SQL for Beginners Part 2

<https://code.tutsplus.com/articles/sql-for-beginners-part-2--net-8274>

RUS

<http://jtest.ru/bazyi-dannyix/sql-dlya-nachinayushhix-chast-2.html>

by [Burak Guzel](https://tutsplus.com/authors/burak-guzel)22 Dec 2009

Length:LongLanguages:

[Databases](https://code.tutsplus.com/categories/databases)[Web Development](https://code.tutsplus.com/categories/web-development)[SQL](https://code.tutsplus.com/categories/sql)

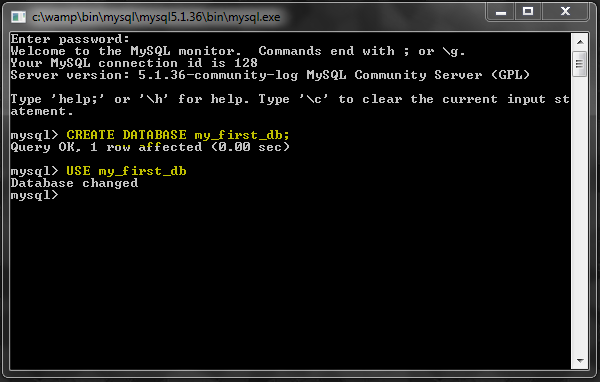
It is important for every web developer to be familiar with database interactions. In part two of the [series](http://net.tutsplus.com/tutorials/other/sql-for-beginners/), we will continue exploring the SQL language and apply what we've learned on a MySQL database. We will learn about **Indexes, Data Types** and more complex query structures.

**What You Need**

Please refer to the "What You Need" section in the first article here: [SQL For Beginners (part 1)](http://net.tutsplus.com/tutorials/other/sql-for-beginners/).

If you would like to follow the examples in this article on your own development server, do the following:

1. Open MySQL Console and login.
2. If you haven't already, create a database named "my\_first\_db" with a CREATE query.
3. **Switch to the database with the USE statement**.



**Database Indexes**

**Indexes (or keys)** are mainly used for improving the speed of data retrieval operations (eg. SELECT) on tables.

They are such an important part of a good database design, it's hard to classify them as "optimization". In most cases they are included in the initial design, but they can also be added later on with an **ALTER TABLE** query.

Most common reasons for indexing database columns are:

* Almost every table should have a PRIMARY KEY index, usually as an "id" column.
* If a column is expected to contain unique values, it should have a **UNIQUE index**.
* If you are going to perform searches on a column often (in the WHERE clause), it should have a **regular INDEX**.
* If a column is used for a relationship with another table, it should be a **FOREIGN KEY** if possible, **or** have just a **regular index** otherwise.

**PRIMARY KEY**

Almost every table should have a PRIMARY KEY, in most cases as an INT with the AUTO\_INCREMET option.

If you recall from the [first article](http://net.tutsplus.com/tutorials/other/sql-for-beginners/), we created a 'user\_id' field in the users table and it was a PRIMARY KEY. This way, in a web application we can refer to all users by their id numbers.

**NOTE!!! The values stored in a PRIMARY KEY column must be unique. Also, there can not be more one PRIMARY KEY on each table.**

Let's see a sample query, creating a table for USA states list:

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE TABLE states (      id INT AUTO\_INCREMENT PRIMARY KEY,      name VARCHAR(20)  ); |

It can also be written like this:

|  |  |
| --- | --- |
| 1  2  3  4  5 | CREATE TABLE states (      id INT AUTO\_INCREMENT,      name VARCHAR(20),      PRIMARY KEY (id)  ); |

**UNIQUE**

Since we are expecting the state name to be a unique value, we should change the previous query example a bit:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | CREATE TABLE states (      id INT AUTO\_INCREMENT,      name VARCHAR(20),      PRIMARY KEY (id),      UNIQUE (name)  ); |

By default, the index will be named after the column name. If you want to, you can assign a different name to it:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | CREATE TABLE states (      id INT AUTO\_INCREMENT,      name VARCHAR(20),      PRIMARY KEY (id),      UNIQUE state\_name (name)  ); |

Now the index is named 'state\_name' instead of 'name'.

