MySQL Schema and State

From CSE330 Wiki

MySQL is a powerful database that allows for complex logic when manipulating data. This guide is an introduction and a reference to common SQL queries for schema and state manipulation.

This guide assumes that you already have a working copy of MySQL. If you don't, read Introduction to MySQL first.

Contents

Schema

The *schema* of your database refers to the way that your data is structured and to how bits of data relate to each other. Every database in your MySQL server has a separate schema, and separate databases should not typically interact with one another.

Tables

A database contains one or more *tables*. Tables contain columns and rows, which you may sometimes call *fields* and *entries*.

You can see all of the tables in the database using this query:

```
SHOW tables;
```

When you create a table, you should provide information about all of that table's columns within the *create table* query. Here is the general syntax for the query:

```
CREATE TABLE TABLENAME (
    FIELDNAME DATATYPE OPTIONS,
    FIELDNAME DATATYPE OPTIONS,
    FIELDNAME DATATYPE OPTIONS,
    FIELDNAME DATATYPE OPTIONS
) TABLE_OPTIONS;
```

You can get details about an existing table using this query:

```
DESCRIBE TABLENAME;
```

Data Types

Just like a programming language, MySQL has data types. There are dozens of data types; some of the most common ones are documented below.

String Data Types

There are four main types of string data types: character, text, blob, and enumerable.

- Character types are useful for short strings, like usernames and e-mail addresses.
- Text types differ from character types in that behind the scenes, they are not actually stored with the rest of the table row; they are useful for longer blocks of text, like forum posts.
- Blobs are for binary data, like files.
- Enumerables are for something that could have one of a set number of values (not more than 65535), like what State a user is from (Alabama, Alaska, and so on). Enums are also useful for storing boolean data, like **ENUM('yes','no')**. An enum is actually stored as an integer in the database, making queries more efficient.

The syntax for these data types is:

- **VARCHAR(#)**, a string *at most* # characters long, where 0 ≤ # ≤ 255
- **CHAR(#)**, a string *exactly* # characters long, where 0 ≤ # ≤ 255
- **TINYTEXT**, a text block at most 255 characters long
- **TEXT**, a text block at most 65535 characters long
- **BLOB**, a binary string at most 65 KB in length
- **MEDIUMBLOB**, a binary string at most 16 MB in length
- **LONGBLOB**, a binary string at most 4 GB in length
- **ENUM('a', 'b', 'c')**, an enumerable with choices a, b, and c

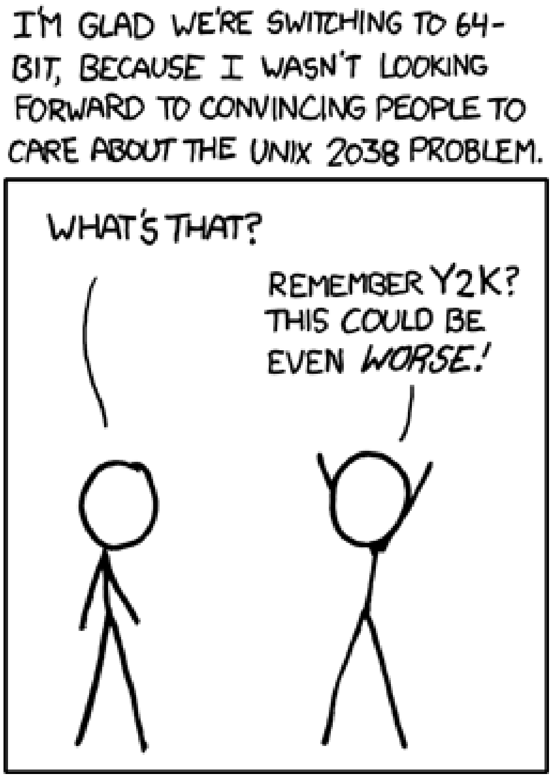
Numeric Data Types

There are two types of numeric data types that you need to worry about: integers and decimals. The difference between the two should be pretty self-explanatory. The syntax for using these data types is:

- **TINYINT**, an integer *n* where -128 ≤ n ≤ 127
- **TINYINT UNSIGNED**, an integer *n* where 0 ≤ n ≤ 255
- **SMALLINT**, an integer *n* where -32,768 ≤ n ≤ 32,767
- **SMALLINT UNSIGNED**, an integer *n* where 0 ≤ n ≤ 65,535
- **MEDIUMINT**, an integer *n* where -8,388,608 ≤ n ≤ 8,388,607
- **MEDIUMINT UNSIGNED**, an integer *n* where 0 ≤ n ≤ 16,777,215
- **INT**, an integer *n* where -2,147,483,648 ≤ n ≤ 2,147,483,647
- **INT UNSIGNED**, an integer *n* where 0 ≤ n ≤ 4,294,967,295
- **BIGINT**, an integer *n* where -9,223,372,036,854,775,808 ≤ n ≤ 9,223,372,036,854,775,807
- **BIGINT UNSIGNED**, an integer *n* where 0 ≤ n ≤ 18,446,744,073,709,551,615
- **DECIMAL(#a, #b)**, a decimal number at most *#a* digits long, *#b* of which are after the decimal point. For example, *DECIMAL(5,2)* can contain *n* where -999.99 ≤ n ≤ 999.99

When choosing the size of an integer, you should use the *smallest integer for which you know you will never "run out of room"*. In most everyday circumstances, **MEDIUMINT** should be the biggest you'll need.

Temporal Data Types


I'M GLAD WE'RE SWITCHING TO 64-BIT, BECAUSE I WASN'T LOOKING FORWARD TO CONVINCING PEOPLE TO CARE ABOUT THE UNIX 2038 PROBLEM.
WHAT'S THAT?
REMEMBER Y2K? THIS COULD BE EVEN WORSE!

MySQL provides a class of data types that specialize in handling dates and times. The syntax for using date types is:

- **DATE**, a date *d* where `1000-01-01` ≤ d ≤ `9999-12-31`
- **DATETIME**, a date with time *d* where `1000-01-01 00:00:00` ≤ d ≤ `9999-12-31 23:59:59`
- **TIMESTAMP**, a date with time *d* where `1970-01-01 00:00:01 UTC` ≤ d ≤ `2038-01-19 03:14:07 UTC` (note: `1970-01-01 00:00:01 UTC` is the Unix epoch)
- **TIME**, a length of time *t* where `-838:59:59` ≤ t ≤ `838:59:59` (note: you cannot store time to a precision greater than 1 second)
- **YEAR**, a year *y* where 1901 ≤ y ≤ 2155

Note that **TIMESTAMP** is a highly versatile data type that can automatically change whenever a row is updated. For more information, refer to the MySQL documentation (http://dev.mysql.com/doc/refman/5.6/en/timestamp-initialization.html) .

Field Options

By default, any "cell" in a MySQL table can contain the *NULL* value. If a certain field in your database schema will never contain a null value, you should declare it as **NOT NULL** in order to make queries more efficient.

You can also specify a default value for a field. If you do, when you insert data into the table, it won't be necessary to specify a value for that column. To specify a default value for a field, use the syntax **DEFAULT 'DEFAULT-VALUE'**

Create Table Example

The following query would create a table called *employees* with six columns: employee ID, first name, last name, nickname, department, and the date that they joined the company. Nickname is the only optional column. The primary key for the table (see the next section) will be the *id* column. The department will be one of five choices, with the default being CSE. The storage engine for the table will be InnoDB, and the default charset for the table will be UTF8.

```
CREATE TABLE employees (
    id MEDIUMINT UNSIGNED NOT NULL AUTO_INCREMENT,
    first_name VARCHAR(30) NOT NULL,
    last_name VARCHAR(40) NOT NULL,
    nickname VARCHAR(20),
    department ENUM('CSE','BME','EECE','ESE','MEMS') NOT NULL
DEFAULT 'CSE',
    joined TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (id)
) engine = INNODB DEFAULT character SET = utf8 COLLATE =
utf8_general_ci;
```

Keys

When MySQL goes to get data out of a database, it does so using an index, like the index of a textbook. In tech talk, we call indices *keys*.

