Chapter 3. Tutorial

Table of Contents [+/-]

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 need to consult the relevant portions of this manual, such as Chapter 5, *MySQL Server Administration*.)

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

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

User Comments

Add your own comment.

« 3 Tutorial
3.2 Entering Queries »

3.1. 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: 5.0.56-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 commands.

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 that 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.1.2, “Common Errors When Using MySQL Programs”.

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`** (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.

Previous / Next / Up / Table of Contents

User Comments

Posted by Morten Simonsen on February 27 2007 12:14pm [Delete] [Edit]

I think it would be nice to include a link to the doc for adding users and doing admin-stuff (look at the doc in the PostgreSQL), since readers of this doc actually is the administrator.

Add your own comment.

3.2. 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'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 here following the **`mysql>`** prompt and press Enter:

```
mysql> SELECT VERSION(), CURRENT_DATE;
+----------------+--------------+
| VERSION()      | CURRENT_DATE |
+----------------+--------------+
| 5.0.7-beta-Max | 2005-07-11   |
+----------------+--------------+
1 row in set (0.01 sec)
mysql>
```

This query illustrates several things about `mysql`:

- A command 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 command, `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 command.
- `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'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 |
+------------------+---------+
| 0.70710678118655 |      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()      |
+----------------+
| 5.0.7-beta-Max |
+----------------+
1 row in set (0.00 sec)

+---------------------+
| NOW()               |
+---------------------+
| 2005-07-11 17:59:36 |
+---------------------+
1 row in set (0.00 sec)
```

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 |
+---------------+--------------+
| jon@localhost | 2005-07-11   |
+---------------+--------------+
```

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 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
`mysql>`	Ready for new command.
`->`	Waiting for next line of multiple-line command.
`'>`	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 `/*`.

In the MySQL 5.0 series, the `/*>` prompt was implemented in MySQL 5.0.6.

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` 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 quote mark.)

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.

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 command.

Previous / Next / Up / Table of Contents

User Comments

Add your own comment.

Top / Previous / Next / Up / Table of Contents

3.3. Creating and Using a Database

[+/-]

Once you know how to enter commands, 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:

- 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 11.5.4.8, "**SHOW DATABASES** Syntax".

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 that you want to call yours **menagerie**. The administrator needs to execute a command 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.

Previous / Next / Up / Table of Contents

User Comments

Posted by Steven Ginzbarg on May 15 2006 5:56pm [Delete] [Edit]

I found the README.txt in the menagerie database download difficult to follow. Here is my revised version.

In what situations would one want to use one's command interpreter rather than the MySQL program? It seems tedious to have to execute mysql and supply connection parameters for each command.

README.txt

This directory contains files that can be used to set up the menagerie database that is used in the tutorial chapter of the MySQL Reference Manual.

First, you should create the database. In the mysql program, issue this statement:

mysql> CREATE DATABASE menagerie;

The examples below assume that you have unzipped menagerie.zip or menagerie.tar.gz to the C: drive creating a temporary folder C:\menagerie containing the downloaded files.

To create the pet table:

mysql> use menagerie
Database changed
mysql> source c:/menagerie/cr_pet_tbl.sql

(Note the use of forward slashes for specifying paths at the mysql> prompt.)

To load the pet table, use this command in mysql:

mysql> LOAD DATA LOCAL INFILE 'c:/menagerie/pet.txt' INTO TABLE pet;

To add Puffball's record, use this command:

mysql> source c:/menagerie/ins_puff_rec.sql

To create the event table:

mysql> source c:/menagerie/cr_event_tbl.sql

To load the event table, use this command:

mysql> LOAD DATA LOCAL INFILE 'c:/menagerie/event.txt' INTO TABLE event;

The commands above were entered in the MySQL program. If you have created an account for yourself and granted

3.3.1. 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, our 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 'monty'@'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 5.4, "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 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 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: ********
```

Note that **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.

User Comments

Posted by Darl Kuhn on February 27 2006 4:46pm [Delete] [Edit]

You can use this command to view the current database that you're connected to:

mysql> select database();

3.3.2. Creating a Table

Creating the database is the easy part, but at this point it's 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'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 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.

> **Note**
>
> Prior to MySQL 5.0.3, the upper limit was 255.) 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.

For more information about MySQL data types, see Chapter 9, *Data Types*.

Previous / Next / Up / Table of Contents

User Comments

Posted by Larry Blanchette on September 1 2005 6:21pm [Delete] [Edit]

you can use: show create table tablename, to get the DDL;

Posted by James Carrig on January 10 2006 7:44pm [Delete] [Edit]

While it is true that VARCHAR(20) means that the lengths in the columns "need not be 20," the more direct meaning is that the maximum column length is 20. Here is actual output which may speak more clearly than words:

mysql> CREATE TABLE demo_varchar (words VARCHAR(5));
Query OK, 0 rows affected (0.09 sec)

mysql> INSERT INTO demo_varchar VALUES ('abcdef');
ERROR 1406 (22001): Data too long for column 'words' at row 1

mysql> INSERT INTO demo_varchar VALUES ('abcde');
Query OK, 1 row affected (0.38 sec)

Posted by Raymond Peck on April 6 2006 8:03pm [Delete] [Edit]

Another issue is that I assume for all or most underlying table implementations a packed string table is used, so only as much space is used as required. Might want to add a link to more detailed info on the implications of various max lengths, and any variations between MyISAM, InnoDB, etc.

Posted by jon doe on October 27 2006 5:32am [Delete] [Edit]

3.3.3. 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 command:

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

Note that if you created the file on Windows with an editor that uses **\r\n** as a line terminator, you should use:

```
mysql> LOAD DATA LOCAL INFILE '/path/pet.txt' INTO TABLE pet
    -> LINES TERMINATED BY '\r\n';
```

(On an Apple machine running OS X, you would likely want to use **LINES TERMINATED BY '\r'**.)

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 5.3.4, “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);
```

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.

