An Introduction to Queries in MySQL

Posted October 17, 2018 11.6k MYSQL DATABASES UBUNTU 18.04

By: Mark Drake

Introduction

Databases are a key component of many websites and applications, and are at the core of how data is stored and exchanged across the internet. One of the most important aspects of database management is the practice of retrieving data from a database, whether it's on an ad hoc basis or part of a process that's been coded into an application. There are several ways to retrieve information from a database, but one of the most commonly-used methods is performed through submitting *queries* through the command line.

In relational database management systems, a *query* is any command used to retrieve data from a table. In Structured Query Language (SQL), queries are almost always made using the `SELECT` statement.

In this guide, we will discuss the basic syntax of SQL queries as well as some of the more commonly-employed functions and operators. We will also practice making SQL queries using some sample data in a MySQL database.

MySQL is an open-source relational database management system. One of the most widely-deployed SQL-databases, MySQL prioritizes speed, reliability, and usability. It generally follows the ANSI SQL standard, although there are a few cases where MySQL performs operations differently than the recognized standard.

Prerequisites

In general, the commands and concepts presented in this guide can be used on any Linux-based operating system running any SQL database software. However, it was written specifically with an Ubuntu 18.04 server running MySQL in mind. To set this up, you will need the following:

- An Ubuntu 18.04 machine with a non-root user with sudo privileges. This can be set up using our Initial Server Setup guide for Ubuntu 18.04.
- MySQL installed on the machine. Our guide on How to Install MySQL on Ubuntu 18.04 can help you set this up.

With this setup in place, we can begin the tutorial.

Creating a Sample Database

Before we can begin making queries in SQL, we will first create a database and a couple tables, then populate these tables with some sample data. This will allow you to gain some hands-on experience when you begin making queries later on.

For the sample database we'll use throughout this guide, imagine the following scenario:

You and several of your friends all celebrate your birthdays with one another. On each occasion, the members of the group head to the local bowling alley, participate in a friendly tournament, and then everyone heads to your place where you prepare the birthday-person's favorite meal.

Now that this tradition has been going on for a while, you've decided to begin tracking the records from these tournaments. Also, to make planning dinners easier, you decide to create a record of your friends' birthdays and their favorite entrees, sides, and desserts. Rather than keep this information in a physical ledger, you decide to exercise your database skills by recording it in a MySQL database.

To begin, open up a MySQL prompt as your **root** MySQL user:

```
$ sudo mysql
```

Note: If you followed the prerequisite the tutorial on Installing MySQL on Ubuntu 18.04, you may have configured your **root** user to authenticate using a password. In this case, you will connect to the MySQL prompt with the following command:

```
  $ mysql -u root -p
```

Next, create the database by running:

```
mysql> CREATE DATABASE `birthdays`;
```

Then select this database by typing:

```
mysql> USE birthdays;
```

Next, create two tables within this database. We'll use the first table to track your friends' records at the bowling alley. The following command will create a table called `tourneys` with columns for the `name` of each of your friends, the number of tournaments they've won (`wins`), their all-time `best` score, and what size bowling shoe they wear (`size`):

```
mysql> CREATE TABLE tourneys (
mysql> name varchar(30),
mysql> wins real,
mysql> best real,
mysql> size real
mysql> );
```

Once you run the `CREATE TABLE` command and populate it with column headings, you'll receive the following output:

```
Output
Query OK, 0 rows affected (0.00 sec)
```

Populate the `tourneys` table with some sample data:

```
mysql> INSERT INTO tourneys (name, wins, best, size)
mysql> VALUES ('Dolly', '7', '245', '8.5'),
mysql> ('Etta', '4', '283', '9'),
mysql> ('Irma', '9', '266', '7'),
mysql> ('Barbara', '2', '197', '7.5'),
mysql> ('Gladys', '13', '273', '8');
```

You'll receive an output like this:

```
Output
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

Following this, create another table within the same database which we'll use to store information about your friends' favorite birthday meals. The following command creates a table named `dinners` with columns for the `name` of each of your friends, their `birthdate`, their favorite `entree`, their preferred `side` dish, and their favorite `dessert`:

```
mysql> CREATE TABLE dinners (
mysql> name varchar(30),
mysql> birthdate date,
mysql> entree varchar(30),
mysql> side varchar(30),
mysql> dessert varchar(30)
mysql> );
```

Similarly for this table, you'll receive feedback confirming that the command ran successfully:

```
Output
Query OK, 0 rows affected (0.01 sec)
```

Populate this table with some sample data as well:

```
mysql> INSERT INTO dinners (name, birthdate, entree, side, dessert)
mysql> VALUES ('Dolly', '1946-01-19', 'steak', 'salad', 'cake'),
mysql> ('Etta', '1938-01-25', 'chicken', 'fries', 'ice cream'),
mysql> ('Irma', '1941-02-18', 'tofu', 'fries', 'cake'),
mysql> ('Barbara', '1948-12-25', 'tofu', 'salad', 'ice cream'),
mysql> ('Gladys', '1944-05-28', 'steak', 'fries', 'ice cream');
```

```
Output
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

Once that command completes successfully, you're done setting up your database. Next, we'll go over the basic command structure of `SELECT` queries.

Understanding SELECT Statements

As mentioned in the introduction, SQL queries almost always begin with the `SELECT` statement. `SELECT` is used in queries to specify which columns from a table should be returned in the result-set. Queries also almost always include `FROM`, which is used to specify which table the statement will query.

Generally, SQL queries follow this syntax:

```
mysql> SELECT column_to_select FROM table_to_select WHERE certain_conditions_apply;
```

By way of example, the following statement will return the entire `name` column from the `dinners` table:

```
mysql> SELECT name FROM dinners;
```

Output

```
+---------+
| name    |
+---------+
| Dolly   |
| Etta    |
| Irma    |
| Barbara |
| Gladys  |
+---------+
5 rows in set (0.00 sec)
```

You can select multiple columns from the same table by separating their names with a comma, like this:

```
mysql> SELECT name, birthdate FROM dinners;
```

Output

```
+---------+------------+
| name    | birthdate  |
+---------+------------+
| Dolly   | 1946-01-19 |
| Etta    | 1938-01-25 |
| Irma    | 1941-02-18 |
| Barbara | 1948-12-25 |
| Gladys  | 1944-05-28 |
+---------+------------+
5 rows in set (0.00 sec)
```

Instead of naming a specific column or set of columns, you can follow the `SELECT` operator with an asterisk (`*`) which serves as a placeholder representing all the columns in a table. The following command returns every column from the `tourneys` table:

```
mysql> SELECT * FROM tourneys;
```

Output

```
+---------+------+------+------+
| name    | wins | best | size |
+---------+------+------+------+
| Dolly   |    7 |  245 |  8.5 |
```

```
| Etta    |    4 |  283 |    9 |
| Irma    |    9 |  266 |    7 |
| Barbara |    2 |  197 |  7.5 |
| Gladys  |   13 |  273 |    8 |
+---------+------+------+------+
5 rows in set (0.00 sec)
```

`WHERE` is used in queries to filter records that meet a specified condition, and any rows that do not meet that condition are eliminated from the result. A `WHERE` clause typically follows this syntax:

```
mysql> . . . WHERE column_name comparison_operator value
```

The comparison operator in a `WHERE` clause defines how the specified column should be compared against the value. Here are some common SQL comparison operators:

Operator	What it does
`=`	tests for equality
`!=`	tests for inequality
`<`	tests for less-than
`>`	tests for greater-than
`<=`	tests for less-than or equal-to
`>=`	tests for greater-than or equal-to
`BETWEEN`	tests whether a value lies within a given range
`IN`	tests whether a row's value is contained in a set of specified values
`EXISTS`	tests whether rows exist, given the specified conditions
`LIKE`	tests whether a value matches a specified string
`IS NULL`	tests for `NULL` values
`IS NOT NULL`	tests for all values other than `NULL`

