MySQL BootCamp

Section 2: Overview

- What is a Database?
    - It is a collection of data.
    - It has methods to access and manipulate the data.

Many times a *Database Management System* is referred to as databases.

- What is the difference between SQL and MySQL?
    - SQL is the language used to manipulate our databases.

Example of SQL: Find All Users who are 18 or older

```
SELECT * FROM Users WHERE age >= 18;
```

- Once someone learns SQL, it's easy to switch to another database that uses SQL. The main difference between MySQL, POstgreSQL and other programs that make use of SQL are the features they have.

Section 3: Creating Databases and Tables

- SQL does not differentiate between **uppercase and lowercase** letters. Nevertheless, it is customary that words related to the SQL language are written in uppercase, while the rest is written in lowercase.
- In SQL every command ends with a **semicolon**.

- Show every database:

```
show databases;
```

- Create new database:

```
CREATE DATABASE <name>;
```

- Delete database:

```
DROP DATABASE <name>;
```

- Specify which database we are going to work with:

```
USE <name>;
```

- Check which database we are working with:

```
SELECT database();
```

    - If there are no active databases, the return will be **NULL**.

- In SQL, the databases are composed of multiple tables.
    - Each column in a table must be related to a data type and it must be consistent for every row, if they are not consistent SQL will raise an error.
        - There are **numeric types**, **string types** and **date types**. The most important ones are:
            - INT
            - VARCHAR is a string with variable length and max of 2
            - VARCHAR(<int>) is varchar string length <int>
- Create a table:

```
CREATE TABLE <table_name>
     (
          <column_name> <data_type>,
          <column_name> <data_type>
     );
```

- Show every existing table in the database.

```
SHOW TABLES
```

- Show the columns in a table

```
SHOW COLUMNS FROM <table_name>
```

```
DESC <table_name>
```

  - The second option does a slightly different thing, but for the effects of what we are trying to do, it does not matter.

- Delete a table

```
DROP TABLE <table_name>
```

Pastries Example

```
CREATE TABLE pastries (
      name VARCHAR(50),
      quantity INT
);
SHOW COLUMNS FROM pastries;
DROP TABLE pastries;
```

Section 4: Inserting Data

- Insert values to a table

```
INSERT INTO <table_name>(<column_name>, <column_name>)
VALUES(<value>, <value>);
```

- Insert multiple values at once

```
INSERT INTO <table_name>(<column_name>, <column_name>)
VALUES
      (<value>, <value>),
      (<value>, <value>);
```

- Show values in a table

```
SELECT * FROM cats;
```

People Example

```
CREATE TABLE people(
      first_name VARCHAR(20),
      last_name VARCHAR(20),
      age INT
);

INSERT INTO people(first_name, last_name, age)
VALUES('Tina', 'Belcher', 13);

INSERT INTO people(last_name, age, first_name)
VALUES('Belcher', 42, 'Bob');

INSERT INTO people(first_name, last_name, age)
VALUES
      ('Linda', 'Belcher', 45),
      ('Phillip', 'Frond', 38),
      ('Calvin', 'Fischoeder', 70);

DROP TABLE people;
```

- Display warnings

```
SHOW WARNINGS;
```

    - This command only shows the warnings of the last command.

- The **NULL** value in a table means a value is unknown, meaning we can insert data without specifying every column.
    - This means that a command as `INSERT INTO cats() VALUES();` and we will have a new row with purely **NULL** values.
- Create a table that does not allow **NULL**:

```
CREATE TABLE <table_name>
     (
          <column_name> <data_type> NOT NULL,
          <column_name> <data_type> NOT NULL
     );
```

- Specify a **default value** to a column:

```
CREATE TABLE <table_name>
     (
          <column_name> <data_type> DEFAULT ‘unnamed’,
          <column_name> <data_type> DEFAULT 0
     );
```

- Upto this moment, we can insert the same data multiple times, but they are not uniquely identifiable. To solve this problem, we make use of a unique ID, which is called **Primary Key** in SQL.
    - As an example, we can use:

```
CREATE TABLE cats(
     cat_id INT NOT NULL,
     name VARCHAR(100),
     age INT,
     PRIMARY KEY (cat_id)
);
```

- This primary key can be automatically specified by adding to the cat_id line the command **AUTO_INCREMENT**.

