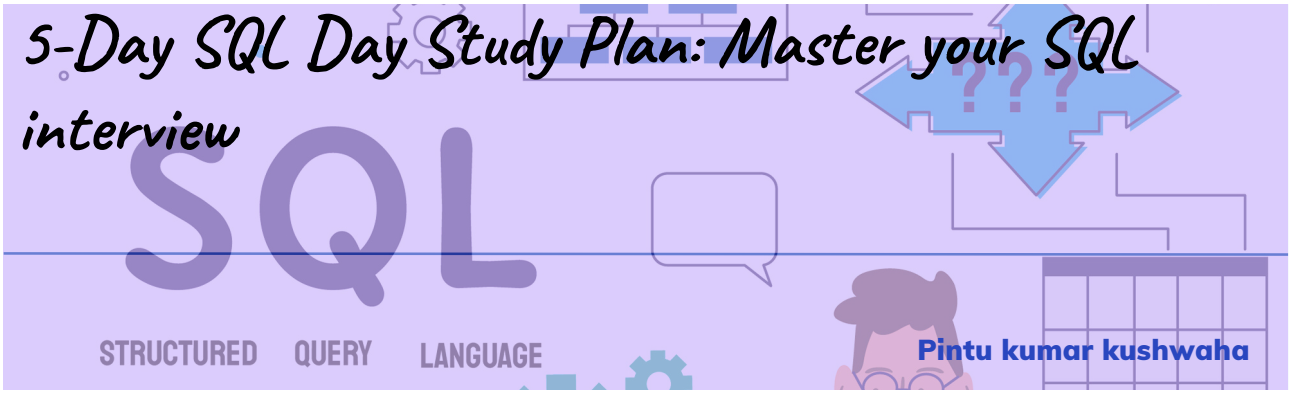


Introduction to SQL

What is SQL?

To manage, manipulate, and retrieve data from relational databases, programmers created SQL (Structured Query Language). It is widely used across sectors for various data-related tasks and acts as a standard interface for working with databases.

At its heart, SQL offers a collection of commands and statements that let users work with databases. Creating and altering database structures (tables, views, and indexes), inserting and updating data, querying and retrieving data, and carrying out intricate data manipulation procedures are all included in these operations.

Because SQL uses a declarative approach, users can indicate what data they wish to obtain or alter without identifying a specific method. The database management system handles streamlining the process and delivering the desired outcomes.

The readability, simplicity, and usability of SQL are well known. Because of its simple syntax and resemblance to natural language, it is usable by both technical and non-technical users. Users can effectively utilize the capabilities of relational databases thanks to its uniform approach to database work.

Why is SQL so popular?

Due to a number of factors, SQL has become extremely popular and is still a mainstay in the data management industry:

DBMSs, including MySQL, PostgreSQL, Oracle, and SQL Server, are compatible with SQL, making it a versatile language. It is a language that efficiently functions across several platforms, allowing users to work with various database systems without substantially changing their SQL code.

- **Standardization:** Since SQL is an ANSI/ISO standard, it ensures that all database systems use the same language syntax and essential capabilities. Thanks to this standardization, users can switch between many DBMSs or work with multiple database systems at once.
- **Relational Power**: Because relational data is the predominant data model in most applications, SQL is a master at handling it. It includes solid mechanisms for data integrity, transaction

management, enforcing relationships between tables, and strong querying capabilities to access and change data.

- **Data Analysis and Reporting:** SQL is a robust data analysis and reporting tool since it can execute complicated queries and aggregations. It enables users to produce reports, create data-driven dashboards, and extract valuable insights from massive amounts of data.
- **Efficiency and Performance:** SQL's declarative character makes efficient data retrieval and manipulation possible, which enables the database management system to optimize query execution. The most effective approach to processing SQL statements is determined by DBMS using advanced query optimizers, which leads to faster and more scalable processes.
- **Integration with Other Technologies:** SQL works well with other technologies and computer languages. Developers can interface with databases and carry out database operations within their code because of its ability to be embedded within apps. SQL can also be used in conjunction with data analysis tools, business intelligence platforms, and data warehouses to execute advanced analytics and decision-making.

Section: How Data Analysts Use SQL

Data analysts play a significant role in enterprises by gaining insights and making recommendations based on data. A data analyst must have access to SQL to effectively retrieve, manage, and analyze data to address critical business concerns.

Using SQL, data analysts can:

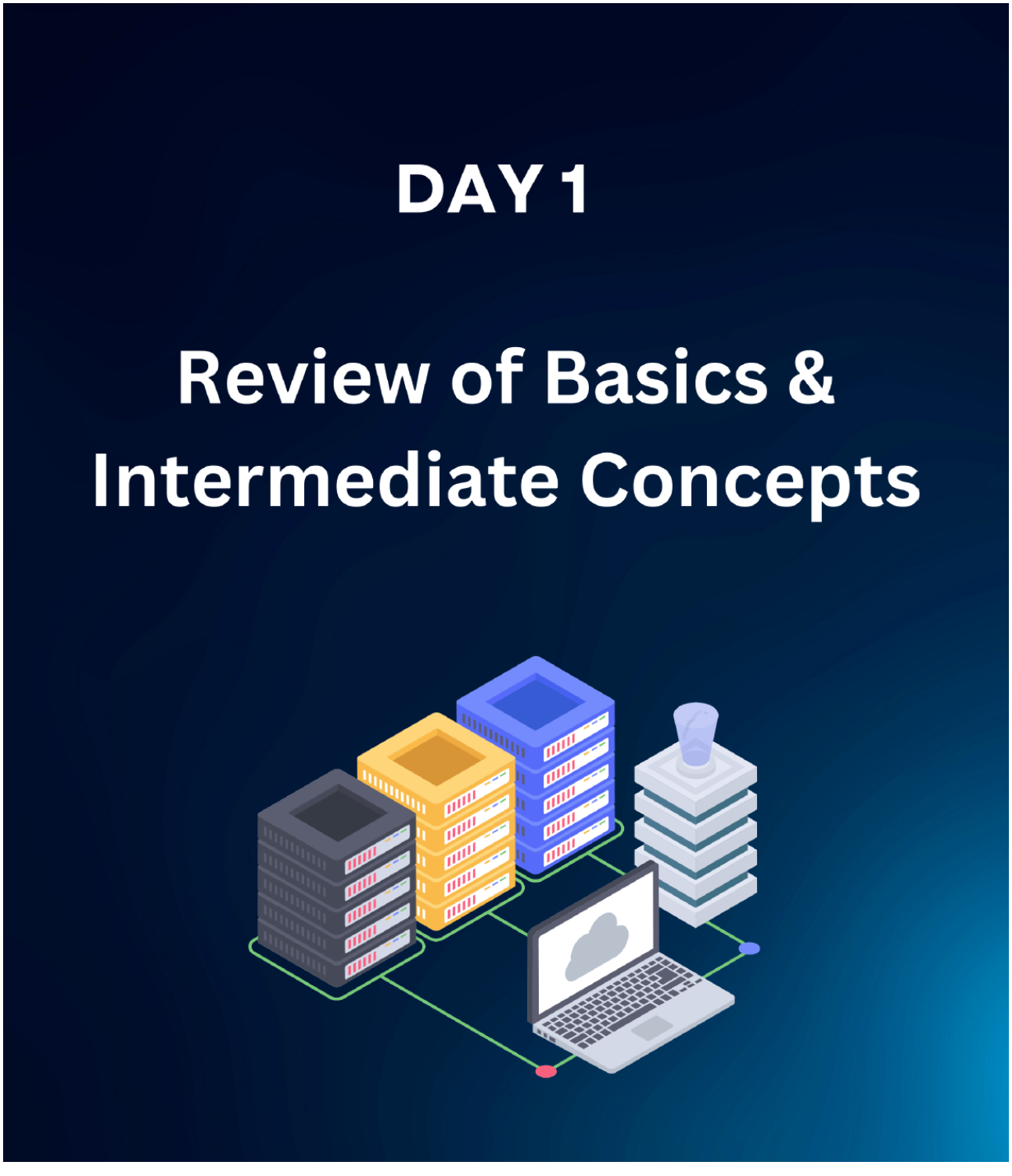
1. **Data Extraction and Cleaning:** SQL allows analysts to access particular subsets of data from databases, apply filters, and carry out data cleaning activities. They can link various tables, aggregate data, and format it appropriately for analysis.
2. **Data Exploration**: By querying databases to find patterns, trends, and anomalies, SQL enables analysts to examine data. They might develop bespoke queries to slice and dice data, execute computations, and summarize outcomes.
3. **Data Analysis and Reporting:** SQL allows analysts to carry out challenging analytical activities like metric calculation, statistical analysis, and cohort analysis. They can produce reports, monitor key performance indicators (KPIs), and give stakeholders helpful information.
4. **Business Intelligence (BI) Operations:** Business intelligence platforms and applications frequently employ SQL. Analysts use SQL to build, enhance, and maintain data pipelines, data models, and data warehouses. They may construct SQL-based reports, dashboards, and ETL (Extract, Transform, Load) operations.
5. **Data verification and quality control:** By doing data quality checks, locating missing values, and finding outliers, SQL enables analysts to verify the integrity of the data. They can create SQL queries to run data validation checks and ensure the data is accurate and consistent.

Data analysts proficient in SQL can easily handle enormous datasets, derive valuable insights, and offer data-driven suggestions to support sensible business decisions.

5-Day SQL Day Study Plan: Master

your SQL interview

Day 1: Review of Basics & Intermediate Concepts



Objective

Review basic and intermediate SQL concepts, including data extraction, joining tables, and working with aggregate functions. Understand how to manipulate and query data from multiple tables in a database.

Part 1: SQL Basics

The basics of SQL consist of understanding how to select data from a database and manipulate it to get desired results. The topics included in this part are the fundamental SQL statements — **SELECT**, **WHERE**, **GROUP BY**, and **ORDER BY**. These SQL commands are used to extract, filter, group, and order the data in the ways needed. For example, you may need to select certain columns from a database table, filter out rows based on specific conditions, group the data by a certain column, or sort the result-set in a particular order.

SELECT

The SELECT statement is used to select data from a database. The data returned is stored in a result table, called the result-set.

```
SELECT
   column1,
   column2
FROM
   table_name;
```

WHERE

The WHERE clause is used to filter records and extract only those records that fulfill a specified condition.

```
SELECT
   column1,
   column2
FROM
   table_name
WHERE
   condition;
```

GROUP BY

The GROUP BY statement groups rows that have the same values in specified columns into aggregated data. It is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

```
SELECT
  column_name
FROM
  table_name
WHERE
  condition
GROUP BY
  column_name
```

ORDER BY

The ORDER BY keyword is used to sort the result-set in ascending or descending order. The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

```
SELECT
    column1,
    column2
FROM
    table_name
ORDER BY
    column1,
    column2 ASC
```

Part 2: Joins

In this part, we discuss the concept of SQL joins. Joins are used to combine rows from two or more tables, based on a related column between them. Understanding SQL joins is essential for working with relational databases, where data is often spread across several related tables. We'll explore different types of joins — INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL JOIN — and understand the use cases for each.

INNER JOIN

The INNER JOIN keyword selects records that have matching values in both tables.

```
SELECT
   column_name
FROM
   table1
INNER JOIN
   table2
ON table1.column_name = table2.column_name;
```

LEFT JOIN

The LEFT JOIN keyword returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL on the right side if there is no match.

```
SELECT
   column_name
FROM
   table1
LEFT JOIN
   table2
ON table1.column_name = table2.column_name;
```

RIGHT JOIN

The RIGHT JOIN keyword returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL on the left side when there is no match.

```
SELECT
   column_name
FROM
   table1
RIGHT JOIN
   table2
ON table1.column_name = table2.column_name;
```

FULL JOIN

The FULL OUTER JOIN keyword returns all records when there is a match in either left (table1) or right (table2) table records.

```
SELECT
   column_name
FROM
   table1
FULL OUTER JOIN
```

```
    table2
ON table1.column_name = table2.column_name;
```

Part 3: Set Operations

Set operations in SQL provide ways to combine rows from two or more tables. We'll delve into the three primary set operations — **UNION**, **INTERSECT**, and **EXCEPT** - and learn how to use them effectively. Set operations can be a powerful tool when you want to combine or compare results from multiple SQL queries.

UNION

The UNION operator is used to combine the result-set of two or more SELECT statements. Each SELECT statement within the UNION must have the same number of columns, the columns must also have similar data types, and they must also be in the same order.

```
SELECT
    column_name
FROM
    table1
UNION
SELECT
    column_name(s)
FROM
    table2;
```

INTERCECT

The INTERSECT operator returns the records that two SELECT statements have in common.

```
SELECT
    column_name
FROM
    table1
INTERSECT
SELECT
    column_name
FROM
    table2;
```

EXCEPT

The EXCEPT operator returns the records from the first SELECT statement that are not in the second SELECT statement.

```
SELECT
    column_name
FROM
    table1
EXCEPT
SELECT
    column_name
FROM
    table2;
```

Part 4: Aggregate Functions

Aggregate functions are used in SQL to perform calculations on a set of values and return a single value. They can be very useful for tasks such as counting the number of records, calculating the sum or average of a column, or finding the minimum or maximum value. This part covers the essential aggregate functions **COUNT**, **AVG**, **SUM**, **MIN**, and **MAX**, and shows how to use them in different scenarios.

COUNT

The COUNT() function returns the number of rows that matches a specified criteria.

```
SELECT
    COUNT(column_name)
FROM
    table_name
WHERE
    condition;
```

AVG

The AVG() function returns the average value of a numeric column.

```
SELECT
    AVG(column_name)
FROM
    table_name
WHERE
    condition;
```

SUM

The SUM() function returns the total sum of a numeric column.

```
SELECT
   SUM(column_name)
FROM
   table_name
WHERE
   condition;
```

MIN and MAX

The MIN() and MAX() functions return the smallest and largest value of the selected column, respectively.

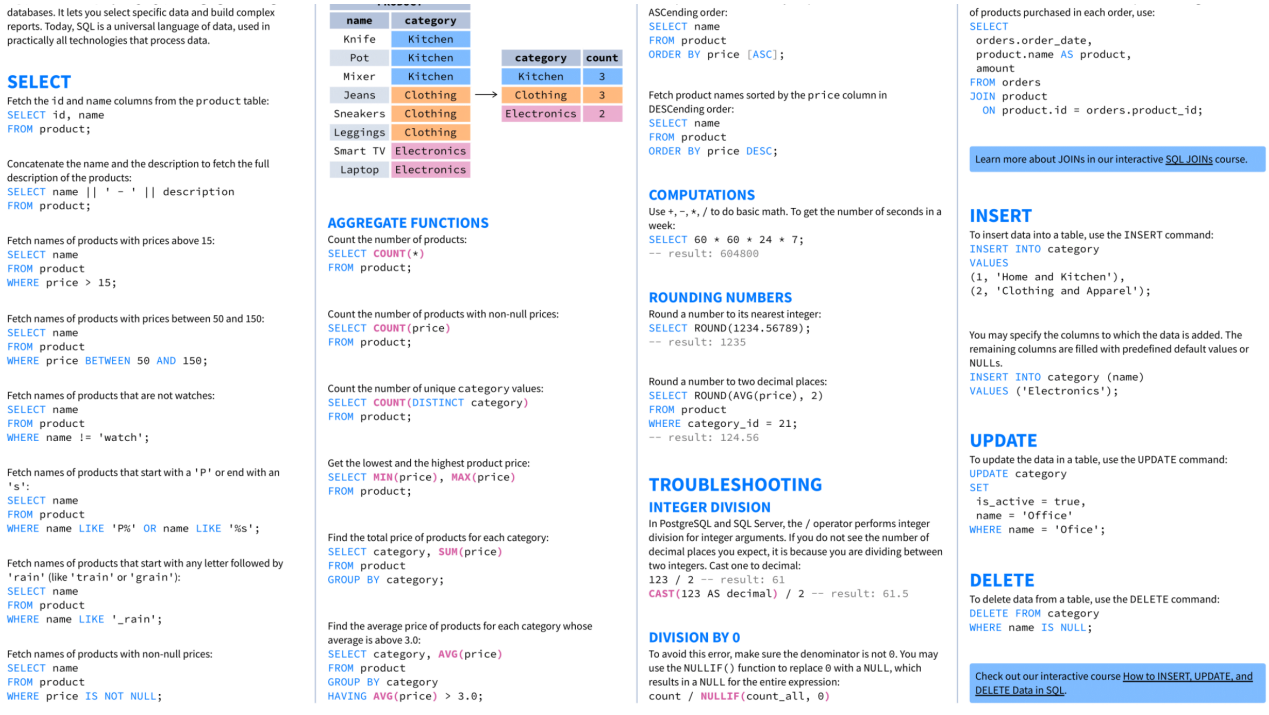
```
SELECT
   MIN(column_name)
FROM
   table_name
WHERE
   condition;
```

```
SELECT
    MAX(column_name)
FROM
    table_name
WHERE
    condition;
```

Part 5: Subqueries

A subquery is a SQL query nested inside a larger query, and it can be used in various ways to create more complex queries. Understanding subqueries is crucial for writing efficient and flexible SQL code. They allow you to perform operations in multiple steps, handle complex data manipulation tasks, and even base your main query on the results of a separate query. We'll learn the syntax for subqueries and look at some common use cases.

```
SELECT
   column_name
FROM
   table_name
WHERE
   column_name OPERATOR
(SELECT column_name FROM table_name WHERE condition);
```



Case Study: Analyzing an E-commerce Database

We will apply the SQL ideas we learnt on Day 1 to a useful, real-world problem in this case study. We will examine data from an online store that offers a variety of goods in a number of categories. The company keeps track of its products and customer orders in a database.