Primary Key

The **primary key** is what MySQL uses to distinguish one row from another. Every table must have exactly one primary key; no more, no less.

It is common practice to add a column to a table with data type **mediumint unsigned not null auto_increment** to serve as the primary key for a table. The **auto_increment** is a neat little feature of MySQL that basically says, "when a new row is inserted into this table, assign it the next-lowest available ID".

To specify a primary key for a table, use the syntax:

```
PRIMARY KEY (column1, column2, column3)
```

Unique Key

If you want to specify that all items in a certain column will be unique from each other (like usernames or e-mail addresses), you can use a **unique key**.

Because a table can have more than one unique key, you need to give your unique key a name. It is common practice to name the key after the columns it is indexing, like *idx_unique_column1_column2*.

The syntax for specifying a unique key is:

```
UNIQUE key INDEX_NAME (column1, column2, column3)
```

Foreign Key

A **foreign key** is the most complicated type of MySQL index. A foreign key says that the values in one column or columns are associated with the values in a column or columns in a *different table*.

For example, suppose you had two tables, one containing users and the other containing posts on a forum. There is a one-to-many relationship between users and posts. You could link a post to a user by having a column in your post table called *user_id*, which would be a *foreign key* to the *id* column in your users table.

Foreign keys can be a source of great confusion in developing database schemas. If you get stuck, keep this tip in mind: **Foreign Keys should always reference the columns used in the Primary Key of a different table.**

The syntax for creating a foreign key is as follows:

```
FOREIGN key (column1, column2, column3) REFERENCES
FOREIGN_TABLENAME (column1, column2, column3)
```

For example, in the user-and-post example above, the foreign key declaration would be:

```
FOREIGN key (user_id) REFERENCES user (id)
```

Further Reading about Foreign Keys

- Wikipedia Entry on Foreign Keys
- An introductory article on Tech Republic (http://www.techrepublic.com/article/an-introduction-to-foreign-keys-and-referential-integrity-in-mysql/6035435)
- MySQL Documentation for Foreign Keys (http://dev.mysql.com/doc/refman/5.1/en/innodb-foreign-key-constraints.html)

Syntax for Keys

You can define keys in your **create table** query, or you can define them later.

If you define them in your **create table** query, do so after the column definitions but still inside the parentheses:

```
CREATE TABLE TABLENAME (
    COLUMN,
    COLUMN,
    ....,
    PRIMARY KEY (column1, column2, ...),
    UNIQUE key INDEXNAME (column1, column2, ...),
    FOREIGN key (column1, column2, ...) REFERENCES
FOREIGN_TABLENAME (column1, column2, ...)
)
```

To define indices later, use an **ALTER TABLE** query:

```
ALTER TABLE TABLENAME add UNIQUE key INDEXNAME (column1, column2,
...);
ALTER TABLE TABLENAME add FOREIGN key (column1, column2, ...)
REFERENCES FOREIGN_TABLENAME (column1, column2, ...);
```

State

The *state* of your database refers to the data that it holds. Whenever you insert, update, or delete data, the state of your database changes.

Inserting Data from a Query

To insert a row (aka *entry*) into a table, use a query of the form

```
INSERT INTO TABLENAME (column1, column2, column3) VALUES
(value1_1, value2_1, value3_1), (value1_2, value2_2, value3_2),
...
```

For example, to insert an employee into the table we created earlier, you could use the query

```
INSERT INTO employees (first_name, last_name, department) VALUES
('John', 'Doe', 'BME'), ('Keith', 'Jones', 'ESE');
```

Because of the various default and auto_increment values we have specified in our database, the database would now contain the following data:

```
+----+------------+-----------+----------+------------+---------------------+
| id | first_name | last_name | nickname | department | joined              |
+----+------------+-----------+----------+------------+---------------------+
|  1 | John       | Doe       | NULL     | BME        | 2012-08-22 22:02:24 |
|  2 | Keith      | Jones     | NULL     | ESE        | 2012-08-22 22:02:24 |
+----+------------+-----------+----------+------------+---------------------+
```

If you want to insert a DATE or DATETIME value into a database, use the YYYY-MM-DD

HH:MM:SS format. The exception is with TIMESTAMP, in which you should use a 16-bit integer representing the number of seconds since January 1, 1970.