Employees Example

```sql
CREATE TABLE employees(
      id INT NOT NULL AUTO_INCREMENT,
      last_name VARCHAR(255) NOT NULL,
      first_name VARCHAR(255) NOT NULL,
      middle_name VARCHAR(255),
      age INT NOT NULL,
      current_status VARCHAR(255) NOT NULL DEFAULT 'employed',
      PRIMARY KEY (id)
);
```

Section 5: CRUD

- Read **every column** in the table:

```
SELECT * FROM <table_name>;
```

- Read a **single column** from the table:

```
SELECT <column_name> FROM <table_name>;
```

- Read **multiple columns** from a table:

```
SELECT <column_name>, <column_name> FROM <table_name>;
```

- If we want to get specific and don't obtain every row in the table we use the **WHERE** command. Which can be used as

```
SELECT * FROM cats WHERE age=4;
```

```
SELECT * FROM cats WHERE name='Egg';
```

  - For this example, it is important to note that by default SQL is case insensitive.

- We can make use of aliases to display column names in another way.

```
SELECT <column_name> as <alias>, <column_name> FROM <table_name>;
```

Rapid Fire Example

```
CREATE TABLE cats(
     cat_id INT NOT NULL AUTO_INCREMENT,
     name VARCHAR(100),
     breed VARCHAR(100),
     age INT,
     PRIMARY KEY (cat_id)
);

INSERT INTO cats(name, breed, age) VALUES
     ('Ringo', 'Tabby', 4),
     ('Cindy', 'Maine Coon', 10),
      ('Dumbledore', 'Maine Coon', 11),
      ('Egg', 'Persian', 4),
      ('Misty', 'Tabby', 13),
      ('George Michael', 'Ragdoll', 9),
      ('Jackson', 'Sphynx', 7);

SELECT cat_id FROM cats;
SELECT name, breed FROM cats;
SELECT name, age FROM cats WHERE breed='Tabby';
SELECT cat_id, age FROM cats WHERE cat_id=age;
```

- To update we use the commands **UPDATE** and **SET**:

```
UPDATE <table_name> SET <column_name>=<value>
WHERE <conditional_expression>;
```

- As a rule of thumb, we should check the **WHERE** condition before actually updating the data.

- To delete something in SQL we use the command **DELETE**:

```
DELETE FROM cats WHERE name='Egg';
```

Section 7: String Functions

- SQL can be run from a **FILENAME.sql**

```
source <filename>.sql
```

- With this, we can create a file such as **first_example.sql**

```
CREATE TABLE books
   (
       book_id INT NOT NULL AUTO_INCREMENT,
       title VARCHAR(100),
       author_fname VARCHAR(100),
       author_lname VARCHAR(100),
       released_year INT,
       stock_quantity INT,
       pages INT,
       PRIMARY KEY(book_id)
   );
```

- We can use the command **CONCAT** to concatenate strings or columns.

```
SELECT CONCAT(<column>, <string>, <column>) FROM <table_name>;
```

- We can also use **CONCAT_WS**

```
SELECT CONCAT_WS(<separator>, <column>, <column>) FROM <table_name>;
```

- To select a substring we can use **SUBSTRING**. It is important to know that in SQL-strings indices start at 1.

```
SELECT SUBSTRING(<column/string>, <start>) FROM <table_name>;
SELECT SUBSTRING(<column/string>, <start>, <end>) FROM <table_name>;
```

- Alternatively, we can also write **SUBSTR**.

- We can use **REPLACE** to replace a piece of a string with another string. This function is **case sensitive**, meaning upper and lowercase are treated differently.

```
SELECT REPLACE(<string>, <replace_this_string>, <with_this_string>);
```

- We can reverse string with the command **REVERSE**

```
SELECT REVERSE(<string/column>) FROM <table_name>;
```

- We can obtain the length of a string with **CHAR_LENGTH**

```
SELECT CHAR_LENGTH(<string/column>) FROM <table_name>;
```

- To transform the string to uppercase we must use **UPPER** and to lowercase we must use **LOWER**

```
SELECT UPPER(<string/column>) FROM <table_name>;
SELECT LOWER(<string/column>) FROM <table_name>;
```

Section 8: Refining Our Selections

- The **DISTINCT** command allows us to obtain the distinct values in a column

```
SELECT DISTINCT <column_name> FROM <table_name>;
```

- **DISTINCT** can be used with **CONCAT**, and it can also be used with multiple column names.

```
SELECT DISTINCT <column_name>, <column_name> FROM <table_name>;
```

- We can sort our results by using the command **ORDER BY**

```
SELECT <column_name> FROM <table_name> ORDER BY <column_name>;
```

which returns the sorted in **ascending order**, we can also be explicit by inserting the command **ASC**.

- To obtain the **descending order** we end the command with a **DESC**

```
SELECT <column_name> FROM <table_name> ORDER BY <column_name> DESC;
```

- If we use **ORDER BY <number>** we are referring to the n-th element in the **SELECT** command.
- We can sort by multiple columns, where the sorting is done by the first column and then by the second column.