**INDEX**

Let's say we want to add a column to represent the year that each state joined.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE TABLE states (      id INT AUTO\_INCREMENT,      name VARCHAR(20),      join\_year INT,      PRIMARY KEY (id),      UNIQUE (name),      INDEX (join\_year)  ); |

I just added the join\_year column and indexed it. This type of index does not have the uniqueness restriction.

**You can also name it KEY instead of INDEX.**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE TABLE states (      id INT AUTO\_INCREMENT,      name VARCHAR(20),      join\_year INT,      PRIMARY KEY (id),      UNIQUE (name),      KEY (join\_year)  ); |

**More About Performance**

**Adding an index reduces the performance of INSERT and UPDATE queries**. Because every time new data is added to the table, the index data is also updated automatically, which requires additional work.

**The performance gains** on the SELECT queries usually outweigh this by far. But still, do not just add indexes on every single table column ***without thinking about the queries you will be running.***

**Sample Table**

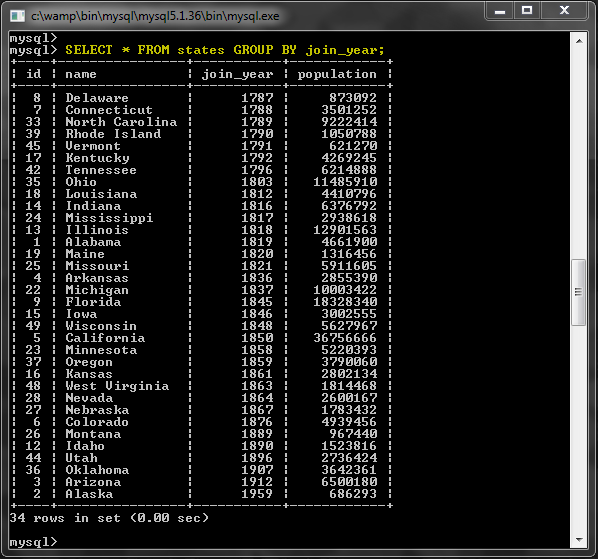
Before we go further with more queries, I would like to create a sample table with some data.

This will be a list of US states, with their join dates (the date the state ratified the United States Constitution or was admitted to the Union) and their current populations. You can copy paste the following to your MySQL console:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | CREATE TABLE states (    id INT AUTO\_INCREMENT,    name VARCHAR(20),    join\_year INT,    population INT,    PRIMARY KEY (id),    UNIQUE (name),    KEY (join\_year)  );      INSERT INTO states VALUES  (1, 'Alabama', 1819, 4661900),  (2, 'Alaska', 1959, 686293),  (3, 'Arizona', 1912, 6500180),  (4, 'Arkansas', 1836, 2855390),  (5, 'California', 1850, 36756666),  (6, 'Colorado', 1876, 4939456),  (7, 'Connecticut', 1788, 3501252),  (8, 'Delaware', 1787, 873092),  (9, 'Florida', 1845, 18328340),  (10, 'Georgia', 1788, 9685744),  (11, 'Hawaii', 1959, 1288198),  (12, 'Idaho', 1890, 1523816),  (13, 'Illinois', 1818, 12901563),  (14, 'Indiana', 1816, 6376792),  (15, 'Iowa', 1846, 3002555),  (16, 'Kansas', 1861, 2802134),  (17, 'Kentucky', 1792, 4269245),  (18, 'Louisiana', 1812, 4410796),  (19, 'Maine', 1820, 1316456),  (20, 'Maryland', 1788, 5633597),  (21, 'Massachusetts', 1788, 6497967),  (22, 'Michigan', 1837, 10003422),  (23, 'Minnesota', 1858, 5220393),  (24, 'Mississippi', 1817, 2938618),  (25, 'Missouri', 1821, 5911605),  (26, 'Montana', 1889, 967440),  (27, 'Nebraska', 1867, 1783432),  (28, 'Nevada', 1864, 2600167),  (29, 'New Hampshire', 1788, 1315809),  (30, 'New Jersey', 1787, 8682661),  (31, 'New Mexico', 1912, 1984356),  (32, 'New York', 1788, 19490297),  (33, 'North Carolina', 1789, 9222414),  (34, 'North Dakota', 1889, 641481),  (35, 'Ohio', 1803, 11485910),  (36, 'Oklahoma', 1907, 3642361),  (37, 'Oregon', 1859, 3790060),  (38, 'Pennsylvania', 1787, 12448279),  (39, 'Rhode Island', 1790, 1050788),  (40, 'South Carolina', 1788, 4479800),  (41, 'South Dakota', 1889, 804194),  (42, 'Tennessee', 1796, 6214888),  (43, 'Texas', 1845, 24326974),  (44, 'Utah', 1896, 2736424),  (45, 'Vermont', 1791, 621270),  (46, 'Virginia', 1788, 7769089),  (47, 'Washington', 1889, 6549224),  (48, 'West Virginia', 1863, 1814468),  (49, 'Wisconsin', 1848, 5627967),  (50, 'Wyoming', 1890, 532668); |

