**School of Computing**

**ST0503 Back End Web Development**

**Practical 4**

**Introduction to Databases**

**Objectives:**

After completing this lab, you should be able to:

* Setup tables in MySQL
* Define Foreign Keys
* Run Select, Insert, Update and Delete Statements
* Join Tables

# 1. Introduction to databases

## 1.1. Why do we need a database?

Let’s say you run a website that provides a service (think Google or Facebook) to millions of users. Whenever a new user signs up on your website, we need to store their information *somewhere*, so that they do not have to sign up again the next time they access your website.

There are many creative ways you can persist data on a hard drive, including storing data on spreadsheets or CSV files. However, the standard for data storage, especially when dealing with millions of records, is a database.

Why are databases suitable for persisting data? They provide the following advantages over other methods (CSV files and spreadsheets):

**1. Data integrity**

A database allows you to define a “schema”, which is the structure of a database. Data that is inserted must conform to the schema, otherwise, it will not be inserted. Data integrity allows us to maintain sanity and certainty when dealing with huge amounts of data.

**2. Querying language (SQL)**

SQL is the language used to fetch and manipulate data in the database. The database is able to understand our queries and fetch the data we request.

## 1.2. What is a database?

A database is a collection of related tables. A table is composed of named columns and a set (not a list!) of rows.

The distinction between set and list is important because:

1. A set does not have order, whereas a list does.
2. Each element (row) in a set must be unique.

Take for example a user table of a social media website named Friendbook that we will be working on during the practicals:

|  |  |  |
| --- | --- | --- |
| Column Name | Data Type | NULL |
| id (Primary key) | INT | NOT NULL |
| username | VARCHAR(255) | NOT NULL |
| full\_name | VARCHAR(255) | NOT NULL |
| bio | VARCHAR(255) | NOT NULL |
| date\_of\_birth | DATE | NULL |
| created\_at | TIMESTAMP | NOT NULL |

Each column must have a data type associated with it, and the column’s nullability must be defined.

When a row is being inserted into the user table, the row values must comply with the rules and structure set in place as seen in the above figure. When the row does not comply, the database will prevent it from being inserted.

The rules and structure shown in the above figure are known formally as the database schema. It is defined when a table is created.

**What is NULL?**

NULL is a value that is used to denote the lack of value. It is similar to null or undefined in JavaScript. When a column is NOT NULL, NULL will not be allowed as a value.

## 1.3. What is MySQL?

MySQL is a relational database that we will use in the Backend Development module.

## 1.4. What is a relational database?

A relational database is a database that is able to recognize and enforce relationships between tables.

This is done through the use of primary keys and foreign keys.

### Primary Key

The primary key is the column (or group of columns) that is used to identify a specific row in the table. This means that the primary key of all rows in a table must be unique.

The user relation’s primary key is id. It is an autogenerated integer that increments after each row is inserted.

### Foreign Key

In our Friendbook API, a user can publish multiple posts. A post must belong to a user. Our database model should be able to represent this one-to-many relationship.

The foreign key is a column that references a column on another table (usually the primary key).

Take for example the post table schema:

|  |  |  |
| --- | --- | --- |
| Column Name | Data Type | NULL |
| id | INT | NOT NULL |
| text\_body | TEXT | NOT NULL |
| fk\_poster\_id (references user.id) | INT | NOT NULL |
| created\_at | TIMESTAMP | NOT NULL |

The post.fk\_poster\_id column is a foreign key that references the user.id column. This ensures that the post.fk\_poster\_id points to a valid and existing user.id.

# 2. Retrieving data from the database

## 2.1. SELECT \* FROM user

Open MySQLWorkbench and connect to the database. Go to **File > New Query Tab**. A new file should be opened.

Enter the following and click on the yellow lightning to execute the command. This command allows us to specify which database we will be using.

use friendbook;

We will query for all users in the database through the use of the SELECT statement.