For example, if you wanted to find Irma's shoe size, you could use the following query:

```
mysql> SELECT size FROM tourneys WHERE name = 'Irma';
```

Output

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

```
+------+
|    7 |
+------+
1 row in set (0.00 sec)
```

SQL allows the use of wildcard characters, and these are especially handy when used in `WHERE` clauses. Percentage signs (`%`) represent zero or more unknown characters, and underscores (`_`) represent a single unknown character. These are useful if you're trying to find a specific entry in a table, but aren't sure of what that entry is exactly. To illustrate, let's say that you've forgotten the favorite entree of a few of your friends, but you're certain this particular entree starts with a "t." You could find its name by running the following query:

```
mysql> SELECT entree FROM dinners WHERE entree LIKE 't%';
```

Output

```
+--------+
| entree |
+--------+
| tofu   |
| tofu   |
+--------+
2 rows in set (0.00 sec)
```

Based on the output above, we see that the entree we have forgotten is `tofu`.

There may be times when you're working with databases that have columns or tables with relatively long or difficult-to-read names. In these cases, you can make these names more readable by creating an alias with the `AS` keyword. Aliases created with `AS` are temporary, and only exist for the duration of the query for which they're created:

```
mysql> SELECT name AS n, birthdate AS b, dessert AS d FROM dinners;
```

Output

```
+---------+------------+-----------+
| n       | b          | d         |
+---------+------------+-----------+
| Dolly   | 1946-01-19 | cake      |
| Etta    | 1938-01-25 | ice cream |
| Irma    | 1941-02-18 | cake      |
| Barbara | 1948-12-25 | ice cream |
| Gladys  | 1944-05-28 | ice cream |
+---------+------------+-----------+
5 rows in set (0.00 sec)
```

Here, we have told SQL to display the `name` column as `n`, the `birthdate` column as `b`, and the `dessert` column as `d`.

The examples we've gone through up to this point include some of the more frequently-used keywords and clauses in SQL queries. These are useful for basic queries, but they aren't helpful if you're trying to perform a calculation or derive a *scalar value* (a single value, as opposed to a set of multiple different values) based on your data. This is where aggregate functions come into play.

Aggregate Functions

Oftentimes, when working with data, you don't necessarily want to see the data itself. Rather, you want information *about* the data. The SQL syntax includes a number of functions that allow you to interpret or run calculations on your data just by issuing a `SELECT` query. These are known as *aggregate functions*.

The `COUNT` function counts and returns the number of rows that match a certain criteria. For example, if you'd like to know how many of your friends prefer tofu for their birthday entree, you could issue this query:

```
mysql> SELECT COUNT(entree) FROM dinners WHERE entree = 'tofu';
```

```
Output
+---------------+
| COUNT(entree) |
+---------------+
|             2 |
+---------------+
1 row in set (0.00 sec)
```

The `AVG` function returns the average (mean) value of a column. Using our example table, you could find the average best score amongst your friends with this query:

```
mysql> SELECT AVG(best) FROM tourneys;
```

```
Output
+-----------+
| AVG(best) |
+-----------+
|     252.8 |
+-----------+
1 row in set (0.00 sec)
```

`SUM` is used to find the total sum of a given column. For instance, if you'd like to see how many games you and your friends have bowled over the years, you could run this query:

```
mysql> SELECT SUM(wins) FROM tourneys;
```

Output

```
+-----------+
| SUM(wins) |
+-----------+
|        35 |
+-----------+
1 row in set (0.00 sec)
```

Note that the `AVG` and `SUM` functions will only work correctly when used with numeric data. If you try to use them on non-numerical data, it will result in either an error or just `0`, depending on which RDBMS you're using:

```
mysql> SELECT SUM(entree) FROM dinners;
```

Output

```
+-------------+
| SUM(entree) |
+-------------+
|           0 |
+-------------+
1 row in set, 5 warnings (0.00 sec)
```

`MIN` is used to find the smallest value within a specified column. You could use this query to see what the worst overall bowling record is so far (in terms of number of wins):

```
mysql> SELECT MIN(wins) FROM tourneys;
```