In this case study, our objective is to extract and analyze the data to provide answers to important business questions, such as determining the top-selling products, comprehending how well each product category is performing in terms of sales, finding high-value clients, and so forth.

Products and orders are the two main tables in the database. The products table includes details about each product, such as the product ID, name, category, and price. Order ID, product ID, customer ID, quantity, and order date are among the details concerning client orders that are included in the orders table.

We'll utilize SQL queries to investigate the data, spot trends, and derive insights throughout the case study so that the firm can make wise decisions. This will enable you to practice the SQL concepts we discussed on Day 1 and will provide you a practical grasp of how SQL is utilized in a commercial context.

Questions

1. How many orders were placed for each product?
2. What is the total quantity sold for each product category?

3. Which product has generated the most sales revenue?
4. Which customers have purchased ‘Product A’?
5. How many unique customers have made purchases?

Complete case study here

Schema SQL

```
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(50),
    product_category VARCHAR(50),
    product_price DECIMAL(6, 2)
);

CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    product_id INT,
    customer_id INT,
    quantity INT,
    order_date DATE,
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);

INSERT INTO products (product_id, product_name, product_category, product_price)
VALUES
(1, 'Product A', 'Category 1', 50.00),
(2, 'Product B', 'Category 1', 25.00),
(3, 'Product C', 'Category 2', 75.00),
(4, 'Product D', 'Category 2', 100.00),
(5, 'Product E', 'Category 3', 150.00);

INSERT INTO orders (order_id, product_id, customer_id, quantity, order_date)
VALUES
(1, 1, 101, 2, '2023-01-01'),
(2, 2, 102, 1, '2023-01-02'),
(3, 3, 103, 3, '2023-01-03'),
(4, 1, 104, 1, '2023-01-04'),
(5, 5, 105, 2, '2023-01-05'),
(6, 4, 106, 4, '2023-01-06'),
(7, 2, 107, 2, '2023-01-07'),
(8, 3, 101, 1, '2023-01-08'),
(9, 1, 108, 2, '2023-01-09'),
(10, 5, 109, 3, '2023-01-10');

-- How many orders were placed for each product?
-- What is the total quantity sold for each product category?
-- Which product has generated the most sales revenue?
```

```sql
-- Which customers have purchased 'Product A'?
-- How many unique customers have made purchases?

-- CHECKING TABLES

SELECT
    *
FROM
    customers;
SELECT
    *
FROM
    products;
SELECT
    *
FROM
    orders;


-- 1. How many orders were placed for each product?

SELECT
    product_id, COUNT(order_id) AS order_count
FROM
    orders
GROUP BY product_id;

-- 2. What is the total quantity sold for each product category?


SELECT
    p.category, t.total_quantity
FROM
    products p
        JOIN
    (SELECT
        product_id, SUM(quantity) AS total_quantity
    FROM
        orders
    GROUP BY product_id) AS t ON p.product_id = t.product_id;

-- 3. Which product has generated the most sales revenue?


SELECT
    p.product_id,
    p.product_name,
    price * quantity AS sales_revenue
```

```
FROM
    products p
        JOIN
    orders o ON p.product_id = o.product_id
ORDER BY sales_revenue DESC
LIMIT 5;


-- 4. Which customers have purchased 'Product A'?


SELECT customer_id

    FROM orders

    WHERE product_id = 4;

-- 5. How many unique customers have made purchases?


SELECT
    COUNT(DISTINCT customer_id)
FROM
    orders;
```

1. How many orders were placed for each product?

```
-- 1. How many orders were placed for each product?

SELECT
    product_id, COUNT(order_id) AS order_count
FROM
    orders
GROUP BY product_id;
```

- The GROUP BY clause is used to group the orders table by the product_id.
- The COUNT(order_id) function is applied to count the number of orders (order_id) in each group, effectively giving the count of orders for each product.
- The result set includes two columns: product_id and order_count, showing the product and the number of orders placed for it.

2. What is the total quantity sold for each product category?

```
--  2. What is the total quantity sold for each product category?

SELECT
```

```
    p.category, t.total_quantity
FROM
    products p
        JOIN
    (SELECT
        product_id, SUM(quantity) AS total_quantity
    FROM
        orders
    GROUP BY product_id) AS t ON p.product_id = t.product_id;
```

- The subquery in parentheses calculates the total quantity sold for each product (product_id) by summing the quantity in the orders table for each product.
- The main query joins the products and subquery results based on the product_id, creating a relationship between product categories and total quantities sold.
- The result set includes two columns: category (product category) and total_quantity (the total quantity sold for each product category).

3. Which product has generated the most sales revenue?

```
-- 3. Which product has generated the most sales revenue?

SELECT
    p.product_id,
    p.product_name,
    price * quantity AS sales_revenue
FROM
    products p
        JOIN
    orders o ON p.product_id = o.product_id
ORDER BY sales_revenue DESC
LIMIT 5;
```

- The query calculates
- The query calculates the sales revenue for each product by multiplying the price and quantity in the products and orders tables.
- The result set is ordered in descending order of sales revenue using the ORDER BY clause, so the product with the highest sales revenue appears at the top.
- The LIMIT 5 clause restricts the result to the top 5 products with the highest sales revenue.

4. Which customers have purchased ‘Product A’?

```
-- 4. Which customers have purchased 'Product A'?
```


```
-- 4. Which customers have purchased 'Product A'?
```


```
SELECT customer_id
   FROM orders
   WHERE product_id = 4;
```

- The query aims to identify customers who have purchased ‘Product A’ by finding their customer_id.
- It does this by querying the orders table.
- The WHERE clause is used to filter the rows in the orders table. It checks if the product_id in each row is equal to 4, assuming that 'Product A' is associated with the product ID 4. This filters the rows to only include those where 'Product A' has been ordered.
- The SELECT statement then retrieves the customer_id from the filtered rows.
- The result set will include one or more customer_id values corresponding to customers who have purchased 'Product A.' These values can be used to identify the customers who have shown an interest in or purchased this specific product.

5. How many unique customers have made purchases?

```
-- 5. How many unique customers have made purchases?

SELECT
   COUNT(DISTINCT customer_id)
FROM
   orders;
```

- The query calculates the count of unique customers who have made purchases by applying the COUNT(DISTINCT customer_id) function to the customer_id column in the orders table.
- This provides the total count of unique customers who have made purchases.

```
-- **Schema (MySQL v8.0)**

   CREATE TABLE products (
      product_id INT PRIMARY KEY,
      product_name VARCHAR(50),
      product_category VARCHAR(50),
      product_price DECIMAL(6, 2)
```

```
);

CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    product_id INT,
    customer_id INT,
    quantity INT,
    order_date DATE,
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);

INSERT INTO products (product_id, product_name, product_category, product_price)
VALUES
(1, 'Product A', 'Category 1', 50.00),
(2, 'Product B', 'Category 1', 25.00),
(3, 'Product C', 'Category 2', 75.00),
(4, 'Product D', 'Category 2', 100.00),
(5, 'Product E', 'Category 3', 150.00);

INSERT INTO orders (order_id, product_id, customer_id, quantity, order_date)
VALUES
(1, 1, 101, 2, '2023-01-01'),
(2, 2, 102, 1, '2023-01-02'),
(3, 3, 103, 3, '2023-01-03'),
(4, 1, 104, 1, '2023-01-04'),
(5, 5, 105, 2, '2023-01-05'),
(6, 4, 106, 4, '2023-01-06'),
(7, 2, 107, 2, '2023-01-07'),
(8, 3, 101, 1, '2023-01-08'),
(9, 1, 108, 2, '2023-01-09'),
(10, 5, 109, 3, '2023-01-10');
```

Query #1

```
select * from  products;
```

product_id	product_name	product_category	product_price
1	Product A	Category 1	50.00
2	Product B	Category 1	25.00
3	Product C	Category 2	75.00
4	Product D	Category 2	100.00
5	Product E	Category 3	150.00

Query #2

```
select * from orders;
```

order_id	product_id	customer_id	quantity	order_date
1	1	101	2	2023-01-01
2	2	102	1	2023-01-02
3	3	103	3	2023-01-03
4	1	104	1	2023-01-04
5	5	105	2	2023-01-05
6	4	106	4	2023-01-06
7	2	107	2	2023-01-07
8	3	101	1	2023-01-08
9	1	108	2	2023-01-09
10	5	109	3	2023-01-10

Query #3

```
select product_id,
       count(order_id) as order_count
from orders
group by product_id;
```

product_id	order_count
1	3
2	2
3	2
4	1
5	2

Query #4

```
SELECT p.product_category, t.total_quantity
FROM products p
JOIN (
    SELECT product_id, SUM(quantity) AS total_quantity
    FROM orders
    GROUP BY product_id
) AS t
ON p.product_id = t.product_id;
```

product_category	total_quantity
Category 1	5
Category 1	3

Category 2	4
Category 2	4
Category 3	5

Query #5

```
select p.product_id, p.product_name, product_price*quantity as sales_revenue
from products p
join orders o
on p.product_id = o.product_id
order by sales_revenue desc
limit 5;
```

product_id	product_name	sales_revenue
5	Product E	450.00
4	Product D	400.00
5	Product E	300.00
3	Product C	225.00
1	Product A	100.00

Query #6

```
SELECT customer_id

    FROM orders

    WHERE product_id = 4;
```

customer_id
106

Query #7

```
select count(distinct customer_id)
from orders;
```

count(distinct customer_id)
9

Day 2: Advanced Filtering, Functions, and Operators

Objective

Explore advanced filtering techniques in SQL, including string functions, date/time functions, numeric functions, and operators. Learn how to apply these functions to filter and manipulate data effectively.

Part 1: String Functions

In this part, we will explore various string functions in SQL. String functions allow us to manipulate and extract information from text data stored in database tables. We'll cover essential string functions such as **LENGTH**, **UPPER**, **LOWER**, **SUBSTRING**, and **CONCAT**. These functions will enable us to perform tasks like finding the length of a string, converting strings to uppercase or lowercase, extracting substrings from strings, and concatenating strings together.

LENGTH

The LENGTH() function returns the length of a string.

```
SELECT
   LENGTH(column_name)
FROM
   table_name;
```

UPPER and LOWER

The UPPER() and LOWER() functions convert a string to uppercase and lowercase, respectively.

```
SELECT SELECT
   UPPER(column_name)
FROM
   table_name;

SELECT
   LOWER(column_name)
FROM
   table_name;
```

SUBSTRING

The SUBSTRING() function extracts a substring from a string.

```
SELECT SELECT
   SUBSTRING(column_name, start_position, length)
```

```
FROM
    table_name;
```

CONCAT

The CONCAT() function concatenates two or more strings.

```
SELECT
    CONCAT(column_name1, column_name2)
FROM
    table_name;
```

Part 2: Date/Time Functions

Dates and times are crucial data types in many databases, and SQL provides several functions to work with them effectively. In this part, we will learn about date/time functions in SQL. We'll cover concepts like **CURRENT_DATE** for obtaining the current date, **DATE_FORMAT** for formatting dates as per specified formats, **DATE_ADD** and **DATE_SUB** for adding or subtracting time intervals from dates, and **DATEDIFF** for calculating the difference between two dates.

CURRENT_DATE

The CURRENT_DATE function returns the current date.

```
SELECT CURRENT_DATE;SELECT CURRENT_DATE;
```

DATE_FORMAT

The DATE_FORMAT() function formats a date as per a specified format.

```
SELECT SELECT
    DATE_FORMAT(column_name, 'format')
FROM
    table_name;
```

DATE_ADD and DATE_SUB

The DATE_ADD() and DATE_SUB() functions add or subtract a specified time interval from a date.

```
SELECT SELECT
    DATE_ADD(column_name, INTERVAL value unit)
FROM
    table_name;
```

```
SELECT
   DATE_SUB(column_name, INTERVAL value unit)
FROM
   table_name;
```

DATEDIFF

The DATEDIFF() function calculates the difference between two dates.

```
SELECT
   DATEDIFF(column_name1, column_name2)
FROM
   table_name;
```

Part 3: Numeric Functions

Numeric functions play a significant role in performing calculations and manipulations on numeric data in SQL. In this part, we will explore important numeric functions like **ABS** for obtaining the absolute value of a number, **ROUND** for rounding numbers to a specified decimal place, **FLOOR** for rounding down to the nearest integer, and **CEILING** for rounding up to the nearest integer. Understanding these functions will allow us to perform various calculations and transformations on numeric data.

ABS

The ABS() function returns the absolute value of a number.

```
SELECT SELECT
   ABS(column_name)
FROM
   table_name;
```

ROUND

The ROUND() function rounds a number to a specified number of decimal places.

```
SELECT
   ROUND(column_name, decimals)
FROM
   table_name;
```

FLOOR and CEILING

The FLOOR() function returns the largest integer less than or equal to a number, while the CEILING() function returns the smallest integer greater than or equal to a

number.

```
SELECT SELECT
   FLOOR(column_name)
FROM
   table_name;

SELECT
    CEILING(column_name)
FROM
    table_name;
```

Part 4: Advanced Filtering using Operators

In this part, we will dive into advanced filtering techniques using operators in SQL. Operators provide powerful tools for filtering and selecting specific data based on certain conditions. We'll cover operators like **LIKE** for pattern matching, **IN** for specifying multiple values, **BETWEEN** for selecting values within a range, **NULL** for handling NULL values, and **NOT** for negating a condition. These operators will enable us to write more complex and precise SQL queries for data extraction and analysis.

LIKE

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

```
SELECT SELECT
   column_name
FROM
   table_name
WHERE
   column_name LIKE pattern;
```

IN

The IN operator allows you to specify multiple values in a WHERE clause.

```
SELECT SELECT
   column_name
FROM
   table_name
WHERE
   column_name IN (value1);
```

BETWEEN

The BETWEEN operator selects values within a specified range.

```
SELECT SELECT
    column_name
FROM
    table_name
WHERE
    column_name BETWEEN value1 AND value2;
```

NULL

The IS NULL operator is used to test for NULL values.

```
SELECT SELECT
    column_name
FROM
    table_name
WHERE
    column_name IS NULL;
```

NOT

The NOT operator is used to negate a condition.

```
SELECT SELECT
    column_name(s)
FROM
    table_name
WHERE
    NOT condition;
```

SQL for Data Analysis Cheat Sheet

LearnSQL.com

DATE AND TIME

There are 3 main time-related types: `date`, `time`, and `timestamp`. Time is expressed using a 24-hour clock, and it can be as vague as just hour and minutes (e.g., `15:30` – 3:30 p.m.) or as precise as microseconds and time zone (as shown below):

```
2021-12-31 14:39:53.662522-05
|  date  |
|      time      |
|          timestamp          |
YYYY-mm-dd HH:MM:SS.ssssss±TZ
```

`14:39:53.662522-05` is almost 2:40 p.m. CDT (e.g., in Chicago; in UTC it'd be 7:40 p.m.). The letters in the above example represent:

In the date part:

- `YYYY` – the 4-digit year.
- `mm` – the zero-padded month (`01`—January through `12`—December).
- `dd` – the zero-padded day.