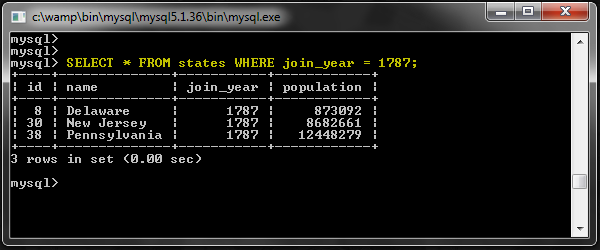
**GROUP BY: Grouping Data**

The GROUP BY clause groups the resulting data rows into groups. Here is an example:



So what just happened? We have 50 rows in the table, but 34 results were returned by this query. This is because the results were grouped by the 'join\_year' column. In other words, **we only see one row for each distinct value of join\_year**. Since some states have the same join\_year, we got less than 50 results.

For example, there was only one row for the year 1787, but there are 3 states in that group:



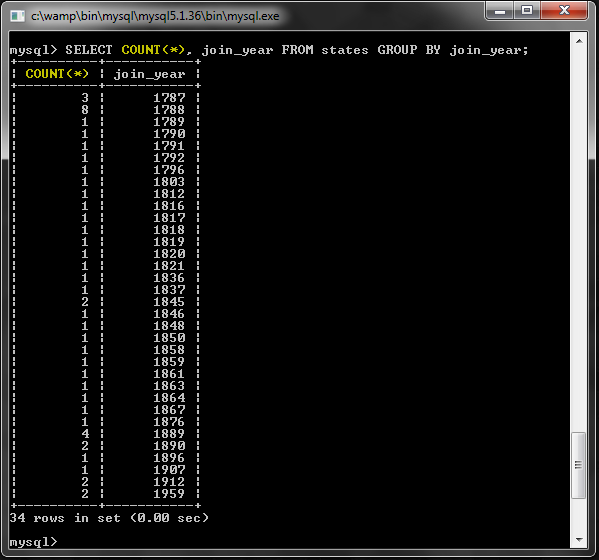
So there are three states here, but only Delaware's name showed up after the GROUP BY query earlier. Actually, it could have been any of the three states and we can not rely on this piece of data. Then what is the point of using the GROUP BY clause?

It would be mostly useless without using an [aggregate function](http://dev.mysql.com/doc/refman/5.1/en/group-by-functions.html) such as COUNT(). Let's see what some of these functions do and how they can get us some useful data.

**COUNT(\*): Counting rows**

This is perhaps the most commonly used function along with GROUP BY queries. It returns the number of rows in each group.

