

# MySQL is the open source data management system: study of its feature and functions

By Goutam Biswas  
Email: [goutambiswas8@gmail.com](mailto:goutambiswas8@gmail.com)  
Mobile: +91 9831092149



## 1.0 Introduction

Open Source Software / Free Software (OSS/FS) has risen to great prominence. Briefly, OSS/FS programs are programs whose licenses give users the freedom to run the program for any purpose, to study and modify the program, and to redistribute copies of either the original or modified program (without having to pay royalties to previous developers).

This goal of this paper is to show that you should consider using OSS/FS when You are looking for software, based on quantitative measures. Some sites provide a few anecdotes on why you should use OSS/FS, but for many that is not enough information to justify using OSS/FS. Instead, this paper emphasizes *quantitative* measures (such as experiments and market studies) on why using OSS/FS products is, in many circumstances, a reasonable or even superior approach. I should note that while I find much to as if about OSS/FS, I am not a rabid advocate; I use both proprietary and OSS/FS products myself. Vendors of proprietary products often work hard to find numbers to support their claims; this page provides a useful antidote of hard figures to aid in comparing prop rietary products to OSS/FS.

MySQL is an open source relational database management system (RDBMS) that uses Structured Query Language (SQL), the most popular language for adding, accessing, and processing data in a database. A database is a structured collection of data. MySQL is a relational database management system. A relational database stores data in separate tables rather than putting all the data in one big storeroom. This adds speed and flexibility.

MySQL is a multithreaded, multi-user database management system. The basic program runs as a server providing multi-user access to multiple databases. MySQL is used to refer to the entire MySQL distribution package or the MySQL server, while mysql refers to a client program. The server and client programs are different entities. Dividing the package into a server and clients separates the actual data from the interface. MySQL Server works in client/server or embedded systems. The MySQL Database Software is a client/server system that consists of a multi-threaded SQL server that supports different back-ends, several different client programs and libraries, administrative tools, and a wide range of application programming interfaces. The MySQL package consists of:

- The MySQL server: This is the heart of MySQL. You can consider it a program that stores and manages your databases.
- MySQL client programs: MySQL comes with many client programs. The one with which we'll be dealing a lot is called mysql (note: smallcaps). This provides an interface through which you can issue SQL statements and have the results displayed.

The server and client programs are different entities. Thus, you can use client programs on your system to access data on a MySQL server running on another computer. (Note: you would need appropriate permissions for this. Consult the system administrator of the remote machine.) Dividing the package into a server and clients separates the actual data from the interface.

## 2.0 Creating MySQL database on Windows system

- Start-Run-cmd
- Press <OK>
- mysql -u root -p
- Enter password [type the password]
- Prompt changes to mysql>
- Type create databases employees;  
(Note: The command ends with a semi-colon).
- The MySQL server responds with something like:  
*Query OK, 1 row affected (0.13 sec).* This means that you have successfully created the database. Now, let's see how many databases you have on your system. Issue the following command.
- Type show databases;
- To come back to the DOS prompt, type **quit** at the mysql prompt.

### Creating tables

We will explore the MySQL commands to create database tables and selecting the database. Databases store data in tables. So what are these tables? In simplest terms, tables consist of rows and columns. Each column defines data of a particular type. Rows contain individual records. Consider the following:

Name	Age	Country	Email
------	-----	---------	-------

Goutam Biswas	28	India	goutambiswas8@gmail.com
John Doe	32	Australia	j.dow@nowhere.com
John Wayne	48	U.S.A.	jw@oldwesterns.com
Alexander	19	Greece	alex@conqueror.com

The table above contains four columns that store the name, age, country and email. Each row contains data for one individual. This is called a **record**. To find the country and email of Alexander, you'd first pick the name from the first column and then look in the third and fourth columns of the same row.

A database can have many tables; it is tables, that contain the actual data. Hence, we can segregate related (or unrelated) data in different tables. For our **employees** database we'll have one table that stores company details of the employees. The other table would contain personal information. Let's make the first table.