In the time part:

- `HH` – the zero-padded hour in a 24-hour clock.
- `MM` – the minutes.
- `SS` – the seconds. *Omissible.*
- `ssssss` – the smaller parts of a second – they can be expressed using 1 to 6 digits. *Omissible.*
- `±TZ` – the timezone. It must start with either `+` or `−`, and use two digits relative to UTC. *Omissible.*

CURRENT DATE AND TIME

Find out what time it is:

```
SELECT CURRENT_TIME;
```

Get today's date:

```
SELECT CURRENT_DATE;
```

In SQL Server:

```
SELECT GETDATE();
```

Get the timestamp with the current date and time:

```
SELECT CURRENT_TIMESTAMP;
```

CREATING DATE AND TIME VALUES

To create a date, time, or timestamp, write the value as a string and cast it to the proper type.

```
SELECT CAST('2021-12-31' AS date);
SELECT CAST('15:31' AS time);
```

SORTING CHRONOLOGICALLY

Using `ORDER BY` on date and time columns sorts rows chronologically from the oldest to the most recent:

```
SELECT order_date, product, quantity
FROM sales
ORDER BY order_date;
```

order_date	product	quantity
2023-07-22	Laptop	2
2023-07-23	Mouse	3
2023-07-24	Sneakers	10
2023-07-24	Jeans	3
2023-07-25	Mixer	2

Use the DESCending order to sort from the most recent to the oldest:

```
SELECT order_date, product, quantity
FROM sales
ORDER BY order_date DESC;
```

COMPARING DATE AND TIME VALUES

You may use the comparison operators `<`, `<=`, `>`, `>=`, and `=` to compare date and time values. Earlier dates are less than later ones. For example, `2023-07-05` is "less" than `2023-08-05`.

Find sales made in July 2023:

```
SELECT order_date, product_name, quantity
FROM sales
WHERE order_date >= '2023-07-01'
  AND order_date <  '2023-08-01';
```

Find customers who registered in July 2023:

```
SELECT registration_timestamp, email
FROM customer
WHERE registration_timestamp >= '2023-07-01'
  AND registration_timestamp <  '2023-08-01';
```

INTERVALS

An interval measures the difference between two points in time. For example, the interval between `2023-07-04` and `2023-07-06` is 2 days.

To define an interval in SQL, use this syntax:

```
INTERVAL '1' DAY
```

The syntax consists of three elements: the `INTERVAL` keyword, a quoted value, and a time part keyword. You may use the following time parts: `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, and `SECOND`.

Adding intervals to date and time values

You may use `+` or `−` to add or subtract an interval to date or timestamp values.

Subtract one year from `2023-07-05`:

```
SELECT CAST('2023-07-05' AS TIMESTAMP)
         - INTERVAL '1' year;
-- result: 2022-07-05 00:00:00
```

Find customers who placed the first order within a month from the registration date:

```
SELECT id
FROM customers
WHERE first_order_date >
  registration_date + INTERVAL '1' month;
```

Filtering events to those in the last 7 days

To find the deliveries scheduled for the last 7 days, use:

```
SELECT delivery_date, address
FROM sales
WHERE delivery_date <= CURRENT_DATE
  AND delivery_date >= CURRENT_DATE
      - INTERVAL '7' DAY;
```

Note: In SQL Server, intervals are not implemented – use the `DATEADD()` and `DATEDIFF()` functions.

Filtering events to those in the last 7 days

EXTRACTING PARTS OF DATES

The standard SQL syntax to get a part of a date is

```
SELECT EXTRACT(YEAR FROM order_date)
FROM sales;
```

You may extract the following fields:
`YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, and `SECOND`.

The standard syntax does not work In SQL Server.
Use the `DATEPART(part, date)` function instead.

```
SELECT DATEPART(YEAR, order_date)
FROM sales;
```

GROUPING BY YEAR AND MONTH

Find the count of sales by month:

```
SELECT
 EXTRACT(YEAR FROM order_date) AS year,
 EXTRACT(MONTH FROM order_date) AS month,
 COUNT(*) AS count
FROM sales
GROUP BY
 year,
 month
ORDER BY
 year
 month;
```

year	month	count
2022	8	51
2022	9	58
2022	10	62
2022	11	76
2022	12	85
2023	1	71
2023	2	69

Note that you must group by both the year and the month. `EXTRACT(MONTH FROM order_date)` only extracts the

```
SELECT CAST('2021-12-31 23:59:29+02'
       AS timestamp);
SELECT CAST('15:31.124769' AS time);
```

Be careful with the last example – it is interpreted as 15 minutes 31 seconds and 124769 microseconds! It is always a good idea to write 00 for hours explicitly: `'00:15:31.124769'`.

Note: Pay attention to the end date in the query. The upper bound `'2023-08-01'` is not included in the range. The timestamp `'2023-08-01'` is actually the timestamp `'2023-08-01 00:00:00.0'`. The comparison operator < is used to ensure the selection is made for all timestamps less than `'2023-08-01 00:00:00.0'`, that is, all timestamps in July 2023, even those close to the midnight of August 1, 2023.

in SQL Server

To find the sales made within the last 7 days, use:

```
SELECT delivery_date, address
FROM sales
WHERE delivery_date <= GETDATE()
  AND delivery_date >=
    DATEADD(DAY, -7, GETDATE());
```

month number (1, 2, ..., 12). To distinguish between months from different years, you must also group by year.

More about working with date and time values in our interactive Standard SQL Functions course.

Case Study: Customer Segmentation for a Telecommunication Company

In this case study, we will analyze data from a subscription-based service that offers various plans to customers. The company maintains two tables: **customers** and **payments**. The **customers** table stores customer information, such as customer ID, customer name, email address, and service type. The **payments** table records payment transactions made by customers, including the payment ID, customer ID, payment amount, and payment date.

Our objective is to gain insights into customer payment behavior and identify patterns that can help improve customer retention and revenue. By applying SQL concepts learned in Day 2, we will perform analysis tasks such as calculating total payment amounts, identifying customers who have made recent payments, segmenting customers based on payment history, and more.

Throughout the case study, we will utilize advanced filtering techniques, string functions, date/time functions, numeric functions, and subqueries to extract and analyze the data effectively. This will provide valuable insights into customer payment behavior, enabling data-driven decision-making for the subscription service company.

Questions

1. Retrieve the email addresses of customers in uppercase.
2. Calculate the total payment amount for each customer.
3. Retrieve the customers who have made payments greater than the average payment amount.
4. Retrieve the customers who have used the ‘Internet’ service.
5. Segment customers based on their payment history: ‘Good’, ‘Average’, or ‘Poor’.

Complete the case study here

Schema SQL

```
CREATE TABLE customers (CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(50),
    email VARCHAR(100),
    service_type VARCHAR(50),
    last_payment_date DATE
);

CREATE TABLE payments (
    payment_id INT PRIMARY KEY,
    customer_id INT,
    payment_amount DECIMAL(10, 2),
    payment_date DATE,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

INSERT INTO customers (customer_id, customer_name, email, service_type, last_payment_date)
VALUES
    (1, 'John Doe', 'johndoe@example.com', 'Internet', '2023-01-10'),
    (2, 'Jane Smith', 'janesmith@example.com', 'Mobile', '2023-01-15'),
    (3, 'Mike Johnson', 'mikejohnson@example.com', 'Internet', '2023-01-05'),
    (4, 'Emily Brown', 'emilybrown@example.com', 'Landline', '2023-01-20'),
    (5, 'David Wilson', 'davidwilson@example.com', 'Internet', '2023-01-12');

INSERT INTO payments (payment_id, customer_id, payment_amount, payment_date)
VALUES
    (1, 1, 50.00, '2023-01-05'),
    (2, 2, 35.00, '2023-01-10'),
    (3, 3, 75.00, '2023-01-15'),
    (4, 4, 20.00, '2023-01-18'),
    (5, 5, 60.00, '2023-01-12'),
    (6, 1, 40.00, '2023-01-20'),
    (7, 2, 30.00, '2023-01-25'),
    (8, 3, 80.00, '2023-01-28'),
    (9, 4, 25.00, '2023-01-22'),
    (10, 5, 65.00, '2023-01-30');

-- -- 1. Retrieve the email addresses of customers in uppercase.1. Retrieve the email addresses of customers in
uppercase.

SELECT UPPER(email)
FROM customers;


-- 2. Calculate the total payment amount for each customer.

SELECT customer_id, SUM(payment_amount)
FROM payments
```

```sql
GROUP BY customer_id;

-- SOLUTION 2


SELECT c.customer_id,
       c.customer_name,
       SUM(payment_amount) AS total_payment_amount
FROM customers c
JOIN payments p
ON c.customer_id = p.customer_id
GROUP BY c.customer_id
ORDER BY total_payment_amount DESC;


-- 3. Retrieve the customers who have made payments greater than the average payment amount.

SELECT * FROM customers
WHERE customer_id IN
(SELECT customer_id FROM payments
WHERE payment_amount >
(SELECT AVG(payment_amount) from payments));


-- 4. Retrieve the customers who have used the 'Internet' service.

SELECT * FROM customers
WHERE service_type = 'Internet';



-- 5.  Segment customers based on their payment history: 'Good', 'Average', or 'Poor'.

SELECT
  c.customer_id,
  c.customer_name,
  SUM(p.payment_amount) AS total_payment_amount,
  CASE
    WHEN SUM(p.payment_amount) > 100 THEN 'Good'
    WHEN SUM(p.payment_amount) > 50 THEN 'Average'
    ELSE 'Poor'
  END AS payment_history
FROM payments p
JOIN customers c
ON p.customer_id = c.customer_id
GROUP BY c.customer_name, c.customer_id;

/*
```

This SQL query will segment customers based on their payment history into 'Good', 'Average', or 'Poor' categories. Here's an explanation of each part of the query:

SELECT c.customer_id, c.customer_name, SUM(p.payment_amount) AS total_payment_amount: This part of the query selects specific columns from both the "customers" and "payments" tables. It selects the customer's ID and name from the "customers" table and calculates the total payment amount for each customer by summing the "payment_amount" column from the "payments" table. The calculated sum is aliased as "total_payment_amount" for readability.

CASE WHEN SUM(p.payment_amount) > 100 THEN 'Good' WHEN SUM(p.payment_amount) > 50 THEN 'Average' ELSE 'Poor' END AS payment_history: In this section, the CASE statement is used to categorize customers' payment history based on the calculated total payment amount. Customers are categorized as follows:

If their total payment amount is greater than 100, they are categorized as 'Good.'
If their total payment amount is greater than 50 but not more than 100, they are categorized as 'Average.'
If their total payment amount is 50 or less, they are categorized as 'Poor.'
FROM payments p JOIN customers c ON p.customer_id = c.customer_id: This part of the query specifies the tables involved in the query and the relationship between them. It joins the "payments" table (aliased as "p") with the "customers" table (aliased as "c") using the "customer_id" column as the common link.

GROUP BY c.customer_name, c.customer_id: The GROUP BY clause groups the results by customer name and customer ID. This is necessary because the SUM function is used to calculate the total payment amount for each customer, and grouping ensures that each customer is treated as a separate group.

When you run this query, it will return a result set with customer information, the total payment amount for each customer, and their payment history category ('Good,' 'Average,' or 'Poor'). This query helps segment customers based on their payment behavior, making it easier to analyze and target different customer groups.

*/

1. Retrieve the email addresses of customers in uppercase.

-- 1. Retrieve the email addresses of customers in uppercase.-- 1. Retrieve the email addresses of customers in uppercase.

```
SELECT UPPER(email)
FROM customers;
```

This SQL query retrieve the email addresses of customers and convert them to uppercase. Here’s an explanation of each part of the query:

SELECT UPPER(email): This part of the query is responsible for selecting data from the “email” column of the “customers” table and converting it to uppercase using the UPPER function. The UPPER function is a SQL function that transforms text to uppercase.

FROM customers: This specifies the table from which we want to retrieve data, which is the “customers” table in this case. It tells the database where to find the “email” column.

So, when you run this query, the database will return a result set with all the email addresses from the “customers” table, but they will all be in uppercase.

2. Calculate the total payment amount for each customer.

```
-- 2. Calculate the total payment amount for each customer.-- 2. Calculate the total payment amount for each customer.

SELECT customer_id, SUM(payment_amount)
FROM payments
GROUP BY customer_id;


-- SOLUTION 2

SELECT c.customer_id,
    c.customer_name,
    SUM(payment_amount) AS total_payment_amount
FROM customers c
JOIN payments p
ON c.customer_id = p.customer_id
GROUP BY c.customer_id
ORDER BY total_payment_amount DESC;
```

This SQL query will calculate the total payment amount for each customer. Here’s a breakdown of each part of the query:

SELECT customer_id, SUM(payment_amount): This part of the query specifies the columns that will be included in the result set. It selects the “customer_id” column, which identifies each customer, and it calculates the sum of the “payment_amount” column using the SUM function. The result set will include these two columns: “customer_id” and the total payment amount for each customer.

FROM payments: This part of the query indicates the source table from which the data will be retrieved. In this case, it’s the “payments” table, which likely contains records of payments made by customers.

GROUP BY customer_id: This is a crucial part of the query. It specifies that the data should be grouped based on the “customer_id” column. The SUM function

will then be applied within each group, calculating the total payment amount separately for each unique customer.

So, when you run this query, the database will aggregate payment data for each customer. It will sum up all the payment amounts associated with each customer (identified by their unique “customer_id”), and the result will display a list of customer IDs along with the total payment amount for each customer.

This type of query is useful for generating reports or analyses that require summarizing payment information on a per-customer basis, allowing businesses to understand the payment behavior of their customers.

3. Retrieve the customers who have made payments greater than the average payment amount.

```
-- 3. Retrieve the customers who have made payments greater than the average payment amount.-- 3. Retrieve the customers who have made payments greater than the average payment amount.

SELECT * FROM customers
WHERE customer_id IN
(SELECT customer_id FROM payments
WHERE payment_amount >
(SELECT AVG(payment_amount) from payments));
```

This SQL retrieve customers who have made payments greater than the average payment amount. Here’s an explanation of how this query works:

Innermost Subquery (SELECT AVG(payment_amount) from payments)): This subquery calculates the average payment amount from the “payments” table. It computes the total sum of all payment amounts and divides it by the number of payments to find the average.

Middle Subquery (SELECT customer_id FROM payments WHERE payment_amount > ...): This subquery selects customer IDs from the “payments” table where the payment amount is greater than the average payment amount calculated in the innermost subquery. In other words, it identifies customers who have made payments above the average.

Outer Query (SELECT * FROM customers WHERE customer_id IN ...): The outermost query retrieves all columns from the “customers” table for customers whose IDs are found in the result of the middle subquery. In other words, it retrieves customer information for those who have made payments greater than the average.

When you run this query, it will return a list of customer records (all columns from the “customers” table) for customers who have made payments exceeding the average payment amount. This allows you to identify and analyze customers who are making above-average payments in your database.

4. Retrieve the customers who have used the ‘Internet’ service.

```
-- 4. Retrieve the customers who have used the 'Internet' service.-- 4. Retrieve the customers who have used the 'Internet' service.

SELECT * FROM customers
WHERE service_type = 'Internet';
```

This SQL query retrieve customers who have used the ‘Internet’ service. Here’s an explanation of the query:

SELECT * FROM customers: This part of the query specifies that you want to retrieve all columns (represented by *) from the “customers” table. It indicates that you want to obtain all customer information.