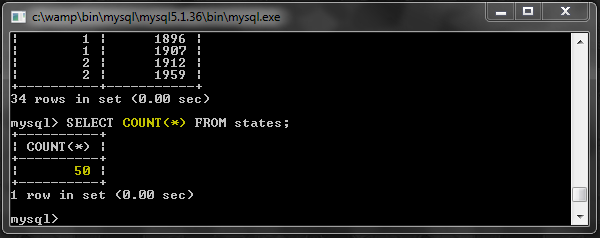
For example we can use it to see the number of states for each join\_year:



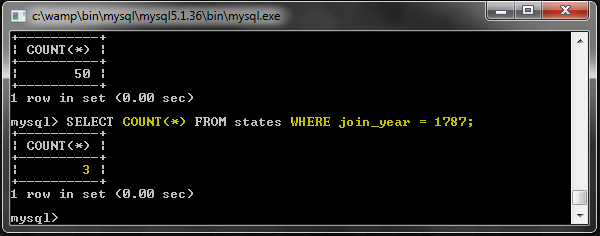
**Grouping Everything**

**If you use a GROUP BY aggregate function, and do not specify a GROUP BY clause, the entire results will be put in a single group.**

Number of all rows in the table:

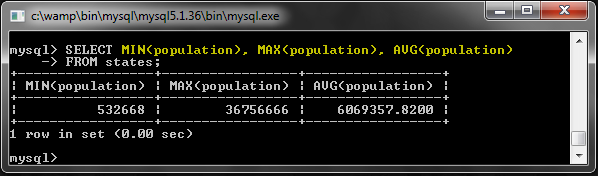


Number of rows satisfying a WHERE clause:



**MIN(), MAX() and AVG()**

These functions return the minimum, maximum and average values:

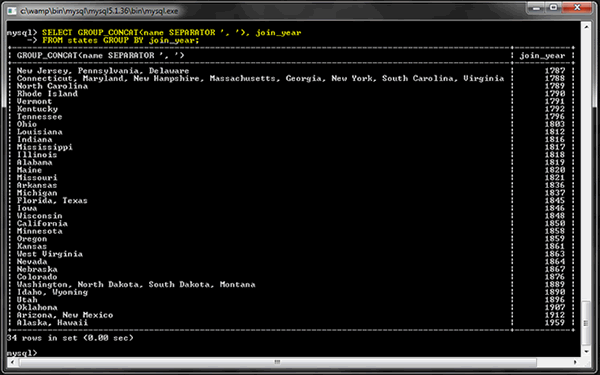


**Trebuie executate toate cazurile ca si pentru functia COUNT() cu GROUP BY, CU WHERE s.a.m.d.**

**GROUP\_CONCAT()**

This function concatenates all values inside the group into a single string, with a given separator.

In the first GROUP BY query example, we could only see one state name per year. You can use this function to see all names in each group:

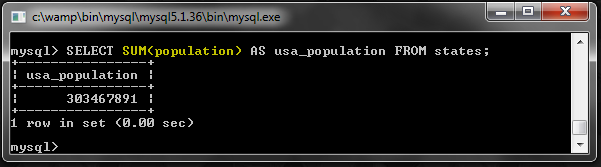


If the resized image is hard to read, this is the query:

|  |  |
| --- | --- |
| 1  2 | SELECT GROUP\_CONCAT(name SEPARATOR ', '), join\_year  FROM states GROUP BY join\_year; |

**SUM()**

You can use this to add up the numerical values.

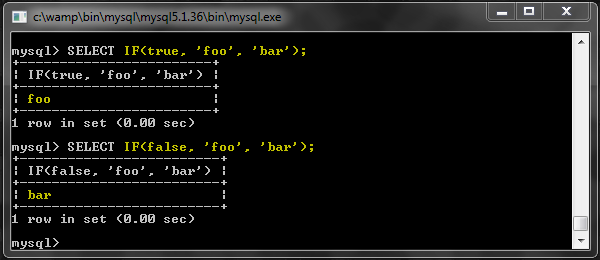


**IF() & CASE: Control Flow**

Similar to other programming languages, SQL has some support for control flow.

**IF()**

This is a function that takes three arguments. **First argument** is the condition, **second argument** is used if the condition ***is true*** and the **third argument** is used if the condition ***is false***.

