## **Tasks**

### **Exploratory data analysis (EDA)**

(Please refer to the **ipynb** format file)

1) Total row count

2) Data Type of columns

3) Check for missing values

4) Unique value counts

5) Top-selling products

6) Revenue by product category

7) Monthly sales trend

8) Sales by region

9) Repeat vs One-time Customers

### **SQL Queries**

**1)Customer Repeat Purchases:** Identify pairs of orders placed by the same customer within 30 days of each other. Return the customer\_id, order\_id of the first order, order\_id of the second order, and the date difference in days.

**Solution**:

SELECT \*

FROM

( SELECT

customer\_id,

order\_id AS first\_order\_id,

LEAD(order\_id) OVER (PARTITION BY customer\_id ORDER BY order\_date) AS second\_order\_id,

DATE\_DIFF('day', order\_date, LEAD(order\_date) OVER (PARTITION BY customer\_id ORDER BY order\_date)) AS day\_difference

FROM ecommerce\_sales ) t

WHERE day\_difference <= 30

order by customer\_id

**Explanation** : This outer query shows only the rows where a customer placed two orders within 30 days. It uses a subquery called t to find and compare those orders.

SubqueryExplanation:

**customer\_id**: Picks the customer’s ID to group their orders.

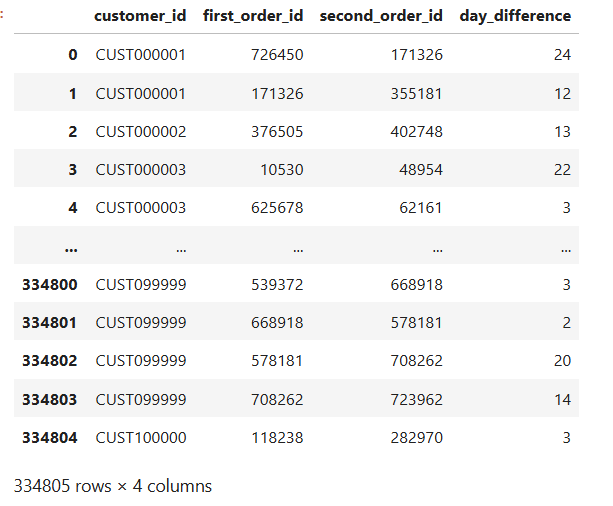
**order\_id AS first\_order\_id**: The current order made by the customer.

**second\_order\_id** : Finds the next order made by the same customer.

**day\_difference** : Calculates how many days passed between the current order and the next one.

This subquery creates a list showing each customer’s order and how soon they ordered again.

**Output:**

****

**2)Products with Similar Prices**: Find pairs of products in the same category where the unit\_price differs by less than 10% (i.e., |price1 - price2| / price1 < 0.1). Return the product\_id and unit\_price for both products, and the percentage price difference.

**Solution**:

WITH ordered\_products AS (

SELECT

product\_id,

product\_category,

unit\_price,

LEAD(product\_id) OVER (PARTITION BY product\_category ORDER BY unit\_price) AS next\_product,

LEAD(unit\_price) OVER (PARTITION BY product\_category ORDER BY unit\_price) AS next\_price

FROM ecommerce\_sales

)

SELECT

product\_category,

product\_id AS product\_id\_1,

next\_product AS product\_id\_2,

unit\_price AS unit\_price\_1,

next\_price AS unit\_price\_2,

ABS(unit\_price - next\_price) / unit\_price AS price\_diff\_ratio

FROM ordered\_products

WHERE ABS(unit\_price - next\_price) / unit\_price < 0.1

order by product\_category

**Explanation** :

Common Table Expression (CTE): ordered\_products

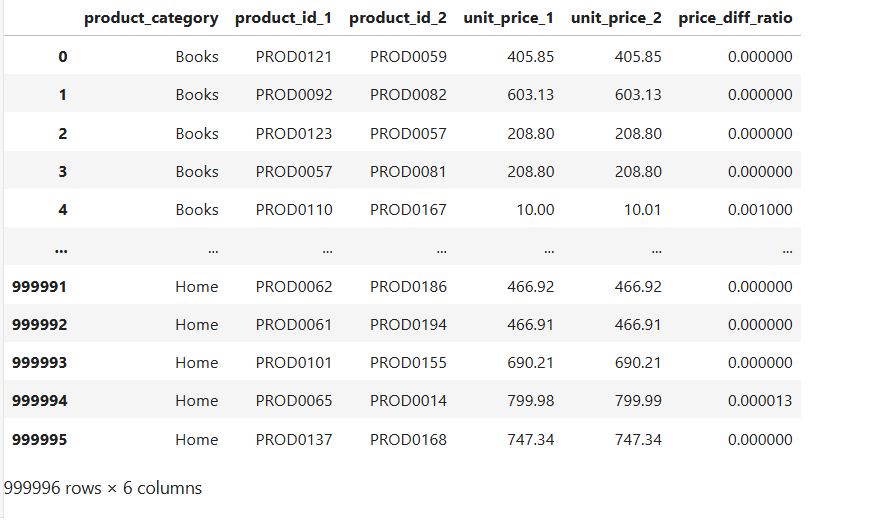
* The CTE pulls data from the ecommerce\_sales table.
* It groups (partitions) the products by product\_category.
* Within each category, products are sorted by unit\_price.
* The LEAD() function is used to get:
  + The next product's ID (next\_product)
  + The next product's price (next\_price)

This helps create pairs of products that are next to each other in price within the same category.

Main Query:

* Selects data from the ordered\_products CTE.
* Gets each product’s category, ID, and price, along with the next product’s ID and price.
* Calculates how different the prices are using:  
  **(absolute price difference) ÷ (current product’s price)**
* Keeps only the product pairs where the price difference is less than 10%.
* Sorts the final results by product\_category.

