Data Analytics Bootcamp

SQL Fundamentals

WeCloud DATA

Prerequisites and Software Requirements

Technology Requirements

Windows or Mac laptop with an internet connection

Software Requirements

Latest versions of MySQL Community Server and MySQL Workbench

Course Description

- The SQL Fundamentals course is designed for students or professionals who want a better understanding of the most common language used to interact with databases
- This course will assume no prior knowledge of SQL or databases
- As always, our goal is to teach you the 20% essential skills that you'll need to solve 80% of the business scenarios you'll likely face in an analytics-based role.

Learning Outcomes

After completing this course, you will be able to:

- Use SELECT statements to query a single table
- Use the WHERE clause to filter data
- Use the ORDER BY clause to sort data
- Use an INNER JOIN, LEFT/RIGHT JOIN, and FULL JOIN to return columns from multiple tables in the same query
- Use mathematical SQL statements to aggregate data
- Use the GROUP BY clause to group data
- Use the HAVING clause to filter groups of data
- Use the OVER and PARTITION BY clauses to show aggregated values for groups of data

Course Syllabus

Day 1
Introduction to SQL for Data Analysts

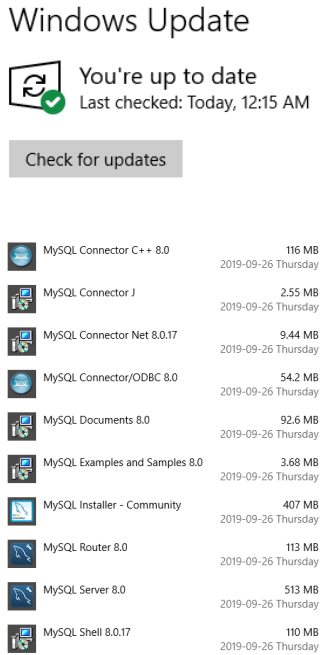
- Install and configure MySQL
- Database Overview
- Use SELECT statement to query data against a single table
- Use WHERE clause to filter data
- Use Boolean expressions in WHERE clause
- Use ORDER BY clause to sort data
- Practice problems using simple SQL queries

Day 2
Intermediate SQL for Data Analysts

- Understand and use the 2 most used joins – INNER JOIN and LEFT JOIN
- Aggregate Functions
- Use GROUP BY clause to group data
- Use HAVING clause to filter groups of data
- Use the OVER and PARTITION BY clauses to append aggregated stats to partitions
- Practice problems using intermediate SQL queries

Windows Install Instructions - MySQL Community Server and Workbench

1. Ensure your Windows laptop is up-to-date (check the status by going into Windows Update settings and ensure all recent patches are installed)



2. Ensure all previous versions of MySQL-related software is uninstalled

3. Once you've uninstalled all previous versions of MySQL, restart your computer.

Windows Install Instructions - MySQL Community Server and Workbench

4. Go to https://dev.mysql.com/downloads/installer/ and select Microsoft Windows from the “Select Operating System” menu.

5. Click to download the larger version of the two MSI Installer files. There’s no need to sign up for an account so don’t click on one of the big buttons. Instead look for “No thanks, just start my download.” and click on that.

6. Find the downloaded file, and double-click to run the installer. Please leave all default settings as they are. If you have any “failing requirements” under the “Check Requirements” install phase, please click “Execute” before clicking “Next”.
   IMPORTANT NOTE When asked to create a root password, please write it down or ensure you remember it. If you forget it, you’ll need to uninstall everything and start the process all over again.

7. Find the “MySQL Workbench” application and click to launch it.

MacOS Install Instructions - MySQL Community Server and Workbench

1. Ensure you have the most recent version (that your current hardware will allow) of MacOS installed.

2. Search for and uninstall any previously installed versions of MySQL and restart your computer.

3. Go to https://dev.mysql.com/downloads/mysql/ and select macOS from the “Select Operating System” menu.

4. Click download on the “DMG Archive” version of the file. It will usually be one of the larger files available for download. There’s no need to sign up for an account so don’t click on one of the big buttons. Instead look for “No thanks, just start my download.” and click on that.

MacOS Install Instructions - MySQL Community Server and Workbench

5. Find the downloaded file, and double-click to run the installer. Please leave all default settings as they are and continue to the end of the process.
   IMPORTANT NOTE When asked to create a root password, please write it down or ensure you remember it. If you forget it, you'll need to uninstall everything and start the process all over again.