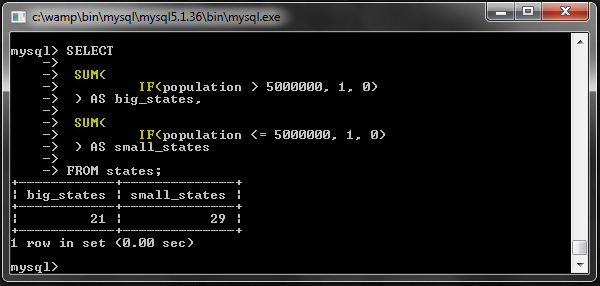


**Here is a more practical example where we use it with the SUM() function:**

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11 | SELECT        SUM(          IF(population > 5000000, 1, 0)      ) AS big\_states,        SUM(          IF(population <= 5000000, 1, 0)      ) AS small\_states    FROM states; |

The first SUM() call counts the number of big states (population over 5 million) and the second one counts the number of small states. The IF() call inside these SUM() calls return either 1 or 0 based on the condition.

Here is the result:



**CASE**

This works similar to the switch-case statements you might be familiar with from programming.

Let's say we want to categorize each state into one of three possible categories.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | SELECT  COUNT(\*),  CASE      WHEN population > 5000000 THEN 'big'      WHEN population > 1000000 THEN 'medium'      ELSE 'small' END      AS state\_size  FROM states GROUP BY state\_size; |

As you can see, we can actually **GROUP BY** the value returned from the CASE statement. Here is what happens:



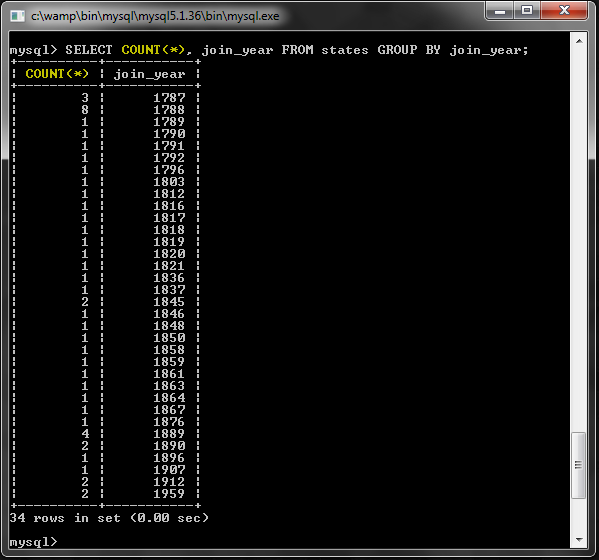
**HAVING: Conditions on Hidden Fields**

***HAVING clause allows us to apply conditions to 'hidden' fields, such as the returned results of aggregate functions. So it is usually used along with GROUP BY.,***

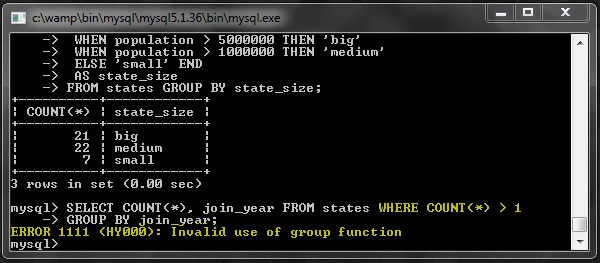
For example, let's look at the query we used for counting number of states by join year:

|  |  |
| --- | --- |
| 1 | SELECT COUNT(\*), join\_year FROM states GROUP BY join\_year; |

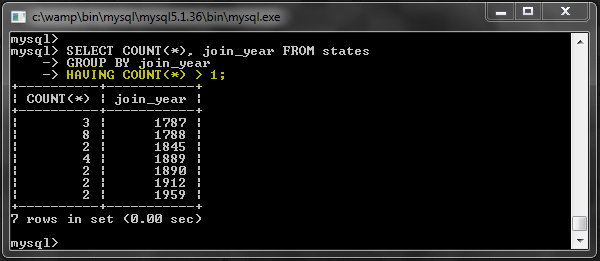
The result was 34 rows.



However, let's say we are only interested in rows that have a count higher than 1. We can not use the WHERE clause for this:



This is where HAVING becomes useful:



Keep in mind that this feature may not be available in all database systems.

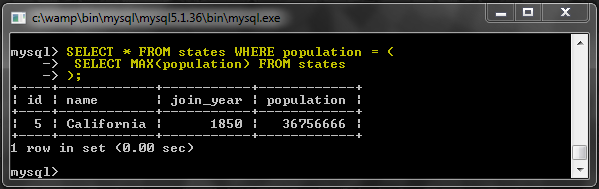
**Subqueries**

It is possible get the results of one query and use it for another query.

