**Step 1: Exploring Dataset:**

**1- Find the total revenue by each customer:**

**Description:**

This query selects the distinct customer IDs from the tableRetail table and calculates the revenue for each customer using the SUM and OVER analytical functions.

The SUM function calculates the total revenue by multiplying the Price and Quantity columns, and the OVER function partitions the calculation by Customer\_ID so that each customer's revenue is calculated separately.

The resulting dataset includes the customer ID and their total revenue.

**How it Works:**

First calculates the revenue for each customer in the table by multiplying the "Price" and "Quantity" columns and summing them up for each customer separately. Using the analytic function "SUM" to sum up the values in the "Price \* Quantity" column for each customer, and returns the result as a new column named "Revenue".

**Query:**

*/\**

*This query selects the customer IDs from the tableRetail table*

*then calculates the revenue for each customer using the SUM and OVER analytical functions.*

*The SUM() function calculates the total revenue by multiplying the Price and Quantity columns,*

*and the OVER() function partitions the calculation result by Customer\_ID so that each customer's revenue is calculated separately.*

*Use DISTINCT to avoid Redundancy, and ORDER BY revenue DESC to be more readable.*

*\*/*

SELECT DISTINCT Customer\_ID,

*-- calculate the revenue for each customer by multiplying the "Price" and "Quantity", and summing them up*

SUM(Price \* Quantity) OVER (PARTITION BY Customer\_ID) AS Revenue

FROM tableRetail

ORDER BY Revenue DESC;

**Output:**

Table

Description automatically generated

**2- Find the top selling Products per Quantity:**

**Description:**

This query provides a list of the products with the total quantity of each product sold. It helps identify the most popular products based on the total quantity sold.

**How it Works:**

Selects the distinct StockCode and calculates the sum of Quantity for each StockCode using the OVER window function with PARTITION BY.

Then, the resulting dataset is ordered by Total\_Quantity in descending order to display the top products first.

**Query:**

*/\**

*This query selects all stock codes and the total quantity sold for each stock code.*

*The SUM() function calculates the total quantity sold for each stock code,*

*and OVER() function partitions the data based on stock code.*

*Use DISTINCT to avoid redundancy, and ORDER BY total\_quantity DESC to get the top sells*

*\*/*

SELECT DISTINCT StockCode, SUM(Quantity) OVER (PARTITION BY StockCode) AS Total\_Quantity

FROM tableRetail

ORDER BY Total\_Quantity DESC;

**Output:**

**Table

Description automatically generated**

**3- Find the monthly revenue for the top 5 customers:**

**Description:**

This query displays the monthly revenue for the top 5 customers who generated the highest over all the months.

**How it Works:**

The query works by creating a CTE called CustomersRev, which retrieves the distinct customer IDs, the order month, the revenue per month, and the total revenue generated by each customer.

The order month is calculated by converting the invoice date to month and year format.

The revenue per month is calculated using the SUM() OVER() window function to get the total revenue of each customer in each month. The total revenue of each customer is also calculated using the same function.

The second CTE, topCustomers is created by assigning a dense rank of the total revenue overall customers in descending order.

Finally, the main query selects all columns from the topCustomers CTE for customers whose dense rank is less than or equal to 5. This gives the top 5 customers who generated the highest revenue per month over all the months.

**Query:**

*/\**

*This query uses CTEs to find the monthly revenue for the top 5 customers based on their total revenue.*

*The first CTE: CustomersRev, calculates the total revenue earned per customer per month and the total revenue earned per customer overall.*

*Then the result is ordered by total revenue in descending order to find the highest customers.*

*The second CTE: topCustomers, uses the CustomersRev CTE to rank the customers based on total revenue earned, using the DENSE\_RANK() analytical function.*

*The final SELECT statement retrieves all data from the topCustomers CTE where the customer ranking is less than or equal to 5.*

*To find the monthly revenue for the top 5 cutomers based on thier total revenue.*

*\*/*

*-- Select DISTINCT customers and calculate revenue for each month and the total revenue*