**Output** :



**3)Top Customers by Region**: For each region, identify the top 3 customers based on total sales (total\_price). Return the region, customer\_id, total sales, and their rank within the region.

**Solution :**

select \* from (

SELECT

region,

customer\_id,

SUM(total\_price) AS total\_sales,

DENSE\_RANK() OVER (PARTITION BY region ORDER BY SUM(total\_price) DESC) AS sales\_rank

FROM ecommerce\_sales

GROUP BY region, customer\_id )

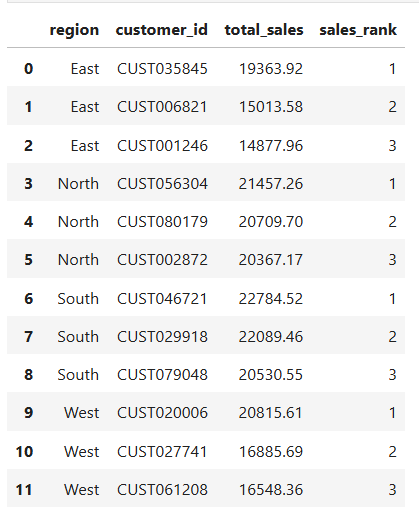
where sales\_rank <= 3

order by region , sales\_rank

**Explanation** :

* **FROM**: Reads data from the ecommerce\_sales table.
* **GROUP BY:** Groups the data by region and customer\_id.
* **SUM**: Calculates total sales (SUM(total\_price)) for each customer in each region.
* **DENSE\_RANK():** Ranks customers within each region based on their total sales (highest first).
* **Subquery output:** Creates a temporary table with columns: region, customer\_id, total\_sales, and sales\_rank.
* **WHERE:** Keeps only the top 3 customers per region (sales\_rank <= 3).
* **ORDER BY:** Sorts the final results by region and sales\_rank.
* **SELECT:** Returns all columns from the filtered, sorted data.

**Output :**

****

**4)Running Total Sales by Category**: Calculate the running total of total\_price for each product\_category over time, ordered by order\_date. Return product\_category, order\_date, total\_price, and the running total.

**Solution :**

**1st Approach:** Add up total\_price by product\_category and order\_date, then calculate the running total each day (Please refer output section)

WITH DailySales AS (

SELECT

product\_category,

order\_date,

SUM(total\_price) AS daily\_total

FROM

ecommerce\_sales

GROUP BY

product\_category,

order\_date

)

SELECT

product\_category,

order\_date,

daily\_total,

SUM(daily\_total) OVER (

PARTITION BY product\_category

ORDER BY order\_date

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS running\_total

FROM

DailySales

ORDER BY

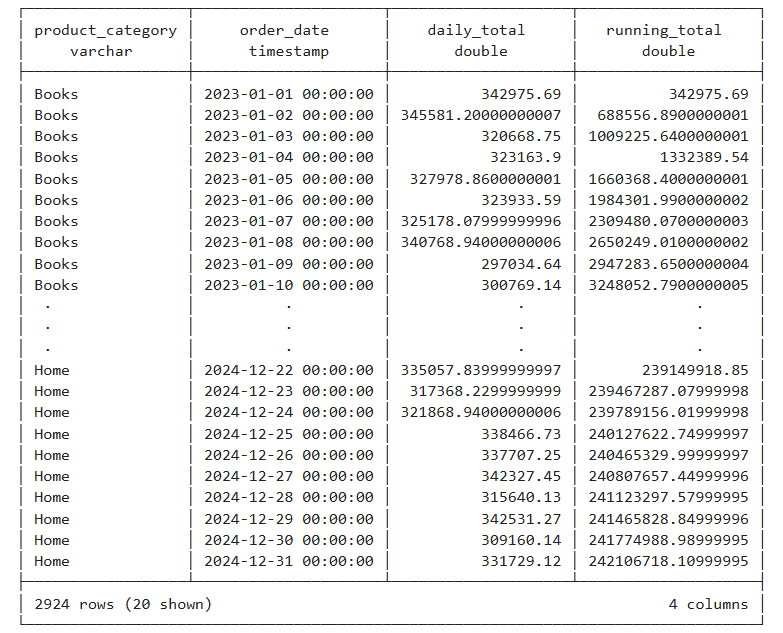
product\_category,

order\_date

**Explaination :**

* **CTE Execution (DailySales)**:  
  Runs first, grouping data by product\_category and order\_date, and calculates daily\_total as the sum of total\_price.
* **FROM (outer query)**:  
  Reads from the DailySales CTE result.
* **Window Function**:  
  Calculates a running total of daily\_total for each category, ordered by date.
* **Window Frame**:  
  Adds up all values from the first date to the current date in the same category.
* **SELECT**:  
  Returns product\_category, order\_date, daily\_total, and running\_total.
* **ORDER BY**:  
  Sorts the final result by product\_category and order\_date.

**OUTPUT :**



**2nd Approach**: Running total per individual order (including multiple orders on the same day)

SELECT

product\_category,

order\_date,

total\_price,

SUM(total\_price) OVER (

PARTITION BY product\_category

ORDER BY order\_date, total\_price

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS running\_total

FROM

ecommerce\_sales

ORDER BY

product\_category,

order\_date,

running\_total

**Explaination :**

* **FROM**: Gets all rows from the ecommerce\_sales table.
* **Window Function (SUM OVER):**
  + Calculates a running total of total\_price.
  + Groups data by product\_category.
  + Sorts each group by order\_date and total\_price.
  + Adds up prices from the start to the current row.
* **SELECT**: Returns product\_category, order\_date, total\_price, and the running\_total.
* **ORDER** **BY**: Sorts the final output by product\_category, order\_date, and running\_total.

