**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!

**Instructions**

**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?

Hint

The column names are id, name, category, downloads, and price.

**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.

**Instructions**

**1.**

Let’s count how many apps are in the table.

In the code editor, run:

SELECT COUNT(\*)

FROM fake\_apps;

Hint

There are 200 apps.

Common errors:

* Missing parenthesis.
* Missing ;.

**2.**

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

Hint

Remember the WHERE statement?

The following code should go inside the previous query, before the semicolon:

SELECT COUNT(\*)

FROM fake\_apps

WHERE price = 0;

* WHERE indicates we want to only include rows where the following condition is true.
* price = 0 is the condition.

There are 73 free apps in the table.

# 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.

**Instructions**

**1.**

Let’s find out the answer!

In the code editor, type:

SELECT SUM(downloads)

FROM fake\_apps;

Hint

There are 3,322,760 total downloads.

# 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.

**Instructions**

**1.**

What is the least number of times an app has been downloaded?

In the code editor, type:

SELECT MIN(downloads)

FROM fake\_apps;

Hint

1,387 downloads.

**2.**

Delete the previous query.

Write a new query that returns the price of the most expensive app.

Hint

SELECT MAX(price)

FROM fake\_apps;

$14.99 is the price of the most expensive app.

# 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.

**Instructions**

**1.**

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

In the code editor, type:

SELECT AVG(downloads)

FROM fake\_apps;

Hint

16,613.8 average downloads.

**2.**

Remove the previous query.

Write a new query that calculates the average price for all the apps in the table.

Hint

Which column should go inside the parenthesis?

SELECT AVG(\_\_\_\_\_)

FROM fake\_apps;

The average price is $2.02365.

**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.

**Instructions**

**1.**

Let’s return the name column and a rounded price column.

In the code editor, type:

SELECT name, ROUND(price, 0)

FROM fake\_apps;

Hint

We are selecting ROUND(price, 0) as the second column in this query.

**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!

Hint

You can treat AVG(price) just like any other value and place it inside the ROUND function like so:

ROUND(AVG(price), 2)

Here, AVG(price) is the 1st argument and 2 is the 2nd argument because we want to round it to two decimal places:

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.

**Instructions**

**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?

Hint

The result contains the total number of apps for each price.

It is organized into two columns, making it very easy to see the number of apps at each price.

**2.**

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.

Hint

Remember, WHERE statement goes before the GROUP BY statement:

SELECT price, COUNT(\*)

FROM fake\_apps

WHERE downloads > 20000

GROUP BY price;

**3.**

Remove the previous query.

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

Select category and SUM(downloads).

Hint

First, select the two columns we want:

SELECT category, SUM(downloads)

FROM fake\_apps;

Next, group the result for each category by adding a GROUP BY:

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).

**Instructions**

**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.

Hint

These numbers represent the selected columns:

* 1 refers to category.
* 2 refers to price.
* 3 refers to AVG(downloads)

Now, change the GROUP BY with numbers:

SELECT category,

price,

AVG(downloads)

FROM fake\_apps

GROUP BY 1, 2;

Note: Even if you use column names instead of numbers, it will still be correct because these two queries are exactly the same!

**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.

**Instructions**

**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.

Hint

The total number of apps at each price point would be given by COUNT(\*).

SELECT price,

ROUND(AVG(downloads)),

COUNT(\*)

FROM fake\_apps

GROUP BY price

HAVING COUNT(\*) > 10;

COUNT(\*) > 10 is the condition.

Because the condition has an aggregate function in it, we have to use HAVING instead of WHERE.

**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.

**Instructions**

Feel free to experiment a bit more with the fake\_apps table before moving on!