The SQL command for creating tables looks complex when you view it for the first time. Don't worry if you get confused, we'll be discussing this in more detail in later sessions.

```
CREATE TABLE employee_data
(
emp_id int unsigned not null auto_increment primary key,
f_name varchar(20),
l_name varchar(20),
title varchar(30),
age int,
yos int,
salary int,
perks int,
email varchar(60)
);
```

In MySQL, commands and column names are not case-sensitive; however, table and database names might be sensitive to case depending on the platform (as in Linux).

The **CREATE TABLE** keywords are followed by the name of the table we want to create, **employee\_data**. Each line inside the parenthesis represents one column. These columns store the employee id, first name, last name, title, age, years of service with the company, salary, perks and emails of our employees and are given descriptive names **emp\_id**, **f\_name**, **l\_name**, **title**, **age**, **yos**, **salary**, **perks** and **email**, respectively.

Each column name is followed by the **column type**. Column types define the *type of data* the column is set to contain. In our example, columns, **f\_name**, **l\_name**, **title** and **email** would contain small text strings, so we set the column type to *varchar*, which means **variable characters**. The maximum number of characters for varchar columns is specified by a number enclosed in parenthesis immediately following the column name. Columns **age**, **yos**, **salary** and **perks** would contain numbers (integers), so we set the column type to *int*. Our first column (**emp\_id**) contains an employee id; **int**: specifies that the column type is an integer (a number). **unsigned**: determines that the number will be *unsigned* (positive integer). **not null**: specifies that the value cannot be null (empty); that is, each row in the column would have a value. **auto\_increment**: When MySQL comes across a column with an auto\_increment attribute, it generates a new value that is one greater than the largest value in the column. Thus, we don't need to supply values for this column, MySQL generates it for us! Also, it follows that each value in this column would be unique. (We'll discuss the benefits of having unique values very shortly). **primary key**: helps in indexing the column that help in faster searches. Each value has to be unique.

### MySQL tables

Now that we've created our **employee\_data** table, let's check its listing. Type at the mysql prompt.

```
SHOW TABLES;
DESCRIBE employee_data;
```

DESCRIBE lists all the column names along with their column types of the table. Now let's see how we can insert data into our table.

### Inserting data in MySQL tables

The **INSERT** SQL statement impregnates our table with data. Here is a general form of INSERT.

```
INSERT into table_name (column1, column2... )
```

```
values (value1, value2...);
```

where *table\_name* is the name of the table into which we want to insert data; column1, column2 etc. are column names and value1, value2 etc. are values for the respective columns. The following statement inserts the first record in **employee\_data** table.

```
INSERT INTO employee_data
(f_name, l_name, title, age, yos, salary, perks, email)
values
("Manish", "Sharma", "CEO", 28, 4, 200000,
50000, "manish@bignet.com");
```

As with other MySQL statements, you can enter this command on one line or span it in multiple lines. Some important points:

- The values for columns f\_name, l\_name, title and email are text strings and surrounded with quotes.
- Values for age, yos, salary and perks are numbers (integers) and without quotes.
- You'll notice that we've inserted data in all columns *except* **emp\_id**. This is because, we leave this job to MySQL, which will check the column for the largest value, increment it by one and insert the new value.

Inserting additional records requires separate INSERT statements. In order to make life easy, I've packed all INSERT statements into a file. You'll notice that it's a plain ASCII file with an INSERT statement on each line. To insert data into employee\_data table with employee.dat file on Windows do the following:

- Move the file to c:\mysql\bin.
- Make sure MySQL is running.
- Issue the following command  
mysql employees <employee.dat

### Querying MySQL tables

Our **employee\_data** table now contains enough data for us to work with. Let us see how we can extract (query) it. Querying involves the use of the MySQL SELECT command. Data is extracted from the table using the **SELECT** SQL command. Here is the format of a SELECT statement:

```
SELECT column_names from table_name [WHERE ...conditions];
```

The *conditions* part of the statement is optional. Basically, you require to know the column names and the table name from which to extract the data. For

example, in order to extract the first and last names of all employees, issue the following command.

```
SELECT f_name, l_name from employee_data;
```

To display the entire table, we can either enter all the column names or use a simpler form of the SELECT statement.

```
SELECT * from employee_data;
```

