AN OPENSOURCE EBOOK

INTRODUCTION TO SQL

Manisha Reddy

Table of Contents

About the book

- **This version was published on May 18 2021**

This is an open-source introduction to SQL guide that will help you learn the basics of SQL and start using relational databases for your SysOps, DevOps, and Dev projects. No matter if you are a DevOps/SysOps engineer, developer, or just a Linux enthusiast, you will most likely have to use SQL at some point in your career.

The guide is suitable for anyone working as a developer, system administrator, or a DevOps engineer and wants to learn the basics of SQL.

Databases

Before we dive deep into SQL, let's quickly define what a database is.

The definition of databases from Wikipedia is:

A database is an organized collection of data, generally stored and accessed electronically from a computer system.

In other words, a database is a collection of data stored and structured in different database tables.

Tables and columns

You've mostlikely worked with spreadsheet systems like Excel or Google Sheets. At the very basic, database tables are quite similar to spreadsheets.

Each table has different **columns** which could contain different types of data.

For example, if you have a todo list app, you would have a database, and in your database, you would have different tables storing different information like:

- Users - In the users table, you would have some data for your users like: `username`, `name`, and `active`, for example.
- Tasks - The tasks table would store all of the tasks that you are planning to do. The columns of the tasks table would be for example, `task_name`, `status`, `due_date` and `priority`.

The Users table will look like this:

```
+----+----------+---------------+--------+
| id | username | name          | active |
+----+----------+---------------+--------+
| 1  |    bobby | Bobby Iliev   |   true |
| 2  |    grisi | Greisi I.     |   true |
| 3  |  devdojo | Dev Dojo      |  false |
+----+----------+---------------+--------+
```

Rundown of the table structure:

- We have 4 columns: `id`, `username`, `name` and `active`
- We also have 3 entries/users
- The `id` column is a unique identifier of each user and is auto-incremented.

In the next chapter, we will learn how to install MySQL and create our first database.

MySQL

Now that you know what a database, table, and column are, the next thing that you would need to do is install a database service where you would be running your SQL queries on.

We would be using MySQL as it is free, open-source, and very widely used.

Installing MySQL

As we are going to use **Ubuntu**, in order to install MySQL run the following commands:

- First update your `apt` repository:

```
sudo apt update -y
```

- Then install MySQL:

```
sudp apt install mysql-server mysql-client
```

We are installing 2 packages, one is the actual MySQL server, and the other is the MySQL client, which would allow us to connect to the MySQL server and run our queries.

In order to check if MySQL is running, run the following command:

```
sudo systemctl status mysql.service
```

In order to secure your MySQL server, you could run the following command:

```
sudo mysql_secure_installation
```

Then follow the prompt and finally choose a secure password and save it in a secure place like a password manager.

With that, you would have MySQL installed on your Ubuntu server. The above should also work just fine on Debina.

Install MySQL on Mac

I would recommend installing MySQL using Homebrew:

```
brew install mysql
```

After that start MySQL:

```
brew services start mysql
```

And finally, secure it:

```
mysql_secure_installation
```

In case that you ever need to stop the MySQL service, you could do so with the following command:

```
brew services stop mysql
```

Install MySQL on Windows

In order to install MySQL on Windows, I would recommend following the steps from the official documentation here:

https://dev.mysql.com/doc/refman/8.0/en/windows-installation.html

Accessing MySQL via CLI

To access MySQL run the `mysql` command followed by your user:

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

Creating a database

After that, switch to the `demo` database that we created in the previous chapter:

```
USE demo;
```

To exit the just type the following:

```
exit;
```

Configuring `.my.cnf`

By configuring the `~/.my.cnf` file in your user's home directory, MySQL would allow you to login without prompting you for a password.

In order to make that change, what you need to do is first create a `.my.cnf` file in your user's home directory:

```
touch ~/.my.cnf
```

After that, set secure permissions so that other regular users could not read the file:

```
chmod 600 ~/.my.cnf
```

Then using your favorite text editor, open the file:

```
nano ~/.my.cnf
```

And add the following configuration:

```
[client]
user=YOUR_MYSQL_USERNAME
password=YOUR_MYSQL_PASSWORD
```

Make sure to update your MySQL credentials accordingly, then save the file and exit.

After that, if you run just `mysql`, you will be authenticated directly with the credentials that you've specified in the `~/.my.cnf` file without being prompted for a password.

The mysqladmin command

As a quick test, you could check all of your open SQL connections by running the following command:

```
mysqladmin proc
```

The `mysqladmin` tool would also use the client details from the `~/.my.cnf` file, and it would list your current MySQL process list.

Another cool thing that you could try doing is combining this with the `watch` command and kind of monitor your MySQL connections in almost real-time:

```
watch -n1 mysqladmin proc
```

To stop the `watch` command, just hit `CTRL+C`

GUI clients

If you prefer using GUI clients, you could take a look a the following ones and install them locally on your laptop:

- MySQL Workbench
- Sequel Pro
- TablePlus

This will allow you to connect to your database via a graphical interface rather than the `mysql` command-line tool.

Tables

Before we get started with SQL, let's learn how to create tables and columns.

As an example, we are going to create a `users` table with the following columns:

- `id` - this is going to be the primary ID of the table and would be the unique identifier of each user.
- `username` - this column would hold the username of our users
- `name` - here, we will store the full name of the users
- `status` - here, we will store the status of a user, which would indicate if a user is active or not.

You need to specify the data type of each column.