6. Go to https://dev.mysql.com/downloads/workbench and select macOS from the "Select Operating System" menu.

7. Click download on the "DMG Archive" version of the file. There's no need to sign up for an account so don't click on one of the big buttons. Instead look for "No thanks, just start my download." and click on that.

8. Find the downloaded file, and drag it into your Applications folder. Double-click to launch the application.

What is a Database?

- A database is simply a container of organized data with multiple tables, columns, and records

- You can think of a database as multiple Excel worksheets (“tables”) in a workbook (“database”) that are organized into rows and columns that contain data in cells (“records”)

What is SQL?

- SQL, which stands for Structured Query Language is a language used to communicate with and retrieve data from a relational database

- SQL is the standard language for most relational database management systems. Although most databases use SQL, many of them have additional proprietary extensions that can only be used with certain types of databases. However, the standard SQL commands used in this course will be applicable with most databases.

Most Common SQL Commands Used by Analysts

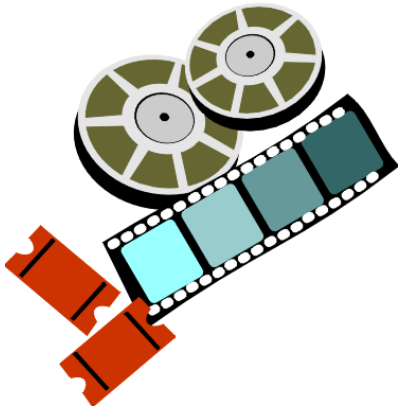
Main SQL Commands:

- SELECT
- WHERE
- ORDER BY
- GROUP BY
- HAVING
- OVER
- PARTITION BY
- INNER JOIN
- LEFT JOIN

SQL Operators:

- AND
- OR
- BETWEEN
- LIKE
- IN
- SUM
- COUNT
- AVG
- MIN/MAX

The WeCloudData Movies Dataset



- Before we jump into the dataset, it's important to understand what type of business we're looking at.
- The WeCloudData movies database is for a traditional brick and mortar DVD rental business
- We're going to pretend that we've just bought this business and using the data in the database, we need to understand everything we can about the business (i.e. the staff, inventory, transactions etc.)

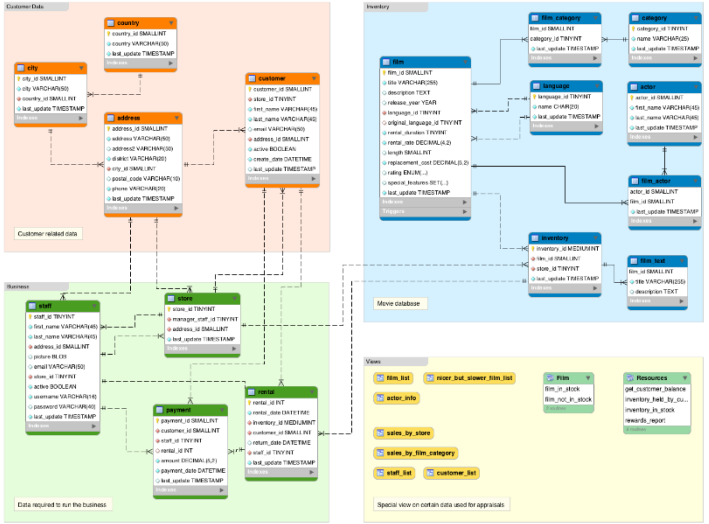
Install Database

- Download the “Create_WeCloudData_Movie_Database.sql” file from Slack
- Click on “File” > “Open SQL Script” and find the downloaded .sql file
- Double-click on the .sql script
- Click on the lightning bolt icon or hit CTRL+SHIFT+ENTER
- Click on “Schemas” on the left hand side (middle of screen) and click on the refresh icon to the left of “SCHEMAS” at the top underneath “Navigator”
- If everything ran correctly, you should see a new database called “weclouddatamovies”

The WeCloudData Movies Database Schema

- A database schema is a graphical representation of the database's structure and the relationship between tables and fields in a database

- A schema allows you to identify not only how many tables and fields you have but how you can potentially join various tables together



Using SELECT and FROM Statements

- The SELECT statement is used to retrieve data from a table within a database. The data is returned in a table structure.