Inserting Data from a File

The **load data infile** command populates a table using data stored in a text file. The general form of the command is:

```
LOAD DATA INFILE 'dataFile.txt' INTO TABLE table_name;
```

where *dataFile.txt* is located relative to the server's data directory if a path name with leading components is given; otherwise, if no leading components are given, the server assumes the file is in the default database's database directory. If an absolute path is specified, the server uses it. For this command to work, you must issue the following command to grant the FILE privilege to your user:

```
GRANT FILE on *.* to 'user'@'host';
```

Alternatively, and quite usefully, you can load data from a local file (a file residing on the same machine as the client) using the LOCAL keyword. When loading data from the local machine you do not need to grant your user the FILE privilege, think about why this might be the case. (Hint: what is going on behind the scenes when you load the data in from a local machine when your database is hosted on a different machine?)

```
LOAD DATA LOCAL INFILE
'/home/linus_torvalds/Documents/dataFile.txt' INTO TABLE
table_name;
```

Note: When the local keyword is used, it is best to specify an absolute path.

Additional Note: If we SSH into our EC2 instance and then run the mysql command, the client is the EC2 instance, not the local computer that we SSH from. The LOCAL keyword in this case is allowing us to load files from the EC2 instance, not the local computer that we are using.

Debian Quirk: If you are using a Debian-based system, you may need to pass a `--local-infile` flag to the `mysql` command.

By default, the text files are assumed to have tab-delimited columns with column values specified in the same order as they are defined in the table (though you can specify additional/alternative delimiters using additional options).

Updating Data

To update a row in a table, use a query of the form

```
UPDATE TABLENAME SET column1=value1, column2=value2,
column3=value3 WHERE CONDITION
```

For example, to set Keith Jones's nickname to "KJ", you would run the query

```
UPDATE employees SET nickname='KJ' WHERE id=2;
```

Deleting Data

To delete a row from a table, use a query of the form

```
DELETE FROM TABLENAME WHERE CONDITION
```

For example, to delete John Doe from the table, you would run the query

```
DELETE FROM employees WHERE id=1;
```

DO NOT FORGET THE WHERE CLAUSE! If you do, *all* rows will be deleted from your table!

Selecting Data

The most simple query to select data from a table is:

```
SELECT * FROM TABLENAME;
```

However, in practice, you will almost always want to perform a more advanced query.

Selecting Specific Columns

In order to save transmission time, you should select only those columns that you need in your application. The syntax for selecting only certain columns is

```
SELECT column1, column2, column3 FROM TABLENAME;
```

Computation

MySQL can perform computation. For example, if you wanted to select the amount of time that a certain employee has been at the company, you could run the query

```
SELECT TIMESTAMPDIFF( SECOND, joined, NOW() ) as time_with_company
FROM employees;
```

Joins

When you have one-to-one, one-to-many, or many-to-many relationships between tables, you will often want to select data from multiple tables in one query. To do this, use **joins**.

Joins are best described by example. Suppose you had the following tables in a MySQL database:

```
mysql> describe employees;
+------------+--------------------------------------+------+-----+-------------------+----------------+
| Field      | Type                                 | Null | Key | Default           | Extra          |
+------------+--------------------------------------+------+-----+-------------------+----------------+
| id         | mediumint(8) unsigned                | NO   | PRI | NULL              | auto_increment |
| first_name | varchar(30)                          | NO   |     | NULL              |                |
| last_name  | varchar(40)                          | NO   |     | NULL              |                |
| nickname   | varchar(20)                          | YES  |     | NULL              |                |
| department | enum('CSE','BME','EECE','ESE','MEMS') | NO   |     | CSE               |                |
| joined     | timestamp                            | NO   |     | CURRENT_TIMESTAMP |                |
+------------+--------------------------------------+------+-----+-------------------+----------------+
6 rows in set (0.00 sec)

mysql> describe projects;
+--------------+-----------------------+------+-----+---------+----------------+
| Field        | Type                  | Null | Key | Default | Extra          |
+--------------+-----------------------+------+-----+---------+----------------+
| id           | mediumint(8) unsigned | NO   | PRI | NULL    | auto_increment |
| employee_id  | mediumint(8) unsigned | NO   | MUL | NULL    |                |
| project_name | tinytext              | NO   |     | NULL    |                |
+--------------+-----------------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)
```