In our case it would be like this:

- `id` - Intiger
- `username` - Varchar
- `name` - Varchar
- `status` - Number

Data types

The most common data types that you would come across are:

- `CHAR`(size): Fixed-length character string with a maximum length of 255 bytes.
- `VARCHAR`(size): Variable-length character string. Max size is specified in parenthesis.
- `TEXT`(size): A string with a maximum length of 65,535 bytes.
- `INTEGER`(size) or `INT`(size): A medium integer.
- `BOOLEAN` or `BOOL`: Holds a true or false value.
- `DATE`: Holds a date.

Let's have the following users table as an example:

- `id`: We would want to set the ID to `INT`.
- `name`: The name should fit in a `VARCHAR` column.
- `about`: As the about section could be longer, we could set the column data type to `TEXT`.
- `birthday`: For the birthday column of the user, we could use `DATE`.

For more information on all data types available, make sure to check out the official documentation here.

Creating a database

As we briefly covered in the previous chapter, before you could create tables, you would need to create a database by running the following:

- First access MySQL:

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

- Then create a database called `demo_db`:

```
CREATE DATABASE demo_db;
```

Note: the database name needs to be unique, if you already have a database named `demo_db` you would receive an error that the database already exists.

You can consider this database as the container where we would create all of the tables in.

Once you've created the database, you need to switch to that database:

```
USE demo_db;
```

You can think of this as accessing a directory in Linux with the `cd` command. With `USE`, we switch to a specific database.

Alternatively, if you do not want to 'switch' to the specific database, you would need to specify the so-called fully qualified table name. For example, if you had a `users` table in the `demo_db`, and you wanted to select all of the entries from that table, you could use one of the

following two approaches:

- Switch to the `demo_db` first and then run a select statement:

```
USE demo_db;
SELECT username FROM demo_db.users;
```

- Alternatively, rather than using the `USE` command first, specify the database name followed by the table name separated with a dot: `db_name.table_name`:

```
SELECT username FROM demo_db.users;
```

We are going to cover the `SELECT` statement more in-depth in the following chapters.

Creating tables

In order to create a table, you need to use the `CREATE TABLE` statement followed by the columns that you want to have in that table and their data type.

Let's say that we wanted to create a `users` table with the following columns:

- `id`: An integer value
- `username`: A varchar value
- `about`: A text type
- `birthday`: Date
- `active`: True or false

The query that we would need to run to create that table would be:

```
CREATE TABLE users
(
    id INT,
    username VARCHAR(255),
    about TEXT,
    birthday DATE,
    active BOOL
);
```

Note: You need to select a database first with the `USE` command as mentioned above. Otherwise you will get the following error: `ERROR 1046 (3D000): No database selected.

To list the available tables, you could run the following command:

```
SHOW TABLES;
```

Output:

```
+--------------------+
| Tables_in_demo_db |
+--------------------+
| users              |
+--------------------+
```

Dropping tables

You can drop or delete tables by using the `DROP TABLE` statement.

Let's test that and drop the table that we've just created:

```
DROP TABLE users;
```

The output that you would get would be:

```
Query OK, 0 rows affected (0.03 sec)
```

And now, if you were to run the `SHOW TABLES;` query again, you would get the following output:

```
Empty set (0.00 sec)
```

Allowing NULL values

By default, each column in your table can hold NULL values. In case that you don't wanted to allow NULL values for some of the columns in a specific table, you need to specify this during the table creation or later on change the table to allow that.

For example, let's say that we want the `username` column to be a required one, we would need to alter the table create statement and include `NOT NULL` right next to the `username` column like this:

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

That way, when you try to add a new user, MySQL will let you know that the `username` column is required.

Specifying a primary key

The primary key column, which in our case is the `id` column, is a unique identifier for our users.

We want the `id` column to be unique, and also, whenever we add new users, we want the ID of the user to autoincrement for each new user.

This can be achieved with a primary key and `AUTO_INCREMENT`. The primary key column needs to be `NOT NULL` as well.

If we were to alter the table creation statement, it would look like this:

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

Updating tables

In the above example, we created a new table and then dropped it as it was empty. However, in a real-life scenario, this would really be the case.

So whenever you need to add or remove a new column from a specific table, you would need to use the `ALTER TABLE` statement.

Let's say that we wanted to add an `email` column with type varchar to our `users` table.

The syntax would be:

```
ALTER TABLE users ADD email VARCHAR(255);
```

After that, if you were to describe the table, you would see the new column:

```
DESCRIBE users;
```

Output:

```
+----------+--------------+------+-----+---------+
| Field    | Type         | Null | Key | Default |
+----------+--------------+------+-----+---------+
| id       | int          | NO   | PRI | NULL    |
| username | varchar(255) | NO   |     | NULL    |
| about    | text         | YES  |     | NULL    |
| birthday | date         | YES  |     | NULL    |
| active   | tinyint(1)   | YES  |     | NULL    |
| email    | varchar(255) | YES  |     | NULL    |
+----------+--------------+------+-----+---------+
```

If you wanted to drop a specific column, the syntax would be:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

Note: keep in mind that this is a permanent change, and if you have any critical data in the specific column, it would be deleted instantly.

You can use the `ALTER TABLE` statement to also change the data type of a specific column. For example, you could change the `about` column from `TEXT` to `LONGTEXT` type, which could hold longer strings.

Note: Important thing to keep in mind is that if a specific table already holds a particular type of data value like an integer, you can't alter it to varchar, for example. Only if the column does not contain any values, then you could make the change.

Basic Syntax

In this chapter, we will go over the basic SQL syntax.