- The SELECT syntax is as follows:

SELECT column(1), column(2), …. ,column(n) AS [Column Alias]
FROM table_name

Note that when you've typed in an recognized SQL command, it will turn blue

- To "SELECT" columns or data from a table, you must specify which table you would like to select "FROM"

- You can also use SELECT DISTINCT to get only unique values, or SELECT * to get all results in a table

Practice Scenarios using SELECT and FROM

1. From the rental table, retrieve all records
2. From the rental table, retrieve only the customer id and rental date columns
3. From the customer table, retrieve the first name, last name, and email
4. From the film table, retrieve only the unique values from the rating column
5. From the film table, retrieve only the unique rental durations

Solutions for SELECT and FROM statements

1. From the rental table, retrieve all records

```
SELECT * FROM rental
```

Solutions for SELECT and FROM statements

2. From the rental table, retrieve only the customer id and rental date columns

```
SELECT
customer_id,
rental_date
FROM rental
```

Solutions for SELECT and FROM statements

3. From the customer table, retrieve the first name, last name, and email

```
SELECT
first_name,
last_name,
Email
FROM customer
```

Solutions for SELECT and FROM statements

4. From the film table, retrieve only the unique values from the rating column

```
SELECT DISTINCT
Rating
FROM film
```

Solutions for SELECT and FROM statements

5. From the film table, retrieve only the unique rental durations

```
SELECT DISTINCT
rental_duration
FROM film
```

Using a WHERE Clause

- The WHERE clause is used to filter rows of data based on a defined criteria

- The WHERE clause is used after the SELECT and FROM operators:

```
SELECT column(1), column(2), …. ,column(n)
FROM table_name
WHERE column(n)='value'
```

- To use the "WHERE" clause, you must specify what value(s) you want to use as a filter

- We can use "=" ">" "<" ">=" "<=" "<>" "between" "in" "like" etc….

Using BOOLEAN expressions within a WHERE clause

- Boolean expressions are logical statements that are evaluated as either TRUE or FALSE

- The most common Boolean expressions used are:
  "AND", "OR","NOT"

- Boolean expressions are used within the WHERE clause to further filter narrow your results:

```
SELECT column(1), column(2), …. ,column(n)
FROM table_name
WHERE column(n)='value x' AND column(1)>='value y'
```

Using an ORDER BY clause

- The ORDER BY clause allows you to sort your data set based on a defined criteria (i.e. column name or column ordinal [position in query])

- The ORDER BY clause is always used at the end of your SQL statement:

```
SELECT column(1), column(2), …. ,column(n)
FROM table_name
WHERE column(n)='value x' AND column(1)>='value y'
ORDER BY column(2) [ASC/DESC]
```

- Note that by default, SQL will always sort data in ascending order based on the first column in the table

Practice Scenarios using WHERE and ORDER BY

1. From the payment table, retrieve the customer id, rental id, amount, payment date and filter for payment dates > '2006-01-01' and order the results by amount in descending order. (182 rows)

2. From the payment table, retrieve the customer id, rental id, amount, and payment date for only those customers that have a customer id of 100 or less. Order by amount in ascending order. (2,711 rows)

3. From the payment table, retrieve the customer id, rental id, amount, and payment date and filter for amounts of only $0.99 and payment dates greater than 2006-01-01. Order by customer id in descending order. (53 rows)

4. From the customer table, retrieve all columns but filter for customers whose first names start with an a. (44 rows)

5. Modify the query above to show customers who do not have an "a" anywhere in the middle of their first name (243 rows)

Practice Scenarios using WHERE and ORDER BY

1. From the payment table, retrieve the customer id, rental id, amount, payment date and filter for payment dates > ‘2006-01-01’ and order the results by amount in descending order. (182 rows)

```
SELECT
customer_id,
rental_id,
amount,
payment_date
FROM Payment
WHERE payment_date>'2006-01-01’
ORDER BY amount DESC
```

Practice Scenarios using WHERE and ORDER BY

2. From the payment table, retrieve the customer id, rental id, amount, and payment date for only those customers that have a customer id of 100 or less. Order by amount in ascending order. (2,711 rows)

```
SELECT
customer_id,
rental_id
amount,
payment_date
FROM payment
WHERE customer_id<=100
ORDER BY amount
```

Practice Scenarios using WHERE and ORDER BY

3. From the payment table, retrieve the customer id, rental id, amount, and payment date and filter for amounts of only $0.99 and payment dates greater than 2006-01-01. Order by customer id in descending order. (53 rows)

```
SELECT
customer_id,
rental_id,
amount,
payment_date
FROM
payment
WHERE
amount=0.99 AND payment_date>'2006-01-01’
ORDER BY
customer_id DESC
```