To selecting data using conditions

```
SELECT column_names from table_name [WHERE ...conditions];
```

```
SELECT f_name, l_name from employee_data where title="Programmer";
```

```
SELECT f_name, l_name from employee_data where age = 32;
```

```
SELECT f_name, l_name from employee_data where age > 32;
```

```
SELECT f_name, l_name from employee_data where yrs < 3;
```

```
select f_name, l_name from employee_data where yrs <= 2;
```

### Pattern Matching with text data

We will now learn at how to match text patterns using the where clause and the **LIKE** operator in this section. The **equal to(=)** comparison operator helps in selecting strings that are identical. Thus, to list the names of employees whose first names are John, we can use the following SELECT statement.

```
select f_name, l_name from employee_data where f_name = "John";
```

What if we wanted to display employees whose first names begin with the alphabet **J**? SQL allows for some pattern matching with string data. Here is how it works.

```
select f_name, l_name from employee_data where f_name LIKE "J%";
```

You'll notice that we've replaced the *Equal To* sign with **LIKE** and we've used a **percentage sign (%)** in the condition. The % sign functions as a wildcard (similar to the usage of \* in DOS and Linux systems). It signifies *any character*.

### Logical Operators

In this section of the SQL primer we look at how to select data based on certain conditions presented through MySQL logical operators. SQL conditions can also contain Boolean (logical) operators. They are

**AND**

**OR**

**NOT**

Their usage is quite simple. Here is a SELECT statement that lists the names of employees who draw more than \$70000 but less than \$90000.

```
SELECT f_name, l_name from employee_data  
where salary > 70000 AND salary < 90000;
```

The **NOT** operator helps in listing all non programmers. (Programmers include Senior programmers, Multimedia Programmers and Programmers).

```
SELECT f_name, l_name, title from employee_data  
where title NOT LIKE "%programmer%";
```

### **IN and BETWEEN**

This section of the tutorial MySQL looks at the In and BETWEEN operators. To list employees who are **Web Designers** and **System Administrators**, we use a SELECT statement as

```
SELECT f_name, l_name, title from  
-> employee_data where  
-> title = 'Web Designer' OR  
-> title = 'System Administrator';
```

SQL also provides an easier method with **IN**. Its usage is quite simple.

```
SELECT f_name, l_name, title from  
-> employee_data where title  
-> IN ('Web Designer', 'System Administrator');
```

Suffixing **NOT** to **IN** will display data that is *NOT* found *IN* the condition. The following lists employees who hold titles other than **Programmer** and **Marketing Executive**.

```
SELECT f_name, l_name, title from  
-> employee_data where title NOT IN  
-> ('Programmer', 'Marketing Executive');
```

**BETWEEN** is employed to specify integer ranges. Thus instead of **age >= 32 AND age <= 40**, we can use **age BETWEEN 32 and 40**.

```
select f_name, l_name, age from
```

```
-> employee_data where age BETWEEN  
-> 32 AND 40;
```

You can use **NOT** with **BETWEEN** as in the following statement that lists employees who draw salaries less than \$90000 and more than \$150000.

```
select f_name, l_name, salary  
-> from employee_data where salary  
-> NOT BETWEEN  
-> 90000 AND 150000;
```

### Ordering data

This section of the online MySQL tutorial looks at how we can change the display order of the data extracted from MySQL tables using the ORDER BY clause of the SELECT statement.

The data that we have retrieved so far was always displayed in the order in which it was stored in the table. Actually, SQL allows for sorting of retrieved data with the **ORDER BY** clause. This clause requires the column name based on which the data will be sorted. Let's see how to display employee names with last names sorted alphabetically (in ascending order).

```
SELECT l_name, f_name from  
employee_data ORDER BY l_name;
```

Here are employees sorted by age.

```
SELECT f_name, l_name, age  
from employee_data  
ORDER BY age;
```

The **ORDER BY** clause can sort in an *ASCENDING (ASC)* or *DESCENDING (DESC)* order depending upon the argument supplied. To list employee first names in descending order, we'll use the statement below.

```
SELECT f_name from employee_data  
ORDER by f_name DESC;
```