**In this example, we will get the state with the highest population:**

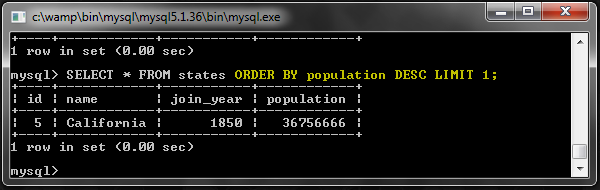
|  |  |
| --- | --- |
| 1  2  3 | SELECT \* FROM states WHERE population = (      SELECT MAX(population) FROM states  ); |

The **inner query** will return the highest population of all states. And the **outer query** will search the table again using that value.



You might be thinking this was a bad example, and I somewhat agree. The same query could be more efficiently written as this:

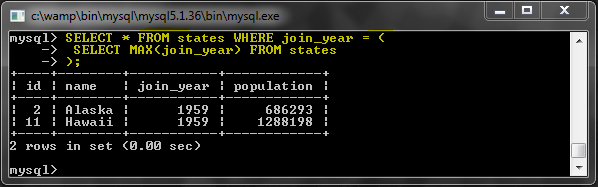
|  |  |
| --- | --- |
| 1 | SELECT \* FROM states ORDER BY population DESC LIMIT 1; |



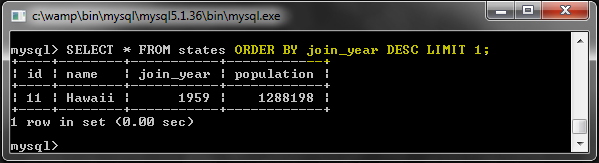
The results in this case are the same, however there is an important difference between these two kinds of queries. Maybe another example will demonstrate that better.

In this example, we will get the last states that joined the Union:

|  |  |
| --- | --- |
| 1  2  3 | SELECT \* FROM states WHERE join\_year = (      SELECT MAX(join\_year) FROM states  ); |



There are two rows in the results this time. If we had used the ORDER BY ... LIMIT 1 type of query here, we would not have received the same result.

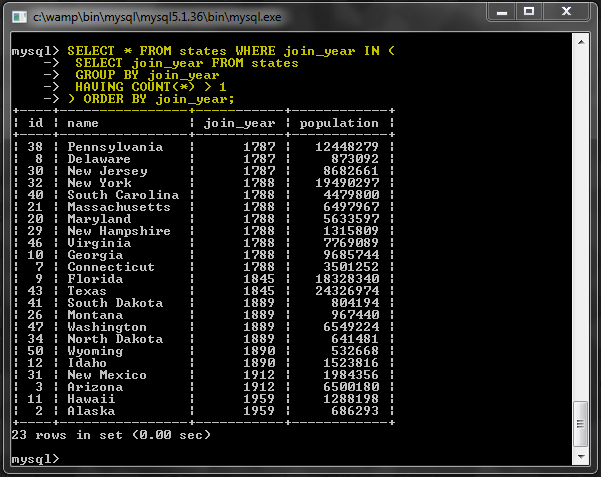


**IN()**

Sometimes you may want to use multiple results **returned by the inner query**.

Following query finds the years, when multiple states joined the Union, and returns the list of those states:

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT \* FROM states WHERE join\_year IN (      SELECT join\_year FROM states      GROUP BY join\_year      HAVING COUNT(\*) > 1  ) ORDER BY join\_year; |



**More on Subqueries**

Subqueries can become quite complex, therefore I will not get much further into them in this article. If you would like to read more about them, check out the [MySQL manual](http://dev.mysql.com/doc/refman/5.1/en/subqueries.html).