Further, suppose the tables had the following data:

```
mysql> select * from employees;
+----+------------+-----------+----------+------------+---------------------+
| id | first_name | last_name | nickname | department | joined              |
+----+------------+-----------+----------+------------+---------------------+
|  1 | John       | Doe       | NULL     | BME        | 2012-08-22 22:02:24 |
|  2 | Keith      | Jones     | KK       | ESE        | 2012-08-22 22:02:24 |
+----+------------+-----------+----------+------------+---------------------+
2 rows in set (0.00 sec)

mysql> select * from projects;
+----+-------------+-----------------------------------+
| id | employee_id | project_name                      |
+----+-------------+-----------------------------------+
|  1 |           1 | Interview the Chancellor          |
|  2 |           1 | Give Presentation in Mumbai       |
|  3 |           2 | Publish Research Paper in Journal |
+----+-------------+-----------------------------------+
3 rows in set (0.00 sec)
```

If you ran the following query:

```sql
SELECT
    project_name,
    employees.first_name,
    employees.last_name
FROM projects
    JOIN employees on (projects.employee_id=employees.id);
```

You would get the result:

```
mysql> select project_name, employees.first_name, employees.last_name from projects
    -> join employees on (projects.employee_id=employees.id);
+-----------------------------------+------------+-----------+
| project_name                      | first_name | last_name |
+-----------------------------------+------------+-----------+
| Interview the Chancellor          | John       | Doe       |
| Give Presentation in Mumbai       | John       | Doe       |
| Publish Research Paper in Journal | Keith      | Jones     |
+-----------------------------------+------------+-----------+
3 rows in set (0.00 sec)
```

Isn't that cool?

Inner Joins, Left Joins, Right Joins, and Full Joins

When you perform a **JOIN** like we did above, you are implicitly performing an **INNER JOIN**. However, there may be times when you want to select data in a query even if it doesn't have a complement in the joined table. For example, suppose Keith Jones cancelled his project. The projects table now contains the following data:

```
mysql> select * from projects;
+----+-------------+-----------------------------+
| id | employee_id | project_name                |
+----+-------------+-----------------------------+
|  1 |           1 | Interview the Chancellor    |
|  2 |           1 | Give Presentation in Mumbai |
+----+-------------+-----------------------------+
2 rows in set (0.00 sec)
```

Now, we could run the following query:

```
SELECT
    count(projects.id) as project_num,
    employees.first_name,
    employees.last_name
FROM employees
    JOIN projects on (employees.id=projects.employee_id)
GROUP BY employees.id;
```

We get the result:

```
mysql> select count(projects.id) as project_num, employees.first_name, employees.last_name from employees
    -> join projects on (employees.id=projects.employee_id) group by employees.id;
+-------------+------------+-----------+
| project_num | first_name | last_name |
+-------------+------------+-----------+
|           2 | John       | Doe       |
+-------------+------------+-----------+
1 row in set (0.00 sec)
```

But where's Keith Jones? Since Keith Jones didn't have any projects, he was omitted from the result. This is probably not what we intended to happen. Instead, you should perform a **LEFT JOIN**, like so:

```
SELECT
    count(projects.id) as project_num,
    employees.first_name,
    employees.last_name
FROM employees
    LEFT JOIN projects on (employees.id=projects.employee_id)
GROUP BY employees.id;
```

The result will now be:

```
mysql> select count(projects.id) as project_num, employees.first_name, employees.last_name from employees
    -> left join projects on (employees.id=projects.employee_id) group by employees.id;
```

```
+-------------+------------+-----------+
| project_num | first_name | last_name |
+-------------+------------+-----------+
|           2 | John       | Doe       |
|           0 | Keith      | Jones     |
+-------------+------------+-----------+
2 rows in set (0.00 sec)
```

(For more information on COUNT and GROUP BY, see a later section.)