WITH CustomersRev AS (

SELECT DISTINCT Customer\_ID,

TO\_CHAR(TO\_DATE(InvoiceDate, 'MM/DD/YYYY HH24:MI'), 'MM-YYYY') AS orderMonth,

SUM(Price \* Quantity) OVER (PARTITION BY Customer\_ID, TO\_CHAR(TO\_DATE(InvoiceDate, 'MM/DD/YYYY HH24:MI'), 'MM-YYYY')) AS RevenuePerMonth,

SUM(Price \* Quantity) OVER (PARTITION BY Customer\_ID) AS totalRevenue

FROM tableRetail

ORDER BY totalRevenue DESC

),

*-- Using previous CTE, rank customers by total revenue,*

*-- so each cutomer will get a rank starts from 1 based on his total revenue*

topCustomers AS (

SELECT Customer\_ID, orderMonth, RevenuePerMonth, totalRevenue,

DENSE\_RANK () OVER (ORDER BY totalRevenue DESC) AS rnk

FROM CustomersRev

)

*-- Using the previuos CTE, select the top 5 customers based on thier total revenue*

SELECT \*

FROM topCustomers

WHERE rnk <=5

*-- Using previous CTE, rank customers by total revenue,*

*-- so each cutomer will get a rank starts from 1 based on his total revenue*

topCustomers AS (

SELECT Customer\_ID, orderMonth, RevenuePerMonth, totalRevenue,

DENSE\_RANK () OVER (ORDER BY totalRevenue DESC) AS rnk

FROM CustomersRev

)

*-- Using the previuos CTE, select the top 5 customers based on thier total revenue*

SELECT \*

FROM topCustomers

WHERE rnk <=5

**Output:**

Table

Description automatically generated

**4- Find the total revenue per month:**

**Description:**

This query selects the distinct order month and the total revenue for each month.

**How it Works:**

First, use the TO\_CHAR function to convert the InvoiceDate to a date format in the form of 'YYYY-MM'. This column is aliased as orderMonth.

Then, use the SUM() function to calculate the revenue for each month. It uses the OVER() clause to sum the revenue for each month, which is partitioned by the InvoiceDate. This column is aliased as RevenuePerMonth.

Finally, the results are ordered by the orderMonth column for better reading.

**Query:**

*/\**

*This query finds the total revenue in each month.*

*The SUM() function calculates the total revenue by multiply quantity sold by price of each one.*

*and OVER() function partitions the data based on month in "MM-YYYY" format*

*Use DISTINCT to avoid redundancy, and ORDER BY ordermonth ASC in "YYYY-MM" format*

*to order per year then month.*

*\*/*

**Output:**

Table

Description automatically generated

*Use DISTINCT to avoid redundancy, and ORDER BY ordermonth ASC in "YYYY-MM" format*

*to order per year then month.*

*\*/*

SELECT DISTINCT TO\_CHAR(TO\_DATE(InvoiceDate, 'MM/DD/YYYY HH24:MI'), 'YYYY-MM') AS orderMonth,

SUM(Price \* Quantity) OVER (PARTITION BY TO\_CHAR(TO\_DATE(InvoiceDate, 'MM/DD/YYYY HH24:MI'), 'MM-YYYY')) AS RevenuePerMonth

FROM tableRetail

ORDER BY orderMonth;

**5- Find the percentage of total revenue generated by each customer compared to the total revenue:**

**Description:**

This query displays the distinct customer IDs along with their percentage of total revenue compared to the total revenue for all customers.

**How it Works:**

The code uses the SUM() function with the OVER() window function to calculate the total revenue for each customer, partitioned by Customer\_ID. The CONCAT() function is used to append the calculated percentage value to the percentage symbol '%'.

The result set is sorted in descending order by the total revenue for each customer to display the top customers first.

