GENERAL ASSEMBLY

Intro to SQL

Database Design + ERD

Basic SQL

BASIC SQL

- SQL is used to communicate questions to the database.
- The two main clauses are **`SELECT`**, and **`FROM`**.

`SELECT`

- Allows you to select certain columns from a table.

`FROM`

- Specifies the tables from which the query extracts data.

BASIC SQL

```
SELECT * FROM restaurants;
```

Translation: Show me all the columns (*) from the restaurants table.

BASIC SQL

We can also get a subset of the columns from a given table.

```
SELECT name, calories FROM foods;
```

Translation: Show me the **name** and **calories** columns from the **foods** table

BASIC SQL

Sometimes you see the columns prefixed by their corresponding table. This is overkill when you're just querying, one table, but becomes important once you query from multiple tables.

```
SELECT foods.name, foods.calories
FROM foods;
```

Translation: Show me the **name** and **calories** columns from the **foods** table

You can also namespace a wildcard.

```
SELECT foods.* FROM foods;
```

Translation: Give me every column from the foods table

BASIC SQL

Writing out the same table can get pretty cumbersome. Thankfully we can give our tables a temporary (and hopefully shorter) name. This is called namespacing.

```
SELECT f.name, f.calories
FROM foods f;
```

Translation: Show me the **name** and **calories** columns from the **foods** table

BASIC SQL

Recall that the foods, restaurants and categories tables all have a **name** column. When we start combining tables into a single query, we might want to give each name column an alias as well.

```
SELECT f.name as food, f.calories
FROM foods f;
```

Translation: Show me the **name** (temporarily renamed to **food**) and **calories** columns from the **foods** table.

BASIC SQL

We can also use DISTINCT to remove any duplicates.

```
SELECT DISTINCT f.name as food
FROM foods f;
```

Translation: Show me the unique names (renamed to “food”) from the foods table.

ORDER BY

BASIC SQL

Sometimes it makes sense to order your query on a certain column. For example, we might want to see our foods by most calories to least.

```
SELECT f.name as food, f.calories
FROM foods f
ORDER BY f.calories DESC;
```

Translation: Show me the **name** (temporarily renamed to **food**) and **calories** columns from the **foods** table, from most caloric to least.

BASIC SQL

You can also order by ascending (increasing) order.

```
SELECT r.name
FROM restaurants r
ORDER BY r.name ASC;
```

Translation: Give me all the restaurants' names in alphabetical order.

You can order on multiple columns. Priority is given to the first column to the nth.

```
SELECT f.restaurant_id AS rid, f.name,
f.calories
FROM foods f
ORDER BY rid ASC, f.calories DESC;
```

Translation: Give me the restaurant id (renamed to rid), name and calories from foods. Order first by restaurant id from smallest to biggest, then by calories from biggest to smallest.

LIMIT

BASIC SQL

Rather than returning ALL rows from a given table, you might only want a subset. This can be achieved with the LIMIT command.

```
SELECT f.name, f.calories
FROM foods f
ORDER BY f.calories DESC
LIMIT 20;
```

Translation: Give me the name and calories of the 20 most caloric items from the foods table.

Filtering with WHERE

BASIC SQL

One of the more important skills in SQL is the ability to filter your queries that meet a certain condition. This is accomplished with the WHERE command.

```
SELECT f.name, f.calories
FROM foods f
WHERE f.calories > 1000;
```

Translation: Give me the name and calories of foods with more than 1,000 calories.

BASIC SQL

Numerical filters can be achieved with the following commands:

- Greater than: >
- Greater than or equal to: >=
- Less than: <
- Less than or equal to: <=
- Equal to: =

BASIC SQL

We can combine multiple conditions into one query using **AND**.

```
SELECT f.name, f.calories
FROM foods f
WHERE f.calories > 1000 AND f.carbs > 30;
```

Translation: Give me the names, calories and carbs from all foods over 1,000 calories and over 30g of carbs.

BASIC SQL

We can also combine multiple conditions into one query using **OR**, which is similar to **AND** but more flexible.

```
SELECT f.name, f.calories
FROM foods f
WHERE f.calories > 1000 OR f.carbs > 30;
```

Translation: Give me the names, calories and carbs from all foods over 1,000 calories **or** over 30g of carbs.

BASIC SQL

You can also filter numerically for results that fall within a given range.

```
SELECT f.name, f.calories
FROM foods f
WHERE f.calories BETWEEN 0 AND 10;
```

Translation: Give me the name and calories of foods that are between 0 and 10 calories (inclusive).

BASIC SQL

Now let's transition to filtering by name.

```
SELECT * FROM restaurants
WHERE restaurants.name = 'McDonald''s';
```

Translation: Give me every column from the restaurants whose name is McDonald's.

NOTE: This example shows how to handle apostrophes.

IN is used for finding multiple matches:

```
SELECT *
FROM categories c
WHERE c.name IN ('Burgers', 'Sandwiches')
```

Translation: Give me everything from the categories table whose name is in the following list: Burgers, Sandwiches.