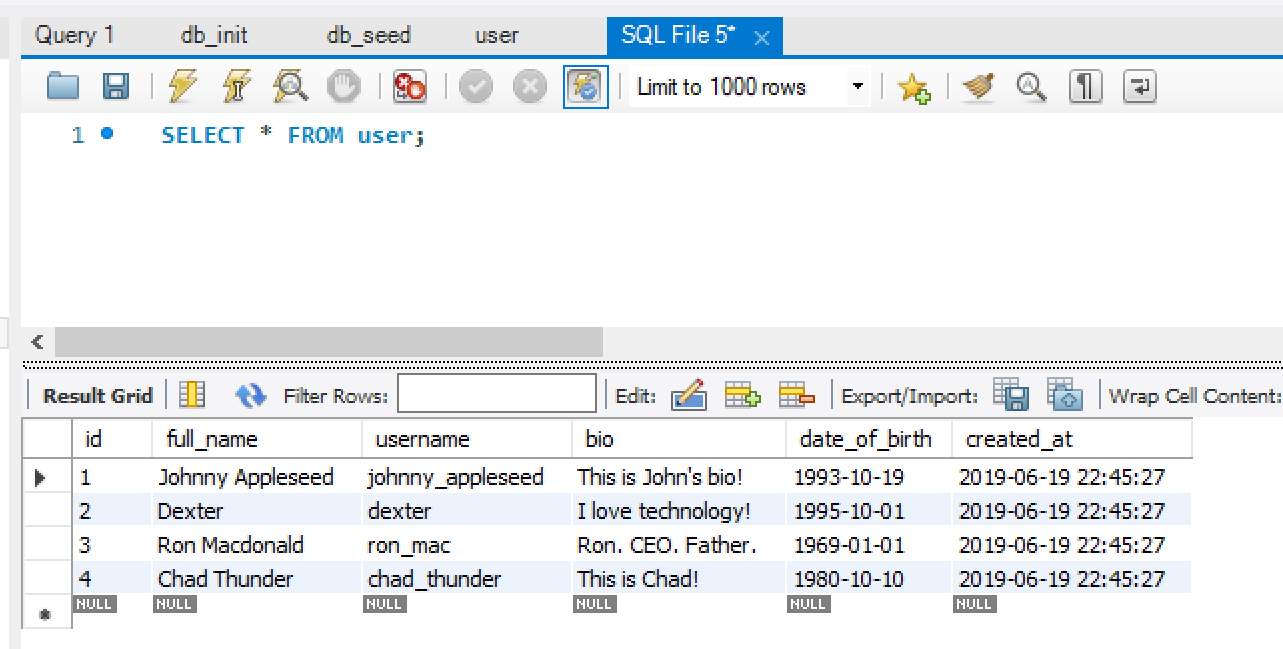
Enter the following into the file:

SELECT \* FROM user;

The asterisk \* signifies that we want all columns of the user table.

Execute the command by clicking on the lightning bolt in the toolbar above:

You should see the following in the “Result Grid”:



As you can see, all the users inserted in the seed\_database.sql file are present.

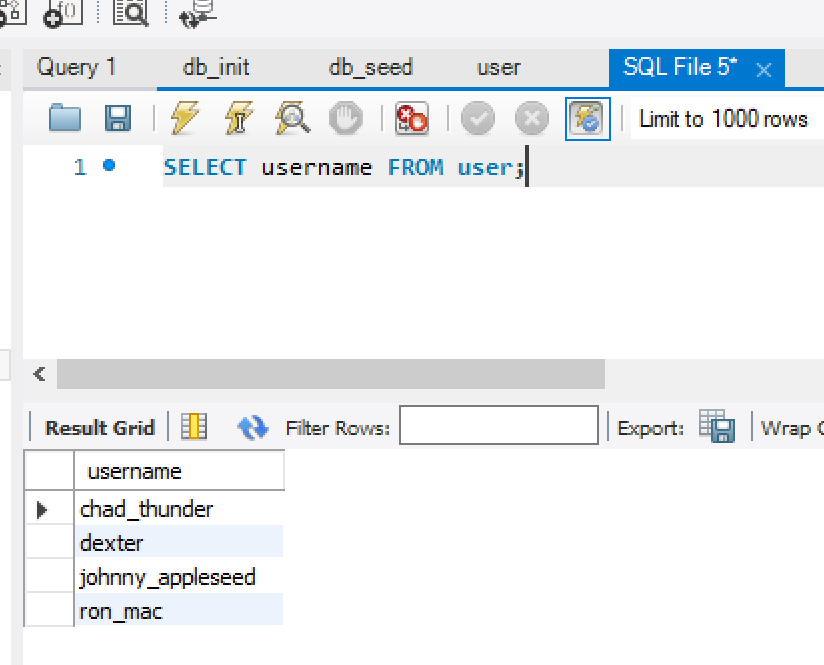
## 2.2. Specifying columns in the SELECT statement

Sometimes we only want certain columns to be returned from our SELECT statement? We can do so by specifying the columns we want in-place of the asterisk \*.

Execute the following query:

SELECT username FROM user;

You should see the following in the Result Grid:



As you can see only the username is present.

## 2.3. Filtering with the WHERE clause

We may not want to retrieve every user from the database. The WHERE clause allows us to filter rows based on the provided condition.

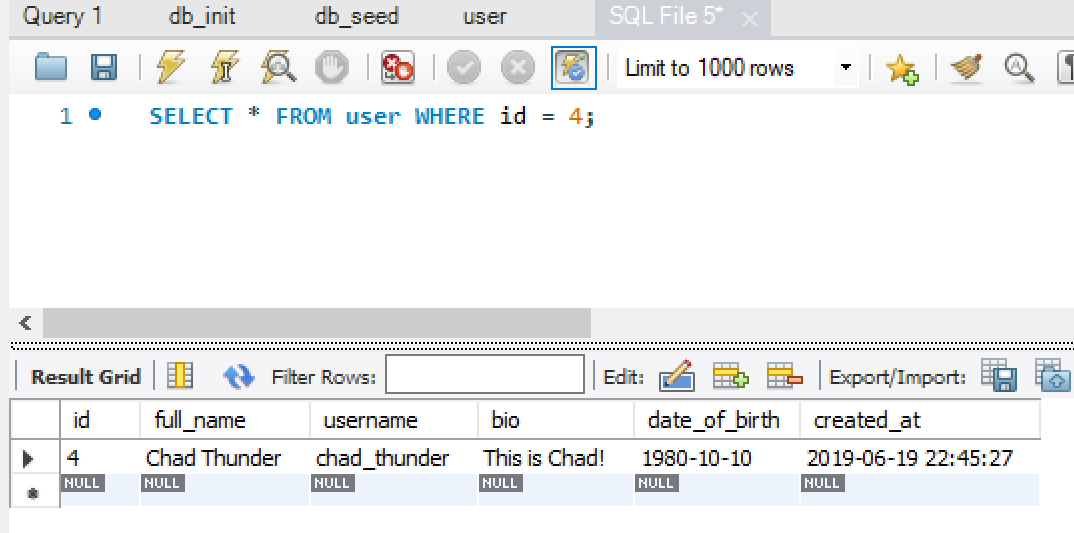
This is the syntax for filtering:

SELECT <columns> FROM <table> WHERE <condition>;

We would like to find a user with their id. Enter the following into the test.sql file:

SELECT \* FROM user WHERE id = 4;

Execute the SQL file. You should see the following output in the Result Grid:

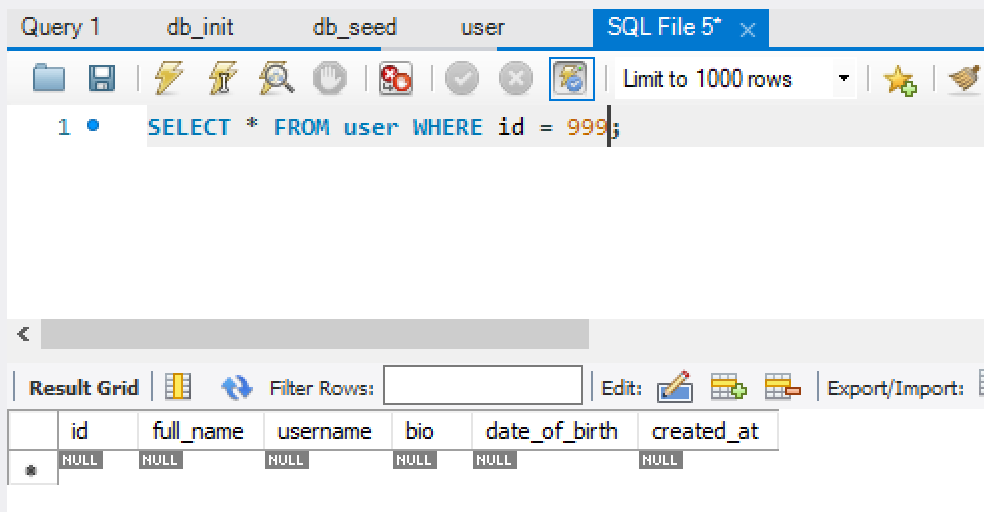


As you can see, only one row is present, because user.id is a PRIMARY KEY, meaning it is unique. There cannot be two users with the same id.

Let’s try to provide a non-existent user.id:

SELECT \* FROM user WHERE id = 999;

Execute the command. You should see that there are zero rows returned:



# 3. Inserting rows with the INSERT statement

Let’s try and insert a post that beings to a user with id of 4.

Before that, let’s inspect the post table’s schema.

## 3.1. Inspecting the schema of the post table

In the “db\_init.sql” file we used to create the tables in our friendbook database, the statement used to create the post table can be found:

CREATE TABLE post (  
 id INT NOT NULL AUTO\_INCREMENT,  
 fk\_poster\_id INT NOT NULL,  
 text\_body TEXT NOT NULL,  
 created\_at TIMESTAMP NOT NULL DEFAULT NOW(),  
 PRIMARY KEY (id),  
 FOREIGN KEY (fk\_poster\_id) REFERENCES user(id) ON DELETE CASCADE  
);

Since id and created\_at is generated automatically for us, we only need to provide fk\_poster\_id and text\_body. Note that due to the foreign key constraint, fk\_poster\_id must be a existing user id (something like 999 will cause an error).

## 3.2. Inserting a post

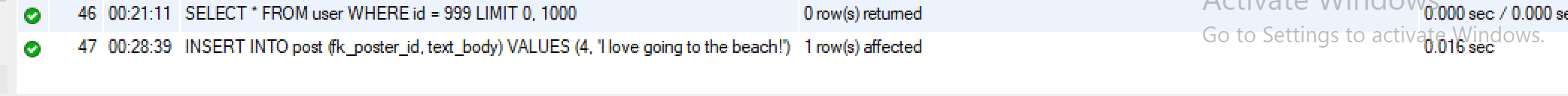
This is the syntax for INSERT statements:

INSERT INTO <table> (<columns>) VALUES (<column values>);

Execute the following command to insert a post belonging to user with id of 4 (Chad Thunder):

INSERT INTO post (fk\_poster\_id, text\_body) VALUES (4, 'I love going to the beach!');