### Limiting data retrieval

This section of the online MySQL lesson looks at how to limit the number of records displayed by the SELECT statement. As your tables grow, you'll find a need to display only a subset of data. This can be achieved with the **LIMIT** clause. For example, to list only the names of first 5 employees in our table, we use LIMIT with 5 as argument.

```
SELECT f_name, l_name from  
employee_data LIMIT 5;
```

You can couple **LIMIT** with **ORDER BY**. Thus, the following displays the 4 senior most employees.

```
SELECT f_name, l_name, age from  
employee_data ORDER BY age DESC  
LIMIT 4;
```

### Extracting Subsets

Limit can also be used to extract a subset of data by providing an additional argument. The general form of this LIMIT is:  
SELECT (whatever) from table LIMIT *starting row, Number to extract*;

```
SELECT f_name, l_name from  
employee_data LIMIT 6,3;
```

### DISTINCT keyword

In this section of the online MySQL guide, we will look at how to select and display records from MySQL tables using the DISTINCT keyword that eliminates the occurrences of the same data. To list all titles in our company database, we can throw a statement as:

```
select title from employee_data;
```

You'll notice that the display contains multiple occurrences of certain data. The SQL **DISTINCT** clause lists only unique data. Here is how you use it.

```
select DISTINCT title from employee_data;
```

Also, you can sort the unique entries using **ORDER BY**.

```
select DISTINCT age from employee_data  
ORDER BY age;
```

### Finding the minimum and maximum values

MySQL provides inbuilt functions to find the minimum and maximum values. SQL provides 5 **aggregate functions**. They are:

**MIN()**: Minimum value

**MAX()**: Maximum value

**SUM()**: The sum of values

**AVG()**: The average values

**COUNT()**: Counts the number of entries

```
select MIN(salary) from employee_data;  
select MAX(salary) from employee_data;
```

### Finding the average and sum

Totalling column values with MySQL SUM. The **SUM()** aggregate function calculates the total of values in a column. You require to give the column name, which should be placed inside parenthesis. Let's see how much *Bignet* spends on salaries.

```
select SUM(salary) from employee_data;  
select sum(salary) + sum(perks) from employee_data;
```

### Finding the Average

The **AVG()** aggregate function is employed for calculating averages of data in columns.

```
select avg(age) from employee_data;
```

### Naming Columns

MySQL allows you to name the displayed columns. So instead of f\_name or l\_name etc. you can use more descriptive terms. This is achieved with **AS**.

```
select avg(salary) AS  
'Average Salary' from  
employee_data;
```

Such *pseudo names* make will the display more clear to users. The important thing to remember here is that if you assign pseudo names that contain spaces, enclose the names in quotes. Here is another example:

```
select (SUM(perks)/SUM(salary) * 100)  
AS 'Perk Percentage' from
```

```
employee_data;
```

### Counting

The **COUNT()** aggregate functions counts and displays the total number of entries. For example, to count the total number of entries in the table, issue the command below.

```
select COUNT(*) from employee_data;
```

As you have learnt, the \* sign means "all data"

Now, let's count the total number of employees who hold the "Programmer" title.

```
select COUNT(*) from employee_data  
where title = 'Programmer';
```

### GROUP BY clause

The **GROUP BY** clause allows us to *group* similar data. Thus, to list all unique *titles* in our table we can issue

```
select title from employee_data  
GROUP BY title;
```

You'll notice that this is similar to the usage of **DISTINCT**, which we encountered in a previous session. Okay, here is how you can count the number of employees with different titles.

```
select title, count(*)  
from employee_data GROUP BY title;
```

### Sorting the data in MySQL

Now, let's find and list the number of employees holding different titles and sort them using ORDER BY.

```
select title, count(*) AS Number  
from employee_data  
GROUP BY title  
ORDER BY Number;
```

### **HAVING clause**

To list the average salary of employees in different departments (titles), we use the GROUP BY clause, as in:

```
select title, AVG(salary)
from employee_data
GROUP BY title;
```

Now, suppose you want to list only the departments where the average salary is more than \$100000, you can't do it, even if you assign a pseudo name to AVG(salary) column. Here, the **HAVING** clause comes to our rescue.

```
select title, AVG(salary)
from employee_data
GROUP BY title
HAVING AVG(salary) > 100000;
```