SQL statements are basically the 'commands' that you run against a specific database. Through the SQL statements, you are telling MySQL what you want it to do, for example, if you wanted to get the `username` of all of your users stored in the `users` table, you would run the following SQL statement:

```
SELECT username FROM users ;
```

Rundown of the statement:

- `SELECT`: First, we specify the `SELECT` keyword, which indicates that we want to select some data from the database. Other popular keywords are: `INSERT`, `UPDATE` and `DELETE`.
- `username`: Then we specify which column we want to select
- `users`: After that, we specify the table that we want to select the data from.
- The `;` is required. Every SQL statement needs to end with a semicolumn.

If you run the above statement, you will get no results as the new `users` table that we've just created is empty.

As a good practice, all SQL keywords should be with uppercase, however, it would work just fine if you use lower case as well.

Let's go ahead and cover the basic operations next.

INSERT

To add data to your database, you would use the `INSERT` statement.

Let's use the table that we created in the last chapter and insert 1 user into our `users` table:

```
INSERT INTO users(username, email, active) VALUES('bobby',
'bobby@bobbyiliev.com', true);
```

Rundown of the insert statement:

- `INSERT INTO users`: first, we specify the `INSERT INTO` keyword, which tells MySQL that we want to insert data into the `users` table.
- `users (username, email, active)`: then, we specify the table name `users` and the columns that we want to insert data into.
- `VALUES`: then, we specify the values that we want to insert in.

SELECT

Once we've inserted that user, let's go ahead and retrieve the information.

To retrieve information from your database, you could use the `SELECT` statement:

```
SELECT * FROM users;
```

Output:

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

As we specify `*` right after the `SELECT` keyword, this means that we want to get all of the columns from the `users` table.

If we wanted to the only the `username` and the `email` columns instead, we would change the statement to:

```
SELECT username,email FROM users;
```

This will return all of the users, but as of the time being we have only 1:

```
+----------+----------------------+
| username | email                |
+----------+----------------------+
| bobby    | bobby@bobbyiliev.com |
+----------+----------------------+
```

UPDATE

In order to modify data in your database, you could use the `UPDATE` statement.

The syntax would look like this:

```
UPDATE users SET username='bobbyiliev' WHERE id=1;
```

Rundown of the statement:

- `UPDATE users`: first, we specify the `UPDATE` keyword followed by the table that we want to update
- `username='bobbyiliev'` Then we specify the columns that we want to update and the new value that we want to set.
- `WHERE id=1`: Finally, by using the `WHERE` clause, we specify which user should be updated. In our case it is the user with ID 1.

NOTE: If we don't specify a `WHERE` clause, all of the entries inside the `users` table would be updated, and all users would have the `username` set to `bobbyiliev`. You need to be careful when you use the `UPDATE` statement without a `WHERE` clause, as every single row will be updated.

We are going to cover `WHERE` more in-depth in the next few chapters.

DELETE

As the name suggests, the `DELETE` statement would remove data from your database.

The syntax is as follows:

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

Similar to the `UPDATE` statement, if you don't specify a `WHERE` clause, all of the entries from the table will be affected, meaning that all of your users will be deleted.

Comments

In case that you are writing a larger SQL script, it might be helpful to add some comments so that later on, when you come back to the script, you would know what each line does.

As with all programming languages, you can add comments in SQL as well.

There are two types of comments:

- Inline comments:

To do so, you just need to add `--` before the text that you want to comment out:

```
SELECT * FROM users; -- Get all users
```

- Multiple-line comments

Similar to some other programming languages in order to comment multiple lines, you could wrap the text in `/* */` as follows:

```
/*
Get all of the users
from your database
*/
SELECT * FROM users;
```

You could write that in `.sql` file and then run it later on, or execute the few lines directly.

Conclusion

Those were some of the most common basic SQL statements.

In the next chapter, we are going to go over each of the above statements more in-depth.

SELECT

As we briefly covered in the previous chapter, the `SELECT` statement allows us to retrieve data from a specific database.

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.

Formatting

As we mentioned in the previous chapters, each SQL statement needs to end with a semi column: `;`. Alternatively, rather than using a semi column, 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.

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 suggest, 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, which is 6 divided by 3 users 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 2 times `bobby`, 2 times `devdjo` and 2 times `tony`.

If we wanted to only show 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.

WHERE

The `WHERE` clause allows you to specify different conditions so that you could filter out the data and get a specific result set.

You would add the `WHERE` clause after the `FROM` clause.

The syntax would look like this:

```
SELECT column_name FROM table_name WHERE column=some_value;
```

WHERE Clause example

If we take the example `users` table from the last chapter, let's say that we wanted to get only the active users. The SQL statement would look like this:

```
SELECT DISTINCT username,email,active FROM users WHERE
active=true;
```

Output:

```
+----------+---------------+--------+
| username | email         | active |
+----------+---------------+--------+
| bobby    | b@devdojo.com |      1 |
| tony     | t@devdojo.com |      1 |
+----------+---------------+--------+
```

As you can see, we are only getting `tony` and `bobby` back as their `active` column is `true` or `1`. If we wanted to get the inactive users, we would have to change the `WHERE` clause and set the `active` to `false`:

```
+----------+---------------+--------+
| username | email         | active |
+----------+---------------+--------+
| devdojo  | d@devdojo.com |      0 |
+----------+---------------+--------+
```

As another example, let's say that we wanted to select all users with the username `bobby`. The query, in this case, would be:

```
SELECT username,email,active FROM users WHERE
username='bobby';
```

The output would look like this:

```
+----------+---------------+--------+
| username | email         | active |
+----------+---------------+--------+
| bobby    | b@devdojo.com |      1 |
| bobby    | b@devdojo.com |      1 |
+----------+---------------+--------+
```

We are getting 2 entries back as we have 2 users in our database with the username `bobby`.