WHERE service_type = ‘Internet’: This is the filtering condition. It tells the database to select only those rows where the value in the “service_type” column is equal to ‘Internet’. In other words, it identifies customers who have used the ‘Internet’ service.

When you run this query, it will return a list of customer records, including all columns from the “customers” table, but only for customers who have used the ‘Internet’ service. This query is helpful for segmenting and analyzing customers based on the specific services they have used.

5. Segment customers based on their payment history: ‘Good’, ‘Average’, or ‘Poor’.

```
-- 5.  Segment customers based on their payment history: 'Good', 'Average', or 'Poor'.-- 5.  Segment customers based on their payment history: 'Good', 'Average', or 'Poor'.

SELECT
  c.customer_id,
  c.customer_name,
  SUM(p.payment_amount) AS total_payment_amount,
  CASE
    WHEN SUM(p.payment_amount) > 100 THEN 'Good'
    WHEN SUM(p.payment_amount) > 50 THEN 'Average'
    ELSE 'Poor'
```

```
  END AS payment_history
FROM payments p
JOIN customers c
ON p.customer_id = c.customer_id
GROUP BY c.customer_name, c.customer_id;
```

This SQL query will segment customers based on their payment history into ‘Good’, ‘Average’, or ‘Poor’ categories. Here’s an explanation of each part of the query:

SELECT c.customer_id, c.customer_name, SUM(p.payment_amount) AS total_payment_amount: This part of the query selects specific columns from both the “customers” and “payments” tables. It selects the customer’s ID and name from the “customers” table and calculates the total payment amount for each customer by summing the “payment_amount” column from the “payments” table. The calculated sum is aliased as “total_payment_amount” for readability.

CASE WHEN SUM(p.payment_amount) > 100 THEN ‘Good’ WHEN SUM(p.payment_amount) > 50 THEN ‘Average’ ELSE ‘Poor’ END AS payment_history: In this section, the CASE statement is used to categorize customers’ payment history based on the calculated total payment amount. Customers are categorized as follows:

If their total payment amount is greater than 100, they are categorized as ‘Good.’
If their total payment amount is greater than 50 but not more than 100, they are categorized as ‘Average.’
If their total payment amount is 50 or less, they are categorized as ‘Poor.’
FROM payments p JOIN customers c ON p.customer_id = c.customer_id: This part of the query specifies the tables involved in the query and the relationship between them. It joins the “payments” table (aliased as “p”) with the “customers” table (aliased as “c”) using the “customer_id” column as the common link.

GROUP BY c.customer_name, c.customer_id: The GROUP BY clause groups the results by customer name and customer ID. This is necessary because the SUM function is used to calculate the total payment amount for each customer, and grouping ensures that each customer is treated as a separate group.

When you run this query, it will return a result set with customer information, the total payment amount for each customer, and their payment history category (‘Good,’ ‘Average,’ or ‘Poor’). This query helps segment customers based on their payment behavior, making it easier to analyze and target different customer groups.

Schema (MySQL v8.0)**Schema (MySQL v8.0)**

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(50),
    email VARCHAR(100),
    service_type VARCHAR(50),
    last_payment_date DATE
);

CREATE TABLE payments (
    payment_id INT PRIMARY KEY,
    customer_id INT,
    payment_amount DECIMAL(10, 2),
    payment_date DATE,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

INSERT INTO customers (customer_id, customer_name, email, service_type, last_payment_date)
VALUES
    (1, 'John Doe', 'johndoe@example.com', 'Internet', '2023-01-10'),
    (2, 'Jane Smith', 'janesmith@example.com', 'Mobile', '2023-01-15'),
    (3, 'Mike Johnson', 'mikejohnson@example.com', 'Internet', '2023-01-05'),
    (4, 'Emily Brown', 'emilybrown@example.com', 'Landline', '2023-01-20'),
    (5, 'David Wilson', 'davidwilson@example.com', 'Internet', '2023-01-12');

INSERT INTO payments (payment_id, customer_id, payment_amount, payment_date)
VALUES
    (1, 1, 50.00, '2023-01-05'),
    (2, 2, 35.00, '2023-01-10'),
    (3, 3, 75.00, '2023-01-15'),
    (4, 4, 20.00, '2023-01-18'),
    (5, 5, 60.00, '2023-01-12'),
    (6, 1, 40.00, '2023-01-20'),
    (7, 2, 30.00, '2023-01-25'),
    (8, 3, 80.00, '2023-01-28'),
    (9, 4, 25.00, '2023-01-22'),
    (10, 5, 65.00, '2023-01-30');
```

Query #1

```
SELECT UPPER(email)
FROM customers;
```

| UPPER(email) |

| ---------------------- |
| JOHNDOE@EXAMPLE.COM |
| JANESMITH@EXAMPLE.COM |
| MIKEJOHNSON@EXAMPLE.COM |
| EMILYBROWN@EXAMPLE.COM |
| DAVIDWILSON@EXAMPLE.COM |

Query #2

```
SELECT customer_id, SUM(payment_amount)
FROM payments
GROUP BY customer_id;
```

customer_id	SUM(payment_amount)
1	90.00
2	65.00
3	155.00
4	45.00
5	125.00

Query #3

```
SELECT c.customer_id,
       c.customer_name,
       SUM(payment_amount) AS total_payment_amount
FROM customers c
JOIN payments p
ON c.customer_id = p.customer_id
GROUP BY c.customer_id
ORDER BY total_payment_amount DESC;
```

customer_id	customer_name	total_payment_amount
3	Mike Johnson	155.00
5	David Wilson	125.00
1	John Doe	90.00
2	Jane Smith	65.00
4	Emily Brown	45.00

Query #4

```
SELECT * FROM customers
WHERE customer_id IN
(SELECT customer_id FROM payments
```

```
    WHERE payment_amount >
    (SELECT AVG(payment_amount) from payments));
```

customer_id	customer_name	email	service_type	last_payment_date
1	John Doe	johndoe@example.com	Internet	2023-01-10
3	Mike Johnson	mikejohnson@example.com	Internet	2023-01-05
5	David Wilson	davidwilson@example.com	Internet	2023-01-12

Query #5

```
    SELECT * FROM customers
    WHERE service_type = 'Internet';
```

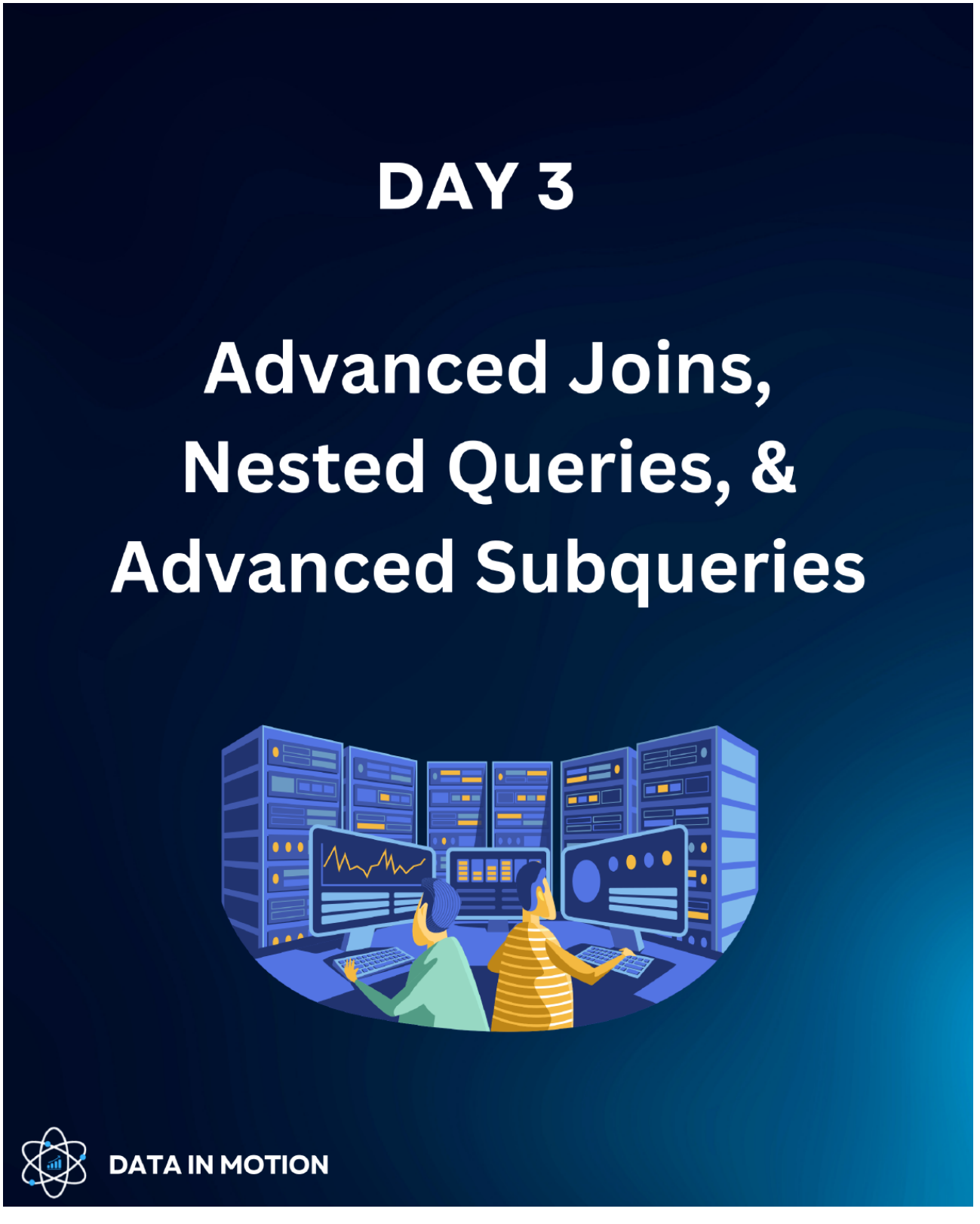
customer_id	customer_name	email	service_type	last_payment_date
1	John Doe	johndoe@example.com	Internet	2023-01-10
3	Mike Johnson	mikejohnson@example.com	Internet	2023-01-05
5	David Wilson	davidwilson@example.com	Internet	2023-01-12

Query #6

```
    SELECT
      c.customer_id,
      c.customer_name,
      SUM(p.payment_amount) AS total_payment_amount,
      CASE
        WHEN SUM(p.payment_amount) > 100 THEN 'Good'
        WHEN SUM(p.payment_amount) > 50 THEN 'Average'
        ELSE 'Poor'
      END AS payment_history
    FROM payments p
    JOIN customers c
    ON p.customer_id = c.customer_id
    GROUP BY c.customer_name, c.customer_id;
```

customer_id	customer_name	total_payment_amount	payment_history
1	John Doe	90.00	Average
2	Jane Smith	65.00	Average
3	Mike Johnson	155.00	Good
4	Emily Brown	45.00	Poor
5	David Wilson	125.00	Good

Day 3: Advanced Joins, Nested Queries, and Advanced Subqueries



Objective

Dive deeper into

Dive deeper into SQL joins, nested queries, and advanced subqueries to enhance your data retrieval and manipulation skills. Understand how to perform complex joins, utilize nested queries to extract data from multiple tables, and leverage advanced subqueries for advanced filtering and data transformation.

Part 1: Advanced Joins

In this part, we will explore advanced join techniques to enhance your data retrieval capabilities. We'll dive into self joins and cross joins. Self joins are useful when comparing records within the same table. Cross joins create a Cartesian product of two tables. By understanding these advanced join types, you'll have more flexibility in combining data from multiple tables and solving complex data retrieval problems.

Self Join

A self join is a technique used to join a table with itself. It is useful when you need to compare records within the same table.

```
SELECT SELECT
    t1.column,
    t2.column
FROM table
    t1
JOIN table t2
ON t1.key = t2.key;
```

Cross Join

A cross join, also known as a Cartesian join, combines each row from the first table with every row from the second table. It results in a Cartesian product of both tables.

```
SELECT *SELECT *
FROM
    table1
CROSS JOIN
    table2;
```

Part 2: Nested Queries

A nested select statement, also known as a subquery, is a query nested within another query. It can be used in the SELECT, FROM, WHERE, or HAVING clause to

retrieve data based on specific conditions.

```
SELECT SELECT
    column1
FROM
    table1
WHERE
    column2 IN (SELECT column3 FROM table2);
```

Part 3: Advanced Subqueries

In this part, we will uncover the potential of advanced subqueries. We'll explore correlated subqueries, which enable dynamic filtering and calculations by referencing columns from the outer query. Scalar subqueries allow us to retrieve a single value as an expression within the SELECT clause or use it within a WHERE clause. We'll also dive into derived tables, which are subqueries used as virtual tables within the main query, allowing us to perform complex calculations or filtering before joining or further processing the data. Understanding advanced subqueries will enhance your ability to perform sophisticated data analysis and manipulation tasks in SQL.

Correlated Subquery

A correlated subquery is a subquery that refers to a column from the outer query. The subquery is evaluated for each row of the outer query, allowing for more dynamic and context-dependent filtering or calculations.

```
SELECT SELECT
    column1
FROM
    table1 t1
WHERE column2 = (SELECT MAX(column2) FROM table2 WHERE t1.column3 = table2.column3);
```

Scalar Subquery

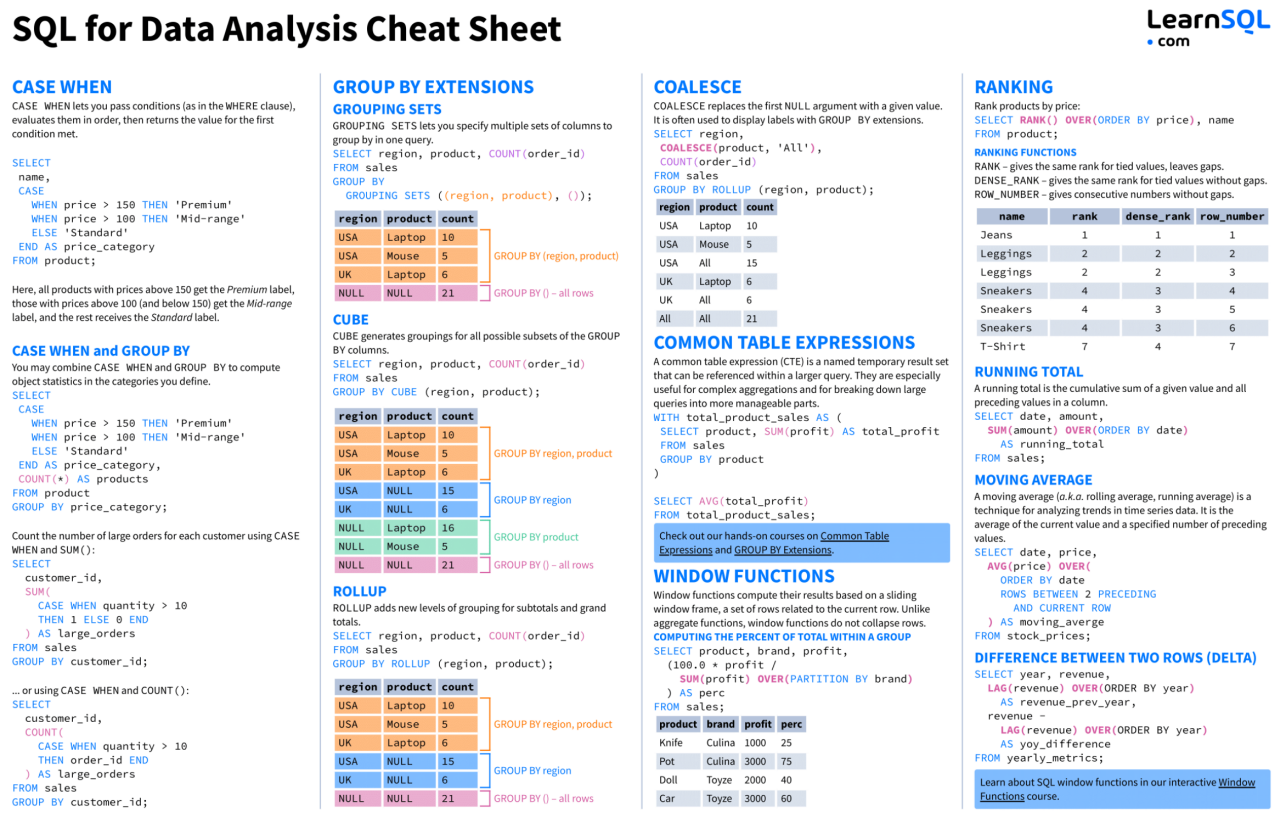
A scalar subquery is a subquery that returns a single value and can be used as an expression in the SELECT clause or within a WHERE clause.