**Query:**

*/\**

*This query calculates the total revenue and percentage of total revenue for each customer in the tableRetail.*

*It uses the SUM() fumction with "OVER (PARTITION BY)" the customer\_id to calculate the total revenue for each customer.*

*and then calculates the percentage of total revenue for each customer using the total revenue and the total revenue of all customers.*

*Use DISTINCT to avoid redundancy, and ORDER BY total\_revenue DESC for better read.*

*\*/*

SELECT DISTINCT Customer\_ID,

SUM(Price \* Quantity) OVER (PARTITION BY Customer\_ID) AS total\_revenue,

CONCAT (ROUND ((SUM(Price \* Quantity) OVER (PARTITION BY Customer\_ID) / SUM(Price \* Quantity) OVER ()) \* 100,2),'%') AS Percentage\_of\_Total\_Revenue

FROM tableRetail

ORDER BY total\_revenue DESC;

*Use DISTINCT to avoid redundancy, and ORDER BY total\_revenue DESC for better read.*

*\*/*

SELECT DISTINCT Customer\_ID,

SUM(Price \* Quantity) OVER (PARTITION BY Customer\_ID) AS total\_revenue,

CONCAT (ROUND ((SUM(Price \* Quantity) OVER (PARTITION BY Customer\_ID) / SUM(Price \* Quantity) OVER ()) \* 100,2),'%') AS Percentage\_of\_Total\_Revenue

FROM tableRetail

ORDER BY total\_revenue DESC;

**Output:**

Table

Description automatically generated

**6- Find the top 10 customers by revenue, their most popular product, and its revenue:**

**Description:**

This query performs an analysis of customer purchase data to find the top 10 customers and their favorite products and how much they spend on them.

**How it Works:**

First, create a CTE called Customer\_Revenue, which calculates the total revenue for each customer and assigns a rank based on their total revenue.

Next, create CTE called Customer\_Product calculates the total quantity and revenue for each product purchased by each customer.

Then, the Ranked\_Products CTE ranks the products for each customer based on their total quantity purchased.

Finally, the three CTEs are joined together to obtain the most popular product and its revenue for each customer. The result only includes the top 10 customers by revenue and their most popular product.

The resulting dataset is ordered by customer rank.

**Query:**

*/\**

*This query uses CTEs to find the top 10 customers by revenue, thier top product and its revenue.*

*The first CTE: Customer\_Revenue, calculates the total revenue earned per customer.*

*The second CTE: Customer\_Product, finds the total quantity and revenue for each product purchased by each customer, the total quantity that customer bought, and the product revenue from this customer.*

*The third CTE: Ranked\_Products, uses Customer\_Product CTE to assign a rank to each product for each customer based on the total quantity of the product purchased by the customer, so the top product can be extracted.*

*The final SELECT statement retrieves all data from the Customer\_Product and Ranked\_Products CTEs where the product ranking is 1 and cutomer ranking is less than or equal to 10*

*to get the most popular product for each customer with their total revenue.*

*\*/*

*-- Get the total revenue for each customer and assign a rank based on the revenue*

WITH Customer\_Revenue AS (

SELECT Customer\_ID, SUM(Price \* Quantity) AS Revenue,

DENSE\_RANK () OVER (ORDER BY SUM(Price \* Quantity) DESC) AS rnk

FROM tableRetail

GROUP BY Customer\_ID

),

*-- Get the total quantity and revenue for each product purchased by each customer*

Customer\_Product AS (

SELECT DISTINCT Customer\_ID, StockCode, SUM(Quantity) AS Total\_Quantity,

SUM(Quantity \* Price) AS product\_revenue

FROM tableRetail

GROUP BY Customer\_ID, StockCode

),

*-- Rank the products for each customer based on their total quantity purchased*

Ranked\_Products AS (

SELECT Customer\_ID, StockCode, Total\_Quantity, product\_revenue,

RANK() OVER (PARTITION BY Customer\_ID ORDER BY Total\_Quantity DESC) AS rnk

FROM Customer\_Product

)

*-- Join the customer revenue and product data to get the most popular product for each customer with their total revenue*

SELECT cr.Customer\_ID, cr.Revenue AS total\_revenue, rp.StockCode AS Most\_Popular\_Product, rp.Total\_Quantity, rp.product\_revenue

FROM Customer\_Revenue cr

JOIN Ranked\_Products rp

ON cr.Customer\_ID = rp.Customer\_ID

WHERE rp.rnk = 1 AND cr.rnk <=10

ORDER BY cr.rnk;

*-- Using previous CTE, rank customers by total revenue,*

*-- so each cutomer will get a rank starts from 1 based on his total revenue*

*5 customers based on thier total revenue*

**Output:**

Table

Description automatically generated

**7- Find the top 10 products by revenue, their total revenue, and their monthly revenue:**

**Description:**

This query displays the monthly revenue for the top 10 products based on their total revenue.