Operators

In the example, we used the = operator, which checks if the result set matches the value that we are looking for.

A list of popular operators are:

- != : Not equal operator
- > : Greater than
- >= : Greater than or equal operator
- < : Less than operator
- <= : Less than or equal operator

For more information about other available operators, make sure to check the official documentation here.

AND keyword

In some cases, you might want to specify multiple criteria. For example, you might want to get all users that are active, and the username matches a specific value. This could be achieved with the `AND` keyword.

Syntax:

```
SELECT * FROM users WHERE username='bobby' AND active=true;
```

The result set would contain the data that matches both conditions. In our case, the output would be:

```
+----+----------+-------+----------+--------+---------------+
| id | username | about | birthday | active | email         |
+----+----------+-------+----------+--------+---------------+
|  2 | bobby    | NULL  | NULL     |      1 | b@devdojo.com |
|  5 | bobby    | NULL  | NULL     |      1 | b@devdojo.com |
+----+----------+-------+----------+--------+---------------+
```

If we were to change the `AND` statement to `active=false`, we would not get any results back as none of the entries in our database match that condition:

```
SELECT * FROM users WHERE username='bobby' AND active=false;

-- Output:
Empty set (0.01 sec)
```

OR keyword

In some cases, you might want to specify multiple criteria. For example, you might want to get all users that are active, or their username matches a specific value. This could be achieved with the `OR` keyword.

As with any other programming language, the main difference between `AND` and `OR` is that with `AND`, the result would only return the values that match the two conditions, and with `OR`, you would get a result that matches either of the conditions.

For example, if we were to run the same query as above but change the `AND` to `OR`, we would get all users that have the username `bobby` and also all users that are not active:

```
SELECT * FROM users WHERE username='bobby' OR active=false;
```

Output:

```
+----+----------+-------+----------+--------+---------------+
| id | username | about | birthday | active | email         |
+----+----------+-------+----------+--------+---------------+
|  2 | bobby    | NULL  | NULL     |      1 | b@devdojo.com |
|  3 | devdojo  | NULL  | NULL     |      0 | d@devdojo.com |
|  5 | bobby    | NULL  | NULL     |      1 | b@devdojo.com |
|  6 | devdojo  | NULL  | NULL     |      0 | d@devdojo.com |
+----+----------+-------+----------+--------+---------------+
```

LIKE operator

Unlike the `=` operator, the `LIKE` operator allows you to do wildcard matching similar to the `*` symbol in Linux.

For example, if you wanted to get all users that have the `y` letter in them, you would run the following:

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

Output

```
+----+----------+-------+----------+--------+---------------+
| id | username | about | birthday | active | email         |
+----+----------+-------+----------+--------+---------------+
|  2 | bobby    | NULL  | NULL     |      1 | b@devdojo.com |
|  4 | tony     | NULL  | NULL     |      1 | t@devdojo.com |
+----+----------+-------+----------+--------+---------------+
```

As you can see, we are getting only `tony` and `bobby` but not `devdojo` as there is no `y` in `devdojo`.

This is quite handy when you are building some search functionality for your application.

IN operator

The `IN` operator allows you to provide a list expression and would return the results that match that list of values.

For example, if you wanted to get all users that have the username `bobby` and `devdojo`, you could use the following:

```
SELECT * FROM users WHERE username IN('bobby', 'devdojo');
```

Output:

```
+----+----------+-------+----------+--------+---------------+
| id | username | about | birthday | active | email         |
+----+----------+-------+----------+--------+---------------+
|  2 | bobby    | NULL  | NULL     |      1 | b@devdojo.com |
|  3 | devdojo  | NULL  | NULL     |      0 | d@devdojo.com |
|  5 | bobby    | NULL  | NULL     |      1 | b@devdojo.com |
|  6 | devdojo  | NULL  | NULL     |      0 | d@devdojo.com |
+----+----------+-------+----------+--------+---------------+
```

This allows you to simplify your `WHERE` expression so that you don't have to add numerous `OR` statements.

IS operator

If you were to run `SELECT * FROM users WHERE about=NULL;` you would get an empty result set as the `=` operator can't be used to check for NULL values. Instead, you would need to use the `IS` operator instead.

The `IS` operator is only used to check `NULL` values, and the syntax is the following:

```
SELECT * FROM users WHERE about IS NULL;
```

If you wanted to get the results where the value is not NULL, you just need to change `IS` to `IS NOT`:

```
SELECT * FROM users WHERE about IS NOT NULL;
```

Conclusion

In this chapter, you've learned how to use the `WHERE` clause with different operators to get different type of results based on the parameters that you provide.

In the next chapter, we will learn how to order the result set.

Sorting with ORDER and GROUP BY

In the last chapter, you've learned how to use the `SELECT` statement with the `WHERE` clause and filter the result set based on some conditions.

More often than not, you would want to order the results in a specific way based on a particular column. For example, you might want to order the users, alphabetically, based on their username.

In this chapter, you will learn how to use the `ORDER BY` and `GROUP BY` clauses.

ORDER BY

The main thing that you need to keep in mind when using `ORDER BY` is that you need to specify the column or columns that you want to order by. In case that you want to specify multiple columns to order by, you need to separate each column with a comma.