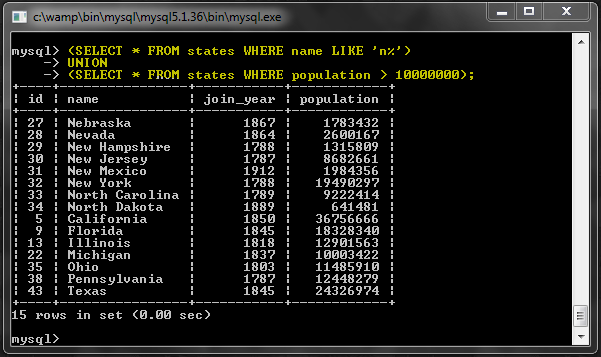
Also it is worth noting that subqueries can sometimes have bad performance, so they should be used with caution.

**UNION: Combining Data**

With a UNION query, we can combine the results of multiple SELECT queries.

This example combines states that start with the letter 'N' and states with large populations:

|  |  |
| --- | --- |
| 1  2  3 | (SELECT \* FROM states WHERE name LIKE 'n%')  UNION  (SELECT \* FROM states WHERE population > 10000000); |



Note that New York is both large and its name starts with the letter 'N'. But it shows up only once because duplicate rows are removed from the results automatically.

Another nice thing about UNION is that you can combine queries on different tables.

**Let's assume** we have tables for **employees, managers and customers**. And each table has an e-mail field. If we want to fetch all e-mails with a single query, we can run this:

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT email FROM employees)  UNION  SELECT email FROM managers)  UNION  SELECT email FROM customers WHERE subscribed = 1); |

It would fetch all emails of all employees and managers, but only the emails of customers that have subscribed to receive emails.

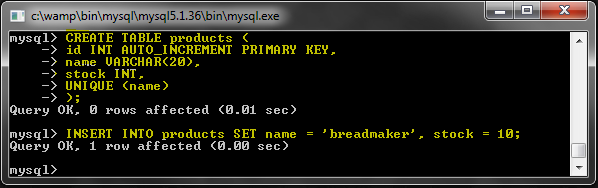
**INSERT Continued**

We have already talked about the INSERT query in the last article. Now that we explored database indexes today, we can talk about more advanced features of the INSERT query.

**INSERT ... ON DUPLICATE KEY UPDATE**

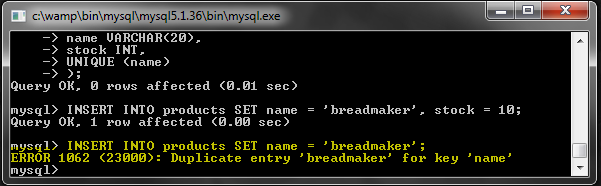
This is almost like a conditional statement. The query first tries to perform a given INSERT, and if it fails due to a duplicate value for a PRIMARY KEY or UNIQUE KEY, it performs an UPDATE instead.

Let's create a test table first.



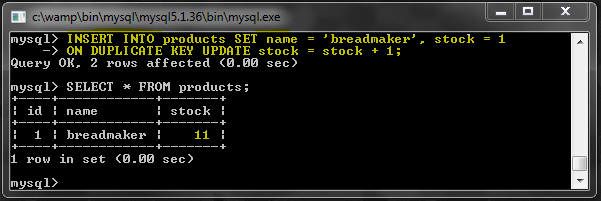
It's a table to hold products. The 'stock' column is the number of products we have in stock.

Now attempt to insert a duplicate value and see what happens.



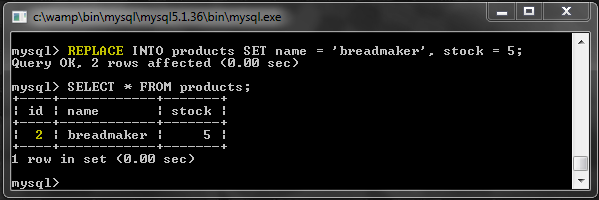
We got an error as expected.

Let's say we received a new breadmaker and want to update the database, and we do not know if there is already a record for it. We could check for existing records and then do another query based on that. Or we could just do it all in one simple query:



**REPLACE INTO**

**This works exactly like INSERT with one important exception.** If a duplicate row is found, it deletes it first and then performs the INSERT, so we get no error messages.



Note that since this is actually an entirely new row, the id was incremented.

**INSERT IGNORE**

This is a way to suppress the duplicate errors, usually to prevent the application from breaking. Sometimes you may want to attempt to insert a new row and just let it fail without any complaints in case there is a duplicate found.