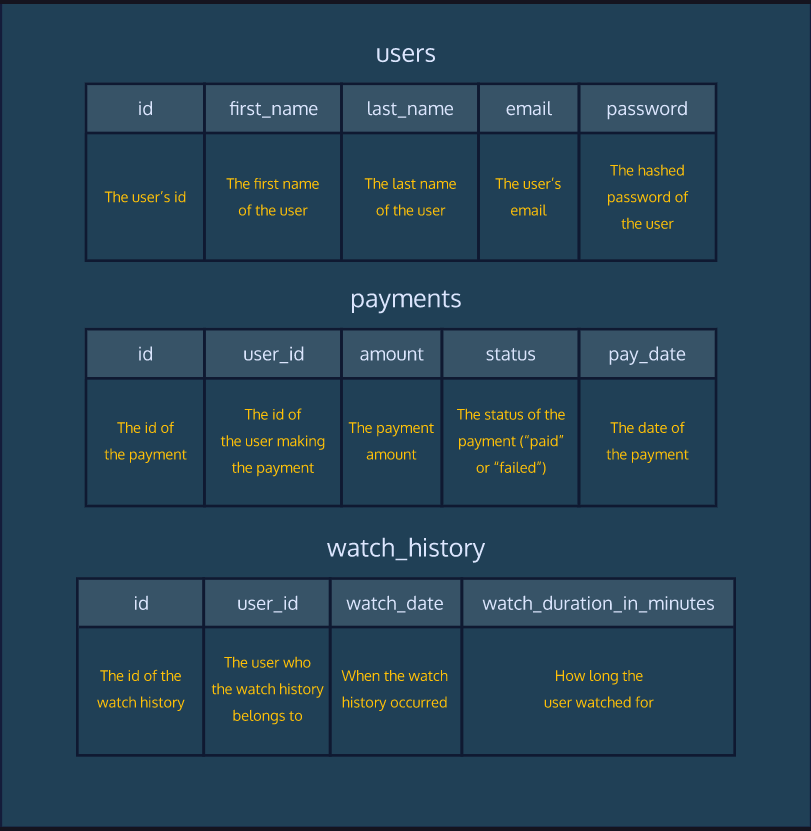
# Codeflix Introduction

Welcome to Code Challenge: Aggregate Functions!

A streaming video company, Codeflix, needs your help analyzing their user data. We’ve imported a portion of their dataset into the SQL workspace for this lesson.

The database contains 3 tables:

* users contains the basic account details for each user
* payments contains payment details for a 3 month period
* watch\_history contains watch events for the users



**Code Challenge 1**

The users table has the following columns:

* id
* first\_name
* last\_name
* email
* password

Click [here](https://s3.amazonaws.com/codecademy-content/courses/learn-sql-aggregates-code-challenge/user-payments-history.png) for the table diagram.

**Instructions**

**1.**

Use COUNT() and a LIKE operator to determine the number of users that have an email ending in ‘.com’.

Hint

As an example,

SELECT \*

FROM users

WHERE first\_name LIKE '%ny';

will give you all the users whose names end in ny.

Using COUNT(\*) instead of \* will return the number of rows that matches the criteria.

SELECT count(\*)

FROM users

WHERE email LIKE '%.com';

**Code Challenge 2**

The users table has the following columns:

* id
* first\_name
* last\_name
* email
* password

Click [here](https://s3.amazonaws.com/codecademy-content/courses/learn-sql-aggregates-code-challenge/user-payments-history.png) for the table diagram.

**Instructions**

**1.**

What are the most popular first names on Codeflix?

Use COUNT(), GROUP BY, and ORDER BY to create a list of first names and occurrences within the users table.

Order the data so that the most popular names are displayed first.

Hint

When you select the count of users, make sure to rename it:

SELECT first\_name, COUNT(\*) AS 'count'

FROM users

GROUP BY first\_name

ORDER BY 2 DESC;

This way, after you GROUP BY first\_name, you can use ORDER BY to sort the data by the count variable.

Also, because we want to display the most popular names first, DESC is needed!

**Code Challenge 3**

The watch\_history table has the following columns:

* id
* user\_id
* watch\_date
* watch\_duration\_in\_minutes

Click [here](https://s3.amazonaws.com/codecademy-content/courses/learn-sql-aggregates-code-challenge/user-payments-history.png) for the table diagram.

**Instructions**

**1.**

The UX Research team wants to see a distribution of watch durations. They want the result to contain:

* duration, which is the watch event duration, rounded to the closest minute
* count, the number of watch events falling into this “bucket”

Your result should look like:

| **duration** | **count** |
| --- | --- |
| 1.0 | 9 |
| 2.0 | 21 |
| 3.0 | 19 |
| … | … |
|  |  |

Use COUNT(), GROUP BY, and ORDER BY to create this result and order this data by increasing duration.

Hint

Use ROUND() to create a column which is the watch duration rounded to the nearest minute. And return a second column that has the number of watch events falling into each duration:

SELECT ROUND(watch\_duration\_in\_minutes) AS 'duration',

COUNT(\*) AS 'count'

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ;

You can then GROUP BY and ORDER BY this variable.

SELECT

  ROUND(watch\_duration\_in\_minutes,0) as duration,

  COUNT(\*) as count

FROM watch\_history

GROUP BY duration

ORDER BY duration ASC;

**Code Challenge 4**

The payments table has the following columns:

* id
* user\_id
* amount
* status
* pay\_date

Click [here](https://s3.amazonaws.com/codecademy-content/courses/learn-sql-aggregates-code-challenge/user-payments-history.png) for the table diagram.

**Instructions**

**1.**

Find all the users that have successfully made a payment to Codeflix and find their total amount paid.

Sort them by their total payments (from high to low).

Use SUM(), WHERE, GROUP BY, and ORDER BY.

Hint

Group the users and calculate their total payments by using SUM().

Filter the successful payments by using WHERE with the condition status = 'paid'.

Sort the total payments in a descending order using ORDER BY and DESC.