Previous / Next / Up / Table of Contents

User Comments

Posted by Doug Hall on February 17 2003 2:11pm [Delete] [Edit]

With Apple OS X: Use the terminal's drag and drop capability to insert the full path of the import file. This cuts down on the amount of typing, if you don't want deal with adding the import file into MySQL's data folder.

example:
%mysql --local-infile -u <username> -p <DatabaseName>
Enter password:<password>
mysql>load data local infile '<drag input file here>' into table <TableName>;

Posted by Brandon Stout on September 22 2004 10:56pm [Delete] [Edit]

you can also drag windows files to the command window, but you'll need to change the backslashes to double-backslashes or forwardslashes, and remove the c: at the beginning. If you have quotes around the path, you'll need to delete them as well.

Posted by tsaiching wong on December 3 2004 6:18am [Delete] [Edit]

mysql> LOAD DATA LOCAL INFILE '<dir>/pet.txt' INTO TABLE pet FIELDS terminated by '<delimiter>';

--> just in case anyone experienced some discomfort following above instructions.

Posted by phil newcombe on December 23 2004 6:40pm [Delete] [Edit]

I used the full path name 'c:/<path to file>' and it worked fine, but my defaults aren't THE defaults. :-)

Posted by Mark Buchanan on January 6 2005 1:52am [Delete] [Edit]

When dragging file in Windows I found the quotes needed to be kept in.

Posted by Mike Hearn on January 17 2005 6:31pm [Delete] [Edit]

Doug Halls trick also works on Linux/BSD using the GNOME or KDE terminal emulator programs.

Posted by Dennis Verbunt on February 27 2005 2:06pm [Delete] [Edit]

I was having some problems getting this working in XP but got it working after checking my syntax multiple times and then ENABLING local infiles.

Also after draging the file into the command window I had to replace the windows style backslashes with linux style forward slashes.
mysql> LOAD DATA LOCAL INFILE "C:\Documents and Settings\Dennis\Desktop\menagerie\pet.txt" INTO TABL

3.3.4. 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.

Previous / Next / Up / Table of Contents

User Comments

Posted by [name withheld] on January 6 2006 7:03am [Delete] [Edit]

like : select name from friends where age>15;

Add your own comment.

Top / Previous / Next / Up / Table of Contents

3.3.4.1. 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 '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.

User Comments

Posted by Vlad Dogaru on May 11 2006 4:33pm [Delete] [Edit]

Or, if, for instance, you added an extra blank line in pet.txt and ended up with an extra empty row, you can use:

DELETE FROM pet WHERE name='';

3.3.4.2. 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  |
+-------+--------+---------+------+------------+-------+
```

Previous / Next / Up / Table of Contents

User Comments

Posted by [Xia Ren] (Chinese Name 任侠) on May 23 2004 8:49am [Delete] [Edit]

If you want to know which animals were cat OR dog AND her owner was Harold.
Following this instruction below :
mysql>SELECT * FROM pet WHERE (species = 'dog' OR species = 'cat') AND owner = 'harold';

Hope it helps.....

Xia Ren comes from People's Republic of China.

Posted by Max Rosan on October 23 2004 7:19pm [Delete] [Edit]

Chat Messenger:

"mysql> DELETE FROM `messenger` WHERE time_row < (UNIX_TIMESTAMP() - 1800)"

time_row : integer

Delete all messages down 30 seconds

Posted by James Herdman on November 6 2005 2:33am [Delete] [Edit]

It should be noted that you can't do something like the following:

select * from pet where (species = 'dog' and species = 'cat');

Any given attribute can only have one value (this is an SQL property called "atomicity"). If you translated the above SQL statement into English this makes more sense: "Show me all pets who are dogs and cats". A pet can't be both a cat and a dog at the same time! A better question is "Show me all pets who are dogs or cats":

select * from pet where (species = 'dog' or species = 'cat');

Posted by Edward Lipchus on October 11 2006 8:35pm [Delete] [Edit]

Here's an example of how the placement of () can make a difference, compared to taking the default precedence order of AND and OR -

mysql> select * from pet where species='cat' and sex='f' or owner='gwen';

3.3.4.3. 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 |
+--------+---------+------------+
```

Previous / Next / Up / Table of Contents

User Comments

Posted by Tanvir Alam (Mitul) Dhaka on October 1 2005 6:15am [Delete] [Edit]

select name sex from pet;
if u miss the ' , ' between name and sex it won't generate error.It will show only sex column.

Posted by Mukhtar Elhadi on October 1 2005 12:38pm [Delete] [Edit]

it is just an alias for name column not sex column

Posted by Alex Grim on December 13 2006 5:30am [Delete] [Edit]

It's like saying:
select cats AS 'dogs' from pets;

It'd return a table something like:

```
+--------+
|  dogs  |
|--------|
| kitty  |
| meow   |
| claws  |
+--------+
```

Posted by Linh Hoang on February 11 2007 8:44pm [Delete] [Edit]

To clarify the comma issue, the token or word after the column's name is a label for the column. Here is exactly what it is:

mysql> SELECT name, species TYPE FROM pet WHERE species = 'dog' OR species = 'cat';

```
+--------+------+
| name   | TYPE |
+--------+------+
| Fluffy | cat  |
| Claws  | cat  |
| Buffy  | dog  |
| Fang   | dog  |
| Bowser | dog  |
+--------+------+
```

3.3.4.4. Sorting Rows

You may have noticed in the preceding examples that the result rows are displayed in no particular order. 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 |
+----------+------------+
```

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 |
+----------+---------+------------+
```

Note that the **DESC** keyword applies only to the column name immediately preceding it (**birth**); it does not affect the **species** column sort order.