Output

```
+-----------+
| MIN(wins) |
+-----------+
|         2 |
+-----------+
1 row in set (0.00 sec)
```

Similarly, `MAX` is used to find the largest numeric value in a given column. The following query will show the best overall bowling record:

```
mysql> SELECT MAX(wins) FROM tourneys;
```

Output

```
+-----------+
| MAX(wins) |
+-----------+
|        13 |
+-----------+
1 row in set (0.00 sec)
```

Unlike `SUM` and `AVG`, the `MIN` and `MAX` functions can be used for both numeric and alphabetic data types. When run on a column containing string values, the `MIN` function will show the first value alphabetically:

```
mysql> SELECT MIN(name) FROM dinners;
```

Output

```
+-----------+
| MIN(name) |
+-----------+
| Barbara   |
+-----------+
1 row in set (0.00 sec)
```

Likewise, when run on a column containing string values, the `MAX` function will show the last value alphabetically:

```
mysql> SELECT MAX(name) FROM dinners;
```

Output

```
+-----------+
| MAX(name) |
+-----------+
| Irma      |
+-----------+
1 row in set (0.00 sec)
```

Aggregate functions have many uses beyond what was described in this section. They're particularly useful when used with the `GROUP BY` clause, which is covered in the next section along with several other query clauses that affect how result-sets are sorted.

Manipulating Query Outputs

In addition to the `FROM` and `WHERE` clauses, there are several other clauses which are used to manipulate the results of a `SELECT` query. In this section, we will explain and provide examples for some of the more commonly-used query clauses.

One of the most frequently-used query clauses, aside from `FROM` and `WHERE`, is the `GROUP BY` clause. It's typically used when you're performing an aggregate function on one column, but in relation to matching values in another.

For example, let's say you wanted to know how many of your friends prefer each of the three entrees you make. You could find this info with the following query:

```
mysql> SELECT COUNT(name), entree FROM dinners GROUP BY entree;
```

Output

```
+-------------+---------+
| COUNT(name) | entree  |
+-------------+---------+
|           1 | chicken |
|           2 | steak   |
|           2 | tofu    |
+-------------+---------+
3 rows in set (0.00 sec)
```

The `ORDER BY` clause is used to sort query results. By default, numeric values are sorted in ascending order, and text values are sorted in alphabetical order. To illustrate, the following query lists the `name` and `birthdate` columns, but sorts the results by birthdate:

```
mysql> SELECT name, birthdate FROM dinners ORDER BY birthdate;
```

Output

```
+---------+------------+
| name    | birthdate  |
+---------+------------+
| Etta    | 1938-01-25 |
| Irma    | 1941-02-18 |
| Gladys  | 1944-05-28 |
| Dolly   | 1946-01-19 |
| Barbara | 1948-12-25 |
+---------+------------+
5 rows in set (0.00 sec)
```

Notice that the default behavior of `ORDER BY` is to sort the result-set in ascending order. To reverse this and have the result-set sorted in descending order, close the query with `DESC`:

```
mysql> SELECT name, birthdate FROM dinners ORDER BY birthdate DESC;
```

Output

```
+---------+------------+
| name    | birthdate  |
+---------+------------+
| Barbara | 1948-12-25 |
| Dolly   | 1946-01-19 |
| Gladys  | 1944-05-28 |
| Irma    | 1941-02-18 |
| Etta    | 1938-01-25 |
+---------+------------+
5 rows in set (0.00 sec)
```

As mentioned previously, the `WHERE` clause is used to filter results based on specific conditions. However, if you use the `WHERE` clause with an aggregate function, it will return an error, as is the case with the following attempt to find which sides are the favorite of at least three of your friends:

```
mysql> SELECT COUNT(name), side FROM dinners WHERE COUNT(name) >= 3;
```

Output

```
ERROR 1111 (HY000): Invalid use of group function
```

The `HAVING` clause was added to SQL to provide functionality similar to that of the `WHERE` clause while also being compatible with aggregate functions. It's helpful to think of the difference between these two clauses as being that `WHERE` applies to individual records, while `HAVING` applies to group records. To this end, any time you issue a `HAVING` clause, the `GROUP BY` clause must also be present.