```
SELECT SELECT
    column1,
    (SELECT MAX(column2) FROM table2) AS max_value
FROM
    table1;
```

Derived Table

A derived table, also known as an inline view, is a subquery that is used as a virtual table within the main query. It allows complex calculations or filtering to be performed before joining or further processing the data.

```
SELECT SELECT
    dt.column1
FROM
    (SELECT column1, column2 FROM table1) AS dt
JOIN
    table2 ON dt.column2 = table2.column2;
```



SQL for Data Analysis Cheat Sheet

LearnSQL.com

CASE WHEN

CASE WHEN lets you pass conditions (as in the WHERE clause), evaluates them in order, then returns the value for the first condition met.

```
SELECT
 name,
 CASE
   WHEN price > 150 THEN 'Premium'
   WHEN price > 100 THEN 'Mid-range'
   ELSE 'Standard'
 END AS price_category
FROM product;
```

Here, all products with prices above 150 get the *Premium* label, those with prices above 100 (and below 150) get the *Mid-range* label, and the rest receives the *Standard* label.

CASE WHEN and GROUP BY

You may combine CASE WHEN and GROUP BY to compute object statistics in the categories you define.

```
SELECT
 CASE
   WHEN price > 150 THEN 'Premium'
   WHEN price > 100 THEN 'Mid-range'
   ELSE 'Standard'
 END AS price_category,
 COUNT(*) AS products
FROM product
GROUP BY price_category;
```

Count the number of large orders for each customer using CASE WHEN and SUM():

```
SELECT
  customer_id,
  SUM(
    CASE WHEN quantity > 10
     THEN 1 ELSE 0 END
  ) AS large_orders
FROM sales
GROUP BY customer_id;
```

... or using CASE WHEN and COUNT():

```
SELECT
  customer_id,
  COUNT(
    CASE WHEN quantity > 10
     THEN order_id END
  ) AS large_orders
FROM sales
GROUP BY customer_id;
```

GROUP BY EXTENSIONS

GROUPING SETS

GROUPING SETS lets you specify multiple sets of columns to group by in one query.

```
SELECT region, product, COUNT(order_id)
FROM sales
GROUP BY
  GROUPING SETS ((region, product), ());
```

region	product	count	
USA	Laptop	10	GROUP BY (region, product)
USA	Mouse	5	
UK	Laptop	6	
NULL	NULL	21	GROUP BY () – all rows

CUBE

CUBE generates groupings for all possible subsets of the GROUP BY columns.

```
SELECT region, product, COUNT(order_id)
FROM sales
GROUP BY CUBE (region, product);
```

region	product	count	
USA	Laptop	10	GROUP BY region, product
USA	Mouse	5	
UK	Laptop	6	
USA	NULL	15	GROUP BY region
UK	NULL	6	
NULL	Laptop	16	GROUP BY product
NULL	Mouse	5	
NULL	NULL	21	GROUP BY () – all rows

ROLLUP

ROLLUP adds new levels of grouping for subtotals and grand totals.

```
SELECT region, product, COUNT(order_id)
FROM sales
GROUP BY ROLLUP (region, product);
```

region	product	count	
USA	Laptop	10	GROUP BY region, product
USA	Mouse	5	
UK	Laptop	6	
USA	NULL	15	GROUP BY region
UK	NULL	6	
NULL	NULL	21	GROUP BY () – all rows

COALESCE

COALESCE replaces the first NULL argument with a given value. It is often used to display labels with GROUP BY extensions.

```
SELECT region,
 COALESCE(product, 'All'),
 COUNT(order_id)
FROM sales
GROUP BY ROLLUP (region, product);
```

region	product	count
USA	Laptop	10
USA	Mouse	5
USA	All	15
UK	Laptop	6
UK	All	6
All	All	21

COMMON TABLE EXPRESSIONS

A common table expression (CTE) is a named temporary result set that can be referenced within a larger query. They are especially useful for complex aggregations and for breaking down large queries into more manageable parts.

```
WITH total_product_sales AS (
 SELECT product, SUM(profit) AS total_profit
 FROM sales
 GROUP BY product
)

SELECT AVG(total_profit)
FROM total_product_sales;
```

Check out our hands-on courses on Common Table Expressions and GROUP BY Extensions.

WINDOW FUNCTIONS

Window functions compute their results based on a sliding window frame, a set of rows related to the current row. Unlike aggregate functions, window functions do not collapse rows.

COMPUTING THE PERCENT OF TOTAL WITHIN A GROUP

```
SELECT product, brand, profit,
  (100.0 * profit /
    SUM(profit) OVER(PARTITION BY brand)
  ) AS perc
FROM sales;
```

product	brand	profit	perc
Knife	Culina	1000	25
Pot	Culina	3000	75
Doll	Toyze	2000	40
Car	Toyze	3000	60

RANKING

Rank products by price:

```
SELECT RANK() OVER(ORDER BY price), name
FROM product;
```

RANKING FUNCTIONS

RANK – gives the same rank for tied values, leaves gaps.
DENSE_RANK – gives the same rank for tied values without gaps.
ROW_NUMBER – gives consecutive numbers without gaps.

name	rank	dense_rank	row_number
Jeans	1	1	1
Leggings	2	2	2
Leggings	2	2	3
Sneakers	4	3	4
Sneakers	4	3	5
Sneakers	4	3	6
T-Shirt	7	4	7

RUNNING TOTAL

A running total is the cumulative sum of a given value and all preceding values in a column.

```
SELECT date, amount,
  SUM(amount) OVER(ORDER BY date)
    AS running_total
FROM sales;
```

MOVING AVERAGE

A moving average (*a.k.a.* rolling average, running average) is a technique for analyzing trends in time series data. It is the average of the current value and a specified number of preceding values.

```
SELECT date, price,
  AVG(price) OVER(
    ORDER BY date
    ROWS BETWEEN 2 PRECEDING
      AND CURRENT ROW
  ) AS moving_averge
FROM stock_prices;
```

DIFFERENCE BETWEEN TWO ROWS (DELTA)

```
SELECT year, revenue,
  LAG(revenue) OVER(ORDER BY year)
    AS revenue_prev_year,
  revenue -
    LAG(revenue) OVER(ORDER BY year)
    AS yoy_difference
FROM yearly_metrics;
```

Learn about SQL window functions in our interactive Window Functions course.

Case Study: Sales Data Analysis for an E-commerce Platform

In this case study, we will analyze sales data for an e-commerce platform. The dataset consists of three tables: **orders**, **customers**, and **products**. Our objective is to gain insights into customer purchasing behavior, product performance, and sales trends. We will utilize advanced joins, nested queries, and advanced subqueries to combine data from multiple tables, extract specific information, and perform complex filtering and calculations. By applying these techniques, we will analyze sales performance by customer and product, identify top-selling products within categories, and determine average order value. This case study will provide hands-on experience in analyzing sales data using advanced SQL techniques.

Questions

1. Retrieve the customer names and their corresponding orders.
2. Find the total quantity and revenue generated from each product category.
3. Retrieve the top-selling products in each category.
4. Retrieve the average order value for each customer.
5. Retrieve the customers who have made more than the average order quantity.

Complete the case study here

Schema SQL

```
CREATE TABLE customers (CREATE TABLE customers (
  customer_id INT PRIMARY KEY,
  customer_name VARCHAR(50),
  email VARCHAR(100),
  location VARCHAR(100)
);

CREATE TABLE products (
  product_id INT PRIMARY KEY,
  product_name VARCHAR(100),
  category VARCHAR(50),
  price DECIMAL(10, 2)
);

CREATE TABLE orders (
  order_id INT PRIMARY KEY,
  customer_id INT,
  product_id INT,
  quantity INT,
  order_date DATE,
  FOREIGN KEY (customer_id) REFERENCES customers(customer_id),
  FOREIGN KEY (product_id) REFERENCES products(product_id)
);

INSERT INTO customers (customer_id, customer_name, email, location)
VALUES
  (1, 'John Doe', 'johndoe@example.com', 'New York'),
  (2, 'Jane Smith', 'janesmith@example.com', 'Los Angeles'),
  (3, 'Mike Johnson', 'mikejohnson@example.com', 'Chicago'),
  (4, 'Emily Brown', 'emilybrown@example.com', 'Houston'),
  (5, 'David Wilson', 'davidwilson@example.com', 'Miami');

INSERT INTO products (product_id, product_name, category, price)
VALUES
  (1, 'iPhone 12', 'Electronics', 999.99),
```

```
  (2, 'Samsung Galaxy S21', 'Electronics', 899.99),
  (3, 'Nike Air Max', 'Fashion', 129.99),
  (4, 'Sony PlayStation 5', 'Gaming', 499.99),
  (5, 'MacBook Pro', 'Electronics', 1499.99);

INSERT INTO orders (order_id, customer_id, product_id, quantity, order_date)
VALUES
  (1, 1, 1, 2, '2023-01-01'),
  (2, 2, 3, 1, '2023-01-02'),
  (3, 3, 2, 3, '2023-01-03'),
  (4, 4, 4, 1, '2023-01-04'),
  (5, 5, 1, 1, '2023-01-05'),
  (6, 1, 3, 2, '2023-01-06'),
  (7, 2, 2, 1, '2023-01-07'),
  (8, 3, 5, 1, '2023-01-08'),
  (9, 4, 4, 2, '2023-01-09'),
  (10, 5, 1, 3, '2023-01-10');

/*/*
## Questions
1. Retrieve the customer names and their corresponding orders.
2. Find the total quantity and revenue generated from each product category.
3. Retrieve the top-selling products in each category.
4. Retrieve the average order value for each customer.
5. Retrieve the customers who have made more than the average order quantity.
*/



-- 1. Retrieve the customer names and their corresponding orders.

SELECT c.customer_name,p.product_name, o.quantity
FROM products p
JOIN orders o
ON p.product_id = o.product_id
JOIN customers c
ON c.customer_id = o.customer_id;

-- 2. Find the total quantity and revenue generated from each product category.

SELECT p.category,
      SUM(o.product_id) AS total_quantity,
      SUM(price*quantity) AS total_revenue
FROM products p
JOIN orders o
ON p.product_id = o.product_id
GROUP BY p.category;

-- 3. Retrieve the top-selling products in each category.
```

```
WITH ProductTotalQuantity AS (
  SELECT p.category, p.product_name, SUM(o.quantity) AS total_quantity
  FROM products p
  JOIN orders o ON p.product_id = o.product_id
  GROUP BY p.category, p.product_name
)
SELECT category, product_name, total_quantity
FROM ProductTotalQuantity
WHERE (category, total_quantity) IN (
  SELECT category, MAX(total_quantity)
  FROM ProductTotalQuantity
  GROUP BY category
);

-- 4. Retrieve the average order value for each customer.

SELECT c.customer_id,
      c.customer_name,
      ROUND(avg(price*quantity),2) as average_order_value
FROM products p
JOIN orders o
ON p.product_id = o.product_id
JOIN customers c
ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.customer_name
ORDER BY average_order_value DESC;

-- 5. Retrieve the customers who have made more than the average order quantity.
SELECT c.customer_id,
      c.customer_name
FROM customers c
JOIN
(SELECT customer_id, AVG(quantity) AS avg_quantity
FROM orders
GROUP BY customer_id) as avg_orders
ON c.customer_id = avg_orders.customer_id
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.quantity > avg_quantity;
```

1. Retrieve the customer names and their corresponding orders.

-- 1. Retrieve the customer names and their corresponding orders.-- 1. Retrieve the customer names and their corresponding orders.

```
SELECT c.customer_name,p.product_name, o.quantity
FROM products p
JOIN orders o
```

```
ON p.product_id = o.product_id
JOIN customers c
ON c.customer_id = o.customer_id;
```

- The goal of this query is to obtain a list of customer names along with the names of the products they ordered and the quantity of each product.
- To achieve this, the query involves joining three tables: products, orders, and customers.
- The JOIN clause connects the products and orders tables based on the common column product_id, linking each product to its corresponding order.
- Then, it connects the resulting dataset (combining products and orders) with the customers table, using the customer_id column to associate each order with the respective customer.
- The SELECT statement specifies the columns to be retrieved in the result set:
- c.customer_name: This column contains the names of the customers.
- p.product_name: It contains the names of the products.
- o.quantity: This column represents the quantity of each product ordered.
- The query combines information from these tables to provide a comprehensive view of customer names, product names, and the corresponding quantities for each order.
- The result set will include multiple rows, each containing the customer's name, the name of the product they ordered, and the quantity of that product. This information can help analyze customer preferences and order details.

2. Find the total quantity and revenue generated from each product category.

```
-- 2. Find the total quantity and revenue generated from each product category.-- 2. Find the total quantity and revenue generated from each product category.

SELECT p.category,
    SUM(o.product_id) AS total_quantity,
    SUM(price*quantity) AS total_revenue
FROM products p
JOIN orders o
ON p.product_id = o.product_id
GROUP BY p.category;
```

- The objective of this query is to calculate the total quantity and revenue generated for each product category.
- To achieve this, the query involves joining the products and orders tables based on the common column product_id, linking each product to its

corresponding order.

- The SELECT statement specifies the columns to be retrieved in the result set:
- p.category: This column contains the product categories.
- SUM(o.product_id) AS total_quantity: It calculates the total quantity sold by summing up the product_id values from the orders table. Note that this is an unusual way to calculate total quantity and is not accurate for this purpose. Typically, you should sum the quantity column from the orders table to get the total quantity sold.
- SUM(price*quantity) AS total_revenue: This calculates the total revenue generated by summing up the product of price and quantity for each order. It provides the revenue for each product category.
- The GROUP BY clause groups the result set by the p.category column, which is the product category. This allows the aggregation functions (SUM) to calculate totals for each unique product category.
- The result set will include multiple rows, each representing a product category. For each category, it will display the total quantity (though this is calculated incorrectly) and the total revenue generated. Please note that the calculation for total quantity may not be accurate in this query, as it sums product_id values instead of the actual quantities.

3. Retrieve the top-selling products in each category.

```
-- 3. Retrieve the top-selling products in each category.-- 3. Retrieve the top-selling products in each category.

WITH ProductTotalQuantity AS (
  SELECT p.category, p.product_name, SUM(o.quantity) AS total_quantity
  FROM products p
  JOIN orders o ON p.product_id = o.product_id
  GROUP BY p.category, p.product_name
)
SELECT category, product_name, total_quantity
FROM ProductTotalQuantity
WHERE (category, total_quantity) IN (
  SELECT category, MAX(total_quantity)
  FROM ProductTotalQuantity
  GROUP BY category
);
```

This query aims to identify the top-selling products within each product category.

- The query uses a Common Table Expression (CTE) named ProductTotalQuantity to calculate the total quantity sold for each product within its respective category.

- It joins the products and orders tables based on the common column product_id.
- It groups the result by both category and product_name, ensuring that the total quantity is calculated for each product within its category.
- The result of this CTE includes three columns: category, product_name, and total_quantity (total quantity sold for each product).
- The main query then selects columns from the ProductTotalQuantity CTE:
- category: This column contains the product category.
- product_name: It contains the names of the products.
- total_quantity: This column represents the total quantity sold for each product within its category.
- The WHERE clause filters the results to include only those rows where the combination of category and total_quantity matches the maximum total quantity within each category. This effectively identifies the top-selling products in each category.
- The result set will include the product category, the names of the top-selling products in each category, and the total quantity sold for each of those products. This information helps identify which products are the most popular within their respective categories.

4. Retrieve the average order value for each customer.