Previous / Next / Up / Table of Contents

User Comments

Posted by [name withheld] on January 29 2002 6:30am [Delete] [Edit]

For an example of sorting based on a dynamically
generated column, see the example of sorting
based on age in section 3.3.4.5 "Date
Calculations".

Posted by Billy Kimble on June 2 2003 7:32am [Delete] [Edit]

If you want to explicity specify the order of which 'order by' comes back in, like if you had a priority field that had the values "Low" "High" or "Medium" .. do this:

select * from tablename order by priority='High' DESC, priority='Medium' DESC, priority='Low" DESC;

Posted by noatun on November 3 2003 1:18am [Delete] [Edit]

Sometimes you might want to sort names. If you have First and Last names in one field, seperated by a blank, you can do this by:
SELECT * FROM my_addressbook ORDER BY SUBSTRING_INDEX(name, ' ', -1) ASC

This works with John Adam, John F. Adam
but not with John F.Adam

Posted by Juan Ignacio Gomez on March 23 2004 4:45am [Delete] [Edit]

** Order By number Like this Number was a Text **
Some times you need to order by a column that contains numbers, but as
if it would be text, example:

Field Name: Numbers
Type: Integer(11)
Data:

```
+----------+
| numbers  |
```

3.3.4.5. 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, compute the difference in the year part of the current date and the birth date, then subtract one if the current date occurs earlier in the calendar year than the birth date. The following query shows, for each pet, the birth date, the current date, and the age in years.

```
mysql> SELECT name, birth, CURDATE(),
    -> (YEAR(CURDATE())-YEAR(birth))
    -> - (RIGHT(CURDATE(),5)<RIGHT(birth,5))
    -> 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 |
+----------+------------+------------+------+
```

Here, **YEAR()** pulls out the year part of a date and **RIGHT()** pulls off the rightmost five characters that represent the **MM-DD** (calendar year) part of the date. The part of the expression that compares the **MM-DD** values evaluates to 1 or 0, which adjusts the year difference down a year if **CURDATE()** occurs earlier in the year than **birth**. The full expression is somewhat ungainly, so an *alias* (**age**) is used to make the output column label more meaningful.

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(),
    -> (YEAR(CURDATE())-YEAR(birth))
    -> - (RIGHT(CURDATE(),5)<RIGHT(birth,5))
    -> 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(),
    -> (YEAR(CURDATE())-YEAR(birth))
    -> - (RIGHT(CURDATE(),5)<RIGHT(birth,5))
    -> 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,
    -> (YEAR(death)-YEAR(birth)) - (RIGHT(death,5)<RIGHT(birth,5))
    -> 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. See Section 3.3.4.6, “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 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()` allows 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;
```

Note that `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`).

Previous / Next / Up / Table of Contents

User Comments

Posted by Dan Fitzpatrick on May 28 2002 10:00am [Delete] [Edit]

In a business context, a more interesting query for this sample db might be the one alluded to earlier in the tutorial - select rows whose birthdays are coming up soon to send out a reminder...

Here is the way I did that:

SET @bdayThreshhold=150;

SELECT name, birth, CONCAT(((RIGHT(birth,5) < RIGHT(CURRENT_DATE,5))

```
+ YEAR(CURRENT_DATE)), RIGHT(birth,6)) AS bday,
```

3.3.4.6. 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, 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 |
+-----------+---------------+
```

Note that 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. See Section B.1.5.3, “Problems with NULL Values”.

3.3.4.7. 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 allows 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. 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 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).

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 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  |
+-------+--------+---------+------+------------+-------+
```

Section 10.4.2, "Regular Expressions", provides more information about the syntax for regular expressions.

Previous / Next / Up / Table of Contents

User Comments

Posted by Angie Ahl on October 14 2005 12:23pm [Delete] [Edit]

Note for regular users of Regex. MySQL doesn't seem to support use of ? as a greedy operator.

ie use this:

UPDATE pages SET sitesection = 'Softwarey' WHERE pages.path REGEXP '^/for_sale/[^/]*\\.html';

NOT

UPDATE pages SET sitesection = 'Softwarey' WHERE pages.path REGEXP '^/for_sale/[^/]*?\\.html';

Posted by Joao Santos on November 5 2005 10:08pm [Delete] [Edit]

In reply to the previews comment.

I believe that what you have in your example is a PERL extension which is not part of the POSIX specification for regular expressions and is not important unless you need to extract the matched substrings (which is not the case).

Posted by Brian Gorby on February 23 2006 6:25pm [Delete] [Edit]

Match a correctly-formatted email address:

RLIKE
'^[[:alnum:][.period.][.hyphen.][.underscore.]]+@([[:alnum:][.hyphen.][.underscore.]]+[.period.])+[[:alnum:][.hyphen.]]{2,3}$

Posted by Dan Stevens on April 11 2006 9:40am [Delete] [Edit]

3.3.4.8. 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 |
+--------+----------+
```

