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	

`Field` indicates the column name, `Type` is the data type for the column, `Null` indicates whether or not the column can contain `NULL` values, `Key` indicates whether or not the column is indexed and `Default`

specifies the column's default value.

If you have indexes on a table, `SHOW INDEX FROM tbl_name` produces information about them.

Using `mysql` in batch mode

In the previous sections, you used `mysql` interactively to enter queries and view the results. You can also run `mysql` in batch mode. To do this, put the commands you want to run in a file, then tell `mysql` to read its input from the file:

```
shell> mysql < batch-file
```

If you need to specify connection parameters on the command line, the command might look like this:

```
shell> mysql -h host -u user -p < batch-file
Enter password: *****
```

When you use `mysql` this way, you are creating a script file, then executing the script.

Why use a script? Here are a few reasons:

- If you run a query repeatedly (say, every day or every week), making it a script allows you to avoid retyping it each time you execute it.
- You can generate new queries from existing ones that are similar by copying and editing script files.
- Batch mode can also be useful while you're developing a query, particularly for multiple-line commands or multiple-statement sequences of commands. If you make a mistake, you don't have to retype everything. Just edit your script to correct the error, then tell `mysql` to execute it again.
- If you have a query that produces a lot of output, you can run the output through a pager rather than watching it scroll off the top of your screen:

```
shell> mysql < batch-file | more
```

- You can catch the output in a file for further processing:

```
shell> mysql < batch-file > mysql.out
```

- You can distribute your script to other people so they can run the commands, too.
- Some situations do not allow for interactive use, for example, when you run a query from a `cron` job. In this case, you must use batch mode.

The default output format is different (more concise) when you run `mysql` in batch mode than when you use it interactively. For example, the output of `SELECT DISTINCT species FROM pet` looks like this when run interactively:

```
+-----+
| species |
+-----+
| bird   |
| cat    |
| dog    |
| hamster|
| snake  |
```

+-----+

But like this when run in batch mode:

```
species
bird
cat
dog
hamster
snake
```

If you want to get the interactive output format in batch mode, use `mysql -t`. To echo to the output the commands that are executed, use `mysql -vvv`.

Queries from twin project

At Analytikerna and Lentus, we have been doing the systems and field work for a big research project. This project is a collaboration between the Institute of Environmental Medicine at Karolinska Institutet Stockholm and the Section on Clinical Research in Aging and Psychology at the University of Southern California.

The project involves a screening part where all twins in Sweden older than 65 years are interviewed by telephone. Twins who meet certain criteria are passed on to the next stage. In this latter stage, twins who want to participate are visited by a doctor/nurse team. Some of the examinations include physical and neuropsychological examination, laboratory testing, neuroimaging, psychological status assessment, and family history collection. In addition, data are collected on medical and environmental risk factors.

More information about Twin studies can be found at:

<http://www.imm.ki.se/TWIN/TWINUKW.HTM>

The latter part of the project is administered with a web interface written using Perl and **MySQL**.

Each night all data from the interviews are moved into a **MySQL** database.