### **Displaying the current date and time**

```
select now();
```

Displaying the current Day, Month and Year

```
SELECT DAYOFMONTH(CURRENT_DATE);
```

### **Displaying text strings**

```
select 'I Love MySQL';
```

**Obviously you can provide pseudo names for these columns using AS.**

```
select 'Manish Sharma' as Name;
```

### **Concatenating in MySQL**

With SELECT you can concatenate values for display. **CONCAT** accepts arguments between parenthesis. These can be column names or plain text strings. Text strings have to be surrounded with quotes (single or double).

```
SELECT CONCAT(f_name, " ", l_name)
from employee_data
where title = 'Programmer';
```

You can also give descriptive names to these columns using **AS**.

```
select CONCAT(f_name, " ", l_name)
AS Name
from employee_data
where title = 'Marketing Executive';
```

### **MySQL mathematical Functions**

In addition to the four basic arithmetic operations addition (+), Subtraction (-), Multiplication (\*) and Division (/), MySQL also has the Modulo (%) operator. This calculates the remainder left after division.

```
select 87 % 9;
```

### **MySQL - MOD(x, y)**

Displays the remainder of x divided by y, Similar to the Modulus operator.

```
select MOD(37, 13);
```

### **MySQL ABS(x)**

Calculates the Absolute value of number **x**. Thus, if x is negative its positive value is returned.

```
select ABS(-4.05022);
```

### **SQL SIGN(x)**

Returns 1, 0 or -1 when x is positive, zero or negative, respectively.

```
select SIGN(-34.22);
```

### **POWER(x,y)**

Calculates the value of x raised to the power of y.

```
select POWER(4,3);
```

### **SQRT(x)**

Calculates the square root of x.

```
select SQRT(3);
```

### **ROUND(x) and ROUND(x,y)**

Returns the value of x rounded to the nearest integer. ROUND can also accept an additional argument **y** that will round x to y decimal places.

```
select ROUND(14.492);
```

### **FLOOR(x)**

Returns the largest integer that is less than or equal to x.

```
select FLOOR(23.544);
```

### **CEILING(x)**

Returns the smallest integer that is greater than or equal to x.

```
select CEILING(54.22);
```

### **Updating records**

The SQL **UPDATE** command updates the data in tables. Its format is quite simple.

```
UPDATE table_name SET  
column_name1 = value1,  
column_name2 = value2,  
column_name3 = value3 ...  
[WHERE conditions];
```

```
UPDATE employee_data SET  
salary=220000, perks=55000  
WHERE title='CEO';
```

Query OK, 1 row affected (0.02 sec)

Rows matched: 1 Changed: 1 Warnings: 0

### **MySQL Date column type part 1**

Till now we've dealt with text (varchar) and numbers (int) data types. To understand **date** type, we'll create one more table. Download *employee\_per.dat* file below and follow the instructions. The file contain the CREATE table command as well as the INSERT statements.

- Move the file to c:\mysql\bin.
- Issue the command at DOS prompt.  
dosprompt> mysql employees <employee\_per.dat
- Start *mysql* client program and check if the table has been created using SHOW TABLES; command.

Notice that column **birth\_date** has **date** as column type. I've also introduced another column type **ENUM**, which we'll discuss later.

**e-id**: are employee ids, same as that in table *employee\_data*

**address**: Addresses of employees

**phone**: Phone numbers

**p\_email**: Personal email addresses

**birth\_date**: Birth dates

**sex**: The sex of the employee, **Male** or **Female**

**m\_status**: Marital status, **Yes** or **No**.

**s\_name**: Name of Spouse (NULL if employee is unmarried)

**children**: Number of children (NULL if employee is unmarried)

MySQL dates are ALWAYS represented with the year followed by the month and then the date. Often you'll find dates written as **YYYY-MM-DD**, where **YYYY** is 4 digit year, **MM** is 2 digit month and **DD**, 2 digit date. We'll look at DATE and related column types in the session on column types.

### Operations on Date

Date column type allow for several operations such as sorting, testing conditions using comparison operators etc.