The following example is another attempt to find which side dishes are the favorite of at least three of your friends, although this one will return a result without error:

```
mysql> SELECT COUNT(name), side FROM dinners GROUP BY side HAVING COUNT(name) >= 3;
```

Output

```
+-------------+-------+
| COUNT(name) | side  |
+-------------+-------+
|           3 | fries |
+-------------+-------+
1 row in set (0.00 sec)
```

Aggregate functions are useful for summarizing the results of a particular column in a given table. However, there are many cases where it's necessary to query the contents of more than one table. We'll go over a few ways you can do this in the next section.

Querying Multiple Tables

More often than not, a database contains multiple tables, each holding different sets of data. SQL provides a few different ways to run a single query on multiple tables.

The `JOIN` clause can be used to combine rows from two or more tables in a query result. It does this by finding a related column between the tables and sorts the results appropriately in the output.

`SELECT` statements that include a `JOIN` clause generally follow this syntax:

```
mysql> SELECT table1.column1, table2.column2
mysql> FROM table1
mysql> JOIN table2 ON table1.related_column=table2.related_column;
```

Note that because `JOIN` clauses compare the contents of more than one table, the previous example specifies which table to select each column from by preceding the name of the column with the name of the table and a period. You can specify which table a column should be selected from like this for any query, although it's not necessary when selecting from a single table, as we've done in the previous sections. Let's walk through an example using our sample data.

Imagine that you wanted to buy each of your friends a pair of bowling shoes as a birthday gift. Because the information about your friends' birthdates and shoe sizes are held in separate tables, you could query both tables separately then compare the results from each. With a `JOIN` clause, though, you can find all the information you want with a single query:

```
mysql> SELECT tourneys.name, tourneys.size, dinners.birthdate
mysql> FROM tourneys
mysql> JOIN dinners ON tourneys.name=dinners.name;
```

Output

```
+---------+------+------------+
| name    | size | birthdate  |
+---------+------+------------+
| Dolly   |  8.5 | 1946-01-19 |
| Etta    |    9 | 1938-01-25 |
| Irma    |    7 | 1941-02-18 |
| Barbara |  7.5 | 1948-12-25 |
| Gladys  |    8 | 1944-05-28 |
+---------+------+------------+
5 rows in set (0.00 sec)
```

The `JOIN` clause used in this example, without any other arguments, is an *inner* `JOIN` clause. This means that it selects all the records that have matching values in both tables and prints them to the results set, while any records that aren't matched are excluded. To illustrate this idea, let's add a new row to each table that doesn't have a corresponding entry in the other:

```
mysql> INSERT INTO tourneys (name, wins, best, size)
```

```
mysql> VALUES ('Bettye', '0', '193', '9');


mysql> INSERT INTO dinners (name, birthdate, entree, side, dessert)
mysql> VALUES ('Lesley', '1946-05-02', 'steak', 'salad', 'ice cream');
```

Then, re-run the previous `SELECT` statement with the `JOIN` clause:

```
mysql> SELECT tourneys.name, tourneys.size, dinners.birthdate
mysql> FROM tourneys
mysql> JOIN dinners ON tourneys.name=dinners.name;
```

Output

```
+---------+------+------------+
| name    | size | birthdate  |
+---------+------+------------+
| Dolly   |  8.5 | 1946-01-19 |
| Etta    |    9 | 1938-01-25 |
| Irma    |    7 | 1941-02-18 |
| Barbara |  7.5 | 1948-12-25 |
| Gladys  |    8 | 1944-05-28 |
+---------+------+------------+
5 rows in set (0.00 sec)
```

Notice that, because the `tourneys` table has no entry for Lesley and the `dinners` table has no entry for Bettye, those records are absent from this output.

It is possible, though, to return all the records from one of the tables using an *outer* `JOIN` clause. In MySQL, `JOIN` clauses are written as either `LEFT JOIN` or `RIGHT JOIN`.