Practice Scenarios using WHERE and ORDER BY

4. From the customer table, retrieve all columns but filter for customers whose first names start with an a. (44 rows)

```
SELECT *
FROM customer
WHERE first_name like 'a%'
```

Practice Scenarios using WHERE and ORDER BY

5. Modify the query above to show customers who do not have an “a” somewhere in their first name (243 rows)

```
SELECT *
FROM customer
WHERE first_name not like '%a%'
```

Functions to Aggregate Data

- The main aggregation functions in SQL are SUM, AVG, COUNT, MAX, MIN
- If you recall, these are similar to widely used Excel functions and the syntax for the formula is nearly identical
- Using aggregation functions in SQL allows you to use mathematical functions to quickly find answers to your questions
- Aggregation functions also allow you to reduce your result size by choosing only the data that you need

Using the GROUP BY clause

- The GROUP BY clause allows us group column values together using aggregate functions (i.e. how many sales did each salesperson do last year)

- The GROUP BY clause is always used after the WHERE clause but before the ORDER BY clause in your SQL statement:

```
SELECT column(1), SUM(column(2)), …. ,column(n)
FROM table_name
WHERE column(n)='value x' AND column(1)>='value y'
GROUP BY column(1)
ORDER BY column(2) [ASC/DESC]
```

- Note that you must have an aggregation function in your query to use a GROUP BY clause

Practice Scenarios using AGGREGATION FUNCTIONS

1. From the payment table, find the average revenue received to-date by each salesperson (answer: 1= $4.16 / 2=$4.25)

2. From the film table, retrieve the count of movies and group them by rental duration. Apply appropriate aliases to columns where necessary. Sort your results by rental duration in descending order. (5 rows)

3. From the film table, find the # of movies, minimum rental rate, the maximum rental rate, and the average rental rate and group them by the movie rating. Apply appropriate aliases to all columns. (5 rows)

Practice Scenarios using AGGREGATE FUNCTIONS

1. From the payment table, find the average revenue received to-date by each salesperson (answer: 1= $4.16 / 2=$4.25)

```
SELECT
staff_id as 'Staff ID’,
AVG(amount) as 'Average Revenue’
FROM payment
GROUP BY staff_id
```

Practice Scenarios using AGGREGATE FUNCTIONS

2. From the film table, retrieve the count of movies and group them by rental duration. Apply appropriate aliases to columns where necessary. Sort your results by rental duration in descending order. (5 rows)

```
SELECT
rental_duration as 'Rental Duration',
COUNT(film_id) as '# of Movies'
FROM film
GROUP BY rental_duration
ORDER BY rental_duration DESC
```

Practice Scenarios using AGGREGATE FUNCTIONS

3. From the film table, find the # of movies, minimum rental rate, the maximum rental rate, and the average rental rate and group them by the movie rating. Apply appropriate aliases to all columns. (5 rows)

```
SELECT
rating as 'Movie Rating',
COUNT(film_id) as '# of Movies',
MIN(rental_rate) as 'Minimum Rental Rate',
MAX(rental_rate) as 'Maximum Rental Rate',
AVG(rental_rate) as 'Average Rental Rate'
FROM film
GROUP BY rating
```

Using the HAVING clause

- The HAVING clause is very similar to the WHERE clause with the key differentiator being that the WHERE clause filters out individual rows of data based on column values whereas the HAVING clause filters out groups of data based on aggregate functions

- The HAVING clause is always used after the GROUP BY clause but before the ORDER BY clause in your SQL statement:

```
SELECT column(1), SUM(column(2)), …. ,column(n)
FROM table_name
WHERE column(n)='value x' AND column(1)>='value y'
GROUP BY column(1)
HAVING Aggregate Function and Criteria
ORDER BY column(2) [ASC/DESC]
```

Practice Scenarios using the HAVING clause

1. We want to find who our **most loyal/frequent** customers are so from the payment table, retrieve all customer ids that have 25 or more life-to-date rentals. Sort your results by total rentals in descending order. (399 rows)

2. We want to find who our **most profitable** customers are so from the payment table, retrieve all customer ids that have $100 or more worth of rentals. Sort your results in descending order by the total amount spent. (395 rows)

3. Combine the results of the 2 queries above to show only those customers who have 25 or more life-to-date rentals AND have spent $100 or more on rentals. Sort your results in descending order by the total amount spent. (353 rows)