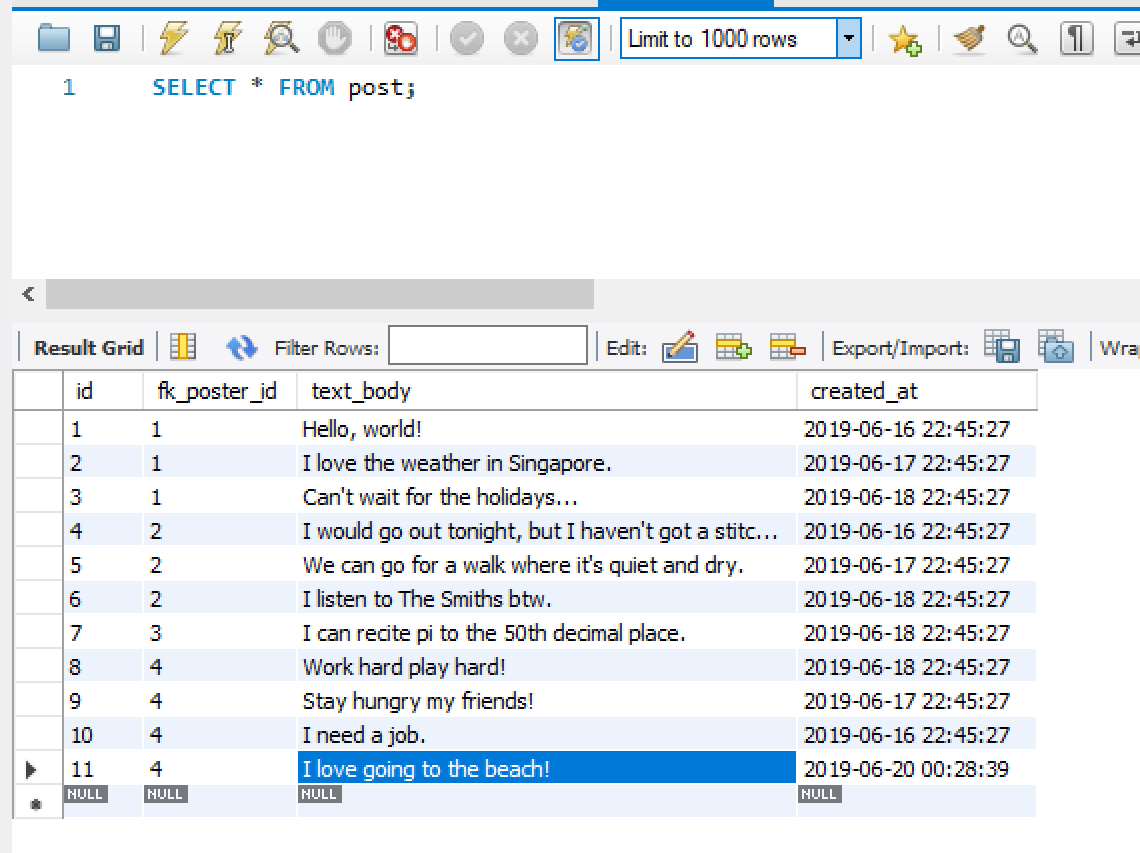
You should see that one row was affected by our query in the Action Output:



Check if the post was inserted using a SELECT statement:

SELECT \* FROM post;

You should see the row that was inserted:



## 3.3. Foreign key in action

What happens if a post.fk\_poster\_id does not refer to any user in the database?

Insert a post with a non-existent fk\_poster\_id:

INSERT INTO post (fk\_poster\_id, text\_body) VALUES (999, 'Great weather today!');

Execute the command. You should see the following error:

Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails (friendbook.post, CONSTRAINT post\_ibfk\_1 FOREIGN KEY (fk\_poster\_id) REFERENCES user (id))

This error occurs because post.fk\_poster\_id is a foreign key that references user.id. Thus, all posts inserted should have a valid fk\_poster\_id.

The foreign key reference was defined in create\_tables.sql used to create the post table.

# 4. Updating rows in the database

UPDATE statements are used to update rows in the database.

This is the syntax of an UPDATE statement:

UPDATE <table>  
SET  
 <column> = <new value>,  
 <column> = <new value>,  
 ...  
WHERE <condition>;

## 4.1. Updating a user’s full\_name

Update the full\_name of the user with id of 2 from Dexter to Dexter Ng:

UPDATE user  
SET  
 full\_name = 'Dexter Ng'  
WHERE id = 2;

The WHERE clause must be provided, otherwise, all rows with be updated.