A `LEFT JOIN` clause returns all the records from the “left” table and only the matching records from the right table. In the context of outer joins, the left table is the one referenced by the `FROM` clause, and the right table is any other table referenced after the `JOIN` statement.

Run the previous query again, but this time use a `LEFT JOIN` clause:

```
mysql> SELECT tourneys.name, tourneys.size, dinners.birthdate
mysql> FROM tourneys
mysql> LEFT JOIN dinners ON tourneys.name=dinners.name;
```

This command will return every record from the left table (in this case, `tourneys`) even if it doesn't have a corresponding record in the right table. Any time there isn't a matching record from the right table, it's returned as `NULL` or just a blank value, depending on your RDBMS:

Output

```
+---------+------+------------+
| name    | size | birthdate  |
+---------+------+------------+
| Dolly   |  8.5 | 1946-01-19 |
| Etta    |    9 | 1938-01-25 |
| Irma    |    7 | 1941-02-18 |
| Barbara |  7.5 | 1948-12-25 |
| Gladys  |    8 | 1944-05-28 |
| Bettye  |    9 | NULL       |
+---------+------+------------+
6 rows in set (0.00 sec)
```

Now run the query again, this time with a `RIGHT JOIN` clause:

```
mysql> SELECT tourneys.name, tourneys.size, dinners.birthdate
mysql> FROM tourneys
mysql> RIGHT JOIN dinners ON tourneys.name=dinners.name;
```

This will return all the records from the right table (`dinners`). Because Lesley's birthdate is recorded in the right table, but there is no corresponding row for her in the left table, the `name` and `size` columns will return as `NULL` values in that row:

Output

```
+---------+------+------------+
| name    | size | birthdate  |
+---------+------+------------+
| Dolly   |  8.5 | 1946-01-19 |
| Etta    |    9 | 1938-01-25 |
| Irma    |    7 | 1941-02-18 |
| Barbara |  7.5 | 1948-12-25 |
| Gladys  |    8 | 1944-05-28 |
| NULL    | NULL | 1946-05-02 |
+---------+------+------------+
6 rows in set (0.00 sec)
```

Note that left and right joins can be written as `LEFT OUTER JOIN` or `RIGHT OUTER JOIN`, although the `OUTER` part of the clause is implied. Likewise, specifying `INNER JOIN` will produce the same result as just writing `JOIN`.

As an alternative to using `JOIN` to query records from multiple tables, you can use the `UNION` clause.

The `UNION` operator works slightly differently than a `JOIN` clause: instead of printing results from multiple tables as unique columns using a single `SELECT` statement, `UNION` combines the results of two `SELECT` statements into a single column.

To illustrate, run the following query:

```
mysql> SELECT name FROM tourneys UNION SELECT name FROM dinners;
```

This query will remove any duplicate entries, which is the default behavior of the `UNION` operator:

```
Output
+---------+
| name    |
+---------+
| Dolly   |
| Etta    |
| Irma    |
| Barbara |
| Gladys  |
| Bettye  |
| Lesley  |
+---------+
7 rows in set (0.00 sec)
```

To return all entries (including duplicates) use the `UNION ALL` operator:

```
mysql> SELECT name FROM tourneys UNION ALL SELECT name FROM dinners;
```

```
Output
+---------+
| name    |
+---------+
| Dolly   |
| Etta    |
| Irma    |
| Barbara |
| Gladys  |
| Bettye  |
| Dolly   |
| Etta    |
| Irma    |
| Barbara |
| Gladys  |
| Lesley  |
+---------+
12 rows in set (0.00 sec)
```

The names and number of the columns in the results table reflect the name and number of columns queried by the first `SELECT` statement. Note that when using `UNION` to query multiple columns from more than one table, each `SELECT` statement must query the same number of columns, the respective columns must have similar data types, and the columns in each `SELECT` statement must be in the same order. The following

example shows what might result if you use a `UNION` clause on two `SELECT` statements that query a different number of columns:

```
mysql> SELECT name FROM dinners UNION SELECT name, wins FROM tourneys;
```

```
Output
ERROR 1222 (21000): The used SELECT statements have a different number of columns
```

Another way to query multiple tables is through the use of *subqueries*. Subqueries (also known as *inner* or *nested queries*) are queries enclosed within another query. These are useful in cases where you're trying to filter the results of a query against the result of a separate aggregate function.