No errors returned and no rows were updated.

**Data Types**

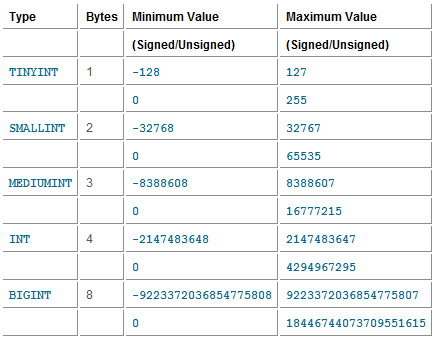
Each table column needs to have a data type. So far we have used INT, VARCHAR and DATE types but we did not talk about them in detail. Also there are several other data types that we should explore.

***First, let's start with the numeric data types. I like to put them into two separate groups: Integers vs. Non-Integers.***

**Integer Data Types**

An integer column can hold only natural numbers (no decimals). By default they can be negative or positive numbers. But if the UNSIGNED option is set, it can only hold positive numbers.

***MySQL supports 5 types of integers, with various sizes and ranges:***



***Non-Integer Numeric Data Types***

These data types can hold decimal numbers: FLOAT, DOUBLE and DECIMAL.

[FLOAT](http://dev.mysql.com/doc/refman/5.0/en/numeric-types.html) is 4 bytes, [DOUBLE](http://dev.mysql.com/doc/refman/5.0/en/numeric-types.html) is 8 bytes and they work similarly. However DOUBLE has better precision.

[DECIMAL(M,N)](http://dev.mysql.com/doc/refman/5.1/en/precision-math-decimal-changes.html) has a varying size based on the precision level, which can be customized. M is the maximum number of digits, and N is the number of digits to the right of the decimal point.

For example, DECIMAL(13,4) has a maximum of 9 integer digits and 4 fractional digits.

***String Data Types***

As the name suggests, we can store strings in these data type columns.

[CHAR(N)](http://dev.mysql.com/doc/refman/5.0/en/char.html) can hold up to N characters, and has a fixed size. For example CHAR(50) will always take 50 characters of space, per row, regardless of the size of the string in it. ***The absolute maximum is 255 characters***

[VARCHAR(N)](http://dev.mysql.com/doc/refman/5.0/en/char.html) works the same, but the storage size is not fixed. N is only used for the maximum size. If a string shorter than N characters is stored, it will take that much less space on the hard drive. The absolute maximum size is **65535 characters.**

Variations of the [TEXT](http://dev.mysql.com/doc/refman/5.1/en/blob.html) data type is more suitable for long strings. TEXT has a limit of 65535 characters, MEDIUMTEXT 16.7 million characters and LONGTEXT 4.3 billion characters. MySQL usually stores them on separate locations on the server so that the main storage for the table remains relatively small and quick.

***Date Types***

[DATE](http://dev.mysql.com/doc/refman/5.1/en/datetime.html) stores dates and displays them in this format 'YYYY-MM-DD' but does not contain the time info. It has a range of 1001-01-01 to 9999-12-31.

[DATETIME](http://dev.mysql.com/doc/refman/5.1/en/datetime.html) contains both the date and the time, and is displayed in this format 'YYYY-MM-DD HH:MM:SS'. It has a range of '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. It takes 8 bytes of space.

[TIMESTAMP](http://dev.mysql.com/doc/refman/5.1/en/datetime.html) works like DATETIME with a few exceptions. It takes only 4 bytes of space and the range is '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC. So, for example it may not be good for storing birth dates.

[TIME](http://dev.mysql.com/doc/refman/5.1/en/time.html) only stores the time, and [YEAR](http://dev.mysql.com/doc/refman/5.1/en/year.html) only stores the year.

**Other**

There are some other data types supported by MySQL. You can see a list of them [here](http://dev.mysql.com/doc/refman/5.1/en/data-types.html). You should also check out the storage sizes of each data type [here](http://dev.mysql.com/doc/refman/5.1/en/storage-requirements.html).

**Conclusion**

Thank you for reading the article. SQL is an important language and a tool in the web developers arsenal.