**How it Works:**

First, create a CTE called Products\_rev, which calculates the total revenue for each product using the SUM aggregate function and grouping by StockCode. Then, the DENSE\_RANK window function ranks the products based on their total revenue.

Next, create CTE called top\_products which selects the top 10 products from the Products\_rev CTE based on their rank.

Finally, the main select statement uses the SUM window function with the PARTITION BY to calculate each product's monthly revenue and total revenue. And the WHERE filters the results to only include the top 10 products, which are determined by the IN operator and the top\_products CTE. The resulting dataset is ordered by StockCode and Month.

**Query:**

*/\**

*This query uses CTEs to find the monthly revenue for the top 10 products by revenue.*

*The first CTE: Products\_rev, finds the total revenue for each product and ranks them by revenue.*

*The second CTE: top\_products, selects the top 10 products by revenue.*

*The final SELECT statement retrieves all data from the top\_products,*

*and orders the results by stock code and month.*

*\*/*

*-- find the total revenue for each product and rank them by the total revenue*

WITH Products\_rev AS (

SELECT StockCode,

SUM(Price \* Quantity) AS Revenue,

DENSE\_RANK () OVER (ORDER BY SUM(Price \* Quantity) DESC) AS rnk

)

SELECT DISTINCT StockCode,

TO\_CHAR(TO\_DATE(InvoiceDate, 'MM/DD/YYYY HH24:MI'), 'YYYY-MM') AS Month,

SUM(Price \* Quantity) OVER (PARTITION BY StockCode, TO\_CHAR(TO\_DATE(InvoiceDate, 'MM/DD/YYYY HH24:MI'), 'YYYY-MM')) AS monthly\_revenue,

SUM(Price \* Quantity) OVER(PARTITION BY StockCode) AS total\_revenue

FROM tableRetail

WHERE StockCode IN (SELECT StockCode FROM top\_products)

ORDER BY StockCode, Month;

FROM tableRetail

GROUP BY StockCode

ORDER BY Revenue DESC

),

*-- select the top 10 product by total revenue*

top\_products AS(

SELECT \* FROM Products\_rev

WHERE rnk <=10

)

SELECT DISTINCT StockCode,

TO\_CHAR(TO\_DATE(InvoiceDate, 'MM/DD/YYYY HH24:MI'), 'YYYY-MM') AS Month,

SUM(Price \* Quantity) OVER (PARTITION BY StockCode, TO\_CHAR(TO\_DATE(InvoiceDate, 'MM/DD/YYYY HH24:MI'), 'YYYY-MM')) AS monthly\_revenue,

SUM(Price \* Quantity) OVER(PARTITION BY StockCode) AS total\_revenue

FROM tableRetail

WHERE StockCode IN (SELECT StockCode FROM top\_products)

ORDER BY StockCode, Month;

**Output:**

Table

Description automatically generated

**8- Find the top 10 most frequently purchased items, their total quantity, and their total revenue:**

**Description:**

This query retrieves information about the top 10 most frequently purchased items based on the total quantity sold.

**How it Works:**

First, create a CTE called products\_quantity to calculate the total quantity and total revenue for each distinct StockCode. The SUM function is used with the OVER window function to calculate the TotalQuantity and TotalRevenue for each StockCode which represents the products. The ORDER BY clause orders the results based on the total quantity sold in descending order.

Next, create CTE called topProductsQuan which selects the top 10 products by the total quantity sold from the products\_quantity CTE. It uses the DENSE\_RANK function to assign a ranking to each StockCode based on the TotalQuantity in descending order.