If we were to run the following statement without providing an `ORDER BY` clause:

```
SELECT id,username FROM users ;
```

We will get the following output:

```
+----+----------+
| id | username |
+----+----------+
|  2 | bobby    |
|  3 | devdojo  |
|  4 | tony     |
|  5 | bobby    |
|  6 | devdojo  |
|  7 | tony     |
+----+----------+
```

As you can see, the result set is sorted by the primary key, which in our case is the `id` of each user. If we wanted to sort the output by `username`, we would run the following query:

```
SELECT id,username FROM users ORDER BY username;
```

Note the `ORDER BY` statement followed by the name of the column that we want to order by.

The output in this case will be:

```
+----+----------+
| id | username |
+----+----------+
|  2 | bobby    |
|  5 | bobby    |
|  3 | devdojo  |
|  6 | devdojo  |
|  4 | tony     |
|  7 | tony     |
+----+----------+
```

Note: You can use `ORDER BY` with and without specifying a `WHERE` clause. But in case that you've specified a `WHERE` clause, you need to put the `ORDER BY` clause after the `WHERE` clause.

The default sorting is ascending and is specified with the `ASC` keyword, and you don't need to explicitly add it, but if you want to sort by descending order, you need to use the `DESC` keyword.

If we use the query above and just add `DESC` at the end as follows:

```
SELECT id,username FROM users ORDER BY username DESC;
```

We will see the following output:

```
+----+----------+
| id | username |
+----+----------+
|  4 | tony     |
|  7 | tony     |
|  3 | devdojo  |
|  6 | devdojo  |
|  2 | bobby    |
|  5 | bobby    |
+----+----------+
```

As you can see, we've got the same list of users sorted alphabetically but in reverse order.

GROUP BY

The `GROUP BY` statement allows you to use a function like `COUNT`, `MIN`, `MAX` and etc., with multiple columns.

For example, let's say that we wanted to get all of the count of all users sorted by username.

In our case, we have 2 users with username bobby, 2 users with username tony, and 2 users with username `devdojo`. This represented in an SQL statement would look like this:

```
SELECT COUNT(username), username FROM users GROUP by username;
```

The output, in this case, would be:

```
+-----------------+----------+
| COUNT(username) | username |
+-----------------+----------+
|               2 | bobby    |
|               2 | devdojo  |
|               2 | tony     |
+-----------------+----------+
```

The `GROUP BY` statement grouped the usernames that were identical. So `bobby`, `tony` and `devdojo`, and then it ran a `COUNT` on each of them.

The main thing to keep in mind here is that the `GROUP BY` should be added after the `FROM` clause and after the WHERE clause in case that you have one.

INSERT

To add data to your database, you would use the `INSERT` statement. You can insert data into one table at a time only.

The syntax is the following:

```
INSERT INTO
table_name(column_name_1,column_name_2,column_name_n)
VALUES('value_1', 'value_2', 'value_3');
```

You would start with the `INSERT INTO` statement, followed by the table that you want to insert the data into. Then you would specify the list of the columns that you want to insert the data into. Finally, with the `VALUES` statement, you specify the data that you want to insert.

The important part is that you need to keep the order of the values based on the order of the columns that you've specified.

In the above example the `value_1` would go into `column_name_1`, the `value_2` would go into `column_name_2` and the `value_3` would go into `column_name_x`

Let's use the table that we created in the last chapter and insert 1 user into our `users` table:

```
INSERT INTO users(username, email, active) VALUES('greisi',
'g@devdojo.com', true);
```

Rundown of the insert statement:

- `INSERT INTO users`: first, we specify the `INSERT INTO` keywords which tells MySQL that we want to insert data into the `users` table.
- `users (username, email, active)`: then, we specify the table name `users` and the columns that we want to insert data into.
- `VALUES`: then, we specify the values that we want to insert in.

Inserting multiple records

We've briefly covered this in one of the previous chapters, but in some cases, you might want to add multiple records in a specific table.

Let's say that we wanted to create 5 new users, rather than running 5 different queries like this:

```
INSERT INTO users(username, email, active) VALUES('user1',
'user1@devdojo.com', true);
INSERT INTO users(username, email, active) VALUES('user1',
'user2@devdojo.com', true);
INSERT INTO users(username, email, active) VALUES('user1',
'user3@devdojo.com', true);
INSERT INTO users(username, email, active) VALUES('user1',
'user4@devdojo.com', true);
INSERT INTO users(username, email, active) VALUES('user1',
'user5@devdojo.com', true);
```

What you could do is to combine this into one `INSERT` statement by providing a list of the values that you want to insert as follows:

```
INSERT INTO users
  ( username, email, active )
VALUES
  ('user1', 'user1@devdojo.com', true),
  ('user2', 'user2@devdojo.com', true),
  ('user3', 'user3@devdojo.com', true),
  ('user4', 'user4@devdojo.com', true),
  ('user5', 'user5@devdojo.com', true);
```

That way, you will add 5 new entries in your `users` table with a single `INSERT` statement. This is going to be much more efficient.

UPDATE

As the name suggests, whenever you have to update some data in your database, you would use the `UPDATE` statement.

You can use the `UPDATE` statement to update multiple columns in a single table.

The syntax would look like this:

```
UPDATE users SET username='bobbyiliev' WHERE id=1;
```

Rundown of the statement:

- `UPDATE users`: first, we specify the `UPDATE` keyword followed by the table that we want to update
- `username='bobbyiliev'` Then we specify the columns that we want to update and the new value that we want to set.
- `WHERE id=1`: Finally, by using the `WHERE` clause, we specify which user should be updated. In our case, it is the user with ID 1.