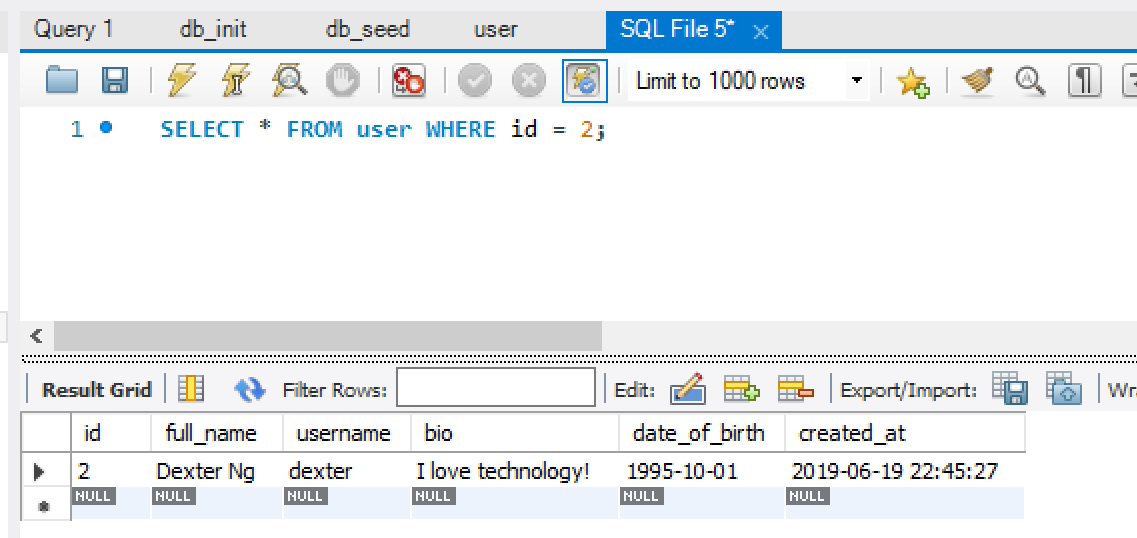
After executing the command, you should see only one row being affected.

## 4.2 Ensuring the update was done

Execute a SELECT statement for user of id to ensure the full\_name was updated:

SELECT \* FROM user WHERE id = 2;

You should see the following:



# 5. Deleting rows from the database

The DELETE statement is used to delete rows from a table.

This is the syntax of a DELETE statement:

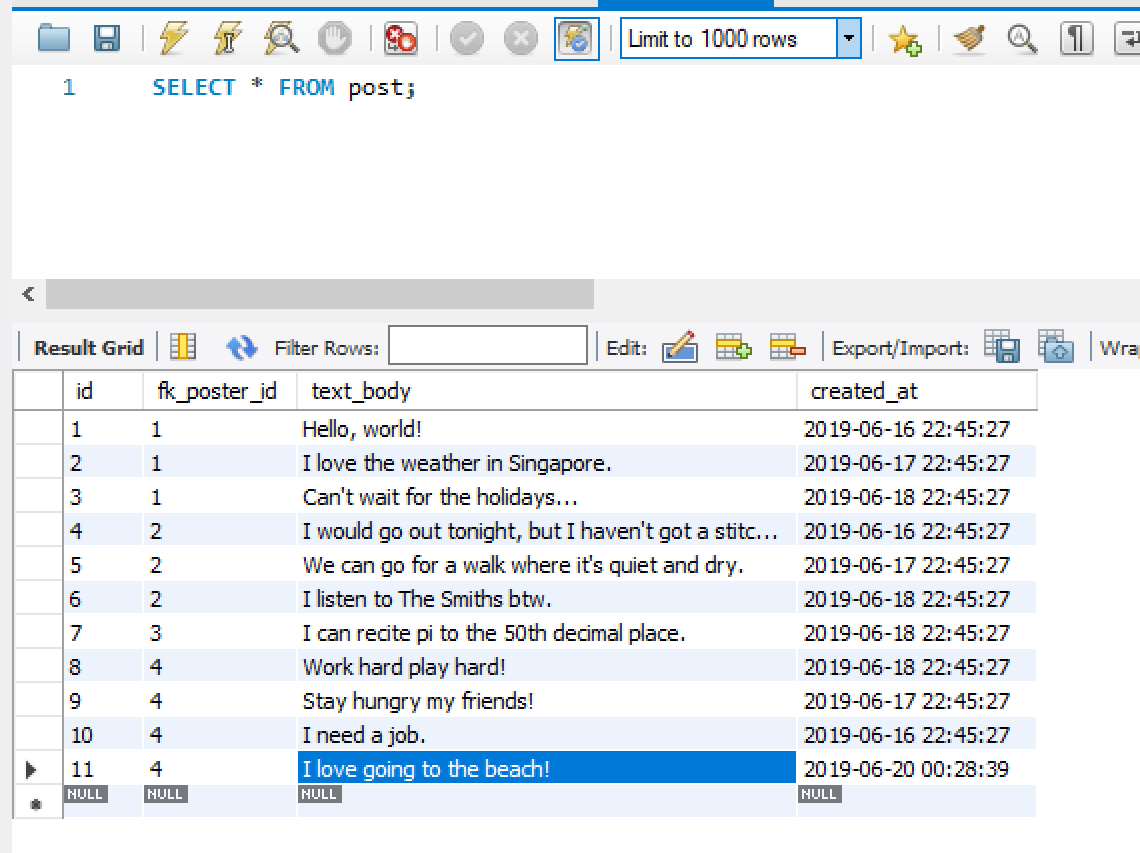
DELETE FROM <table> WHERE <condition>;

## 5.1. Finding a post to delete

Inspect all rows of the post table:

SELECT \* FROM post;

You should see the following:



## 5.2. Deleting a post

Let’s say Johnny Appleseed regrets posting "I love the weather in Singapore" and would like to delete it. From the previous SELECT query we can see that it has an id of 2.