Using = and != operators on dates

```
select p_email, phone
from employee_per
where birth_date = '1969-12-31';
```

**Note:** MySQL requires the dates to be enclosed in quotes.

Using >= and <= operators

```
select e_id, birth_date
from employee_per where
birth_date >= '1970-01-01';
```

### Specifying date ranges in MySQL

```
select e_id, birth_date
```

```
from employee_per where  
birth_date BETWEEN  
'1969-01-01' AND '1974-01-01';
```

## MySQL Date column type part 2

Using Date to sort data

```
select e_id, birth_date  
from employee_per  
ORDER BY birth_date;
```

### Selecting data using Dates

Here is how we can select employees born in March.

```
select e_id, birth_date  
from employee_per  
where MONTH(birth_date) = 3;
```

Alternatively, we can use month names instead of numbers.

```
select e_id, birth_date  
from employee_per  
where MONTHNAME(birth_date) = 'January';
```

Be careful when using month names as they are case sensitive. Thus, *January* will work but *JANUARY* will not! Similarly, you can select employees born in a specific year or under specific dates.

```
select e_id, birth_date  
from employee_per  
where year(birth_date) = 1972;
```

```
select e_id, birth_date  
from employee_per
```

```
where DAYOFMONTH(birth_date) = 20;
```



### Current dates

We had seen in the session on SELECT statement ([A little more on the SELECT statement](#)) that current date, month and year can be displayed with **CURRENT\_DATE** argument to DAYOFMONTH(), MONTH() and YEAR() clauses, respectively. The same can be used to select data from tables.

```
select e_id, birth_date
from employee_per where
MONTH(birth_date) = MONTH(CURRENT_DATE);
```

### MySQL table joins

Till now, we've used SELECT to retrieve data from only one table. However, we can extract data from two or more tables using a single SELECT statement. The strength of RDBMS lies in allowing us to *relate* data from one table with data from another. This correlation can only be made if at least one column in the two tables contain related data. In our example, the columns that contain related data are **emp\_id** of *employee\_data* and **e\_id** of *employee\_per*. Let's conduct a table join and extract the names (from *employee\_data*) and spouse names (from *employee\_per*) of married employee.

```
select CONCAT(f_name, " ", l_name) AS Name,
s_name as 'Spouse Name' from
employee_data, employee_per
where m_status = 'Y' AND
emp_id = e_id;
```

The **FROM** clause takes the names of the two tables from which we plan to extract data. Also, we specify that data has to be retrieved for only those entries where the *emp\_id* and *e\_id* are same.

The names of columns in the two tables are unique. However, this may not be true always, in which case we can explicitly specify column names along with table name using the **dot notation**.

```
select CONCAT(employee_data.f_name, " ", employee_data.l_name)
AS Name, employee_per.s_name AS 'Spouse Name'
```

```
from employee_data, employee_per
```

```
where employee_per.m_status = 'Y'
```

```
AND employee_data.emp_id = employee_per.e_id;
```

### Deleting entries from tables

The SQL delete statement requires the table name and optional conditions.

```
DELETE from table_name [WHERE conditions];
```

### **NOTE: If you don't specify any conditions ALL THE DATA IN THE TABLE WILL BE DELETED!!!**

One of the Multimedia specialists 'Hasan Rajabi' (employee id 10) leaves the company. We'll delete his entry.

```
DELETE from employee_data  
WHERE emp_id = 10;  
Query OK, 1 row affected (0.00 sec)
```

### Dropping tables

To remove all entries from the table we can issue the DELETE statement without any conditions.

```
DELETE from employee_data;  
Query OK, 0 rows affected (0.00 sec)
```

However, this does not delete the table. The table still remains, which you can check with SHOW TABLES;

```
mysql> SHOW TABLES;
```

To delete the table, we issue a DROP table command.

```
DROP TABLE employee_data;  
Query OK, 0 rows affected (0.01 sec)
```

### MySQL database Column Types

The three major types of column types used in MySQL are

- Integer
- Text
- Date

Choosing a column data type is very important in order to achieve speed, effective storage and retrieval. Hence, I've dedicated two sessions to this topic.

Now, I'll be touching only the surface; for a thorough explanation refer the resources in [What Next?](#) session.

### Numeric Column Types

In addition to **int** (Integer data type), MySQL also has provision for floating-point and double precision numbers. Each integer type can take also be UNSIGNED and/or AUTO\_INCREMENT.