The most important thing that you need to keep in mind is that if you don't specify a `WHERE` clause, all of the entries inside the `users` table would be updated, and all users would have the `username` set to `bobbyiliev`.

Important: You need to be careful when you use the `UPDATE` statement without a `WHERE` clause as every single row will be updated.

If you have been following along all of the user entries in our `users` table currently have no data in the `about` column:

```
+----+----------+-------+
| id | username | about |
+----+----------+-------+
|  2 | bobby    | NULL  |
|  3 | devdojo  | NULL  |
|  4 | tony     | NULL  |
|  5 | bobby    | NULL  |
|  6 | devdojo  | NULL  |
|  7 | tony     | NULL  |
+----+----------+-------+
```

Let's go ahead and update this for all users and set the column value to `404 bio not found` for example:

```
UPDATE users SET about='404 bio not found;
```

The output would let you know how many rows have been affected by the query:

```
Query OK, 6 rows affected (0.02 sec)
Rows matched: 6  Changed: 6  Warnings: 0
```

Now, if you were to run a select for all `users`, you would get the following result:

```
+----+----------+------------------+
| id | username | about            |
+----+----------+------------------+
|  2 | bobby    | 404 bio not found |
|  3 | devdojo  | 404 bio not found |
|  4 | tony     | 404 bio not found |
|  5 | bobby    | 404 bio not found |
|  6 | devdojo  | 404 bio not found |
|  7 | tony     | 404 bio not found |
+----+----------+------------------+
```

Let's now say that we wanted to update the `about` column for the user with an id of 2. In this case, we need to specify a `WHERE` clause followed by the ID of the user that we want to update as follows:

```
UPDATE users SET about='Hello World :)' WHERE id=2;
```

The output here should indicate that only 1 row was updated:

```
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Now, if you again run the `SELECT id,username,about FROM users` query, you would see that the user with `id` '2' now has an updated `about` column data:

```
+----+----------+-------------------+
| id | username | about             |
+----+----------+-------------------+
|  2 | bobby    | Hello World :)    |
|  3 | devdojo  | 404 bio not found |
|  4 | tony     | 404 bio not found |
|  5 | bobby    | 404 bio not found |
|  6 | devdojo  | 404 bio not found |
|  7 | tony     | 404 bio not found |
+----+----------+-------------------+
```

DELETE

As the name suggests, the `DELETE` statement would remove data from your database.

The syntax is as follows:

```
DELETE FROM users WHERE id=5;
```

The output should indicate that 1 row was affected:

```
Query OK, 1 row affected (0.01 sec)
```

Important: Just like the `UPDATE` statement, if you don't specify a `WHERE` clause, all of the entries from the table will be affected, meaning that all of your users will be deleted. So it is critical to always add a `WHERE` clause when executing a `DELETE` statement.

Similar to the Linux `rm` command, when you use the `DELETE` statement, the data would be gone permanently, and the only way to recover your data would be by restoring a backup.

JOIN

The `JOIN` clause allows you to combine the data from 2 or more tables into one result set.

As we will be selecting from multiple columns, we would need to include the list of the columns that we want to select data from after the `FROM` clause separated by a comma.

In this chapter, we will go over the following `JOIN` types:

- `CROSS` Join
- `INNER` Join
- `LEFT` Join
- `RIGHT` Join

Before we get started, let's create a new database and 2 tables that we are going to work with:

- We are going to call the database `demo_joins`:

```
CREATE DATABASE demo_joins;
```

- Then switch to the new database:

```
USE demo_joins;
```

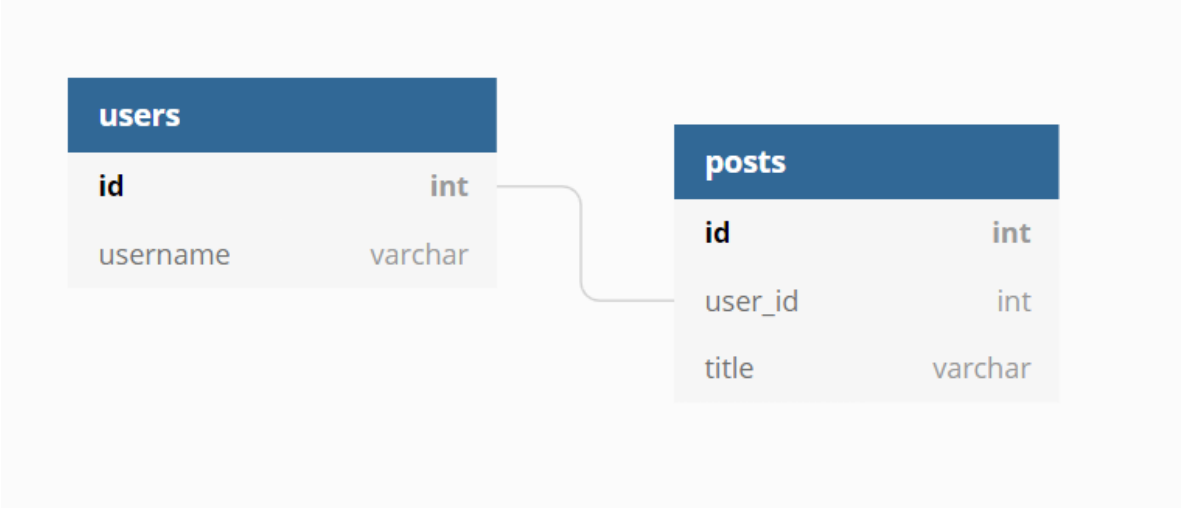
- Then the first table will be called `users` and it will only have 2 columns: `id` and `username`:

```
CREATE TABLE users
(
    id INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(255) NOT NULL
);
```

- Then let's create a second table called `posts`, and to keep things simple we will have three two columns: `id`, `user_id` and `title`:

```
CREATE TABLE posts
(
    id INT PRIMARY KEY AUTO_INCREMENT,
    user_id INT,
    title VARCHAR(255) NOT NULL
);
```

The `user_id` column would be used to reference the ID of the user that the post belongs to. It is going to be a one to many relation, e.g. one user could have many post:



- Now, let's add some data into the two tables first by creating a few users:

```
INSERT INTO users
  ( username )
VALUES
  ('bobby'),
  ('devdojo'),
  ('tony'),
  ('greisi');
```

- And finally add some posts:

```
INSERT INTO posts
  ( user_id, title )
VALUES
  ('1', 'Hello World!'),
  ('2', 'Getting started with SQL'),
  ('3', 'SQL is awesome'),
  ('2', 'MySQL is up!'),
  ('1', 'SQL - structured query language');
```

Now that we've got our tables and demo data ready let's go ahead and learn how to use joins.

Cross join

The `CROSS` join allows you to basically put the result of two tables next to each other without specifying any `WHERE` conditions. This makes the `CROSS` join the simplest one, but it also not of much use in a real-life scenario.

So if we were to select all of the users and all of the posts side by side, we will use the following query:

```
SELECT * FROM users CROSS JOIN posts;
```

The output will be all of your users and all of the posts side by side:

```
+----+----------+----+--------+-----------------+
| id | username | id |user_id | title           |
+----+----------+----+--------+-----------------+
|  4 | greisi   |  1 |      1 | Hello World!    |
|  3 | tony     |  1 |      1 | Hello World!    |
|  2 | devdojo  |  1 |      1 | Hello World!    |
|  1 | bobby    |  1 |      1 | Hello World!    |
|  4 | greisi   |  2 |      2 | Getting started |
|  3 | tony     |  2 |      2 | Getting started |
|  2 | devdojo  |  2 |      2 | Getting started |
|  1 | bobby    |  2 |      2 | Getting started |
|  4 | greisi   |  3 |      3 | SQL is awesome  |
|  3 | tony     |  3 |      3 | SQL is awesome  |
|  2 | devdojo  |  3 |      3 | SQL is awesome  |
|  1 | bobby    |  3 |      3 | SQL is awesome  |
|  4 | greisi   |  4 |      2 | MySQL is up!    |
|  3 | tony     |  4 |      2 | MySQL is up!    |
|  2 | devdojo  |  4 |      2 | MySQL is up!    |
|  1 | bobby    |  4 |      2 | MySQL is up!    |
|  4 | greisi   |  5 |      1 | SQL             |
|  3 | tony     |  5 |      1 | SQL             |
|  2 | devdojo  |  5 |      1 | SQL             |
|  1 | bobby    |  5 |      1 | SQL             |
+----+----------+----+--------+-----------------+
```

As mentioned above, in a real-life scenario, you will highly unlikely run a `CROSS` join for two whole tables. You would most likey use one of the following joins instead combined with a specific condition.

Inner join

The `INNER` join is used to join two tables, however, unlike the `CROSS` join, it is based on a condition. By using an `INNER` join, you can match the first table to the second one.

As we have a one-to-many relationship, a best practice would be to use a primary key for the posts `id` column and a foreign key for the `user_id`; that way, we can 'link' or relate the users table to the posts table. However, this is beyond the scope of this SQL basics eBook, though I might extend it in the future and add more chapters.

As an example and to make things a bit clearer, let's say that you wanted to get all of your users and the posts associated with each user. The query that we would use will look like this:

```
SELECT * FROM users
INNER JOIN posts
ON users.id = posts.user_id;
```

Rundown of the query:

- `SELECT * FROM users`: this is a standard select that we've covered many times in the previous chapters.
- `INNER JOIN posts`: then we specify the second table, which table we want to join the result set with
- `ON users.id = posts.user_id`: finally, we specify the logic on how we want the data in these two tables to be merged together. The `user.id` is the `id` column of the `user` table, which is also the primary ID, and `posts.user_id` is the foreign key in the email address table referring to the ID column in the users table.

The output will be the following, associating each user with their post based on the `user_id` column:

```
+----+----------+----+---------+-----------------+
| id | username | id | user_id | title           |
+----+----------+----+---------+-----------------+
|  1 | bobby    |  1 |       1 | Hello World!    |
|  2 | devdojo  |  2 |       2 | Getting started |
|  3 | tony     |  3 |       3 | SQL is awesome  |
|  2 | devdojo  |  4 |       2 | MySQL is up!    |
|  1 | bobby    |  5 |       1 | SQL             |
+----+----------+----+---------+-----------------+
```

The main things that you need to keep in mind here are the `INNER JOIN` and `ON` clauses.

With the inner join, the `NULL` values are discarded. For example, if you have a user who does not have a post associated with it, when running the above `INNER` join query, the user that has NULL posts will not be displayed.

To get the null values as well, you would need to use an outer join.

Left join

By using the `LEFT OUTER` join, you would get all rows from the first table that you've specified, and if there are no associated records with it within the second table, you will get a `NULL` value.

In our case, we have a user called `graisi`, which is not associated with a specific post. As you can see from the output from the previous query, the `graisi` user was not present in there. To show that user even though it does not have an associated post with it, you could use a `LEFT OUTER` join:

```
SELECT *
FROM users
LEFT JOIN posts ON users.id = posts.user_id;
```

The output will look like this:

```
+----+----------+------+---------+-----------------+
| id | username | id   | user_id | title           |
+----+----------+------+---------+-----------------+
|  1 | bobby    |    1 |       1 | Hello World!    |
|  2 | devdojo  |    2 |       2 | Getting started |
|  3 | tony     |    3 |       3 | SQL is awesome  |
|  2 | devdojo  |    4 |       2 | MySQL is up!    |
|  1 | bobby    |    5 |       1 | SQL             |
|  4 | greisi   | NULL |    NULL | NULL            |
+----+----------+------+---------+-----------------+
```

Right join

The `RIGHT OUTER` join is the exact opposite of the `LEFT OUTER` join. It will display all of the rows from the second table and give you a `NULL` value in case that it does not match with an entry from the first table.

Let's create a post that does not have a matching user id:

```
INSERT INTO posts
  ( user_id, title )
VALUES
  ('123', 'No user post!');
```

We are specifying `123` as the user ID, but we don't have such a user in our `users` table.

Now, if you were to run the `LEFT` outer join, you would not see the post as it has a null value for the corresponding `users` table.

But if you were to run a `RIGHT` outer join, you would see the post but not the `greisi` user as it does not have any posts:

```
SELECT * FROM users RIGHT JOIN posts ON users.id =
posts.user_id;
```

Output:

```
+------+----------+----+---------+-----------------+
| id   | username | id | user_id | title           |
+------+----------+----+---------+-----------------+
|    1 | bobby    |  1 |       1 | Hello World!    |
|    2 | devdojo  |  2 |       2 | Getting started |
|    3 | tony     |  3 |       3 | SQL is awesome  |
|    2 | devdojo  |  4 |       2 | MySQL is up!    |
|    1 | bobby    |  5 |       1 | SQL             |
| NULL | NULL     |  6 |     123 | No user post!   |
+------+----------+----+---------+-----------------+
```

Conclusion

The whole concept of joins might be very confusing in the beginning but would make a lot of sense once you get used to it.

The best way to wrap your head around it is to write some queries and play around with each type of `JOIN` and see how the result set changes.

For more information, you could take a look at the official documentation here.

The MySQL dump command

There are many ways and tools on how to export or backup your MySQL databases. In my opinion, `mysqldump` is a great tool to accomplish this task.

The mysqldump tool can be used to dump a database or a collection of databases for backup or transfer to another database server (not necessarily MariaDB or MySQL). The dump typically contains SQL statements to create the table, populate it, or both.

One of the main benefits of mysqldump is that it is available out of the box on almost all shared hosting servers. So if you are hosting your database on a cPanel server that you don't have root access to, you could still use it to export larger databases.

Exporting a Database

In order to export/backup a database, all you need to do is run the following command:

```
mysqldump -u your_username -p your_database_name >
your_database_name-$(date +%F).sql
```

Note that you need to change the your_database_name with the actual name of your database and the `your_username` part with your actual MySQL username.

Rundown of the arguments:

- `-u`: needs to be followed by your MySQL username
- `-p`: indicates that you would be prompted for your MySQL password
- `>`: indicates that the output of the command should be stored in the .sql file that you specify after that sign

By running the above command, you would create an export of your database, which you could later use as a backup or even transfer it to another server.

Exporting all databases

If you have root access to the server, you could use the `--all-databases` flag in order to export all of the databases hosted on the particular MySQL server. The downside of this approach is that this would create one single `.sql` export, which would contain all of the databases.

Let's say that you would like to export each database into a separate .sql file. You could do that with the following script:

```
#!/bin/bash

##
# Get a list of all databases except the system databases that
are not needed
##
DATABASES=$(echo "show databases;" | mysql | grep -Ev
"(Database|information_schema|mysql|performance_schema)")

DATE=$(date +%d%m%Y)
TIME=$(date +%s)
BACKUP_DIR=/home/your_user/backup

##
# Create Backup Directory
##

if [ ! -d ${BACKUP_DIR} ]; then
  mkdir -p ${BACKUP_DIR}
fi

##
# Backup all databases
##

for DB in $DATABASES;
do
    mysqldump --single-transaction --skip-lock-tables $DB |
gzip > ${BACKUP_DIR}/$DATE-$DB.sql.gz
done
```

The script would backup each database and would store the .sql dumps in the `/home/your_user/backup` folder. Make sure to adjust the path to your backup folder.

For more information on Bash scripting check out this opensource eBook here.

Automated backups

You can even set a cronjob to automate the backups above, that way you would have regular backups of your databases.

In order to do that, you need to make sure that you have the following content in your `.my.cnf` file. The file should be stored at:

```
/home/your_user/.my.cnf
```

You should make sure that it has secure permissions:

```
chmod 600 /home/your_user/.my.cnf
```

And you should add the following content:

```
[client]
user=your_mysql_user
password=your_mysql_password
```

Once you have your `.my.cnf` file configured, you set up a cronjob to trigger the mysqldump export of your database:

```
0 10,22 * * * /usr/bin/mysqldump -u your_username -p
your_database_name > your_database_name-$(date +%F).sql
```

The above would run at 10 AM and 10 PM every day, so you will have 2 daily backups of your database.

You can even expand the logic and add a compression step so that the .sql dumps do not consume too much webspace.