Practice Scenarios using the HAVING clause

1. We want to find who our **most loyal/frequent** customers are so from the payment table, retrieve all customer ids that have 25 or more life-to-date rentals. Sort your results by total rentals in descending order. (399 rows)

```
SELECT
customer_id as 'Customer ID',
COUNT(payment_id) as 'Number of Transactions'
FROM payment
GROUP BY customer_id
HAVING COUNT(payment_id)>=25
ORDER BY COUNT(payment_id) DESC
```

Practice Scenarios using the HAVING clause

2. We want to find who our **most profitable** customers are so from the payment table, retrieve all customer ids that have $100 or more worth of rentals. Sort your results in descending order by the total amount spent. (395 rows)

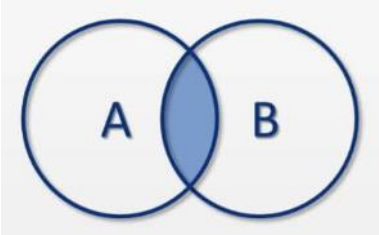
```
SELECT
customer_id as 'Customer ID',
SUM(amount) as 'Total Spent'
FROM payment
GROUP BY customer_id
HAVING SUM(amount)>=100
ORDER BY SUM(amount) DESC
```

Practice Scenarios using the HAVING clause

3. Combine the results of the 2 queries above to show only those customers who have 25 or more life-to-date rentals AND have spent $100 or more on rentals. Sort your results in descending order by the total amount spent. (353 rows)

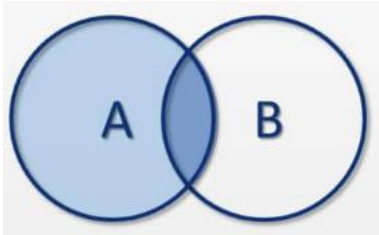
```
SELECT
customer_id as 'Customer ID',
COUNT(payment_id) as 'Number of Transactions',
SUM(amount) as 'Total Spent'
FROM payment
GROUP BY customer_id
HAVING COUNT(payment_id)>=25 AND SUM(amount)>=100
ORDER BY SUM(amount) DESC
```

The 2 Most Used SQL JOINS



INNER JOIN

- The INNER JOIN clause selects records that have matching values in both tables



LEFT JOIN

- The LEFT JOIN clause returns all records from the left table (A), and the matched records from the right table (B)

JOIN Clauses and Primary vs. Foreign Keys

- A JOIN clause is used to combine rows from 2 or more tables in a single query using a common column (Primary/Secondary keys) between the multiple tables
- A relational database will typically contain a Primary Key and a Foreign Key in each column. Sometimes, a column may be both a Primary Key and a Foreign Key
- A Primary Key is one or more columns that is used to uniquely identify each row in the table. These values must be unique and cannot contain duplicates or NULL values
- A Foreign Key is one or more columns in a table that refers to the primary key in another table. These values can be duplicated and can contain NULL values

Practice Scenarios using JOINS

1. Retrieve all distinct titles and their descriptions from store 2. (you'll need to use the film and inventory tables) (762 rows)

2. From the film table and the film_actor tables, find the title and how many actors acted in each movie. (Note – you will need to use a join, aggregate function and group by clause). Sort by highest number of actors to lowest in descending order. (Top Movie with most actors = Lambs Cincinatti with 15 actors)

3. Retrieve the title from the film table and the category from the category table. Note that you will need to use a 3rd "bridge" table in order to get the solution.

4. We're going to send holiday gift cards to our 10 most profitable customers. Create a mailing list for these 10 customers. Get a list of all customer first names, last names, address, city, province, postal code, country, and email.

Practice Scenarios using JOINS

1. Retrieve all distinct titles and their descriptions from store 2. (you'll need to use the film and inventory tables) (762 rows)

```
SELECT distinct
b.store_id,
a.title as 'Movie Title',
a.description as 'Movie Description'
FROM film as a
INNER JOIN inventory as b
on a.film_id=b.film_id
WHERE b.store_id=2
```

Practice Scenarios using JOINS

2. From the film table and the film_actor tables, find the title and how many actors acted in each movie. (Note – you will need to use a join, aggregate function and group by clause). Sort by highest number of actors to lowest in descending order. (Top Movie with most actors = Lambs Cincinatti with 15 actors)

```
SELECT
title as 'Movie Title',
COUNT(actor_id) as '# of Actors'
FROM film as a
LEFT JOIN film_actor as b
on a.film_id=b.film_id
GROUP BY title
ORDER BY COUNT(actor_id) DESC
```