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

```
mysql> SELECT owner, COUNT(*) FROM pet;
ERROR 1140 (42000): 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 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 |
+---------+------+----------+
```

Previous / Next / Up / Table of Contents

User Comments

Posted by Michael Dibbets on April 28 2005 6:42pm [Delete] [Edit]

For example, this is a table of yours:

```
|---ID---|---Amount--|
|X:183812|---$3.00---|
|00391123|---$4.00---|
|X:203985|---$0.00---|
```

And you wish to select all ID's with an X: in front of it, but you have like 10000 of them, but you need to know how many you have, without having to use the php function mysql_numrows after a mysql_query, because that would mean that all the data is loaded into memory, held ready by the mysql server... that's a bit of a memory hog.
Then use this code(only for php, modify to your own liking)

```
$test1 = mysql_query("SELECT COUNT(*) FROM [tablename] WHERE `ID` LIKE 'X:%'");
$test2 = mysql_fetch_row($test1);
echo "$test2[0]";
```

Just my 2 cents ;-)

Michael

Posted by Patrick Mineault on August 10 2005 7:22pm [Delete] [Edit]

If you are using SELECT DISTINCT on a complex join clause, you might be stumped to find the count without returning the whole recordset. For example:

SELECT DISTINCT COUNT(*) AS theCount FROM table1, table2 WHERE table1.user_id = table2.user_id

Will not return the same number for 'theCount' as you would have using mysql_num_rows and:

SELECT DISTINCT table1.user_id FROM table1, table2 WHERE table1.user_id = table2.user_id

This is because DISTINCT affects the user_id, not the count. If you try something like:

SELECT DISTINCT table1.user_id, COUNT(*) AS theCount FROM table1, table2 WHERE table1.user_id = table2.user_id

Then you'll get an error saying there's no GROUP BY clause. The key is to collapse all of the rows using GROUP BY NULL:

SELECT DISTINCT table1.user_id, COUNT(*) AS theCount FROM table1, table2 WHERE table1.user_id = table2.user_id GROUP BY NULL

Posted by Sherzod Ruzmetov on October 4 2005 5:41am [Delete] [Edit]

Regarding the DISTINCT with COUNT() on joins, try the following instead. It works for me:

SELECT COUNT(DISTINCT table_name.rowname) FROM table_name WHERE ....

Posted by negnin on January 10 2007 4:58pm [Delete] [Edit]

If you use distinct and joins and just want a count, you can also put your complete query between parentheses and

3.3.4.9. 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 that you know which animal each event pertains to.
- A date so that you know when the event occurred.
- A field to describe the event.
- An event type field, if you want to be able to categorize events.

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:

name	date	type	remark
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 have learned from the queries that you have 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 that you want to find out the ages at which each pet had its litters. We saw earlier how to calculate ages from two dates. The litter date of the mother is in the `event` table, but to calculate her age on that date you need her birth date, which is stored in the `pet` table. This means the query requires both tables:

```
mysql> SELECT pet.name,
```

```
    -> (YEAR(date)-YEAR(birth)) - (RIGHT(date,5)<RIGHT(birth,5)) AS age,
    -> remark
    -> FROM pet INNER JOIN event
    ->   ON pet.name = event.name
    -> WHERE event.type = 'litter';
+--------+------+-----------------------------+
| name   | age  | remark                      |
+--------+------+-----------------------------+
| Fluffy |    2 | 4 kittens, 3 female, 1 male |
| Buffy  |    4 | 5 puppies, 2 female, 3 male |
| Buffy  |    5 | 3 puppies, 3 female         |
+--------+------+-----------------------------+
```

There are several things to note about this query:

- The **FROM** clause joins two tables because the query needs to pull information from both of them.
- When combining (joining) information from multiple tables, you need to specify how records in one table can be matched to records in the other. This is easy because they both have a **name** column. The query uses **ON** clause to match up records in the two tables based on the **name** values.

  The query uses an **INNER JOIN** to combine the tables. An **INNER JOIN** allows for rows from either table to appear in the result if and only if both tables meet the conditions specified in the ON clause. In this example, the **ON** clause specifies that the **name** column in the **pet** table must match the **name** column in the **event** table. If a name appears in one table but not the other, the row will not appear in the result because the condition in the **ON** clause fails.
- Because 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 produce candidate pairs of males and females of like species:

```
mysql> SELECT p1.name, p1.sex, p2.name, p2.sex, p1.species
    -> FROM pet AS p1 INNER JOIN pet AS p2
    ->   ON 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 to refer to the columns and keep straight which instance of the table each column reference is associated with.

Previous / Next / Up / Table of Contents

User Comments

Posted by Steve Seliquini on November 30 2003 8:46am [Delete] [Edit]

Depending on when this query was run, Bowser might not be available for mating. :) In this case we can add additional criteria to the where clause to ensure that the animal is actually alive to perform.

3.4. 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 (for example, what its columns are called)? MySQL addresses this problem through several statements that provide information about the databases and tables it supports.

You have previously 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 have not yet selected any database, the result is **NULL**.

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