```
-- 4. Retrieve the average order value for each customer.-- 4. Retrieve the average order value for each customer.

SELECT c.customer_id,
    c.customer_name,
    ROUND(avg(price*quantity),2) as average_order_value
FROM products p
JOIN orders o
ON p.product_id = o.product_id
JOIN customers c
ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.customer_name
ORDER BY average_order_value DESC;
```

- The goal of this query is to calculate the average order value for each customer. An average order value is the average amount spent by each customer in their orders.
- The query involves joining three tables: products, orders, and customers, to combine information about products, orders, and customer details.
- The JOIN clauses link the tables based on common columns (product_id and customer_id) to associate products with orders and customers.

- The SELECT statement specifies the columns to be retrieved in the result set:
- c.customer_id: This column contains the unique customer IDs.
- c.customer_name: It contains the names of the customers.
- ROUND(avg(price*quantity), 2) as average_order_value: This calculates the average order value for each customer by first multiplying the price and quantity for each order and then calculating the average (avg). The ROUND function is used to round the result to two decimal places.
- The GROUP BY clause groups the result set by both c.customer_id and c.customer_name, ensuring that the average order value is calculated for each individual customer.
- The ORDER BY clause arranges the result set in descending order of average order value (average_order_value), so the customers with the highest average order values appear at the top of the list.
- The result set will include multiple rows, each representing a customer. For each customer, it displays their customer ID, customer name, and the average order value. This information helps identify customers with higher average spending.

5. Retrieve the customers who have made more than the average order quantity.

```
-- 5. Retrieve the customers who have made more than the average order quantity.-- 5. Retrieve the customers who have made more than the average order quantity.
SELECT c.customer_id,
    c.customer_name
FROM customers c
JOIN
(SELECT customer_id, AVG(quantity) AS avg_quantity
FROM orders
GROUP BY customer_id) as avg_orders
ON c.customer_id = avg_orders.customer_id
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.quantity > avg_quantity;
```

- This query aims to find customers who have placed orders with a quantity greater than the average order quantity across all customers.
- The query involves joining the customers table with a subquery (avg_orders) that calculates the average quantity of orders for each customer.
- The subquery calculates the average quantity (AVG(quantity) AS avg_quantity) for each customer by grouping the orders table by customer_id.
- The main query then joins the customers table with the subquery (avg_orders) on the customer_id to associate customers with their average order quantities.

- The WHERE clause filters the results to include only those rows where the quantity of orders placed by each customer (o.quantity) is greater than their respective average order quantity (avg_quantity).
- The result set will include customer IDs and customer names of those customers who have placed orders with quantities exceeding the average order quantity. This helps identify customers who are purchasing more than the average.

Schema (MySQL v8.0)**Schema (MySQL v8.0)**

```
CREATE TABLE customers (
  customer_id INT PRIMARY KEY,
  customer_name VARCHAR(50),
  email VARCHAR(100),
  location VARCHAR(100)
);

CREATE TABLE products (
  product_id INT PRIMARY KEY,
  product_name VARCHAR(100),
  category VARCHAR(50),
  price DECIMAL(10, 2)
);

CREATE TABLE orders (
  order_id INT PRIMARY KEY,
  customer_id INT,
  product_id INT,
  quantity INT,
  order_date DATE,
  FOREIGN KEY (customer_id) REFERENCES customers(customer_id),
  FOREIGN KEY (product_id) REFERENCES products(product_id)
);

INSERT INTO customers (customer_id, customer_name, email, location)
VALUES
  (1, 'John Doe', 'johndoe@example.com', 'New York'),
  (2, 'Jane Smith', 'janesmith@example.com', 'Los Angeles'),
  (3, 'Mike Johnson', 'mikejohnson@example.com', 'Chicago'),
  (4, 'Emily Brown', 'emilybrown@example.com', 'Houston'),
  (5, 'David Wilson', 'davidwilson@example.com', 'Miami');

INSERT INTO products (product_id, product_name, category, price)
VALUES
  (1, 'iPhone 12', 'Electronics', 999.99),
  (2, 'Samsung Galaxy S21', 'Electronics', 899.99),
  (3, 'Nike Air Max', 'Fashion', 129.99),
```

```
    (4, 'Sony PlayStation 5', 'Gaming', 499.99),
    (5, 'MacBook Pro', 'Electronics', 1499.99);

INSERT INTO orders (order_id, customer_id, product_id, quantity, order_date)
VALUES
    (1, 1, 1, 2, '2023-01-01'),
    (2, 2, 3, 1, '2023-01-02'),
    (3, 3, 2, 3, '2023-01-03'),
    (4, 4, 4, 1, '2023-01-04'),
    (5, 5, 1, 1, '2023-01-05'),
    (6, 1, 3, 2, '2023-01-06'),
    (7, 2, 2, 1, '2023-01-07'),
    (8, 3, 5, 1, '2023-01-08'),
    (9, 4, 4, 2, '2023-01-09'),
    (10, 5, 1, 3, '2023-01-10');
```

Query #1

```
SELECT c.customer_name,p.product_name, o.quantity
FROM products p
JOIN orders o
ON p.product_id = o.product_id
JOIN customers c
ON c.customer_id = o.customer_id;
```

customer_name	product_name	quantity
John Doe	iPhone 12	2
Jane Smith	Nike Air Max	1
Mike Johnson	Samsung Galaxy S21	3
Emily Brown	Sony PlayStation 5	1
David Wilson	iPhone 12	1
John Doe	Nike Air Max	2
Jane Smith	Samsung Galaxy S21	1
Mike Johnson	MacBook Pro	1
Emily Brown	Sony PlayStation 5	2
David Wilson	iPhone 12	3

Query #2

```
SELECT p.category,
    SUM(o.product_id) AS total_quantity,
    SUM(price*quantity) AS total_revenue
FROM products p
JOIN orders o
```

```
    ON p.product_id = o.product_id
    GROUP BY p.category;
```

category	total_quantity	total_revenue
Electronics	12	11099.89
Fashion	6	389.97
Gaming	8	1499.97

Query #3

```
    WITH ProductTotalQuantity AS (
      SELECT p.category, p.product_name, SUM(o.quantity) AS total_quantity
      FROM products p
      JOIN orders o ON p.product_id = o.product_id
      GROUP BY p.category, p.product_name
    )
    SELECT category, product_name, total_quantity
    FROM ProductTotalQuantity
    WHERE (category, total_quantity) IN (
      SELECT category, MAX(total_quantity)
      FROM ProductTotalQuantity
      GROUP BY category
    );
```

category	product_name	total_quantity
Electronics	iPhone 12	6
Fashion	Nike Air Max	3
Gaming	Sony PlayStation 5	3

Query #4

```
    SELECT c.customer_id,
           c.customer_name,
           ROUND(avg(price*quantity),2) as average_order_value
    FROM products p
    JOIN orders o
    ON p.product_id = o.product_id
    JOIN customers c
    ON c.customer_id = o.customer_id
    GROUP BY c.customer_id, c.customer_name
    ORDER BY average_order_value DESC;
```

customer_id	customer_name	average_order_value

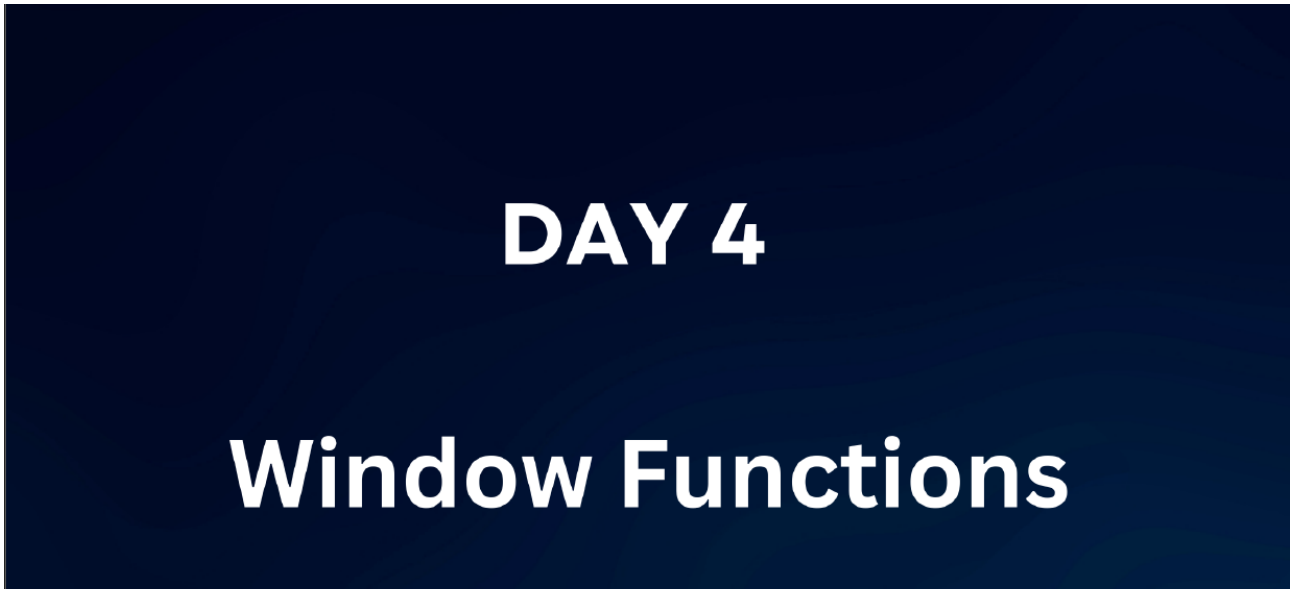
3	Mike Johnson	2099.98
5	David Wilson	1999.98
1	John Doe	1129.98
4	Emily Brown	749.99
2	Jane Smith	514.99

Query #5

```
SELECT c.customer_id,
       c.customer_name
FROM customers c
JOIN
(SELECT customer_id, AVG(quantity) AS avg_quantity
FROM orders
GROUP BY customer_id) as avg_orders
ON c.customer_id = avg_orders.customer_id
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.quantity > avg_quantity;
```

customer_id	customer_name
3	Mike Johnson
4	Emily Brown
5	David Wilson

Day 4: Window Functions



Objective

Learn about window functions in SQL for advanced data analysis. Understand how to perform calculations and aggregations on specific windows or partitions of data. Explore concepts such as ranking, row numbering, aggregating over windows, and more.

Part 1: Introduction to Window Functions

Window functions are SQL functions that perform calculations and aggregations over a set of rows called a “window” or “partition” within a table. They allow you to apply calculations to a subset of data based on specified criteria or order.

```
SELECT SELECT
    column1,
    column2,
  window_function() OVER (PARTITION BY column1 ORDER BY column2)
FROM
    table_name;
```

Part 2: Ranking Functions

Ranking functions allow us to assign a rank or row number to each row within a window based on a specific order or criteria. In this part, we'll explore three ranking functions: **ROW_NUMBER**, **RANK**, and **DENSE_RANK**. These functions are useful for tasks such as identifying top performers, finding the highest or lowest values within a group, and understanding the relative positions of rows within a window. By mastering ranking functions, we can easily analyze and compare data based on specific ranking criteria.

ROW_NUMBER

The ROW_NUMBER() function assigns a unique sequential number to each row within a window.

```
SELECT SELECT
    column1,
    column2,
  ROW_NUMBER() OVER (PARTITION BY column1 ORDER BY column2)
FROM
    table_name;
```

RANK

The RANK() function calculates the rank of each row within a window, with ties receiving the same rank and leaving gaps in the ranking sequence.

```
SELECT SELECT
    column1,
    column2,
  RANK() OVER (PARTITION BY column1 ORDER BY column2)
FROM
    table_name;
```

DENSE_RANK

The DENSE_RANK() function calculates the rank of each row within a window, with ties receiving the same rank but with no gaps in the ranking sequence.

```
SELECT SELECT
    column1,
    column2,
  DENSE_RANK() OVER (PARTITION BY column1 ORDER BY column2)
FROM
    table_name;
```

Part 3: Aggregate Functions with Window

Aggregate functions, such as **SUM**, **AVG**, **MIN**, and **MAX**, are commonly used to perform calculations on groups of rows. In this part, we'll discover how to combine aggregate functions with window functions to perform calculations and aggregations on specific partitions or windows of data. This enables us to calculate running totals, moving averages, and other aggregations based on customized windows within our data. By leveraging aggregate functions with windows, we can gain deeper insights into the patterns and trends within our data.

SUM, AVG, MIN, MAX

Aggregate functions can be used in conjunction with window functions to calculate aggregated values over specific windows or partitions of data.