Finally, the main SELECT statement retrieves the StockCode, TotalQuantity, and TotalRevenue from the topProductsQuan CTE where the rank is less than or equal to 10.

**Query:**

*/\**

*This query uses CTEs to find the top 10 products with the highest total quantity sold and their total revenue.*

*The first CTE: products\_quantity, calculate the total quantity and total revenue for each product.*

*The second CTE: topProductsQuan, ranks the product based on the total quantity.*

*The final SELECT statement retrieves all data for these top 10 products from the topProductsQuan.*

*\*/*

*-- find the total quantity and totla revenue for each product*

WITH products\_quantity AS (

SELECT

DISTINCT StockCode,

SUM(Quantity) OVER (PARTITION BY StockCode) AS TotalQuantity,

SUM(Quantity \* Price) OVER (PARTITION BY StockCode) AS TotalRevenue

FROM

tableRetail

ORDER BY

SUM(Quantity) OVER (PARTITION BY StockCode) DESC

*-- ranks the product based on the total quantity*

), topProductsQuan AS (

SELECT StockCode, TotalQuantity, TotalRevenue,

DENSE\_RANK () OVER (ORDER BY TotalQuantity DESC) AS rnk

FROM products\_quantity

)

SELECT StockCode, TotalQuantity, TotalRevenue

FROM topProductsQuan

WHERE rnk <=10;

**Output:**

Table

Description automatically generated

**9- Find the average basket size (average number of items purchased per transaction) for each customer:**

**Description:**

This query displays information about the average basket size for each customer, which is the average number of items per transaction.

**How it Works:**

The query starts with a SELECT DISTINCT statement, which selects unique values of Customer\_ID. Then, the COUNT (\*) OVER (PARTITION BY Customer\_ID) function counts the number of transactions for each Customer\_ID, essentially counting the number of orders each customer has made. The SUM (Quantity) OVER (PARTITION BY Customer\_ID) function sums up the Quantity column for each Customer\_ID, giving the total number of products sold to each customer. The AVG(Quantity) OVER (PARTITION BY Customer\_ID) function calculates the average Quantity for each Customer\_ID, giving the average number of products purchased per order. Finally, the resulting dataset is ordered by AvgBasketSize in descending order.

**Query:**

*/\**

*This query displays the number of orders, total sold quantities, and average basket size for each unique customer.*

*It counts the number of orders for each customer, and the total quantities that bought from each user,*

*then calculate the average basket size by divide the total quantities on the number of orders for each customer*

*\*/*

SELECT DISTINCT Customer\_ID,

COUNT (\*) OVER (PARTITION BY Customer\_ID) AS NumberOfOrders,

SUM (Quantity) OVER (PARTITION BY Customer\_ID) AS total\_sold\_quantities,

ROUND (AVG(Quantity) OVER (PARTITION BY Customer\_ID), 2) AS AvgBasketSize

FROM tableRetail

ORDER BY AvgBasketSize DESC;

**Output:**

Table

Description automatically generated

**Step 2: Implementing RFM (Monetary) Model:**

**Description:**

This query performs customer segmentation based on the analysis of RFM (Recency, Frequency, Monetary).

**How it Works:**

The code consists of three CTEs and a final SELECT statement that generates the customer segments based on their RFM scores.

The first CTE, ref\_date, extracts the maximum date in the InvoiceDate column and formats it as a reference date for calculating recency.

The second CTE, customer\_rfm, calculates the RFM values for each customer by subtracting the maximum InvoiceDate from each invoice date to calculate recency, counting the number of unique invoices to calculate frequency, and summing the product of Price and Quantity to calculate monetary value.

The third CTE, scores, calculates the average F (frequency) and M (monetary) scores for each customer and assigns an FM score based on the NTILE function, which divides the customers into equal groups based on their RFM values.

Finally, the customers are assigned to a segment based on their RFM scores using a CASE statement in the final SELECT statement.