Practice Scenarios using JOINS

3. Retrieve the title from the film table and the category from the category table. Note that you will need to use a 3rd "bridge" table in order to get the solution.

```
SELECT
title as 'Movie Title',
name as 'Movie Category'
FROM film as a
LEFT JOIN film_category as b
on a.film_id=b.film_id
LEFT JOIN category as c
on b.category_id=c.category_id
```

Practice Scenarios using JOINS

4. We're going to send holiday gift cards to our 10 most profitable customers. Create a mailing list for these 10 customers. Get a list of all customer first names, last names, address, city, province, postal code, country, and email.

```
SELECT a.customer_id as 'Customer ID', SUM(amount) as 'Total Spent',
first_name as 'First Name', last_name as 'Last Name', address as 'Address',
city as City, district as 'Province or State', postal_code as 'Postal Code',
country as Country, email as Email
FROM customer as a JOIN address as b on a.address_id=b.address_id
JOIN city as c on b.city_id=c.city_id JOIN country as d on c.country_id=d.country_id
JOIN payment as e on a.customer_id=e.customer_id
GROUP BY a.customer_id
ORDER BY SUM(e.amount) DESC
LIMIT 10
```

Using “Window” Functions - OVER and PARTITION BY clauses

- The OVER and PARTITION clause are “Window” functions. They are used to perform calculations on a subset of your data. The OVER clause determines “windows” or sets of rows. The PARTITION BY clause splits the results into partitions on which the Window function is applied.

- The OVER clause is always used after an aggregation clause and before a PARTITION BY clause in your SQL statement:

```
SELECT column(1), SUM(column(2)) OVER(PARTITION BY column(1))
FROM table_name
ORDER BY column(2) [ASC/DESC]
```

Practice Scenarios using the OVER and PARTITION BY clauses

1. From the payment table, find the total amount spent by each customer but also ensure all the details for each individual transaction are still displayed.

2. Add to the query in question 1 to also show the average amount spent by each customer while still keeping all the details for each individual transaction

3. Add to the query from question 2 to show what each transaction represents as a % of the customer's total spending

4. Add to the query in Question 3 to show what each transaction represents as a % of the customers average spending.

Practice Scenarios using the OVER and PARTITION BY clauses

1. From the payment table, find the total amount spent by each customer but also ensure all the details for each individual transaction are still displayed.

```
SELECT
customer_id as 'Customer ID',
payment_id as 'Payment ID',
payment_date as 'Payment Date',
amount as 'Transaction Amount',
SUM(AMOUNT) OVER(PARTITION BY customer_id) as 'Total Amount Spent by
Customer'
FROM payment
```

Practice Scenarios using the OVER and PARTITION BY clauses

2. Add to the query in question 1 to also show the average amount spent by each customer while still keeping all the details for each individual transaction

```
SELECT
customer_id as 'Customer ID',
payment_id as 'Payment ID',
payment_date as 'Payment Date',
amount as 'Transaction Amount',
SUM(amount) OVER(PARTITION BY customer_id) as 'Total Amount Spent by Customer',
AVG(amount) OVER(PARTITION BY customer_id) as 'Average Spent by Customer',
FROM payment
```

Practice Scenarios using the OVER and PARTITION BY clauses

3. Modify the query from question 2 to show what each transaction represents as a % of the customer's total spending

```
SELECT
customer_id as 'Customer ID',
payment_id as 'Payment ID',
payment_date as 'Payment Date',
amount as 'Transaction Amount',
SUM(amount) OVER(PARTITION BY customer_id) as 'Total Amount Spent by Customer',
AVG(amount) OVER(PARTITION BY customer_id) as 'Average Spent by Customer',
amount/(SUM(Amount) OVER(PARTITION BY customer_id)) as 'Transaction as % of Total Spent',
FROM payment
```

Practice Scenarios using the OVER and PARTITION BY clauses

4. Add to the query in Question 3 to show what each transaction represents as a % of the customers average spending.

```
SELECT
customer_id as 'Customer ID',
payment_id as 'Payment ID',
payment_date as 'Payment Date',
amount as 'Transaction Amount',
SUM(amount) OVER(PARTITION BY customer_id) as 'Total Amount Spent by Customer',
AVG(amount) OVER(PARTITION BY customer_id) as 'Average Spent by Customer',
amount/(SUM(Amount) OVER(PARTITION BY customer_id)) as 'Transaction as % of Total Spent',
amount/(AVG(Amount) OVER(PARTITION BY customer_id)) as 'Transaction as % of Average Spent'
FROM payment
```