Find all non-distributed twins

The following query is used to determine who goes into the second part of the project:

```
select
    concat(p1.id, p1.tvab) + 0 as tvid,
    concat(p1.christian_name, " ", p1.surname) as Name,
    p1.postal_code as Code,
    p1.city as City,
    pg.abrev as Area,
    if(td.participation = "Aborted", "A", " ") as A,
    p1.dead as dead1,
    l.event as event1,
    td.suspect as tsuspect1,
    id.suspect as isuspect1,
    td.severe as tsevere1,
    id.severe as isevere1,
    p2.dead as dead2,
    l2.event as event2,
    h2.nurse as nurse2,
```

```

h2.doctor as doctor2,
td2.suspect as tsuspect2,
id2.suspect as isuspect2,
td2.severe as tsevere2,
id2.severe as isevere2,
l.finish_date
from
twin_project as tp
/* For Twin 1 */
left join twin_data as td on tp.id = td.id and tp.tvab = td.tvab
left join informant_data as id on tp.id = id.id and tp.tvab = id.tvab
left join harmony as h on tp.id = h.id and tp.tvab = h.tvab
left join lentus as l on tp.id = l.id and tp.tvab = l.tvab
/* For Twin 2 */
left join twin_data as td2 on p2.id = td2.id and p2.tvab = td2.tvab
left join informant_data as id2 on p2.id = id2.id and p2.tvab = id2.tvab
left join harmony as h2 on p2.id = h2.id and p2.tvab = h2.tvab
left join lentus as l2 on p2.id = l2.id and p2.tvab = l2.tvab,
person_data as p1,
person_data as p2,
postal_groups as pg
where
/* p1 gets main twin and p2 gets his/her twin. */
/* ptvab is a field inverted from tvab */
p1.id = tp.id and p1.tvab = tp.tvab and
p2.id = p1.id and p2.ptvab = p1.tvab and
/* Just the sceening survey */
tp.survey_no = 5 and
/* Skip if partner died before 65 but allow emigration (dead=9) */
(p2.dead = 0 or p2.dead = 9 or
 (p2.dead = 1 and
  (p2.death_date = 0 or
   ((to_days(p2.death_date) - to_days(p2.birthdate)) / 365)
   >= 65))))
and
(
/* Twin is suspect */
(td.future_contact = 'Yes' and td.suspect = 2) or
/* Twin is suspect - Informant is Blessed */
(td.future_contact = 'Yes' and td.suspect = 1 and id.suspect = 1) or
/* No twin - Informant is Blessed */
(ISNULL(td.suspect) and id.suspect = 1 and id.future_contact = 'Yes') or
/* Twin broken off - Informant is Blessed */
(td.participation = 'Aborted'
 and id.suspect = 1 and id.future_contact = 'Yes') or
/* Twin broken off - No inform - Have partner */
(td.participation = 'Aborted' and ISNULL(id.suspect) and p2.dead = 0))
and
l.event = 'Finished'
/* Get at area code */
and substring(p1.postal_code, 1, 2) = pg.code
/* Not already distributed */
and (h.nurse is NULL or h.nurse=00 or h.doctor=00)
/* Has not refused or been aborted */
and not (h.status = 'Refused' or h.status = 'Aborted'
 or h.status = 'Died' or h.status = 'Other')
order by
tvid;

```

Some explanations:

```
concat(p1.id, p1.tvab) + 0 as tvid
```

We want to sort on the concatenated `id` and `tvab` in numerical order. Adding 0 to the result causes

MySQL to treat the result as a number.

column `id`

This identifies a pair of twins. It is a key in all tables.

column `tvab`

This identifies a twin in a pair. It has a value of 1 or 2.

column `ptvab`

This is an inverse of `tvab`. When `tvab` is 1 this is 2, and vice versa. It exists to save typing and to make it easier for **MySQL** to optimize the query.

This query demonstrates, among other things, how to do lookups on a table from the same table with a join (`p1` and `p2`). In the example, this is used to check whether a twin's partner died before the age of 65. If so, the row is not returned.

All of the above exist in all tables with twin-related information. We have a key on both `id, tvab` (all tables) and `id, ptvab` (`person_data`) to make queries faster.

On our production machine (A 200MHz UltraSPARC), this query returns about 150-200 rows and takes less than one second.

The current number of records in the tables used above:

Table	Rows
<code>person_data</code>	71074
<code>lentus</code>	5291
<code>twin_project</code>	5286
<code>twin_data</code>	2012
<code>informant_data</code>	663
<code>harmony</code>	381
<code>postal_groups</code>	100

Show a table on twin pair status

Each interview ends with a status code called `event`. The query shown below is used to display a table over all twin pairs combined by event. This indicates in how many pairs both twins are finished, in how many pairs one twin is finished and the other refused, and so on.

```
select
    t1.event,
    t2.event,
    count(*)
from
    lentus as t1,
    lentus as t2,
    twin_project as tp
where
    /* We are looking at one pair at a time */
    t1.id = tp.id
    and t1.tvab=tp.tvab
    and t1.id = t2.id
    /* Just the sceening survey */
    and tp.survey_no = 5
```

```
        /* This makes each pair only appear once */  
        and t1.tvab='1' and t2.tvab='2'  
group by  
        t1.event, t2.event;
```