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:

CUSTOMER_ID	REVENUE
12931	42055.96
12748	33719.73
12901	17654.54
12921	16587.09
12939	11581.8
12830	6814.64
12839	5591.42

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:

STOCKCODE	TOTAL_QUANTITY
84077	7824
84879	6117
22197	5918
21787	5075
21977	4691
21703	2996
17096	2019
15036	1920
23203	1803

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
```

Output:

CUSTOMER_ID	ORDERMONTH	REVENUEPERMONTH	TOTALREVENUE	RNK
12931	02-2011	1696.4	42055.96	1
12931	03-2011	62.5	42055.96	1
12931	04-2011	1488	42055.96	1
12931	05-2011	496.8	42055.96	1
12931	08-2011	28610	42055.96	1
12931	10-2011	1909.36	42055.96	1
12931	11-2011	7615.9	42055.96	1
12931	12-2010	177	42055.96	1
12748	01-2011	418.77	33719.73	2
12748	02-2011	389.64	33719.73	2
12748	03-2011	1179.37	33719.73	2
12748	04-2011	1100.37	33719.73	2
12748	05-2011	2234.5	33719.73	2
12748	06-2011	2006.26	33719.73	2
12748	07-2011	1113.27	33719.73	2

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.

*/
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;
```

Output:

ORDERMONTH	REVENUEPERMONTH
2010-12	13422.96
2011-01	9541.29
2011-02	13336.84
2011-03	17038.01
2011-04	10980.51
2011-05	19496.18
2011-06	13517.01
2011-07	15664.54
2011-08	38374.64
2011-09	27853.82

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;
```

Output:

CUSTOMER_ID	TOTAL_REV...	PERCENTAGE_OF_TOTAL_REVENUE
12931	42055.96	16.45%
12748	33719.73	13.19%
12901	17654.54	6.9%
12921	16587.09	6.49%
12939	11581.8	4.53%
12830	6814.64	2.66%
12839	5591.42	2.19%
12971	5190.74	2.03%
12955	4757.16	1.86%
12747	4196.01	1.64%

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:

CUSTOMER_ID	TOTAL_REVENUE	MOST_POPULAR_PRODUCT	TOTAL_QUANTITY	PRODUCT_REVENUE
12931	42055.96	22197	5340	3844.8
12748	33719.73	21135	595	147.31
12901	17654.54	84077	6768	1482.72
12921	16587.09	84879	320	540.8
12939	11581.8	22570	624	2115.36
12830	6814.64	21703	2880	777.6
12839	5591.42	22197	252	214.2
12971	5190.74	40016	1200	300
12955	4757.16	15039	188	159.8
12747	4196.01	82484	240	1519.2

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
```

```
  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:

STOCKCODE	MONTH	MONTHLY_REVENUE	TOTAL_REVENUE
21479	2010-12	52.5	2736.01
21479	2011-01	15	2736.01
21479	2011-08	1305.51	2736.01
21479	2011-09	42.5	2736.01
21479	2011-10	68	2736.01
21479	2011-11	1214.25	2736.01
21479	2011-12	38.25	2736.01
21787	2010-12	3.4	4059.35
21787	2011-02	0.85	4059.35

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:

STOCKCODE	TOTALQUANTITY	TOTALREVENUE
84077	7824	1788.96
84879	6117	9114.69
22197	5918	4323.1
21787	5075	4059.35
21977	4691	2063.69
21703	2996	826.32
17096	2019	343.23
15036	1920	1329.36
23203	1803	3357.44
21790	1579	1011.67

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:

CUSTOMER_ID	NUMBEROFORDERS	TOTAL_SOLD_QUANTITIES	AVGBASKETSIZE
12875	2	2019	1009.5
12908	2	1200	600
12931	82	28004	341.51
12891	3	950	316.67
12830	38	9848	259.16
12901	116	23075	198.92
12939	47	4876	103.74
12864	3	216	72

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;
```

Output:

CUSTOMER_ID	NUMBEROFORDERS	TOTAL_SOLD_QUANTITIES	AVGBASKETSIZE
12875	2	2019	1009.5
12908	2	1200	600
12931	82	28004	341.51
12891	3	950	316.67
12830	38	9848	259.16
12901	116	23075	198.92
12939	47	4876	103.74
12864	3	216	72
12917	2	120	60
12950	23	1380	60
12823	5	230	46
12863	5	188	37.6
12879	10	370	37
12829	11	376	34.18

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:

CUST_ID	MAX_CONSECUTIVE_DAYS
625527	37
646553	8
664981	5
811892	43
1073321	3
1331618	12
1357409	20
1444226	3
1490919	3
1505992	34
1519955	60
1798791	5
1807965	13
1988054	18

- 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:

CUST_ID	THRESHOLD_OF_250
151293	10
259866	22
664981	12
811892	52
1331618	18
1505992	44
2146867	60
2208179	23
2284100	61
2792126	34
2842713	59
3130002	47
3259850	6
3394425	61
3399415	16