- To obtain the first **N** elements from a table, we can make use of **LIMIT**

```
SELECT <column_name> FROM <table_name> LIMIT <number>;
```

Combined with **ORDER BY** it becomes a very powerful command. As an example we can use it to obtain the 5 most recent books.

```
SELECT released_year, title FROM books ORDER BY released_year DESC 5;
```

- If **ORDER BY** is used with two numbers. The **first number** means the number of the row in the table and the **second number** is the number of rows the program will show. In this case the row numbers start at 0.

- We can use the command **LIKE** to search for items that contains a particular string

```
WHERE <column> LIKE '%<string>%'
```

where the symbols **%** represent wild-cards, that can be any type of string.

- We can use only one **%** or we could use a **%** in the middle of the string.
- We could use **LIKE '_ _ _ _'** to represent anything with a specific number of characters/digits. In this case it would mean 4 characters long and **LIKE '_ _'** would be 2 characters long.
- To search for something with a **%** or **_** we can escape them with a backslash.

Section 9: Aggregate Functions

- To count the number of elements we use **COUNT**

```
SELECT COUNT(*) FROM <table_name>;
```

- To count unique number of elements we can use **DISTINCT**

```
SELECT COUNT(DISTINCT <column_name>) FROM <table_name>;
```

  where we can also use multiple column names to count unique tuples.
- To combine it with **LIKE** we use

```
SELECT COUNT(*) FROM <table_name>
WHERE <column_name> LIKE <string_pattern>;
```

- **GROUP BY** summarized identical data into single rows.
  - As an example we might want to count the number of books written by each author

```
SELECT author_fname, author_lname, COUNT(*)
FROM books GROUP BY author_fname, author_lname;
```

- To find the minimum and maximum we use **MIN** and **MAX**

```
SELECT MIN(<column_name>) FROM <table_name>;
SELECT MAX(<column_name>) FROM <table_name>;
```

- To find the other columns related to the **MIN** and **MAX** we can make use of **subqueries**. As an example:

```
SELECT * FROM books
WHERE pages = (SELECT MIN(pages) FROM books);
```

  - The main problem of this is that it is slow.

  Another option is to make use of **ORDER BY**

```
SELECT * FROM books
ORDER BY pages ASC LIMIT 1;
```

- **MIN/MAX** can be used in conjunction with **GROUP BY**. As an example, we can find each author's first book.

```
SELECT author_fname, author_lname, MIN(released_year)
FROM books
GROUP BY author_fname, author_lname;
```

- To sum things together, we can use **SUM:**

```
SELECT SUM(<column_name>) FROM <table_name>;
```

- **SUM** and **GROUP BY** can be used the same way **MIN/MAX** is used with **GROUP BY**.

- To calculate average we can use **AVG**:

```
SELECT AVG(<column_name>) FROM <table_name>;
```

- **AVG** and **GROUP BY** can be used the same way **MIN/MAX** is used with **GROUP BY**.

Section 10: Data Types

Text

- **CHAR(n)**: has a fixed length of n. It can be any number from 0 to 255. Trailing spaces are removed unless the **PAD_CHAR_TO_FULL_LENGTH** SQL mode is enabled. It is faster than **VARCHAR(n)**.
- **VARCHAR(n)**: has maximum length of n. It can be any number from 0 to 255.

Numbers

- **INT**: whole numbers
- **DECIMAL(n,m)**: fixed-point decimal numbers. **n** is the total number of digits a number can have and **m** is the number of decimals after the decimal point. **n** has a range from 1 to 65 and **m** has a range from 0 to 65.
  - If the value inserted is greater than the maximum number, SQL will assign the largest possible decimal.
  - If the value inserted has more decimals than the assigned by **m**, SQL rounds the number
- **FLOAT**: floating point numbers. Have ~7 digits precision.
- **DOUBLE**: floating point with double precision. Have ~15 digits of precision.

- Due to precision, it is usually preferred to use **DECIMAL** in databases. Unless we really don't care about precision.

Dates and Times

- **DATE**: Stores date without time. Formatted as **YYYY-MM-DD**
- **TIME**: Stores time without date.   Formatted as **HH:MM:SS**
- **DATETIME**: Stores date and time Formatted as **YYYY-MM-DD HH:MM:SS**
- **TIMESTAMP**: Stores datetime and time. It has a smaller range than **DATETIME** and it uses less memory.

- We can obtain the **current date** with **CURDATE()**, the **current time** with **CURTIME()** and **NOW()** to obtain the **current datetime**.