You can see that a LEFT JOIN preserves all rows from the "left" table in the join. Likewise, a RIGHT JOIN preserves all rows from the "right" table. A FULL JOIN preserves all rows from both tables.

WHERE Clauses

If you want to select data that satisfies a condition, use a **WHERE** clause.

A simple **WHERE** clause can use equality, less than, greater than, and most other comparison operators. For example, the following query selects all last names of employees with an Employee ID of less than 10:

```
SELECT
    last_name
FROM employees
WHERE
    employee_id < 10;
```

If we wanted to select everyone from our employees table whose name started with a J, we could use a **LIKE** statement in our **WHERE** clause:

```
SELECT
    first_name,
    last_name
FROM employees
WHERE
    first_name LIKE 'J%';
```

The **LIKE** clause is SQL's way of letting you perform simple tests on strings. (SQL's built-in string comparison operations are not as powerful as regular expressions, but they'll get the job done in most everyday situations.)

Sorting Data

You can sort data using an **ORDER BY** clause like so:

```
SELECT
    first_name,
    last_name
FROM employees
ORDER BY last_name asc;
```

If you wanted to select only the first few results from a query, use the **LIMIT** clause:

```
SELECT
    first_name,
    last_name
FROM employees
ORDER BY last_name asc
LIMIT 10;
```

It is important to note that the ORDER BY on a field is performed *after* aggregation functions. Currently, there is no way in MySQL to perform sorting *before* aggregation has occurred in a single query; we need to resort to nested queries. (For example, in the baseball example below, it's not easy to select "the player from each team with the highest batting average". It's even harder, if not impossible, to select "the *three* players from each team with the highest batting averages".)

Aggregation Clauses

When performing computations on an entire database, you will probably find yourself wanting to use *aggregation functions*. Aggregation functions include:

- **AVG(column_name)** to compute the average value of column_name
- **STD(column_name)** to compute the standard deviation of column_name
- **SUM(column_name)** to compute the sum
- **MIN(column_name)** to find the minimum value
- **MAX(column_name)** to fund the maximum value
- **COUNT(column_name)** to count the total number of entries having a not null entry in column_name

For example, if you had a table containing batter averages, you could calculate the mean batter average using this query:

```
SELECT
    avg(batter_average) as avg_batter_average
FROM baseball_players;
```

Since WHERE clauses are performed *before* aggregation, you can also perform a query like

```
SELECT
    avg(batter_average) as avg_batter_average
FROM baseball_players
WHERE
    team_id=5;
```

If you want to aggregate all rows that share a certain value in an index, use a **GROUP BY** clause. For example, to select the average batter average for all teams, you could do

```
SELECT
    avg(batter_average) as avg_batter_average
FROM baseball_players
```

```
GROUP BY team_id;
```

You may find yourself wanting to apply a condition after the aggregation has occurred. You can do this using a **HAVING** clause. For example, the following query would select all baseball teams whose average batter average is more than 300:

```
SELECT
    avg(batter_average) as avg_batter_average
FROM baseball_players
GROUP BY team_id
HAVING
    avg_batter_average > 300;
```

It is perfectly valid to combine WHERE and HAVING clauses in a single query. For example, the following query first finds all players whose first name starts with "J", averages their batting averages together by team, and then selects those teams whose average batting average of the "J" players is more than 300; that is, it selects those teams "whose average batting average of players whose first name starts with J is more than 300":

```
SELECT
    avg(batter_average) as avg_batter_average
FROM baseball_players
WHERE
    first_name LIKE "J%"
GROUP BY team_id
HAVING
    avg_batter_average > 300;
```