Delete that post by specifying its id in the WHERE clause:

DELETE FROM post  
WHERE id = 2;

Execute the statement. One row should be affected, assuming the id provided was valid.

Perform a SELECT statement and ensure that the post is no longer present.

# 6. Representing many-to-many relationships (Likes)

In Section 1.4, it was mentioned that the user-post relationship is a one-to-many relationship. This is because a user can have multiple posts, but a post can only belong to one user. Hence, the foreign key belongs to the post table.

We would like to implement a “like” feature, where a user can like a post.

Since a user can like many posts, and a post can be liked by multiple users, this is known as a many-to-many relationship in SQL.

To represent this relationship, we need a likes table that references a single user and a single post. The likes table is known as an associative table, as it is used to resolve a many-to-many relationship between user and post.

This is the CREATE TABLE statement used to create the likes table (taken from db\_init.sql):

CREATE TABLE likes (  
 id INT NOT NULL AUTO\_INCREMENT,  
 fk\_user\_id INT NOT NULL,  
 fk\_post\_id INT NOT NULL,  
 created\_at TIMESTAMP NOT NULL DEFAULT NOW(),  
 PRIMARY KEY (id),  
 UNIQUE KEY (fk\_user\_id, fk\_post\_id),  
 FOREIGN KEY (fk\_user\_id) REFERENCES user(id) ON DELETE CASCADE,  
 FOREIGN KEY (fk\_post\_id) REFERENCES post(id) ON DELETE CASCADE  
);

We have a UNIQUE KEY with both the user id and the post id so that a user cannot like the same post more than once.

## 6.1. Liking a post

Lets make user of id 4 like post of id 3.

Execute the following command to insert a like:

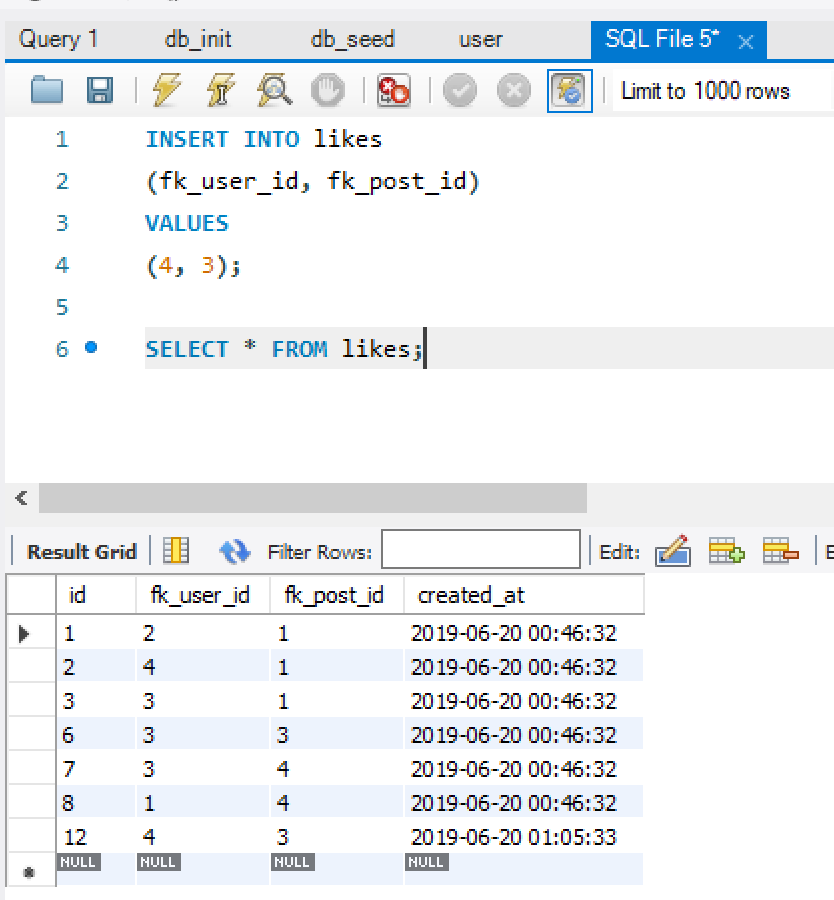
INSERT INTO likes  
(fk\_user\_id, fk\_post\_id)  
VALUES  
(4, 3);

## 6.2. Check that the like is inserted

Check that likes has one new row:

SELECT \* FROM likes;

Output:



## 6.3. Can a user like a post more than once?

Execute the same query from Section 6.1:

INSERT INTO likes  
(fk\_user\_id, fk\_post\_id)  
VALUES  
(4, 3);

You should see an error. Why can’t you insert the row this time?

Notice how the ids are not contiguous? This is because when we deleted the post of id 2, all likes with fk\_post\_id of 2 are automatically deleted. This occurs because of ON DELETE CASCADE.

## 6.4. Inserting a non-existent post.id

Insert a likes row with an invalid fk\_post\_id:

INSERT INTO likes  
(fk\_user\_id, fk\_post\_id)  
VALUES  
(2, 999);

You should see an error. What property of the schema prevents the row from being inserted?