- To format dates we can use
  - **DAY**(<date/datetime>)
  - **DAYNAME**(<date/datetime>)
  - **DAYOFWEEK**(<date/datetime>)
  - **DAYOFYEAR**(<date/datetime>)
  - **DATE_FORMAT**(<date/datetime/string>, <format>)
- To format time we can use
  - **MINUTE**(<time/datetime>)
  - **HOUR**(<time/datetime>)

- To calculate the number of days difference between two dates we can use **DATEDIFF**(date1, date2), which is equivalent to date1 - date2.
- To add a **DATE/TIME** to a **DATETIME** we can use **DATE_ADD**(<datetime>, <interval>)
- **DATETIME** arithmetics can also be done with **+** and **-** symbols.

Section 11: Logical Operators

- We have already seen the **equal** operator, which is written as =
- **Not equal** is expressed as !=
- The **NOT LIKE** is used to find strings that do not contain a certain pattern.
- To obtain values **greater than** a certain value we use > and **greater than or equal than** is expressed as >=
- **Lesser than** is obtained using < and equivalently, <= is **lesser than or equal than**.
    - When comparing strings it is important to remember that in SQL, uppercase and lowercase strings are equivalent.
- The **logical and** is written as && and we can also use the word **AND**
- The **logical or** is expressed as **OR** or ||
- To obtain a value between a lower and upper range we can use **BETWEEN**

```
SELECT * FROM <table_name>
WHERE <column_name> BETWEEN <lower_bound> AND <upper_bound>
```

  - We can also use **NOT BETWEEN**, which does the exact opposite.
  - When using **BETWEEN** for date values it is important to use **CAST** to ensure correct results.

```
SELECT CAST(<original_value> AS <type>);
```

- To find if a column is in a list of values we can use the operator **IN**

```
SELECT * FROM books
WHERE <column_name> IN (<value>, <value>);
```

  - We can make use of **NOT IN** for the exact opposite.
- To calculate the modulo between two numbers we can use %

- We can stand in different cases with the operator **CASE**. Example:

```
SELECT title, released_year,
   CASE
       WHEN released_year >= 2000 THEN 'Modern Lit'
       ELSE '20th Century Lit'
   END AS 'genre'
FROM books;
```

- To use multiple cases we can change the example as

```
SELECT
   title,
   released_year AS 'year',
   CASE
       WHEN stock_quantity <= 50 THEN '*'
       WHEN stock_quantity <= 100 THEN '**'
       ELSE '***'
   END AS 'stock'
FROM books;
```

Section 12: One To Many

- We are now going to start with messy and interrelated data. This is usually stored in multiple tables.
- Relationship types between tables in a database:
    - One to One
    - One to Many
    - Many to Many
- We are going to work with an example of two tables: **customers** and **orders**
- To implement a One to Many relationship we have to make use of **FOREIGN KEY**.

```
CREATE TABLE customers(
   id INT AUTO_INCREMENT PRIMARY KEY,
   first_name VARCHAR(100),
   last_name VARCHAR(100),
   email VARCHAR(255)
);

CREATE TABLE orders(
   id INT AUTO_INCREMENT PRIMARY KEY,
   order_date DATETIME,
   amount DECIMAL(8,2),
   customer_id INT,
   FOREIGN KEY(customer_id) REFERENCES customers(id)
);
```

- If we make the following call:

```
SELECT * FROM customers, orders;
```

SQL will make a cartesian product between every row of the two tables. This is called a **cross join**

- If we select both elements of both tables with a where we can make an **implicit inner join**

```
SELECT customers.first_name, customers.last_name, orders.order_date,
orders.amount
FROM customers, orders
WHERE customers.id = orders.customer_id;
```

- Which can also be accomplished with an **explicit inner join**

```
SELECT customers.first_name, customers.last_name, orders.order_date,
orders.amount FROM customers
JOIN orders
   ON customers.id = orders.customer_id;
```

- We can use **LEFT JOIN** to obtain the table in the left joined with the table in the right. The main difference with **INNER JOIN** is that when there is missing data in the right table, it will append **NULL** data, while in **INNER JOIN** it does nothing and that value does not appear. As an example we have

```
SELECT customers.first_name,
   customers.last_name,
   IFNULL(SUM(orders.amount), 0)
FROM customers
LEFT JOIN orders
   ON customers.id = orders.customer_id
GROUP BY customers.id
ORDER BY SUM(orders.amount) DESC;
```

  where we have also used **IFNULL**, which checks if the first value is **NULL**, if it is, then it returns the second value, which in this case is 0.
- **RIGHT JOIN** does the same thing but on the other way around. As an example we have that

- To delete a parent key, we need to include **ON DELETE CASCADE**