# AGGREGATE FUNCTIONS

## Introduction

We've learned how to write queries to retrieve information from the database. Now, we are going to learn how to perform calculations using SQL.

Calculations performed on multiple rows of a table are called **aggregates**.

In this lesson, we have given you a table named `fake_apps` which is made up of fake mobile applications data.

Here is a quick preview of some important aggregates that we will cover in the next five exercises:

- `COUNT()`: count the number of rows
- `SUM()`: the sum of the values in a column
- `MAX()`/`MIN()`: the largest/smallest value
- `AVG()`: the average of the values in a column
- `ROUND()`: round the values in the column

Let's get started!

1. Before getting started, take a look at the data in the `fake_apps` table. In the code editor, type the following:

```
SELECT *
FROM fake_apps;
```

What are the column names?

## Count

The fastest way to calculate how many rows are in a table is to use the `COUNT()` function. `COUNT()` is a function that takes the name of a column as an argument and counts the number of non-empty values in that column.

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

Here, we want to count every row, so we pass `*` as an argument inside the parenthesis.

1. Let's count how many apps are in the table. In the code editor, run:

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

**2.** Add a `WHERE` clause in the previous query to count how many *free* apps are in the table.

```
SELECT COUNT(*)
FROM fake_apps
WHERE price  = 0.0;
```

**Sum**

SQL makes it easy to add all values in a particular column using `SUM()`. `SUM()` is a function that takes the name of a column as an argument and returns the sum of all the values in that column. What is the total number of downloads for all of the apps combined?

```
SELECT SUM(downloads)
FROM fake_apps;
```

This adds all values in the `downloads` column.

**1.** Let's find out the answer! In the code editor, type:

```
SELECT SUM(downloads)
FROM fake_apps;
```

**Max / Min**

The `MAX()` and `MIN()` functions return the highest and lowest values in a column, respectively. How many downloads does the most popular app have?

```
SELECT MAX(downloads)
FROM fake_apps;
```

The most popular app has 31,090 downloads! `MAX()` takes the name of a column as an argument and returns the largest value in that column. Here, we returned the largest value in the `downloads` column. `MIN()` works the same way but it does the exact opposite; it returns the smallest value.

**1.** What is the least number of times an app has been downloaded? In the code editor, type:

```
SELECT MIN(downloads)
FROM fake_apps;
```

**2.** Delete the previous query. Write a new query that returns the price of the most expensive app.

```
SELECT MAX(price)
FROM fake_apps;
```

**Average**

SQL uses the `AVG()` function to quickly calculate the average value of a particular column. The statement below returns the average number of downloads for an app in our database:

```
SELECT AVG(downloads)
FROM fake_apps;
```

The `AVG()` function works by taking a column name as an argument and returns the average value for that column.

**1.** Calculate the average number of downloads for all the apps in the table.

```
SELECT AVG(downloads)
FROM fake_apps;
```

**3.** Remove the previous query. Write a new query that calculates the average price for all the apps in the table.

```
SELECT AVG(price)
FROM fake_apps;
```

**Round**

By default, SQL tries to be as precise as possible without rounding. We can make the result table easier to read using the `ROUND()` function. `ROUND()` function takes two arguments inside the parenthesis:

1. a column name
2. an integer

It rounds the values in the column to the number of decimal places specified by the integer.

```
SELECT ROUND(price, 0)
FROM fake_apps;
```

Here, we pass the column `price` and integer `0` as arguments. SQL rounds the values in the column to 0 decimal places in the output.

**1.** Let's return the `name` column and a rounded `price` column.

```
SELECT name, ROUND(price, 0)
FROM fake_apps;
```

**2.** Remove the previous query. In the last exercise, we were able to get the average price of an app ($2.02365) using this query:

```
SELECT AVG(price)
FROM fake_apps;
```

Now, let's edit this query so that it rounds this result to 2 decimal places. This is a tricky one!

```
SELECT ROUND (AVG(price), 2)
FROM fake_apps;
```

**Group By I**

Oftentimes, we will want to calculate an aggregate for data with certain characteristics. For instance, we might want to know the mean IMDb ratings for all movies each year. We could calculate each number by a series of queries with different `WHERE` statements, like so:

```
SELECT AVG(imdb_rating)
FROM movies
WHERE year = 1999;

SELECT AVG(imdb_rating)
FROM movies
WHERE year = 2000;

SELECT AVG(imdb_rating)
FROM movies
WHERE year = 2001;
```

and so on.
Luckily, there's a better way! We can use `GROUP BY` to do this in a single step:

```
SELECT year,
   AVG(imdb_rating)
FROM movies
GROUP BY year
ORDER BY year;
```

`GROUP BY` is a clause in SQL that is used with aggregate functions. It is used in collaboration with the `SELECT` statement to arrange identical data into *groups*. The `GROUP BY` statement comes after any `WHERE` statements, but before `ORDER BY` or `LIMIT`.