```
mysql> SHOW TABLES;
+---------------------+
| Tables_in_menagerie |
+---------------------+
| event               |
| pet                 |
+---------------------+
```

The name of the column in the output produced by this statement is always **Tables_in_***db_name*, where *db_name* is the name of the database. See Section 11.5.4.25, “**SHOW TABLES** Syntax”, for more information.

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 the column can contain **NULL** values, **Key** indicates whether the column is indexed, and **Default** specifies the column's default value. **Extra** displays special information about columns; for example, if a column was created with the

AUTO_INCREMENT option, this is shown here.

DESC is a short form of **DESCRIBE**. See Section 11.3.1, "**DESCRIBE** Syntax", for more information.

You can obtain the **CREATE TABLE** statement necessary to create an existing table using the **SHOW CREATE TABLE** statement. See Section 11.5.4.6, "**SHOW CREATE TABLE** Syntax".

If you have indexes on a table, **SHOW INDEX FROM** *tbl_name* produces information about them. See Section 11.5.4.13, "**SHOW INDEX** Syntax", for more about this statement.

Previous / Next / Up / Table of Contents

User Comments

Add your own comment.

Top / Previous / Next / Up / Table of Contents

3.5. 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 are running `mysql` under Windows and have some special characters in the file that cause problems, you can do this:

```
C:\> mysql -e "source 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.

If you want the script to continue even if some of the statements in it produce errors, you should use the `--force` command-line option.

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 that they can also run the commands.
- 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 `mysql` is run interactively:

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

In batch mode, the output looks like this instead:

```
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`**.

You can also use scripts from the `mysql` prompt by using the **`source`** command or **`\.`** command:

```
mysql> source filename;
mysql> \. filename
```

See Section 4.5.1.4, “Executing SQL Statements from a Text File”, for more information.

Previous / Next / Up / Table of Contents

User Comments

Posted by Yurii Zborovs'kyi on March 6 2003 12:57am [Delete] [Edit]

How to measure total batch running time for several SQLs:

```
# at start of your script file
SET @start=UNIX_TIMESTAMP();

# great job
...
...
...

# at bottom of your script file
SET
@s=@seconds:=UNIX_TIMESTAMP()-@start,
@d=TRUNCATE(@s/86400,0), @s=MOD(@s,86400),
@h=TRUNCATE(@s/3600,0), @s=MOD(@s,3600),
@m=TRUNCATE(@s/60,0), @s=MOD(@s,60),
@day=IF(@d>0,CONCAT(@d,' day'),''),
@hour=IF(@d+@h>0,CONCAT(IF(@d>0,LPAD(@h,2,'0'),@h),' hour'),''),
@min=IF(@d+@h+@m>0,CONCAT(IF(@d+@h>0,LPAD(@m,2,'0'),@m),' min.'),''),
@sec=CONCAT(IF(@d+@h+@m>0,LPAD(@s,2,'0'),@s),' sec.');

SELECT
CONCAT(@seconds,' sec.') AS seconds,
```

3.6. Examples of Common Queries

[+/-]

Here are examples of how to solve some common problems with MySQL.

Some of the examples use the table **shop** to hold the price of each article (item number) for certain traders (dealers). Supposing that each trader has a single fixed price per article, then (**article**, **dealer**) is a primary key for the records.

Start the command-line tool `mysql` and select a database:

```
shell> mysql your-database-name
```

(In most MySQL installations, you can use the database named **test**).

You can create and populate the example table with these statements:

```
CREATE TABLE shop (
    article INT(4) UNSIGNED ZEROFILL DEFAULT '0000' NOT NULL,
    dealer  CHAR(20)                 DEFAULT ''     NOT NULL,
    price   DOUBLE(16,2)             DEFAULT '0.00' NOT NULL,
    PRIMARY KEY(article, dealer));
INSERT INTO shop VALUES
    (1,'A',3.45),(1,'B',3.99),(2,'A',10.99),(3,'B',1.45),
    (3,'C',1.69),(3,'D',1.25),(4,'D',19.95);
```

After issuing the statements, the table should have the following contents:

```
SELECT * FROM shop;

+---------+--------+-------+
| article | dealer | price |
+---------+--------+-------+
|    0001 | A      |  3.45 |
|    0001 | B      |  3.99 |
|    0002 | A      | 10.99 |
|    0003 | B      |  1.45 |
|    0003 | C      |  1.69 |
|    0003 | D      |  1.25 |
|    0004 | D      | 19.95 |
+---------+--------+-------+
```


User Comments

Add your own comment.

3.6.1. The Maximum Value for a Column

“What's the highest item number?”

```
SELECT MAX(article) AS article FROM shop;

+---------+
| article |
+---------+
|       4 |
+---------+
```

Previous / Next / Up / Table of Contents

User Comments

Posted by Najeem M Illyas on November 13 2004 6:03am [Delete] [Edit]

Similarly MIN can be used to find minium value

SELECT MIN(jnr_id) AS jnr_id FROM chain

```
+----------+
|  jnr_id  |
+----------+
|   1000   |
+----------+
```

Posted by Andy Pieters on February 6 2005 8:06pm [Delete] [Edit]

You can use just about any function in a select query.

Besides the MAX, and MIN functions, there is also the AVG function to quickly calculate the Average value of a field.

* The AS parameter sets the name of the field that is outputted

Example:

SELECT avg(article) AS article_avg FROM shop;

```
+-------------+
| article_avg |
+-------------+
|      2.4286 |
+-------------+
```

1 row in set (0.00 sec)

3.6.2. The Row Holding the Maximum of a Certain Column

Task: Find the number, dealer, and price of the most expensive article.

This is easily done with a subquery:

```
SELECT article, dealer, price
FROM   shop
WHERE  price=(SELECT MAX(price) FROM shop);
```

Another solution is to sort all rows descending by price and get only the first row using the MySQL-specific `LIMIT` clause:

```
SELECT article, dealer, price
FROM shop
ORDER BY price DESC
LIMIT 1;
```

> **Note**
>
> If there were several most expensive articles, each with a price of 19.95, the `LIMIT` solution would show only one of them.

Previous / Next / Up / Table of Contents

User Comments

Posted by Andrew Ford on July 20 2003 5:10am [Delete] [Edit]

Problem: To find out which team scored the most points in a given week for my Fantasy Football League.

Background Info: My game Schedule table had five columns (awayTeam, awayPoints, homeTeam, homePoints, & week). So, I decided to make a temporary table to make the team IDs and points indifferent.
CREATE TEMPORARY TABLE tmp (team INT NOT NULL,
points INT,
PRIMARY KEY (team));

Populate table:
INSERT INTO tmp
SELECT awayTeam, awayPoints
FROM Schedule
WHERE week=1;
(Rinse & repeat for home team).

Here's where everything gets *stupid* (for lack of a better term). You'd expect to just do a nested select to figure the team with the most points. For example:
SELECT t.team, f.name
FROM tmp t, fflTeam f

3.6.3. Maximum of Column per Group

Task: Find the highest price per article.

```
SELECT article, MAX(price) AS price
FROM   shop
GROUP BY article

+---------+-------+
| article | price |
+---------+-------+
|    0001 |  3.99 |
|    0002 | 10.99 |
|    0003 |  1.69 |
|    0004 | 19.95 |
+---------+-------+
```

Previous / Next / Up / Table of Contents

User Comments

Posted by Marco Gergele on September 30 2004 10:24am [Delete] [Edit]

There seems to be no NULL=Infinity maximum version. In a column is stored end (type date) with NULL = no end. I need the maximum of that column, which is NULL if a NULL-value exists. That should be easy to implement as a function, but not so easy with the existing function.

The same for sum, avg and so on. The versions with NULL=ignore are usefull most of the time, but sum() of NULL and 5 can sometimes be NULL instead of 5. NULL+5 is NULL.

Add your own comment.

3.6.4. The Rows Holding the Group-wise Maximum of a Certain Field

Task: For each article, find the dealer or dealers with the most expensive price.

This problem can be solved with a subquery like this one:

```
SELECT article, dealer, price
FROM   shop s1
WHERE  price=(SELECT MAX(s2.price)
              FROM shop s2
              WHERE s1.article = s2.article);

+---------+--------+-------+
| article | dealer | price |
+---------+--------+-------+
|    0001 | B      |  3.99 |
|    0002 | A      | 10.99 |
|    0003 | C      |  1.69 |
|    0004 | D      | 19.95 |
+---------+--------+-------+
```

The preceding example uses a correlated subquery, which can be inefficient (see Section 11.2.8.7, "Correlated Subqueries"). Other possibilities for solving the problem are to use a uncorrelated subquery in the **FROM** clause or a **LEFT JOIN**:

```
SELECT s1.article, dealer, s1.price
FROM shop s1
JOIN (
  SELECT article, MAX(price) AS price
  FROM shop
  GROUP BY article) AS s2
  ON s1.article = s2.article AND s1.price = s2.price;

SELECT s1.article, s1.dealer, s1.price
FROM shop s1
LEFT JOIN shop s2 ON s1.article = s2.article AND s1.price < s2.price
WHERE s2.article IS NULL;
```

The **LEFT JOIN** works on the basis that when **s1.price** is at its maximum value, there is no **s2.price** with a greater value and the **s2** rows values will be **NULL**. See Section 11.2.7.1, "**JOIN** Syntax".

User Comments

Posted by Tim Henderson on February 10 2004 3:55am [Delete] [Edit]

OK, Some BAD News about performance:

3.6.5. Using User-Defined Variables

You can employ MySQL user variables to remember results without having to store them in temporary variables in the client. (See Section 7.4, “User-Defined Variables”.)

For example, to find the articles with the highest and lowest price you can do this:

```
mysql> SELECT @min_price:=MIN(price),@max_price:=MAX(price) FROM shop;
mysql> SELECT * FROM shop WHERE price=@min_price OR price=@max_price;
+---------+--------+-------+
| article | dealer | price |
+---------+--------+-------+
|    0003 | D      |  1.25 |
|    0004 | D      | 19.95 |
+---------+--------+-------+
```

> **Note**
>
> It is also possible to store the name of a database object such as a table or a column in a user variable and then to use this variable in an SQL statement; however, this requires the use of a prepared statement. See Section 11.7, “SQL Syntax for Prepared Statements”, for more information.

Previous / Next / Up / Table of Contents

User Comments

Posted by Daniel on September 30 2005 5:48pm [Delete] [Edit]

Remember that using user variables the query will not be cached if you use the query caching option on the server, and therefore the answer could be slower than what we might expect.

Add your own comment.

3.6.6. Using Foreign Keys

In MySQL, **InnoDB** tables support checking of foreign key constraints.
See Section 12.2, "The **InnoDB** Storage Engine", and Section 1.8.5.4, "Foreign Keys".

A foreign key constraint is not required merely to join two tables. For storage engines other than **InnoDB**, it is possible when defining a column to use a **REFERENCES** *tbl_name(col_name)* clause, which has no actual effect, and *serves only as a memo or comment to you that the column which you are currently defining is intended to refer to a column in another table*. It is extremely important to realize when using this syntax that:

- MySQL does not perform any sort of **CHECK** to make sure that *col_name* actually exists in *tbl_name* (or even that *tbl_name* itself exists).
- MySQL does not perform any sort of action on *tbl_name* such as deleting rows in response to actions taken on rows in the table which you are defining; in other words, this syntax induces no **ON DELETE** or **ON UPDATE** behavior whatsoever. (Although you can write an **ON DELETE** or **ON UPDATE** clause as part of the **REFERENCES** clause, it is also ignored.)
- This syntax creates a *column*; it does ***not*** create any sort of index or key.
- This syntax will cause an error if used in trying to define an **InnoDB** table.

You can use a column so created as a join column, as shown here:

```
CREATE TABLE person (
    id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
    name CHAR(60) NOT NULL,
    PRIMARY KEY (id)
);

CREATE TABLE shirt (
    id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
    style ENUM('t-shirt', 'polo', 'dress') NOT NULL,
    color ENUM('red', 'blue', 'orange', 'white', 'black') NOT NULL,
    owner SMALLINT UNSIGNED NOT NULL REFERENCES person(id),
    PRIMARY KEY (id)
);

INSERT INTO person VALUES (NULL, 'Antonio Paz');

SELECT @last := LAST_INSERT_ID();

INSERT INTO shirt VALUES
(NULL, 'polo', 'blue', @last),
(NULL, 'dress', 'white', @last),
(NULL, 't-shirt', 'blue', @last);

INSERT INTO person VALUES (NULL, 'Lilliana Angelovska');

SELECT @last := LAST_INSERT_ID();

INSERT INTO shirt VALUES
```

```
(NULL, 'dress', 'orange', @last),
(NULL, 'polo', 'red', @last),
(NULL, 'dress', 'blue', @last),
(NULL, 't-shirt', 'white', @last);

SELECT * FROM person;
+----+--------------------+
| id | name               |
+----+--------------------+
|  1 | Antonio Paz        |
|  2 | Lilliana Angelovska |
+----+--------------------+

SELECT * FROM shirt;
+----+---------+--------+-------+
| id | style   | color  | owner |
+----+---------+--------+-------+
|  1 | polo    | blue   |     1 |
|  2 | dress   | white  |     1 |
|  3 | t-shirt | blue   |     1 |
|  4 | dress   | orange |     2 |
|  5 | polo    | red    |     2 |
|  6 | dress   | blue   |     2 |
|  7 | t-shirt | white  |     2 |
+----+---------+--------+-------+


SELECT s.* FROM person p INNER JOIN shirt s
   ON s.owner = p.id
 WHERE p.name LIKE 'Lilliana%'
   AND s.color <> 'white';

+----+-------+--------+-------+
| id | style | color  | owner |
+----+-------+--------+-------+
|  4 | dress | orange |     2 |
|  5 | polo  | red    |     2 |
|  6 | dress | blue   |     2 |
+----+-------+--------+-------+
```

When used in this fashion, the **REFERENCES** clause is not displayed in the output of **SHOW CREATE TABLE** or **DESCRIBE**:

```
SHOW CREATE TABLE shirt\G
*************************** 1. row ***************************
Table: shirt
Create Table: CREATE TABLE `shirt` (
`id` smallint(5) unsigned NOT NULL auto_increment,
`style` enum('t-shirt','polo','dress') NOT NULL,
`color` enum('red','blue','orange','white','black') NOT NULL,
`owner` smallint(5) unsigned NOT NULL,
PRIMARY KEY  (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1
```

The use of **REFERENCES** in this way as a comment or “reminder” in a column definition works with both **MyISAM** and **BerkeleyDB** tables.

Previous / Next / Up / Table of Contents

3.6.7. Searching on Two Keys

An **OR** using a single key is well optimized, as is the handling of **AND**.

The one tricky case is that of searching on two different keys combined with **OR**:

```
SELECT field1_index, field2_index FROM test_table
WHERE field1_index = '1' OR  field2_index = '1'
```

This case is optimized from MySQL 5.0.0. See Section 6.2.6, “Index Merge Optimization”.

You can also solve the problem efficiently by using a **UNION** that combines the output of two separate **SELECT** statements. See Section 11.2.7.3, “**UNION** Syntax”.

Each **SELECT** searches only one key and can be optimized:

```
SELECT field1_index, field2_index
    FROM test_table WHERE field1_index = '1'
UNION
SELECT field1_index, field2_index
    FROM test_table WHERE field2_index = '1';
```

Previous / Next / Up / Table of Contents

User Comments

Posted by David Thompson on March 9 2005 3:01pm [Delete] [Edit]

One thing to remember when using the 'union' statement (as I found out): the resulting set removes all duplicate entries unless you proceed the 'union' statement with the word 'all'. Assuming the table:

mysql> select * from ourpets;

```
+----------+--------+------------+------+------------+------------+
| 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       | f    | 1997-12-09 | NULL       |
| Slim     | Benny  | snake      | m    | 1996-04-29 | NULL       |
| Dalli    | Alli   | canine     | m    | 2001-12-20 | NULL       |
| Tara     | David  | canine     | f    | 2002-05-17 | NULL       |
| Mimi     | Alli   | guinea pig | m    | 2004-05-17 | NULL       |
+----------+--------+------------+------+------------+------------+
```

11 rows in set (0.00 sec)

3.6.8. Calculating Visits Per Day

The following example shows how you can use the bit group functions to calculate the number of days per month a user has visited a Web page.