**Query:**

*/\**

*This query uses the CTEs to calculate the RFM scores and find the customers segments based on thier scores.*

*The first CTE: ref\_date, defines the reference date as the maximum date in the InviceDate.*

*The second CTE: customer\_rfm, calculates the RFM (recency, frequency, monetary) score for each customer, using the reference date.*

*The third CTE: scores, calculates the FM score for each customer, based on the RFM scores calculated in the customer\_rfm CTE.*

*The final SELECT statement assigns a customer segment to each customer, based on their R and FM scores.*

*\*/*

WITH

*-- Extract the Maximum Date in the InvoiceDate column as a reference date*

ref\_date AS (

SELECT

MAX(TO\_DATE(InvoiceDate, 'MM/DD/YYYY HH24:MI')) AS reference\_date,

TO\_CHAR(MAX(TO\_DATE(InvoiceDate, 'MM/DD/YYYY HH24:MI')), 'DD') AS reference\_day

FROM tableRetail

),

*-- Find the recency, frequency, monetary values*

customer\_rfm AS (

SELECT

Customer\_ID,

TO\_CHAR(ref\_date.reference\_date - MAX(TO\_DATE(InvoiceDate, 'MM/DD/YYYY HH24:MI')), '9999999') AS recency,

COUNT(DISTINCT Invoice) AS frequency,

SUM(Price \* Quantity) AS monetary

FROM tableRetail

CROSS JOIN ref\_date

GROUP BY Customer\_ID, ref\_date.reference\_date

),

scores AS (

*-- Find the average F and M scores*

SELECT Customer\_ID, recency, frequency, monetary, r\_score,

NTILE(5) OVER (ORDER BY AVG (f\_score + m\_score)) AS fm\_score

FROM(

*-- use NTILE() to devide the customers into equal groups based on the RFM values*

SELECT Customer\_ID, recency, frequency, monetary,

NTILE (5) OVER (ORDER BY CAST(recency AS INT) DESC) AS r\_score,

NTILE (5) OVER (ORDER BY frequency ) AS f\_score,

NTILE(5) OVER (ORDER BY monetary ) AS m\_score

FROM customer\_rfm

)

GROUP BY Customer\_ID, recency, frequency, monetary, r\_score

)

*-- Assign segments to customers based on thier RFM scores*

SELECT Customer\_ID, recency, frequency, monetary, r\_score, fm\_score,

CASE

WHEN (r\_score = 5 AND (fm\_score = 5 OR fm\_score = 4))

OR (r\_score = 4 AND fm\_score = 5) THEN 'Champions'

WHEN (fm\_score = 2 AND (r\_score = 5 OR r\_score = 4))

OR (fm\_score = 3 AND (r\_score = 3 OR r\_score = 4))

THEN 'Potential Loyalists'

WHEN (r\_score = 3 AND (fm\_score = 5 OR fm\_score = 4))

OR (r\_score = 4 AND fm\_score = 4 )

OR (r\_score = 5 AND fm\_score = 3)

THEN 'Loyal Customers'

WHEN r\_score = 5 AND fm\_score = 1

THEN 'Recent Customers'

WHEN fm\_score = 1 AND (r\_score = 3 OR r\_score = 4)

THEN 'Promising'

WHEN r\_score = 2 AND (fm\_score = 3 OR fm\_score = 2)

OR (r\_score = 3 AND fm\_score = 2)

THEN 'Customers Needing Attention'

WHEN (r\_score = 2 AND (fm\_score = 5 OR fm\_score = 4 OR fm\_score = 1))

OR (r\_score = 1 AND fm\_score = 3)

THEN 'At Risk'

WHEN r\_score = 1 AND (fm\_score = 5 OR fm\_score = 4)

THEN 'Cant Lose Them'

WHEN r\_score = 1 AND fm\_score = 2

THEN 'Hibernating'

WHEN r\_score = 1 AND fm\_score = 1

THEN 'Lost'

END

AS cust\_segment

FROM scores;

),

*-- Find the recency, frequency, monetary values*

customer\_rfm AS (

SELECT

Customer\_ID,

TO\_CHAR(ref\_date.reference\_date - MAX(TO\_DATE(InvoiceDate, 'MM/DD/YYYY HH24:MI')), '9999999') AS recency,