**1.** In the code editor, type:

```
SELECT price, COUNT(*)
FROM fake_apps
GROUP BY price;
```

Here, our aggregate function is `COUNT()` and we arranged `price` into groups. What do you expect the result to be?

3. In the previous query, add a `WHERE` clause to count the total number of apps that have been downloaded more than 20,000 times, at each price.

```
SELECT price, COUNT(*)
FROM fake_apps
WHERE downloads > 20000
GROUP BY price;
```

4. Remove the previous query. Write a new query that calculates the total number of downloads for each category.

```
Select category and SUM(downloads).
SELECT category, SUM(downloads)
FROM fake_apps
GROUP BY category;
```

**Group By II**

Sometimes, we want to `GROUP BY` a calculation done on a column. For instance, we might want to know how many movies have IMDb ratings that round to 1, 2, 3, 4, 5. We could do this using the following syntax:

```
SELECT ROUND(imdb_rating),
   COUNT(name)
FROM movies
GROUP BY ROUND(imdb_rating)
ORDER BY ROUND(imdb_rating);
```

However, this query may be time-consuming to write and more prone to error. SQL lets us use column reference(s) in our `GROUP BY` that will make our lives easier.

- `1` is the first column selected
- `2` is the second column selected
- `3` is the third column selected

and so on.

The following query is equivalent to the one above:

```
SELECT ROUND(imdb_rating),
   COUNT(name)
FROM movies
GROUP BY 1
ORDER BY 1;
```

Here, the `1` refers to the first column in our `SELECT` statement, `ROUND(imdb_rating)`.

1. Suppose we have the query below:

```
SELECT category,
   price,
   AVG(downloads)
FROM fake_apps
GROUP BY category, price;
```

Write the exact query, but use column reference numbers instead of column names after `GROUP BY`.

```
SELECT category,
   price,
   AVG(downloads)
FROM fake_apps
GROUP BY 1, 2;
```

**Having**

In addition to being able to group data using `GROUP BY`, SQL also allows you to filter which groups to include and which to exclude. For instance, imagine that we want to see how many movies of different genres were produced each year, but we only care about years and genres with at least 10 movies. We can't use `WHERE` here because we don't want to filter the rows; we want to *filter groups*.
This is where `HAVING` comes in.

`HAVING` is very similar to `WHERE`. In fact, all types of `WHERE` clauses you learned about thus far can be used with `HAVING`. We can use the following for the problem:

```
SELECT year,
   genre,
   COUNT(name)
FROM movies
GROUP BY 1, 2
HAVING COUNT(name) > 10;
```

- When we want to limit the results of a query based on values of the individual rows, use `WHERE`.
- When we want to limit the results of a query based on an aggregate property, use `HAVING`.

`HAVING` statement always comes after `GROUP BY`, but before `ORDER BY` and `LIMIT`.

1. Suppose we have the query below:

```
SELECT price,
   ROUND(AVG(downloads)),
   COUNT(*)
FROM fake_apps
GROUP BY price;
```

It returns the average downloads (rounded) and the number of apps – at each price point.
However, certain price points don't have very many apps, so their average downloads are less meaningful.

Add a `HAVING` clause to restrict the query to price points that have more than 10 apps.

```
SELECT price, ROUND(AVG(downloads)), COUNT(*)
FROM fake_apps
GROUP BY price
HAVING COUNT(*) > 10;
```

**Review**

Congratulations!

You just learned how to use aggregate functions to perform calculations on your data. What can we generalize so far?

- `COUNT()`: count the number of rows
- `SUM()`: the sum of the values in a column
- `MAX()`/`MIN()`: the largest/smallest value
- `AVG()`: the average of the values in a column
- `ROUND()`: round the values in the column

*Aggregate functions* combine multiple rows together to form a single value of more meaningful information.

- `GROUP BY` is a clause used with aggregate functions to combine data from one or more columns.
- `HAVING` limit the results of a query based on an aggregate property.

Download the Aggregate Functions: Cheat Sheet to help you remember the content covered in this lesson.