- **TINYINT**: very small numbers; suitable for ages. Actually, we should have used this data type for employee ages and number of children. Can store numbers between 0 to 255 if UNSIGNED clause is applied, else the range is between -128 to 127.
- **SMALLINT**: Suitable for numbers between 0 to 65535 (UNSIGNED) or -32768 to 32767.
- **MEDIUMINT**: 0 to 16777215 with UNSIGNED clause or -8388608 to 8388607.
- **INT**: UNSIGNED integers fall between 0 to 4294967295 or -2147683648 to 2147683647.
- **BIGINT**: Huge numbers. (-9223372036854775808 to 9223372036854775807)
- **FLOAT**: Floating point numbers (single precision)
- **DOUBLE**: Floating point numbers (double precision)
- **DECIMAL**: Floating point numbers represented as strings.

### Date and time column types

- **DATE**: YYYY-MM-DD (Four digit year followed by two digit month and date)
- **TIME**: hh:mm:ss (Hours:Minutes:Seconds)
- **DATETIME**: YYYY-MM-DD hh:mm:ss (Date and time separated by a space character)
- **TIMESTAMP**: YYYYMMDDhhmmss
- **YEAR**: YYYY (4 digit year)

### MySQL Text data type

Text can be fixed length (**char**) or variable length strings. Also, text comparisons can be case sensitive or insensitive depending on the type you choose.

**CHAR(x)**: where x can range from 1 to 255.

**VARCHAR(x)**: x ranges from 1 - 255

**TINYTEXT**: small text, case insensitive

**TEXT**: slightly longer text, case insensitive

**MEDIUMTEXT**: medium size text, case insensitive

**LONGTEXT**: really long text, case insensitive

**TINYBLOB**: Blob means a **B**inary **L**arge **O**bject. You should use blobs for case sensitive searches.

**BLOB**: slightly larger blob, case sensitive.

**MEDIUMBLOB**: medium sized blobs, case sensitive.

**LOBLOB**: really huge blobs, case sensitive.

**ENUM:** Enumeration data type have fixed values and the column can take only one value from the given set. The values are placed in parenthesis following ENUM declaration. An example, is the marital status column we encountered in *employee\_per* table.

- `m_status ENUM("Y", "N")`

Thus, `m_status` column will take only **Y** or **N** as values. If you specify any other value with the INSERT statement, MYSQL will not return an error, it just inserts a NULL value in the column.

**SET:** An extension of ENUM. Values are fixed and placed after the SET declaration; however, SET columns can take multiple values from the values provided. Consider a column with the SET data type as

- `hobbies SET ("Reading", "Surfing", "Trekking", "Computing")`

You can have 0 or all the four values in the column.

```
INSERT tablename (hobbies) values ("Surfing", "Computing");
```

### Alter Table

ALTER TABLE enables you to change the structure of an existing table. For example, you can add or delete columns, create or destroy indexes, change the type of existing columns, or rename columns or the table itself. You can also change the comment for the table and type of the table.

### ALTER TABLE *tablename* CLAUSE

Clause	Usages	Remarks
ADD COLUMN	ALTER TABLE <i>tablename</i> ADD Column <i>column-name</i> ...	Add a new columns
CHANGE COLUMN	ALTER TABLE <i>tablename</i> CHANGE COLUMN <i>column_name</i> <i>column_name</i> VARCHAR(25)	Change the data type and properties of a column
DROP COLUMN	ALTER TABLE <i>tablename</i> DROP COLUMN <i>column_name</i>	Remove a column from a table
ADD INDEX	ALTER TABLE <i>tablename</i> ADD INDEX	Add a new index

	indexname(column_name)	
DROP INDEX	ALTER TABLE tablename DROP INDEX indexname	Remove an existing index
RENAME AS	ALTER TABLE tablename RENAME AS new_tablename	

### Reference:

1. David A. Wheeler "Why Open Source Software / Free Software (OSS/FS)? Look at the Numbers!" URL- <http://www.dwheeler.com/contactme.html>

### About the author:



**Goutam Biswas, Research Fellow,  
Dept. of Lib. & Inf. Sc, University of Kalyani, West Bengal, India .**