COUNT(DISTINCT Invoice) AS frequency,

SUM(Price \* Quantity) AS monetary

FROM tableRetail

CROSS JOIN ref\_date

GROUP BY Customer\_ID, ref\_date.reference\_date

),

scores AS (

*-- Find the average F and M scores*

SELECT Customer\_ID, recency, frequency, monetary, r\_score,

NTILE(5) OVER (ORDER BY AVG (f\_score + m\_score)) AS fm\_score

FROM(

*-- use NTILE() to devide the customers into equal groups based on the RFM values*

SELECT Customer\_ID, recency, frequency, monetary,

NTILE (5) OVER (ORDER BY CAST(recency AS INT) DESC) AS r\_score,

NTILE (5) OVER (ORDER BY frequency ) AS f\_score,

NTILE(5) OVER (ORDER BY monetary ) AS m\_score

FROM customer\_rfm

)

GROUP BY Customer\_ID, recency, frequency, monetary, r\_score

)

*-- Assign segments to customers based on thier RFM scores*

SELECT Customer\_ID, recency, frequency, monetary, r\_score, fm\_score,

CASE

WHEN (r\_score = 5 AND (fm\_score = 5 OR fm\_score = 4))

OR (r\_score = 4 AND fm\_score = 5) THEN 'Champions'

WHEN (fm\_score = 2 AND (r\_score = 5 OR r\_score = 4))

OR (fm\_score = 3 AND (r\_score = 3 OR r\_score = 4))

THEN 'Potential Loyalists'

WHEN (r\_score = 3 AND (fm\_score = 5 OR fm\_score = 4))

OR (r\_score = 4 AND fm\_score = 4 )

OR (r\_score = 5 AND fm\_score = 3)

THEN 'Loyal Customers'

WHEN r\_score = 5 AND fm\_score = 1

THEN 'Recent Customers'

WHEN fm\_score = 1 AND (r\_score = 3 OR r\_score = 4)

THEN 'Promising'

WHEN r\_score = 2 AND (fm\_score = 3 OR fm\_score = 2)

OR (r\_score = 3 AND fm\_score = 2)

THEN 'Customers Needing Attention'

WHEN (r\_score = 2 AND (fm\_score = 5 OR fm\_score = 4 OR fm\_score = 1))

OR (r\_score = 1 AND fm\_score = 3)

THEN 'At Risk'

WHEN r\_score = 1 AND (fm\_score = 5 OR fm\_score = 4)

THEN 'Cant Lose Them'

WHEN r\_score = 1 AND fm\_score = 2

THEN 'Hibernating'

WHEN r\_score = 1 AND fm\_score = 1

THEN 'Lost'

END

AS cust\_segment

FROM scores;

**Output:**

END

AS cust\_segment

FROM scores;

**Table

Description automatically generated**

**Step 3: Analyzing Daily Transactions:**

**1- Find the maximum number of consecutive days a customer made purchases:**

**Description:**

This code calculates the maximum number of consecutive days a customer made purchases using the daily transaction data in the "dailycustomers" table.

**How it Works:**It works by first calculating whether each transaction for a given customer is part of a consecutive day streak using the lag function and a conditional statement.   
Then finds the running total of consecutive days for each customer using the sum and partition functions.   
Finally, it selects the distinct customer IDs and their corresponding maximum consecutive days using the max function and the partition function. The resulting dataset contains two columns: customer ID and their maximum consecutive days of purchase.