To illustrate this idea, say you want to know which of your friends have won more matches than Barbara. Rather than querying how many matches Barbara has won then running another query to see who has won more games than that, you can calculate both with a single query:

```
mysql> SELECT name, wins FROM tourneys
mysql> WHERE wins > (
mysql> SELECT wins FROM tourneys WHERE name = 'Barbara'
mysql> );
```

```
Output
+--------+------+
| name   | wins |
+--------+------+
| Dolly  |    7 |
| Etta   |    4 |
| Irma   |    9 |
| Gladys |   13 |
+--------+------+
4 rows in set (0.00 sec)
```

The subquery in this statement was run only once; it only needed to find the value from the `wins` column in the same row as `Barbara` in the `name` column, and the data returned by the subquery and outer query are independent of one another. There are cases, though, where the outer query must first read every row in a table and compare those values against the data returned by the subquery in order to return the desired data. In this case, the subquery is referred to as a *correlated subquery*.

The following statement is an example of a correlated subquery. This query seeks to find which of your friends have won more games than is the average for those with the same shoe size:

```
mysql> SELECT name, size FROM tourneys AS t
mysql> WHERE wins > (
mysql> SELECT AVG(wins) FROM tourneys WHERE size = t.size
```

```
mysql> );
```

In order for the query to complete, it must first collect the `name` and `size` columns from the outer query. Then, it compares each row from that result-set against the results of the inner query, which determines the average number of wins for individuals with identical shoe sizes. Because you only have two friends that have the same shoe size, there can only be one row in the result-set:

```
Output
+------+------+
| name | size |
+------+------+
| Etta |    9 |
+------+------+
1 row in set (0.00 sec)
```

As mentioned earlier, subqueries can be used to query results from multiple tables. To illustrate this with one final example, say you wanted to throw a surprise dinner for the group's all-time best bowler. You could find which of your friends has the best bowling record and return their favorite meal with the following query:

```
mysql> SELECT name, entree, side, dessert
mysql> FROM dinners
mysql> WHERE name = (SELECT name FROM tourneys
mysql> WHERE wins = (SELECT MAX(wins) FROM tourneys));
```

```
Output
+--------+--------+-------+-----------+
| name   | entree | side  | dessert   |
+--------+--------+-------+-----------+
| Gladys | steak  | fries | ice cream |
+--------+--------+-------+-----------+
1 row in set (0.00 sec)
```

Notice that this statement not only includes a subquery, but also contains a subquery within that subquery.

Conclusion

Issuing queries is one of the most commonly-performed tasks within the realm of database management. There are a number of database administration tools, such as phpMyAdmin or pgAdmin, that allow you to perform queries and visualize the results, but issuing `SELECT` statements from the command line is still a widely-practiced workflow that can also provide you with greater control.

If you're new to working with SQL, we encourage you to use our SQL Cheat Sheet as a reference and to review the official MySQL documentation. Additionally, if you'd like to learn more about SQL and relational databases, the following tutorials may be of interest to you:

- Understanding SQL And NoSQL Databases And Different Database Models
- How To Create a Multi-Node MySQL Cluster on Ubuntu 18.04
- How To Reset Your MySQL or MariaDB Root Password on Ubuntu 18.04

By: Mark Drake

Related Tutorials

How To Ensure Code Quality with SonarQube on Ubuntu 18.04

How To Sync and Share Your Files with Seafile on Ubuntu 18.04

How To Troubleshoot Issues in MySQL

How To Set Up a Remote Database to Optimize Site Performance with MySQL on Ubuntu 18.04

How To Set Up Laravel, Nginx, and MySQL with Docker Compose

0 Comments

Log In to Comment