```
SELECT SELECT
    column1,
    column2,
  SUM(column3) OVER (PARTITION BY column1)
FROM
    table_name;
```

Part 4: Analytical Functions

Analytical functions provide access to data from previous or subsequent rows within a window, allowing us to perform complex analyses and comparisons. In this part, we'll explore two fundamental analytical functions: **LEAD** and **LAG**. These functions enable us to retrieve data from the next or previous rows, which is particularly useful for tasks like calculating differences, identifying trends, and detecting anomalies. Additionally, we'll delve into **FIRST_VALUE** and **LAST_VALUE**, which allow us to extract the first and last values within a window, respectively. Understanding analytical functions empowers us to perform advanced analyses and gain deeper insights into our data.

LEAD and LAG

The LEAD() and LAG() functions access data from subsequent and previous rows within a window, respectively.

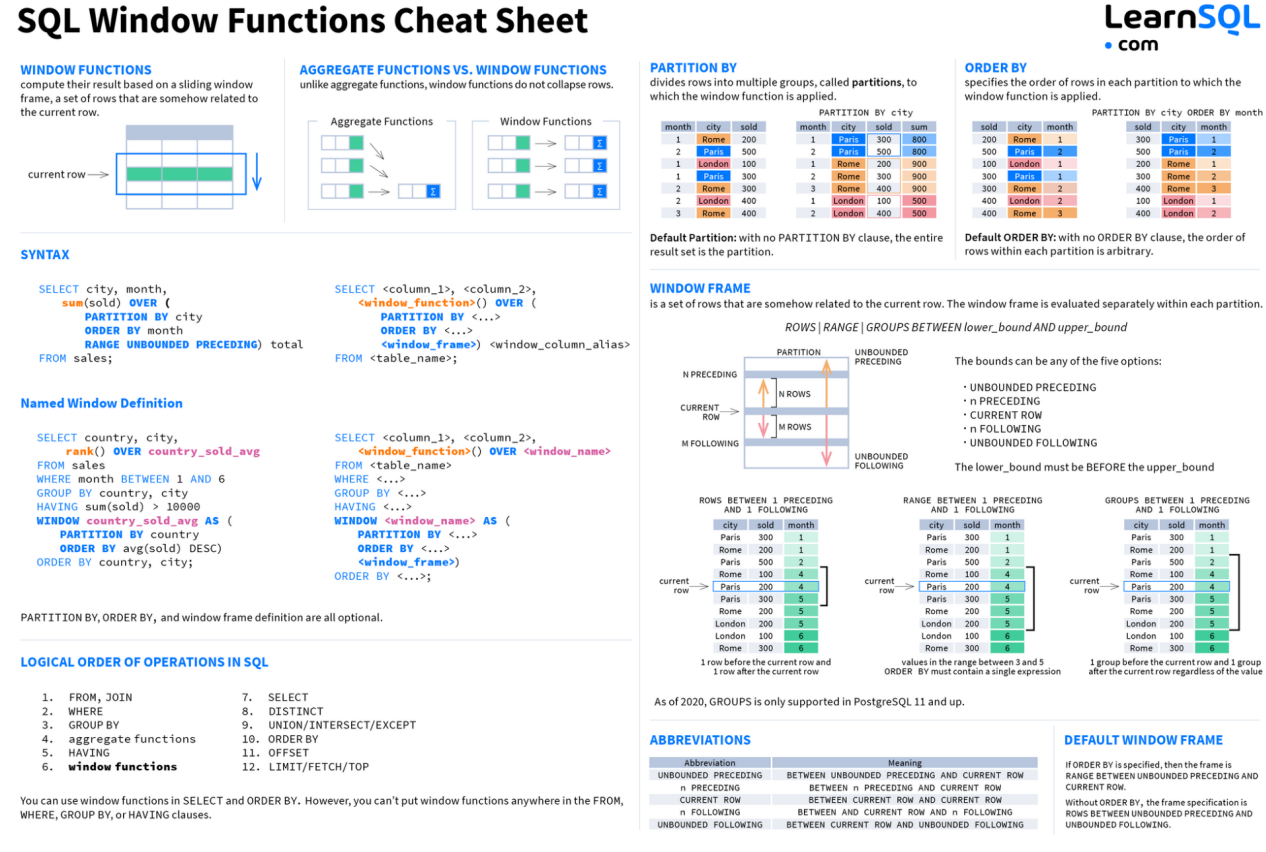
```
SELECT SELECT
    column1,
    column2,
  LEAD(column2) OVER (PARTITION BY column1 ORDER BY column2)
```

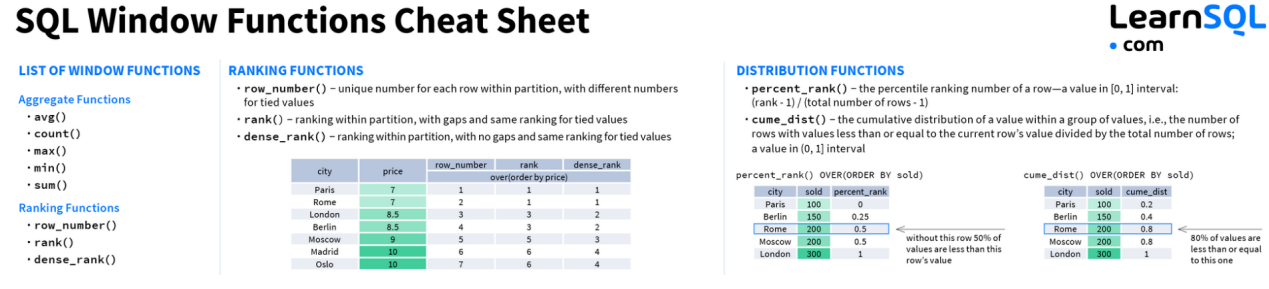
```
FROM
    table_name;
```

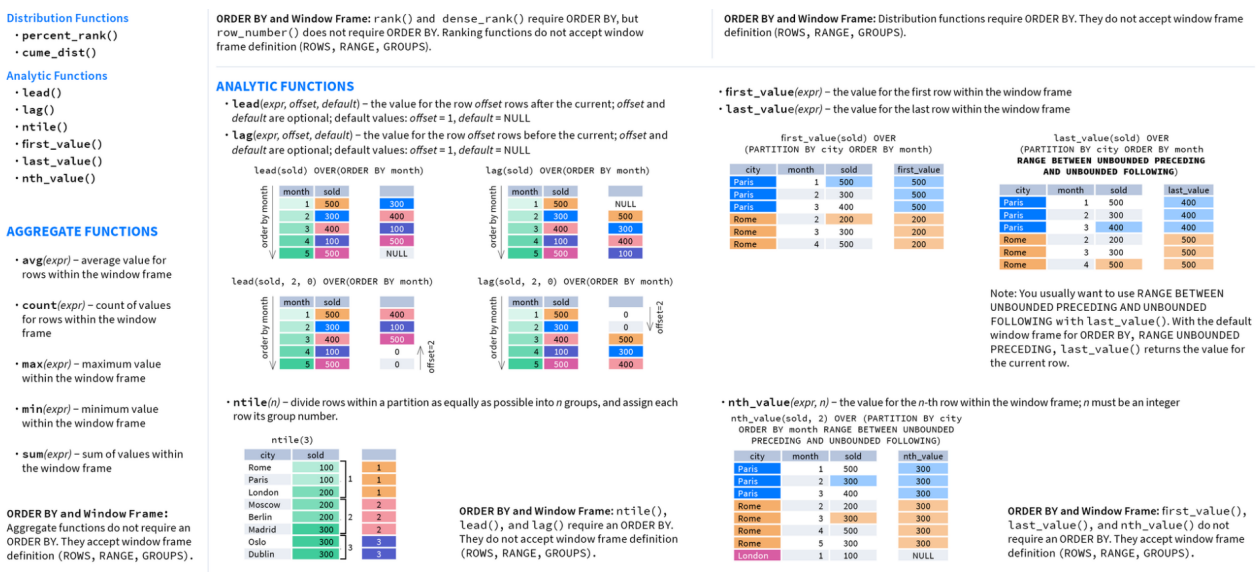
FIRST_VALUE and LAST_VALUE

The FIRST_VALUE() and LAST_VALUE() functions retrieve the first and last values within a window, respectively.

```
SELECT SELECT
    column1,
    column2,
 FIRST_VALUE(column2) OVER (PARTITION BY column1 ORDER BY column2)
FROM
    table_name;
```







Case Study: Sales Analysis with Window Functions

In this case study, we will dive into sales analysis using window functions in MySQL. We have two tables: **products** and **sales**. The **products** table stores information about various products, including the product ID, name, and category. The **sales** table contains data related to sales transactions, including the sale ID, product ID, region, sale date, quantity, and revenue.

Our objective is to gain insights into the sales performance by applying window functions to perform advanced calculations and aggregations on specific partitions or windows of data. We will explore concepts such as calculating cumulative revenue, ranking products within categories, analyzing revenue differences between sales, identifying top-performing regions, and more.

Questions

1. Retrieve the sales revenue for each product, along with the maximum revenue achieved for each product across all sales.
2. Calculate the average revenue for each product, considering only the three most recent sales for each product.
3. Calculate the difference in revenue between each sale and the previous sale for each product, sorted by product and sale date.
4. Retrieve the sales revenue for each product, along with the cumulative revenue for each product over time.
5. Rank the sales regions based on the total revenue generated, and display the top three regions along with their respective total revenue.

Complete the case study here

```
CREATE TABLE products (CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(50),
    category VARCHAR(50)
);

CREATE TABLE sales (
    sale_id INT PRIMARY KEY,
    product_id INT,
    region VARCHAR(50),
    sale_date DATE,
    quantity INT,
    revenue DECIMAL(10, 2),
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);

INSERT INTO products (product_id, product_name, category)
VALUES
    (1, 'Product A', 'Category 1'),
    (2, 'Product B', 'Category 1'),
    (3, 'Product C', 'Category 2'),
    (4, 'Product D', 'Category 3'),
    (5, 'Product E', 'Category 3');

INSERT INTO sales (sale_id, product_id, region, sale_date, quantity, revenue)
VALUES
    (1, 1, 'Region 1', '2023-01-01', 10, 100.00),
    (2, 2, 'Region 1', '2023-01-02', 5, 75.00),
    (3, 3, 'Region 2', '2023-01-02', 8, 120.00),
    (4, 4, 'Region 2', '2023-01-03', 12, 150.00),
    (5, 5, 'Region 1', '2023-01-03', 6, 90.00),
    (6, 1, 'Region 2', '2023-01-04', 15, 200.00),
    (7, 3, 'Region 1', '2023-01-04', 10, 150.00),
    (8, 2, 'Region 2', '2023-01-05', 7, 105.00),
    (9, 4, 'Region 1', '2023-01-05', 9, 135.00),
    (10, 5, 'Region 2', '2023-01-05', 3, 45.00);

/*/*
## Questions

1. Retrieve the sales revenue for each product, along with the maximum revenue achieved for each product across all sales.
2. Calculate the average revenue for each product, considering only the three most recent sales for each product.
3. Calculate the difference in revenue between each sale and the previous sale for each product, sorted by product and sale date.
4. Retrieve the sales revenue for each product, along with the cumulative revenue for each product over time.
```

5. Rank the sales regions based on the total revenue generated, and display the top three regions along with their respective total revenue.

*/

-- 1. Retrieve the sales revenue for each product, along with the maximum revenue achieved for each product across all sales.

```
SELECT
    p.product_name,
    s.revenue AS sales_revenue,
    MAX(s.revenue) OVER (PARTITION BY p.product_id) AS max_revenue
FROM
    products p
JOIN
    sales s
ON
    p.product_id = s.product_id;
```

-- 2. Calculate the average revenue for each product, considering only the three most recent sales for each product.

```
WITH RankedSales AS (
    SELECT
        product_id,
        sale_date,
        revenue,
        ROW_NUMBER() OVER (PARTITION BY product_id ORDER BY sale_date DESC) AS row_num
    FROM
        sales
)
SELECT
    p.product_name,
    AVG(s.revenue) AS average_recent_revenue
FROM
    products p
JOIN
    RankedSales s
ON
    p.product_id = s.product_id
WHERE
    s.row_num <= 3
GROUP BY
    p.product_name;
```

-- 3. Calculate the difference in revenue between each sale and the previous sale for each product, sorted by product and sale date.

```sql
WITH RevenueChanges AS (
    SELECT
        product_id,
        sale_date,
        revenue - LAG(revenue, 1, 0) OVER (PARTITION BY product_id ORDER BY sale_date) AS revenue_change
    FROM
        sales
)
SELECT
    p.product_name,
    s.sale_date,
    s.revenue_change
FROM
    products p
JOIN
    RevenueChanges s
ON
    p.product_id = s.product_id
ORDER BY
    p.product_name,
    s.sale_date;

-- 4. Retrieve the sales revenue for each product, along with the cumulative revenue for each product over time.

SELECT
    p.product_name,
    s.sale_date,
    s.revenue,
    SUM(s.revenue) OVER (PARTITION BY p.product_id ORDER BY s.sale_date) AS cumulative_revenue
FROM
    products p
JOIN
    sales s
ON
    p.product_id = s.product_id
ORDER BY
    p.product_name,
    s.sale_date;


-- 5. Rank the sales regions based on the total revenue generated, and display the top three regions along with their respective total revenue.

SELECT
    region,
    total_region_revenue
FROM (
    SELECT
```

```
        region,
        SUM(revenue) AS total_region_revenue,
        DENSE_RANK() OVER (ORDER BY SUM(revenue) DESC) AS region_rank
    FROM
        sales
    GROUP BY
        region
) AS ranked_regions
WHERE region_rank <= 3;
```

Schema (MySQL v8.0)**Schema (MySQL v8.0)**

```
    CREATE TABLE products (
        product_id INT PRIMARY KEY,
        product_name VARCHAR(50),
        category VARCHAR(50)
    );

    CREATE TABLE sales (
        sale_id INT PRIMARY KEY,
        product_id INT,
        region VARCHAR(50),
        sale_date DATE,
        quantity INT,
        revenue DECIMAL(10, 2),
        FOREIGN KEY (product_id) REFERENCES products(product_id)
    );

    INSERT INTO products (product_id, product_name, category)
    VALUES
        (1, 'Product A', 'Category 1'),
        (2, 'Product B', 'Category 1'),
        (3, 'Product C', 'Category 2'),
        (4, 'Product D', 'Category 3'),
        (5, 'Product E', 'Category 3');

    INSERT INTO sales (sale_id, product_id, region, sale_date, quantity, revenue)
    VALUES
        (1, 1, 'Region 1', '2023-01-01', 10, 100.00),
        (2, 2, 'Region 1', '2023-01-02', 5, 75.00),
        (3, 3, 'Region 2', '2023-01-02', 8, 120.00),
        (4, 4, 'Region 2', '2023-01-03', 12, 150.00),
        (5, 5, 'Region 1', '2023-01-03', 6, 90.00),
        (6, 1, 'Region 2', '2023-01-04', 15, 200.00),
        (7, 3, 'Region 1', '2023-01-04', 10, 150.00),
        (8, 2, 'Region 2', '2023-01-05', 7, 105.00),
        (9, 4, 'Region 1', '2023-01-05', 9, 135.00),
        (10, 5, 'Region 2', '2023-01-05', 3, 45.00);
```

Query #1

```
    SELECT
        p.product_name,
        s.revenue AS sales_revenue,
        MAX(s.revenue) OVER (PARTITION BY p.product_id) AS max_revenue
    FROM
        products p
    JOIN
        sales s
    ON
        p.product_id = s.product_id;
```

product_name	sales_revenue	max_revenue
Product A	100.00	200.00
Product A	200.00	200.00
Product B	75.00	105.00
Product B	105.00	105.00
Product C	120.00	150.00
Product C	150.00	150.00
Product D	150.00	150.00
Product D	135.00	150.00
Product E	90.00	90.00
Product E	45.00	90.00

Query #2

```
    WITH RankedSales AS (
        SELECT
            product_id,
            sale_date,
            revenue,
            ROW_NUMBER() OVER (PARTITION BY product_id ORDER BY sale_date DESC) AS row_num
        FROM
            sales
    )
    SELECT
        p.product_name,
        AVG(s.revenue) AS average_recent_revenue
    FROM
        products p
    JOIN
        RankedSales s
    ON
```

```
    p.product_id = s.product_id
WHERE
    s.row_num <= 3
GROUP BY
    p.product_name;
```

product_name	average_recent_revenue
Product A	150.000000
Product B	90.000000
Product C	135.000000
Product D	142.500000
Product E	67.500000

Query #3

```
WITH RevenueChanges AS (
    SELECT
        product_id,
        sale_date,
        revenue - LAG(revenue, 1, 0) OVER (PARTITION BY product_id ORDER BY sale_date) AS revenue_change
    FROM
        sales
)
SELECT
    p.product_name,
    s.sale_date,
    s.revenue_change
FROM
    products p
JOIN
    RevenueChanges s
ON
    p.product_id = s.product_id
ORDER BY
    p.product_name,
    s.sale_date;
```

product_name	sale_date	revenue_change
Product A	2023-01-01	100.00
Product A	2023-01-04	100.00
Product B	2023-01-02	75.00
Product B	2023-01-05	30.00
Product C	2023-01-02	120.00
Product C	2023-01-04	30.00
Product D	2023-01-03	150.00

Product D	2023-01-05	-15.00
Product E	2023-01-03	90.00
Product E	2023-01-05	-45.00

Query #4

```
SELECT
    p.product_name,
    s.sale_date,
    s.revenue,
    SUM(s.revenue) OVER (PARTITION BY p.product_id ORDER BY s.sale_date) AS cumulative_revenue
FROM
    products p
JOIN
    sales s
ON
    p.product_id = s.product_id
ORDER BY
    p.product_name,
    s.sale_date;
```

product_name	sale_date	revenue	cumulative_revenue
Product A	2023-01-01	100.00	100.00
Product A	2023-01-04	200.00	300.00
Product B	2023-01-02	75.00	75.00
Product B	2023-01-05	105.00	180.00
Product C	2023-01-02	120.00	120.00
Product C	2023-01-04	150.00	270.00
Product D	2023-01-03	150.00	150.00
Product D	2023-01-05	135.00	285.00
Product E	2023-01-03	90.00	90.00
Product E	2023-01-05	45.00	135.00

Query #5

```
SELECT
    region,
    total_region_revenue
FROM (
    SELECT
        region,
        SUM(revenue) AS total_region_revenue,
        DENSE_RANK() OVER (ORDER BY SUM(revenue) DESC) AS region_rank
    FROM
        sales
```

```
    GROUP BY
        region
) AS ranked_regions
WHERE region_rank <= 3;
```

region	total_region_revenue
Region 2	620.00
Region 1	550.00

Day 5: Advanced SQL Queries with Common Table Expressions (CTEs)

Objective

Learn about Common Table Expressions (CTEs), a powerful feature in SQL that allows us to create temporary named result sets. Understand how to use CTEs to write complex and efficient SQL queries. Explore concepts such as recursive CTEs, multiple CTEs, and inline views.

Part 1: Introduction to Common Table Expressions (CTEs)

In this part, we will explore Common Table Expressions (CTEs), a powerful feature in SQL that allows us to create temporary named result sets. CTEs provide a way to break down complex queries into smaller, more manageable parts, improving query readability and maintainability. We will learn the syntax of CTEs and understand how to use them to write efficient and concise SQL queries.