**Query:**

*/\**

*This query uses the CTEs to calculate the maximum number of consecutive days for each customer made a purchase.*

*The first CTE: consecutive\_days, uses the LAG function to compare the current date with the previous date and assign a value of 1 if the dates are consecutive, or 0 if they are not.*

*It is partitioned by customer and ordered by calendar date.*

*The second CTE: running\_total, calculates the total consecutive days for each purchase using the SUM function over the is\_consecutive column.*

*It is partitioned by customer and ordered by calendar date.*

*The final SELECT statement selects the distinct cust\_id values from the running\_total CTE and calculates the maximum number of consecutive days for each customer using the MAX function over the total\_consecutive\_days column.*

*\*/*

*-- find the consecutive days for each customer*

WITH consecutive\_days AS (

SELECT

cust\_id,

calendar\_dt,

CASE

WHEN calendar\_dt = LAG(calendar\_dt) OVER (PARTITION BY cust\_id ORDER BY calendar\_dt) + 1

THEN 1

ELSE 0

END AS is\_consecutive

FROM dailycustomers

),

*-- Find the total consecutive days*

running\_total AS (

SELECT

cust\_id,

calendar\_dt,

SUM(is\_consecutive) OVER (PARTITION BY cust\_id ORDER BY calendar\_dt) AS total\_consecutive\_days

FROM consecutive\_days

)

*-- select the max consecutive days*

SELECT

DISTINCT cust\_id,

MAX(total\_consecutive\_days) OVER (PARTITION BY cust\_id) AS max\_consecutive\_days

FROM running\_total;

**Output:**

Table

Description automatically generated

**- Find the number of days or transactions it takes a customer to reach a spent threshold of 250 L.E:**

**Description:**

The query calculates the number of days or transactions it takes for a customer to reach a spent threshold of 250 L.E using CTEs and window functions.

**How it Works:**The first CTE: customer\_spend, calculates the total amount spent by each customer using a window function to accumulate the spend over time.

The second CTE: customer\_days, joins the dailycustomers table with customer\_spend and uses a window function to count the number of distinct days each customer has made a purchase.

The third CTE: customer\_threshold, calculates the number of days it takes for each customer to reach a spend of 250 L.E using another window function to find the minimum number of days preceding the threshold date.

The final SELECT statement selects the cust\_id and days\_to\_250 columns from the customer\_threshold CTE, filtering out any NULL values which indicate that the customers didn't reach 250 L.E, and grouping by cust\_id and days\_to\_250.

**Query:**

*/\**

*This query uses the CTEs to calculate the average number of days or transactions it takes for a customer to reach a spend threshold of 250 LE.*

*The first CTE: customer\_spend, calculates the total amount spent by each customer, using a window function to accumulate the spend over time.*

*The second CTE: customer\_days, joins the dailycustomers table with customer\_spend and uses a window function to count the number of distinct days each customer has made a purchase.*

*The third CTE: customer\_threshold, calculates the number of days it takes for each customer to reach a spend of 250 LE,*

*using another window function to find the minimum number of days preceding the threshold date.*

*The final SELECT statement selects the cust\_id and days\_to\_250 columns from the customer\_threshold CTE,*

*filtering out any NULL values which is the customers didn't reach 250 LE, and grouping by cust\_id and days\_to\_250.*

*\*/*

*-- find total amount spent by each customer*

WITH customer\_spend AS (

SELECT

cust\_id,

SUM(amt\_le) OVER (PARTITION BY cust\_id ORDER BY calendar\_dt) AS total\_spend

FROM dailycustomers

),

*-- find the number of distinct days each customer has made a purchase*

customer\_days AS (

SELECT

dc.cust\_id, cs.total\_spend,

COUNT(DISTINCT dc.calendar\_dt) OVER (PARTITION BY dc.cust\_id ) AS num\_days

FROM dailycustomers dc

JOIN customer\_spend cs

ON dc.cust\_id = cs.cust\_id

),

*-- find the number of days it takes for each customer to reach a spend of 250 LE*

customer\_threshold AS (

SELECT

cust\_id,

MIN(num\_days) OVER (PARTITION BY cust\_id ORDER BY num\_days ASC

ROWS BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING) AS days\_to\_250

FROM customer\_days

WHERE total\_spend >= 250

)

SELECT

cust\_id,

days\_to\_250 AS threshold\_of\_250

FROM customer\_threshold

WHERE days\_to\_250 IS NOT NULL

GROUP BY cust\_id, days\_to\_250;

**Output:**

Table

Description automatically generated