```
CREATE TABLE t1 (year YEAR(4), month INT(2) UNSIGNED ZEROFILL,
             day INT(2) UNSIGNED ZEROFILL);
INSERT INTO t1 VALUES(2000,1,1),(2000,1,20),(2000,1,30),(2000,2,2),
            (2000,2,23),(2000,2,23);
```

The example table contains year-month-day values representing visits by users to the page. To determine how many different days in each month these visits occur, use this query:

```
SELECT year,month,BIT_COUNT(BIT_OR(1<<day)) AS days FROM t1
       GROUP BY year,month;
```

Which returns:

```
+------+-------+------+
| year | month | days |
+------+-------+------+
| 2000 |    01 |    3 |
| 2000 |    02 |    2 |
+------+-------+------+
```

The query calculates how many different days appear in the table for each year/month combination, with automatic removal of duplicate entries.

Previous / Next / Up / Table of Contents

User Comments

Posted by [name withheld] on May 17 2005 2:39pm [Delete] [Edit]

It seems to me that it would be much simpler and intuitive to use something like this to query the days per month:

SELECT year, month, COUNT(DISTINCT day) AS days FROM t1 GROUP BY year,month;

Posted by Anders Henke on December 19 2005 1:55pm [Delete] [Edit]

Note that this example is not suitable for large-scale processing of weblogs ... it's missing indexes and usually one might use an aggregated counter if aggregated values are goal of the statistics.

Add your own comment.

3.6.9. Using AUTO_INCREMENT

The **AUTO_INCREMENT** attribute can be used to generate a unique identity for new rows:

```
CREATE TABLE animals (
     id MEDIUMINT NOT NULL AUTO_INCREMENT,
     name CHAR(30) NOT NULL,
     PRIMARY KEY (id)
 );

INSERT INTO animals (name) VALUES
    ('dog'),('cat'),('penguin'),
    ('lax'),('whale'),('ostrich');

SELECT * FROM animals;
```

Which returns:

```
+----+---------+
| id | name    |
+----+---------+
|  1 | dog     |
|  2 | cat     |
|  3 | penguin |
|  4 | lax     |
|  5 | whale   |
|  6 | ostrich |
+----+---------+
```

You can retrieve the most recent **AUTO_INCREMENT** value with the **LAST_INSERT_ID()** SQL function or the **mysql_insert_id()** C API function. These functions are connection-specific, so their return values are not affected by another connection which is also performing inserts.

> **Note**
>
> For a multiple-row insert, **LAST_INSERT_ID()** and **mysql_insert_id()** actually return the **AUTO_INCREMENT** key from the *first* of the inserted rows. This allows multiple-row inserts to be reproduced correctly on other servers in a replication setup.

For **MyISAM** and **BDB** tables you can specify **AUTO_INCREMENT** on a secondary column in a multiple-column index. In this case, the generated value for the **AUTO_INCREMENT** column is calculated as **MAX(*auto_increment_column*) + 1 WHERE prefix=*given-prefix***. This is useful when you want to put data into ordered groups.

```
CREATE TABLE animals (
    grp ENUM('fish','mammal','bird') NOT NULL,
    id MEDIUMINT NOT NULL AUTO_INCREMENT,
    name CHAR(30) NOT NULL,
    PRIMARY KEY (grp,id)
);
```

```
INSERT INTO animals (grp,name) VALUES
    ('mammal','dog'),('mammal','cat'),
    ('bird','penguin'),('fish','lax'),('mammal','whale'),
    ('bird','ostrich');

SELECT * FROM animals ORDER BY grp,id;
```

Which returns:

```
+--------+----+---------+
| grp    | id | name    |
+--------+----+---------+
| fish   |  1 | lax     |
| mammal |  1 | dog     |
| mammal |  2 | cat     |
| mammal |  3 | whale   |
| bird   |  1 | penguin |
| bird   |  2 | ostrich |
+--------+----+---------+
```

Note that in this case (when the **AUTO_INCREMENT** column is part of a multiple-column index), **AUTO_INCREMENT** values are reused if you delete the row with the biggest **AUTO_INCREMENT** value in any group. This happens even for **MyISAM** tables, for which **AUTO_INCREMENT** values normally are not reused.

If the **AUTO_INCREMENT** column is part of multiple indexes, MySQL will generate sequence values using the index that begins with the **AUTO_INCREMENT** column, if there is one. For example, if the **animals** table contained indexes **PRIMARY KEY (grp, id)** and **INDEX (id)**, MySQL would ignore the **PRIMARY KEY** for generating sequence values. As a result, the table would contain a single sequence, not a sequence per **grp** value.

To start with an **AUTO_INCREMENT** value other than 1, you can set that value with **CREATE TABLE** or **ALTER TABLE**, like this:

```
mysql> ALTER TABLE tbl AUTO_INCREMENT = 100;
```

More information about **AUTO_INCREMENT** is available here:

- How to assign the **AUTO_INCREMENT** attribute to a column: Section 11.1.5, "**CREATE TABLE** Syntax", and Section 11.1.2, "**ALTER TABLE** Syntax".
- How **AUTO_INCREMENT** behaves depending on the SQL mode: Section 5.1.6, "SQL Modes".
- Find the row that contains the most recent AUTO_INCREMENT value: Section 10.2.3, "Comparison Functions and Operators".
- Set the **AUTO_INCREMENT** value to be used: Section 11.5.3, "**SET** Syntax".
- **AUTO_INCREMENT** and replication: Section 14.3.1, "Replication Features and Issues".
- Server-system variables related to **AUTO_INCREMENT** (**auto_increment_increment** and **auto_increment_offset**) that can be used for replication: Section 5.1.3, "System Variables".

Previous / Next / Up / Table of Contents

User Comments

Posted by Jim Martin on October 1 2002 9:57am [Delete] [Edit]