```
WITH cte_name (column1, column2, ...)WITH cte_name (column1, column2, ...)
AS (
    SELECT column1, column2, ...
    FROM table_name
    WHERE condition
)
SELECT column1, column2, ...
FROM cte_name;
```

Part 2: Recursive CTEs

Recursive CTEs are a fascinating aspect of CTEs that enable us to perform hierarchical or graph-based queries. In this part, we will delve into the world of recursive CTEs and learn how they can be used to traverse tree structures, find connected components in graphs, and perform other iterative operations. We will understand the recursive CTE syntax, including the anchor member and recursive member, and see practical examples of their usage.

```
WITH RECURSIVE cte_name (column1, column2, ...)WITH RECURSIVE cte_name (column1, column2, ...)
AS (
    -- Anchor member
    SELECT column1, column2, ...
    FROM table_name
    WHERE condition
    UNION ALL
    -- Recursive member
    SELECT column1, column2, ...
    FROM cte_name
    WHERE condition
)
SELECT column1, column2, ...
FROM cte_name;
```

Part 3: Multiple CTEs

Multiple CTEs allow us to define and use multiple CTEs within a single SQL statement. In this part, we will explore the concept of multiple CTEs and understand how they can help break down complex queries into smaller logical parts. We will learn how to define and reference multiple CTEs in a query and discover their usefulness in improving query organization and reusability.

```
WITH cte1 (column1, column2, ...)WITH cte1 (column1, column2, ...)
AS (
    SELECT column1, column2, ...
    FROM table1
),
cte2 (column3, column4, ...)
AS (
    SELECT column3, column4, ...
    FROM table2
)
SELECT column1, column2, column3, column4, ...
FROM cte1
JOIN cte2 ON condition;
```

Part 4: Inline Views

Inline views, also known as derived tables, provide a powerful way to treat the result of a subquery as a temporary table within a SQL statement. In this part, we will explore the concept of inline views and understand how they can be used to manipulate and reference intermediate result sets in the main query. We will learn the syntax of inline views and see practical examples of their application.

```
SELECT column1, column2, ...SELECT column1, column2, ...
FROM (
    SELECT column1, column2, ...
    FROM table_name
    WHERE condition
) AS inline_view_name;
```

Part 5: Combining CTEs and Inline Views

Combining CTEs and inline views allows us to create even more complex SQL queries. In this part, we will learn how to leverage the power of both CTEs and inline views together to break down complex queries, create temporary result sets, and perform advanced analysis. We will explore examples demonstrating how CTEs and inline views can work harmoniously to enhance query readability and maintainability.

```
WITH cte_name (column1, column2, ...)WITH cte_name (column1, column2, ...)
AS (
    SELECT column1, column2, ...
    FROM table1
),
inline_view_name AS (
    SELECT column3, column4, ...
    FROM table2
    WHERE condition
)
SELECT column1, column2, column3, column4, ...
FROM cte_name
JOIN inline_view_name ON condition;
```

Case Study: Employee Management System

Questions

1. Calculate the total salary expenditure for each department and display the departments in descending order of the total salary expenditure.
2. Retrieve the employees who have at least two subordinates.
3. Calculate the average salary for each department, considering only employees with a salary greater than the department average.
4. Find the employees who have the highest salary in their respective departments.
5. Calculate the running total of salaries for each department, ordered by department ID and employee name.

Complete the case study here

Calculate the total salary expenditure for each department and display the departments in descending order of the total salary expenditure.

```
WITH department_salary ASWITH department_salary AS
(
SELECT
    department_id,
    SUM(salary) as total_salary
FROM employees
GROUP BY department_id
)
SELECT
     d1.department_name,
     d2.total_salary as total_salary_expenditure
FROM departments d1
JOIN department_salary d2
ON d1.department_id = d2.department_id
```

Sure, let's break down the SQL query step by step:

Step 1:

```
WITH department_salary ASWITH department_salary AS
(
  SELECT
      department_id,
      SUM(salary) as total_salary
  FROM employees
  GROUP BY department_id
)
```

In this step, we are using a Common Table Expression (CTE) named department_salary. This CTE calculates the total salary expenditure for each department. It does so by selecting the department_id from the employees table and using the SUM function to sum up the salary for each department. The result is grouped by department_id, so you get the total salary for each department.

Step 2:

```
SELECT SELECT
     d1.department_name,
     d2.total_salary as total_salary_expenditure
FROM departments d1
```

```
JOIN department_salary d2
ON d1.department_id = d2.department_id
```

In this step, we are querying the result of the CTE department_salary. We want to retrieve the department names and their respective total salary expenditures.

- We select d1.department_name from the departments table as the department name.
- We select d2.total_salary from the CTE result department_salary as the total salary expenditure.
- We use a JOIN clause to join the departments table (d1) with the CTE result (d2) using the department_id as the common column.

The result is a list of department names along with their total salary expenditures, which is calculated using the CTE. This gives you the total salary expenditure for each department, and it is displayed as a result set.

Another solution

```
SELECTSELECT
    d.department_name,
    SUM(e.salary) AS total_salary_expenditure
FROM
    departments d
LEFT JOIN
    employees e ON d.department_id = e.department_id
GROUP BY
    d.department_name
ORDER BY
    total_salary_expenditure DESC;
```

Your SQL query is written to retrieve the total salary expenditure for each department, sorting the results in descending order of total salary expenditure. Here's an explanation of your query:

1. You are selecting two columns in your result set:
- department_name from the departments table.
- The sum of salary from the employees table, aliased as total_salary_expenditure.
2. You are using a LEFT JOIN to combine data from the departments and employees tables based on the department_id column. This allows you to associate each department with its employees.
3. You are grouping the results by department_name using the GROUP BY clause. This groups all the employees within each department together.

4. Finally, you are ordering the results in descending order of total_salary_expenditure using the ORDER BY clause.

The result of this query will give you a list of departments along with the total salary expenditure for each department, sorted from the highest expenditure to the lowest.

Query #1

```
WITH department_salary AS
(
SELECT
    department_id,
    SUM(salary) as total_salary
FROM employees
GROUP BY department_id
)
SELECT
    d1.department_name,
    d2.total_salary as total_salary_expenditure
FROM departments d1
JOIN department_salary d2
ON d1.department_id = d2.department_id
ORDER BY total_salary_expenditure DESC;
```

department_name	total_salary_expenditure
Sales	15200.00
Finance	11500.00
IT	9900.00
Marketing	8800.00
HR	5000.00

Query #2

```
SELECT
    d.department_name,
    SUM(e.salary) AS total_salary_expenditure
FROM
    departments d
LEFT JOIN
    employees e ON d.department_id = e.department_id
GROUP BY
```

```
    d.department_name
ORDER BY
    total_salary_expenditure DESC;
```

department_name	total_salary_expenditure
Sales	15200.00
Finance	11500.00
IT	9900.00
Marketing	8800.00
HR	5000.00

Question 2: Retrieve the employees who have at least two subordinates.

```
-- Question 2: Retrieve the employees who have at least two subordinates.-- Question 2: Retrieve the employees who have at least two subordinates.
-- Query 1
SELECT
    e1.employee_name AS employee_with_subordinates,
    COUNT(*) AS num_subordinates
FROM
    employees e1
INNER JOIN
    employees e2 ON e1.employee_id = e2.manager_id
GROUP BY
    e1.employee_name
HAVING
    COUNT(*) >= 2;



-- --- -----
--   Query 2

WITH subordinate_count AS (
    SELECT manager_id, COUNT(*) AS num_subordinates
    FROM employees
    GROUP BY manager_id
)
SELECT e.employee_name, sc.num_subordinates
FROM employees e
JOIN subordinate_count sc ON e.employee_id = sc.manager_id
WHERE sc.num_subordinates >= 2;
```

Query 1

- In this query, you are joining the employees table with itself using aliases e1 and e2 to create a self-join.
- You are counting the number of subordinates for each employee (employee_with_subordinates) by counting the matching rows in the self-join.
- The GROUP BY clause groups the results by the names of employees.
- The HAVING clause filters the results to only include employees with at least two subordinates (COUNT(*) >= 2).

Query2

- In this query, you first create a Common Table Expression (CTE) named subordinate_count that calculates the number of subordinates for each manager.
- Then, you select the employee_name and the num_subordinates from the employees table, joining it with the subordinate_count CTE on the employee_id and manager_id.
- The WHERE clause filters the results to only include employees with at least two subordinates (sc.num_subordinates >= 2).

Both queries achieve the same result of identifying employees with at least two subordinates, but they use slightly different approaches.

Question 3: Calculate the average salary for each department, considering only employees with a salary greater than the department average.

-- Question 3: Calculate the average salary for each department, considering only employees with a salary greater than the department average.-- Question 3: Calculate the average salary for each department, considering only employees with a salary greater than the department average.

```
WITH DepartmentAvgSalary AS (
  SELECT
    d.department_id,
    AVG(e.salary) AS avg_salary
  FROM
    departments d
  LEFT JOIN
    employees e ON d.department_id = e.department_id
  GROUP BY
    d.department_id
)
```

```
SELECT
    d.department_name,
    ROUND(AVG(e.salary),2) AS average_salary
FROM
    departments d
LEFT JOIN
    employees e ON d.department_id = e.department_id
JOIN
    DepartmentAvgSalary a ON d.department_id = a.department_id
WHERE
    e.salary > a.avg_salary
GROUP BY
    d.department_name;


-- ---

WITH department_avg AS (
    SELECT
        department_id,
        AVG(salary) AS avg_salary
    FROM employees
    GROUP BY department_id
)
SELECT
       d.department_name,
       ROUND(AVG(e.salary),2) AS avg_salary
FROM employees e
JOIN department_avg da ON e.department_id = da.department_id
JOIN departments d ON e.department_id = d.department_id
WHERE e.salary > da.avg_salary
GROUP BY  d.department_name;

---
---
**Query #7**

    WITH DepartmentAvgSalary AS (
        SELECT
            d.department_id,
            AVG(e.salary) AS avg_salary
        FROM
            departments d
        LEFT JOIN
            employees e ON d.department_id = e.department_id
        GROUP BY
            d.department_id
    )
    SELECT
```

```
    d.department_name,
    ROUND(AVG(e.salary),2) AS average_salary
FROM
    departments d
LEFT JOIN
    employees e ON d.department_id = e.department_id
JOIN
    DepartmentAvgSalary a ON d.department_id = a.department_id
WHERE
    e.salary > a.avg_salary
GROUP BY
    d.department_name;
```

department_name	average_salary
Finance	6000.00
Sales	5350.00
Marketing	4800.00
IT	5200.00

Query #8

```
WITH department_avg AS (
    SELECT department_id, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY department_id
)
SELECT  d.department_name, ROUND(AVG(e.salary),2) AS avg_salary
FROM employees e
JOIN department_avg da ON e.department_id = da.department_id
JOIN departments d ON e.department_id = d.department_id
WHERE e.salary > da.avg_salary
GROUP BY  d.department_name;
```

department_name	avg_salary
Finance	6000.00
Sales	5350.00
Marketing	4800.00
IT	5200.00

Question 4: Find the employees who have the highest salary in

their respective departments.

V

Schema (MySQL v8.0)**Schema (MySQL v8.0)**

Query #1

```
    WITH max_salary_per_department AS
    (SELECT department_id, MAX(salary) as max_salary
    FROM employees
    GROUP BY department_id)
    SELECT
        employee_name,
        salary,
        department_name
    FROM employees e
    JOIN max_salary_per_department d1
    ON e.department_id = d1.department_id
    JOIN departments d2
    ON d1.department_id = d2.department_id
    WHERE e.salary = d1.max_salary;
```

employee_name	salary	department_name
John Doe	5000.00	HR
Jane Smith	6000.00	Finance
Emily Brown	5500.00	Sales
Sarah Davis	4800.00	Marketing
Robert Anderson	5200.00	IT

Query #2

```
    SELECT
        d.department_name,
        e.employee_name,
        e.salary
    FROM
        departments d
    LEFT JOIN
        employees e ON d.department_id = e.department_id
    WHERE
        (e.department_id, e.salary) IN (
```

```
        SELECT
            department_id,
            MAX(salary)
        FROM
            employees
        GROUP BY
            department_id
    );
```

department_name	employee_name	salary
HR	John Doe	5000.00
Finance	Jane Smith	6000.00
Sales	Emily Brown	5500.00
Marketing	Sarah Davis	4800.00
IT	Robert Anderson	5200.00

Question 5: Calculate the running total of salaries for each department, ordered by department ID and employee name.

-- Question 5: Calculate the running total of salaries for each department, ordered by department ID and employee name.-- Question 5: Calculate the running total of salaries for each department, ordered by department ID and employee name.

```
WITH running_total AS (
    SELECT department_id, employee_name, salary,
        SUM(salary) OVER (PARTITION BY department_id ORDER BY employee_name) AS total_salary
    FROM employees
)
SELECT department_id, employee_name, salary, total_salary
FROM running_total
ORDER BY department_id, employee_name;
```

```
WITH DepartmentSalaries AS (
    SELECT
        d.department_id,
        e.employee_name,
        e.salary,
        ROW_NUMBER() OVER (PARTITION BY d.department_id ORDER BY e.employee_name) AS row_num
    FROM
        departments d
    LEFT JOIN
```

```
        employees e ON d.department_id = e.department_id
)
SELECT
    d.department_name,
    ds.employee_name,
    ds.salary,
    SUM(ds.salary) OVER (PARTITION BY ds.department_id ORDER BY ds.row_num) AS running_total
FROM
    departments d
LEFT JOIN
    DepartmentSalaries ds ON d.department_id = ds.department_id
ORDER BY
    ds.department_id,
    ds.row_num;
```

Output

Schema (MySQL v8.0)**Schema (MySQL v8.0)**

Query #1

```
    WITH running_total AS (
        SELECT department_id, employee_name, salary,
            SUM(salary) OVER (PARTITION BY department_id ORDER BY employee_name) AS total_salary
        FROM employees
    )
    SELECT department_id, employee_name, salary, total_salary
    FROM running_total
    ORDER BY department_id, employee_name;
```

department_id	employee_name	salary	total_salary
1	John Doe	5000.00	5000.00
2	Jane Smith	6000.00	6000.00
2	Olivia Harris	5500.00	11500.00
3	Daniel Turner	5200.00	5200.00
3	Emily Brown	5500.00	10700.00
3	Michael Johnson	4500.00	15200.00
4	David Wilson	4000.00	4000.00
4	Sarah Davis	4800.00	8800.00
5	Laura Clark	4700.00	4700.00
5	Robert Anderson	5200.00	9900.00

Query #2

```
WITH DepartmentSalaries AS (
    SELECT
        d.department_id,
        e.employee_name,
        e.salary,
        ROW_NUMBER() OVER (PARTITION BY d.department_id ORDER BY e.employee_name) AS row_num
    FROM
        departments d
    LEFT JOIN
        employees e ON d.department_id = e.department_id
)
SELECT
    d.department_name,
    ds.employee_name,
    ds.salary,
    SUM(ds.salary) OVER (PARTITION BY ds.department_id ORDER BY ds.row_num) AS running_total
FROM
    departments d
LEFT JOIN
    DepartmentSalaries ds ON d.department_id = ds.department_id
ORDER BY
    ds.department_id,
    ds.row_num;
```

department_name	employee_name	salary	running_total
HR	John Doe	5000.00	5000.00
Finance	Jane Smith	6000.00	6000.00
Finance	Olivia Harris	5500.00	11500.00
Sales	Daniel Turner	5200.00	5200.00
Sales	Emily Brown	5500.00	10700.00
Sales	Michael Johnson	4500.00	15200.00
Marketing	David Wilson	4000.00	4000.00
Marketing	Sarah Davis	4800.00	8800.00
IT	Laura Clark	4700.00	4700.00
IT	Robert Anderson	5200.00	9900.00

CHECK THE COMMENTS FOR THE LINK!

DATA IN MOTION