SELECT user\_id, SUM(amount)

FROM payments

WHERE status = 'paid'

GROUP BY user\_id

ORDER BY SUM(amount) DESC;

You can also rename the SUM(amount) column:

SELECT user\_id, SUM(amount) AS 'total'

FROM payments

WHERE status = 'paid'

GROUP BY user\_id

ORDER BY total DESC;

And add column reference numbers:

SELECT user\_id, SUM(amount) AS 'total'

FROM payments

WHERE status = 'paid'

GROUP BY 1

ORDER BY 2 DESC;

# Code Challenge 5

The watch\_history table has the following columns:

* id
* user\_id
* watch\_date
* watch\_duration\_in\_minutes

Click [here](https://s3.amazonaws.com/codecademy-content/courses/learn-sql-aggregates-code-challenge/user-payments-history.png) for the table diagram.

**Instructions**

**1.**Generate a table of user ids and total watch duration for users who watched more than 400 minutes of content.

Use SUM(), GROUP BY, and HAVING to achieve this.

Hint

Use SUM() and GROUP BY to calculate a total amount for each user\_id.

The HAVING allows us to filter the result to users that have a total amount greater than 400:

SELECT user\_id,

SUM(watch\_duration\_in\_minutes)

FROM watch\_history

GROUP BY user\_id

HAVING SUM(watch\_duration\_in\_minutes) > 400;

If you give the total amount an alias, you can use it in HAVING:

SELECT user\_id,

SUM(watch\_duration\_in\_minutes) AS 'total\_duration'

FROM watch\_history

GROUP BY 1

HAVING total\_duration > 400;

# Code Challenge 6

The watch\_history table has the following columns:

* id
* user\_id
* watch\_date
* watch\_duration\_in\_minutes

Click [here](https://s3.amazonaws.com/codecademy-content/courses/learn-sql-aggregates-code-challenge/user-payments-history.png) for the table diagram.

**Instructions**

**1.**

To the nearest minute, how many minutes of content were streamed on Codeflix?

Hint

Use the SUM() and ROUND() functions with the watch\_history table.

You can treat SUM(watch\_duration\_in\_minutes) just like any other value and place it inside the ROUND() function like so:

ROUND(SUM(watch\_duration\_in\_minutes), 0)

Here, SUM(watch\_duration\_in\_minutes) is the 1st argument and 0 is the 2nd argument because we want to round it to zero decimal places:

SELECT ROUND(SUM(watch\_duration\_in\_minutes), 0)

FROM watch\_history;

# Code Challenge 7

The payments table has the following columns:

* id
* user\_id
* amount
* status
* pay\_date

Click [here](https://s3.amazonaws.com/codecademy-content/courses/learn-sql-aggregates-code-challenge/user-payments-history.png) for the table diagram.

**Instructions**

**1.**

Which days in this period did Codeflix collect the most money?

Your result should have two columns, pay\_date and total amount, sorted by the latter descending.

This should only include successful payments (status = 'paid').

Use SUM(), GROUP BY, and ORDER BY.

Hint

Use SUM() to calculate the total amount paid each day and rename it to ‘total’ using AS.

Be sure to GROUP BY the day and ORDER BY the total in a decreasing order.

SELECT pay\_date, SUM(amount)

FROM payments

WHERE status = 'paid'

GROUP BY pay\_date

ORDER BY SUM(amount) DESC;

You can also rename pay\_date → day and SUM(amount) → total using AS:

SELECT pay\_date AS 'day',

SUM(amount) AS 'total'

FROM payments

WHERE status = 'paid'

GROUP BY day

ORDER BY total DESC;

# Code Challenge 8

The payments table has the following columns:

* id
* user\_id
* amount
* status
* pay\_date

Click [here](https://s3.amazonaws.com/codecademy-content/courses/learn-sql-aggregates-code-challenge/user-payments-history.png) for the table diagram.

**Instructions**

**1.**

When users successfully pay Codeflix (status = 'paid'), what is the average payment amount?

Hint

Use the AVG() function after selecting the correct rows from payments:

SELECT AVG(\_\_\_\_\_\_)

FROM payments

WHERE status = 'paid';

You don’t need GROUP BY, ORDER BY, and HAVING for this query!

# Code Challenge 9

The watch\_history table has the following columns:

* id
* user\_id
* watch\_date
* watch\_duration\_in\_minutes

Click [here](https://s3.amazonaws.com/codecademy-content/courses/learn-sql-aggregates-code-challenge/user-payments-history.png) for the table diagram.

**Instructions**

**1.**

Of all the events in the watch\_history table, what is the duration of the longest individual watch event? What is the duration of the shortest?

Use one query and rename the results to max and min.

Hint

Use the MAX() function to find the longest duration event and MIN() to find the shortest.

SELECT MAX(\_\_\_\_\_), MIN(\_\_\_\_\_)

FROM \_\_\_\_\_\_ ;

You can rename the results using AS.

select max(watch\_duration\_in\_minutes) as max,min(watch\_duration\_in\_minutes) as min from watch\_history;