BASIC SQL

Try getting information on the Whopper from the foods table.

```
SELECT * FROM foods
WHERE foods.name = 'Whopper';
```

Translation: Give me every column from the foods table whose name exactly matches Whopper.

What is wrong with this query?

BASIC SQL

There are 20 Whoppers in the database. The problem is our query is exact. We're looking for foods whose name is exactly Whopper, thus excluding items like the Texas Triple Whopper Sandwich. We can use LIKE to broaden our search. Try the following queries:

```
SELECT * FROM foods
WHERE foods.name LIKE 'Whopper%';
```

```
SELECT * FROM foods
WHERE foods.name LIKE '%Whopper';
```

BASIC SQL

Can you infer what the % is doing in the query? Now try this one:

```
SELECT * FROM foods
WHERE foods.name LIKE '%Whopper%';
```

Translation: Give me everything from the foods table, where the word Whopper occurs somewhere in the name.

The % acts as a wild card.

BASIC SQL

Now try this:

```
SELECT * FROM foods
WHERE foods.name LIKE '%whopper%';
```

Translation: Give me everything from the foods table, where the word whopper (lowercase w) occurs somewhere in the name.

Note that LIKE is case sensitive; capitalization matters.

BASIC SQL

If you wanted your search to be case insensitive, use the following:

```
SELECT * FROM foods
WHERE foods.name ILIKE '%whopper%';
```

Translation: Give me everything from the foods table, where the word whopper (irrespective of capitalization) occurs somewhere in the name.

BASIC SQL

You can also use the inverse of LIKE.

```
SELECT * FROM foods
WHERE foods.name NOT LIKE '%Whopper%';
```

Translation: Give me everything from the foods table that **does not** have Whopper **anywhere** in the name.

You can also look to see if there are empty values.

```
SELECT *
FROM foods f
WHERE f.name IS NULL;
```

Translation: Give me everything from the foods table that **does not** have a name.

BASIC SQL

You can also look to see if there is a value of some kind, present.

```
SELECT *
FROM foods f
WHERE f.name IS NOT NULL;
```

Translation: Give me everything from the foods table where the name is not null.

Joining

BASIC SQL

It's very common to want to combine information from multiple tables into one query. For example, we might want to run a query returning all food items with their associated restaurant. We can do this by joining. There are several types of joins:

- Inner join
- Left/Right join
- Left/Right outer join
- Unions

The most common join is the inner join.

BASIC SQL

Let's try a few examples:

```
SELECT f.name, r.name AS restaurant
FROM foods f
INNER JOIN restaurants r ON r.id =
f.restaurant_id;
```

Translation: Give me the names of every food item with their associated restaurant.

BASIC SQL

Let's try a few examples:

```
SELECT f.name
FROM foods f
INNER JOIN restaurants r ON r.id =
f.restaurant_id
WHERE r.name = 'Burger King';
```

Translation: Give me the names of every food item from Burger King.

BASIC SQL

Let's try a few examples:

```
SELECT f.name, c.name as category
FROM foods f
INNER JOIN categories_foods cf ON
cf.food_id = f.id
INNER JOIN categories c ON c.id =
cf.category_id;
```

Translation: Give me the names of every food item with their associated category.

BASIC SQL

```
SELECT f.name FROM foods f
INNER JOIN categories_foods cf ON
cf.food_id = f.id
INNER JOIN categories c ON c.id =
cf.category_id
WHERE c.name = 'Desserts';
```

Translation: Give me the names of every food item in the Desserts category.

Aggregating

BASIC SQL

Sometimes we might want to reduce our query to a single value. For example, we may want to know how many foods our in our database:

```
SELECT COUNT(f.id) FROM foods f;
```

Translation: How many rows are in my foods table?

GUIDED PRACTICE

The **COUNT** in the previous query is what's known as an aggregate function. The most common aggregate functions are:

- COUNT
- AVG
- MIN
- MAX
- SUM

Why are min and max considered aggregate functions?

BASIC SQL

Often we'll want to **group** our data into buckets and then run some sort of aggregate function. For example:

```
SELECT r.name, AVG(f.calories)
FROM foods f
INNER JOIN restaurants r ON r.id =
f.restaurant_id
GROUP BY r.name;
```

Translation: What is the average number of calories **per restaurant**?

BASIC SQL

NOTE: Your non-aggregating columns (in this case restaurant id and name) need to **all** be included in the **GROUP BY**

```
SELECT r.id, r.name, AVG(f.calories)
FROM foods f
INNER JOIN restaurants r ON r.id =
f.restaurant_id
GROUP BY r.id, r.name;
```

Translation: Give me the id, name and average calories per restaurant.

BASIC SQL

Sometimes you might want to use the result of an aggregate function as a filter. We can do this with **HAVING**, which is like **WHERE** but for aggregates:

```
SELECT r.name, AVG(f.calories)
FROM restaurants r
INNER JOIN foods f ON f.restaurant_id = r.id
GROUP BY r.name
HAVING AVG(f.calories) > 700;
```

Translation: Give me the name and average calories for all restaurants who average more than 700 calories per menu item.