# 7. Representing friendships in SQL

In the same way the like table ties many users to many posts, our friendship tables ties many users to many users.

This is the CREATE TABLE statement used to create the friendship table (taken from db\_init.sql):

CREATE TABLE friendship (  
 id INT NOT NULL AUTO\_INCREMENT,  
 fk\_friend\_one\_id INT NOT NULL,  
 fk\_friend\_two\_id INT NOT NULL,  
 created\_at TIMESTAMP NOT NULL DEFAULT NOW(),  
 PRIMARY KEY (id),  
 UNIQUE KEY (fk\_friend\_one\_id, fk\_friend\_two\_id),  
 FOREIGN KEY (fk\_friend\_one\_id) REFERENCES user(id) ON DELETE CASCADE,  
 FOREIGN KEY (fk\_friend\_two\_id) REFERENCES user(id) ON DELETE CASCADE  
);

Take note of the two foreign keys that point to the user.id. The ON DELETE CASCADE causes friendship rows to be delete whenever users involved in the friendship are deleted.

## 7.1. Adding a friendship (naive)

Execute this query to add a friendship between user with id 2 and user with id 3.

INSERT INTO friendship  
(fk\_friend\_one\_id, fk\_friend\_two\_id)  
VALUES  
(2, 3);

## 7.2. Checking if friendship exists

Execute the following statement to check if a friendship exists between user 2 and user 3:

SELECT \* FROM friendship  
WHERE  
fk\_friend\_one\_id = 2  
AND  
fk\_friend\_two\_id = 3;

Friendships by nature are a two-way relationship. If user A is a friend of user B, user B must be a friend of user A.

What happens if we reverse the order we provide the user ids?

Execute the following:

SELECT \* FROM friendship  
WHERE  
fk\_friend\_one\_id = 3  
AND  
fk\_friend\_two\_id = 2;

There should be zero rows returned because of the order of the user ids.

So how do we know which order to provide the ids? We want to avoid missing some friendships when we query from the friendship table.

We can avoid worrying about the order of the ids by inserting both orders. Execute this query to complete the insertion of a friendship:

INSERT INTO friendship  
(fk\_friend\_one\_id, fk\_friend\_two\_id)  
VALUES  
(3, 2);

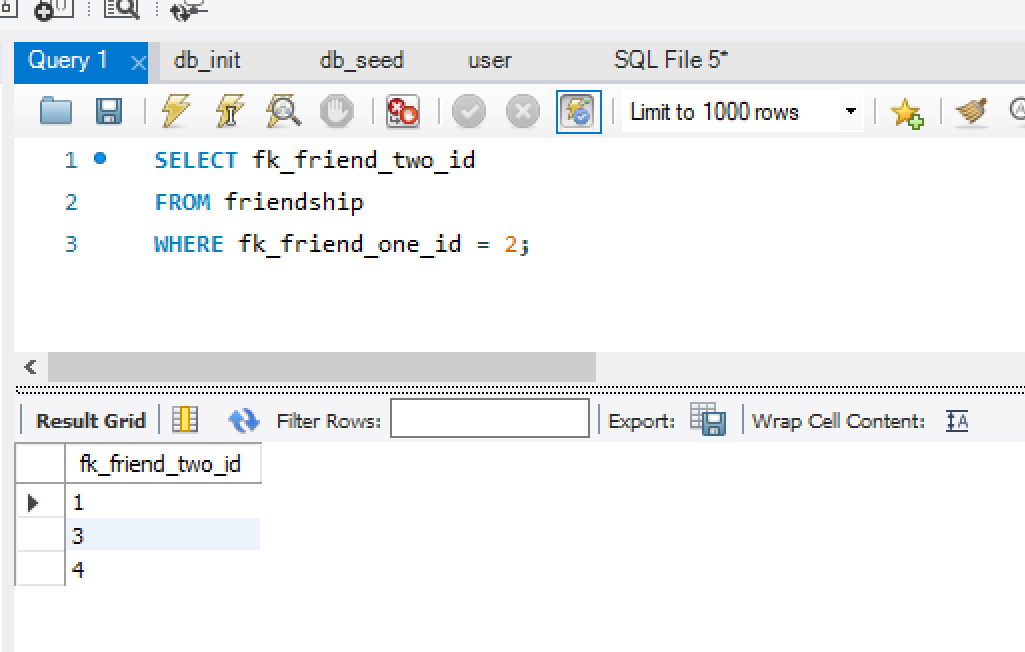
That way, we can query for the friendship with the user ids in both orders and the friendship will still be shown.

