SELECT

As we briefly covered in the previous chapter, the `SELECT` statement allows us to retrieve data from single or multiple tables on the database. In this chapter, we will be performing the query on a single table.

It corresponds to the projection operation of Relational Algebra.

You can use `SELECT` to get all of your users or a list of users that match a certain criteria.

Before we dive into the `SELECT` statement let's quickly create a database:

```
CREATE DATABASE sql_demo;
```

Switch to that database:

```
USE sql_demo;
```

Create a new users table:

```
CREATE TABLE users
(
    id INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(255) NOT NULL,
    about TEXT,
    email VARCHAR(255),
    birthday DATE,
    active BOOL
);
```

Insert some data that we could work with:

```
INSERT INTO users
  ( username, email, active )
VALUES
  ('bobby', 'b@devdojo.com', true),
  ('devdojo', 'd@devdojo.com', false),
  ('tony', 't@devdojo.com', true);
```

Output:

```
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

We are going to learn more about the `INSERT` statement in the following chapters.

SELECT all columns

Now that we've got some data in the `users` table, let's go ahead and retrieve all of the entries from that table:

```
SELECT * FROM users;
```

Rundown of the statement:

- `SELECT`: First, we specify the action that we want to execute, in our case, we want to select or get some data from the database.
- `*`: The star here indicates that we want to get all of the columns associated with the table that we are selecting from.
- `FROM`: The from statement tells MySQL which table we want to select the data from. You need to keep in mind that you can select from multiple tables, but this is a bit more advanced, and we are going to cover this in the next few chapters.
- `users`: This is the table name that we want to select the data from.

This will return all of the entries in the `users` table along with all of the columns:

```
+----+----------+-------+----------+--------+---------------+
| id | username | about | birthday | active | email         |
+----+----------+-------+----------+--------+---------------+
|  1 | bobby    | NULL  | NULL     |      1 | b@devdojo.com |
|  2 | devdojo  | NULL  | NULL     |      0 | d@devdojo.com |
|  3 | tony     | NULL  | NULL     |      1 | t@devdojo.com |
+----+----------+-------+----------+--------+---------------+
3 rows in set (0.00 sec)
```

As you can see, we get a list of the 3 users that we've just created, including all of the columns in that table. In some cases, the table might

have a lot of columns, and you might not want to see all of them. For example, we have the `about` and `birthday` columns that are all `NULL` at the moment. So let's see how we could limit that and get only a list of specific columns.

Pattern matching

SQL pattern matching let's you to search for patterns if you don't know the exact word or phrase you are looking for. To do this, we use so-called wildcard characters to match a pattern together with LIKE and ILIKE operators.

Two of the most common wildcard characters are `_` and `%`.

`_` matches any single character and `%` matches an arbitrary number of characters.

Let's see an example how you would look for a `username` ending with `y`:

```
SELECT * FROM users WHERE username LIKE '%y';
```

Output:

```
+----+----------+-------+----------+--------+---------------+
| id | username | about | birthday | active | email         |
+----+----------+-------+----------+--------+---------------+
|  1 | bobby    | NULL  | NULL     |      1 | b@devdojo.com |
|  3 | tony     | NULL  | NULL     |      1 | t@devdojo.com |
+----+----------+-------+----------+--------+---------------+
```

As you can see above, we used `%` to match any number of characters preceding the character `y`.

If we know the exact number of characters we want to match, we can use `_`. Each `_` represents a single character.

So, if we want to look up an username that has `e` as its second character, we would do something like this:

```
SELECT * FROM users WHERE username LIKE '_e%';
```

Output:

```
+----+----------+-------+----------+--------+---------------+
| id | username | about | birthday | active | email         |
+----+----------+-------+----------+--------+---------------+
|  2 | devdojo  | NULL  | NULL     |      0 | d@devdojo.com |
+----+----------+-------+----------+--------+---------------+
```

Please, keep in mind that `LIKE` operator is case sensitive, meaning it won't mach capital letters with lowercase letters and vice versa. If you wish to ignore capitalization, use `ILIKE` operator instead.

Formatting

As we mentioned in the previous chapters, each SQL statement needs to end with a semi-colon: `;`. Alternatively, rather than using a semi-colon, you could use the `\G` characters which would format the output in a list rather than a table.

The syntax is absolutely the same but you just change the `;` with `\G`:

```
SELECT * FROM users \G
```

The output will be formatted like this:

```
*************************** 1. row ***************************
      id: 1
username: bobby
   about: NULL
birthday: NULL
  active: 1
   email: b@devdojo.com
*************************** 2. row ***************************
      id: 2
username: devdojo
   about: NULL
birthday: NULL
  active: 0
   email: d@devdojo.com
...
```

This is very handy whenever your table consists of a large number of columns and they can't fit on the screen, which makes it very hard to read the result set.

SELECT specific columns only

You could limit this to a specific set of columns. Let's say that you only needed the `username` and the `active` columns. In this case, you would change the `*` symbol with the columns that you want to select divided by a comma:

```
SELECT username,active FROM users;
```

Output:

```
+----------+--------+
| username | active |
+----------+--------+
| bobby    |      1 |
| devdojo  |      0 |
| tony     |      1 |
+----------+--------+
```

As you can see, we are getting back only the 2 columns that we've specified in the `SELECT` statement.

NOTE: *SQL names are case insensitive. For example, username ≡ USERNAME ≡ userName.*

SELECT with no FROM Clause

In a SQL statement, a column can be a literal with no `FROM` clause.

```
SELECT 'Sunil' as username;
```

Output:

```
+----------+
| username |
+----------+
| Sunil    |
+----------+
```

SELECT with Arithmetic Operations

The select clause can contain arithmetic expressions involving the operation +, –, *, and /.

```
SELECT username, active*5 as new_active FROM users;
```

Output:

```
+----------+------------+
| username | new_active |
+----------+------------+
| bobby    |          5 |
| devdojo  |          0 |
| tony     |          5 |
+----------+------------+
```

LIMIT

The `LIMIT` clause is very handy in case that you want to limit the number of results that you get back. For example, at the moment, we have 3 users in our database, but let's say that you only wanted to get 1 entry back when you run the `SELECT` statement.

This can be achieved by adding the `LIMIT` clause at the end of your statement, followed by the number of entries that you want to get. For example, let's say that we wanted to get only 1 entry back. We would run the following query:

```
SELECT * FROM users LIMIT 1;
```

Output:

```
+----+----------+-------+----------+--------+---------------+
| id | username | about | birthday | active | email         |
+----+----------+-------+----------+--------+---------------+
|  2 | bobby    | NULL  | NULL     |      1 | b@devdojo.com |
+----+----------+-------+----------+--------+---------------+
```

If you wanted to get 2 entries, you would change `LIMIT 2` and so on.

COUNT

In case that you wanted to get only the number of entries in a specific column, you could use the `COUNT` function. This is a function that I personally use very often.

The syntax is the following:

```
SELECT COUNT(*) FROM users;
```

Output:

```
+----------+
| COUNT(*) |
+----------+
|        3 |
+----------+
```

MIN, MAX, AVG, and SUM

Another useful set of functions similar to `COUNT` that would make your life easier are:

- `MIN`: This would give you the smallest value of a specific column. For example, if you had an online shop and you wanted to get the lowest price, you would use the `MIN` function. In our case, if we wanted to get the lowest user ID, we would run the following:

```
SELECT MIN(id) FROM users;
```

This would return `1` as the lowest user ID that we have is 1.

- `MAX`: Just like `MIN`, but it would return the highest value:

```
SELECT MAX(id) FROM users;
```

In our case, this would be `3` as we have only 3 users, and the highest value of the `id` column is 3.

- `AVG`: As the name suggests, it would sum up all of the values of a specific column and return the average value. As we have 3 users with ids 1, 2, and 3, the average would be 6 divided by 3 users which is 2.

```
SELECT AVG(id) FROM users;
```

- `SUM`: This function takes all of the values from the specified column and sums them up:

```
SELECT SUM(id) FROM users;
```

DISTINCT

In some cases, you might have duplicate entries in a table, and in order to get only the unique values, you could use `DISTINCT`.

To better demonstrate this, let's run the insert statement one more time so that we could duplicate the existing users and have 6 users in the users table:

```
INSERT INTO users
  ( username, email, active )
VALUES
  ('bobby', 'b@devdojo.com', true),
  ('devdojo', 'd@devdojo.com', false),
  ('tony', 't@devdojo.com', true);
```

Now, if you run `SELECT COUNT(*) FROM users;` you would get `6` back.

Let's also select all users and show only the `username` column:

```
SELECT username FROM users;
```

Output:

```
+----------+
| username |
+----------+
| bobby    |
| devdojo  |
| tony     |
| bobby    |
| devdojo  |
| tony     |
+----------+
```

As you can see, each name is present multiple times in the list. We have `bobby`, `devdjo` and `tony` showing up twice.

If we wanted to show only the unique `usernames`, we could add the `DISTINCT` keyword to our select statement:

```
SELECT DISTINCT username FROM users;
```

Output:

```
+----------+
| username |
+----------+
| bobby    |
| devdojo  |
| tony     |
+----------+
```

As you can see, the duplicate entries have been removed from the output.

Conclusion

The `SELECT` statement is essential whenever working with SQL. In the next chapter, we are going to learn how to use the `WHERE` clause and take the `SELECT` statements to the next level.