## 7.3. Querying for all friends of user 2

We can query for the id of friends of user of id 2 with the following command:

SELECT fk\_friend\_two\_id  
FROM friendship  
WHERE fk\_friend\_one\_id = 2;

Output:



## 7.4. Exercise: Deleting a friendship

Using two SQL commands, delete the friendship between user with id 2 and user with id 3.

delete from friendship where fk\_friend\_one\_id = 3 and fk\_friend\_two\_id = 2;

delete from friendship where fk\_friend\_one\_id = 2 and fk\_friend\_two\_id = 3;

After that has been done, select all friendships:

SELECT \* FROM friendship;

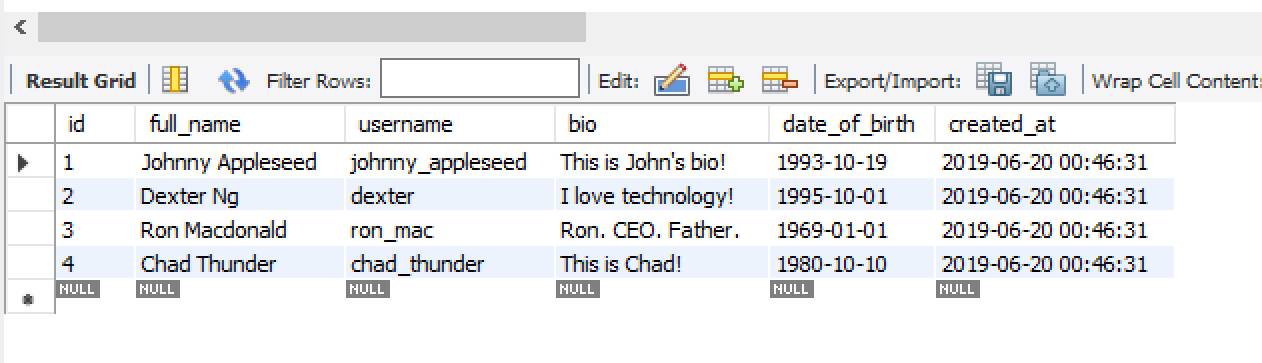
The friendship should no longer be present in the database.

# 8. Join tables

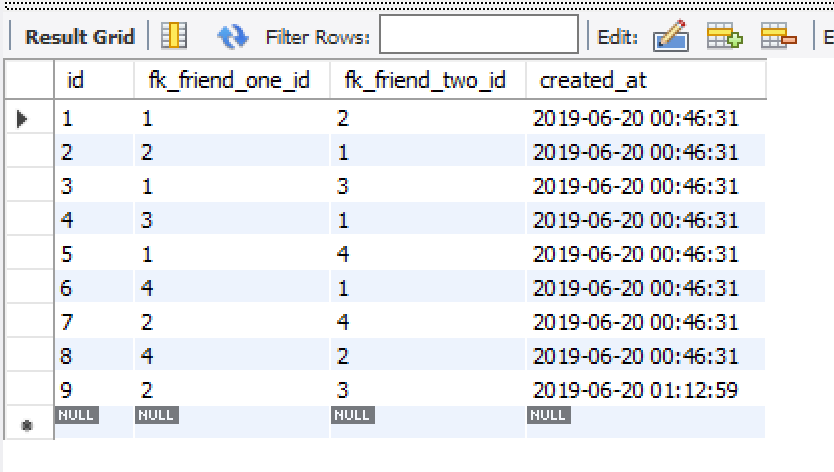
In Section 7.3, we wrote a simple SELECT statement to retrieve the user.id of friends of a user with id 2.

What if we would like rows of users instead of rows of user.ids? We could execute two queries, one to get the friend ids, and another to retrieve all users with those ids. However, a more performant solution may be to execute one statement through the use of join tables.

Here is our current user table:



Here is our current friendship table:



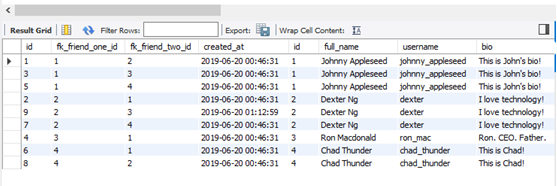
friendship.fk\_friend\_one\_id and friendship.fk\_friend\_two\_id both must point to existing user.ids because of the foreign key constraints (see the CREATE TABLE statement for friendship).

We would like to join the rows in user and friendship that share the same user.id and friendship.fk\_friend\_one\_id.

We can perform a join as below:

SELECT \* FROM friendship, user  
where friendship.fk\_friend\_one\_id = user.id;

Output:



Notice that friendship.fk\_friend\_one\_id is equal to user.id in every single row.

This means that we can find the friends of user with user.id of x by showing only rows where friendship.fk\_friend\_two\_id = x.

**Exercise: How can you find all the friends of user 2?**

Ie, show the rows of friend\_one and friend\_two in the friendship table

# SELECT user.id,full\_name,username FROM user,friendship WHERE fk\_friend\_one\_id = 2 and fk\_friend\_two\_id = user.id

# 9. Resetting the database

We will now revert the database back to when it was initially created and seeded.

Execute the following statement to drop the friendbook database and all of its tables:

DROP DATABASE friendbook;

Open the “db\_init.sql” file and execute the entire file to create all the tables.

Open the “db\_seed.sql” file and execute the entire file to seed the database.

# Conclusion

Now that you have an idea of what a database is, and how we can fetch and manipulate the data it contains, we will be doing the same thing in the next practical, except we will do it programmatically in a Node project.