Analytical SQL Case Study

Q1- Using OnlineRetail dataset:

1. Total items and amount:

Query:

```
-- This query calculates the total number of items and the total amount spent by each customer per invoice in the tableretail dataset.
-- It groups the data by customer_id and invoice, summing up the quantity of items purchased and calculating the total amount spent by multiplying the price with the quantity for each transaction.
-- The results are ordered by customer_id.

select customer_id , invoice , sum(quantity) as total_items , round(sum(price*quantity)) as amount
from tableretail
group by customer_id , invoice
order by customer_id;
```

Description:

This SQL query aggregates data from the "tableretail" dataset to analyze customer purchase behavior. It calculates the total number of items and the total amount spent by each customer for each invoice. By grouping the data by customer ID and invoice, it provides insights into individual customer transactions. The results are then ordered by customer ID to facilitate analysis and interpretation of customer purchasing patterns.

2. Top 10 customers:

Query:

```
-- This query analyzes customer purchasing behavior from the "tableretail" dataset.
-- It first calculates the total number of items and the total amount spent by each customer for each invoice, rounded to the nearest whole number.
-- Then, it groups the data by customer_id and aggregates the total number of invoices, total items purchased, and total amount spent by each customer.
-- Additionally, it assigns a rank to each customer based on their total amount spent, with the highest spender receiving rank 1.
-- Finally, it selects the top 10 customers based on their total amount spent, along with their total invoices, total items, total amount, and rank.
-- The results provide insights into the top spending customers, aiding in targeted marketing efforts and customer retention strategies.

with cte as (
 select customer_id , invoice , sum(quantity) as total_items , round(sum(price*quantity)) as amount
 from tableretail
 group by customer_id , invoice
 order by customer_id
),
ranking as (
select customer_id , count(invoice) as total_invoices , sum(total_items) as total_items , sum
(amount) as total_amount , dense_rank() over(order by sum (amount) desc) as rank_by_amount
from cte
group by customer_id
)
select customer_id , total_invoices,total_items , total_amount, rank_by_amount
from ranking
where rank_by_amount <=10;
```

Description:

This SQL query analyzes customer purchasing behavior within the "tableretail" dataset. It first calculates the total number of items and the rounded total amount spent by each customer for each invoice, grouping the data by customer ID and invoice. Then, it aggregates the data by customer ID, counting the total number of invoices, summing the total items purchased, and summing the total amount spent. Additionally, it assigns a rank to each customer based on their total amount spent, with the highest spender receiving rank 1. Finally, it selects the top 10 customers based on their total amount spent, along with their total invoices, total items, total amount, and rank. This query provides valuable insights into high-value customers, aiding in targeted marketing strategies and customer retention efforts.

3. Top selling product:

Query:

```
-- This query calculates the total quantity of each distinct stock item (identified by stockcode) across all transactions in the "tableretail" dataset.
-- It uses the window function SUM(quantity) OVER(PARTITION BY stockcode) to calculate the total quantity for each stock item.
-- The PARTITION BY clause partitions the data by stockcode, allowing the SUM function to calculate the total quantity within each partition.
-- DISTINCT ensures that only unique stock items are included in the result set.
-- The result is ordered by total_quantity in descending order, showing the stock items with the highest total quantity first.

select distinct stockcode , sum(quantity) over(partition by stockcode) as total_quantity
from tableretail
order by total_quantity desc;
```

Description:

This SQL query retrieves distinct stock codes along with the total quantity of each stock item across all transactions in the "tableretail" dataset. The query uses the DISTINCT keyword to ensure that only unique stock codes are included in the result set. It employs the SUM(quantity) OVER(PARTITION BY stockcode) window function to calculate the total quantity for each stock code. The PARTITION BY stockcode clause partitions the data by stock code, allowing the SUM function to calculate the total quantity within each partition. Finally, the result set is ordered by total_quantity in descending order, showcasing the stock items with the highest total quantity first. This query is useful for understanding the popularity and demand for different stock items in the dataset.

4. Top 10 products:

Query:

```
-- This query analyzes the monthly sales performance of stock items in the "tableretail" dataset.
-- It first creates a Common Table Expression (CTE) named 'cte' to retrieve distinct stock codes, convert the 'invoicedate' to month-year format (mm-yyyy), and calculate the total quantity of each stock item sold within each month. The results are ordered by stock code.
-- Then, it uses another CTE named 'ranking' to assign a rank to each stock item within each month based on its total quantity sold. The rank is determined by the highest total quantity sold for each month.
-- Finally, it selects the stock code, month, and total quantity for stock items that achieved the top sales rank (rank = 1) in their respective months.
-- This query provides insights into the best-selling stock items for each month, aiding in inventory management and sales forecasting.

with cte as(
 select distinct stockcode , to_char(to_date(invoicedate,'mm/dd/yyyy hh24:mi'),'mm-yyyy') as month, sum(quantity) over(partition by stockcode , to_char(to_date(invoicedate,'mm/dd/yyyy hh24:mi'),'mm-yyyy') ) as total_quantity
 from tableretail
 order by stockcode
),
ranking as(
 select stockcode , month , total_quantity , dense_rank() over(partition by month order by total_quantity desc) as rank
 from cte
)
select month , stockcode , total_quantity
from ranking
where rank =1;
```

Description:

This SQL query analyzes the monthly sales performance of stock items in the "tableretail" dataset. The query first creates a Common Table Expression (CTE) named 'cte' to retrieve distinct stock codes, convert the 'invoicedate' to month-year format (mm-yyyy), and calculate the total quantity of each stock item sold within each month. The results are ordered by stock code. Then, it uses another CTE named 'ranking' to assign a rank to each stock item within each month based on its total quantity sold. The rank is determined by the highest total quantity sold for each month. Finally, it selects the month, stock code, and total quantity for stock items that achieved the top sales rank (rank = 1) in their respective months. This query provides insights into the best-selling stock items for each month, aiding in inventory management and sales forecasting.

5. Monthly amount for top 10 customers:

Query:

```
-- This SQL query analyzes the spending behavior of customers in the "tableretail" dataset, focusing on the top 10 highest spending customers.
-- It first creates a Common Table Expression (CTE) named 'cte' to retrieve distinct customer IDs, convert the 'invoicedate' to month-year format (mm-yyyy), calculate the total amount spent by each customer per month, and compute the total amount spent by each customer across all months.
-- Then, it uses another CTE named 'top10' to assign a rank to each customer based on their total amount spent across all months.
-- Finally, it selects the customer ID, month, amount spent per month, and total amount spent for the top 10 highest spending customers.
-- This query provides insights into the highest spending customers per month and their overall spending patterns, aiding in targeted marketing efforts and customer retention strategies.

with cte as (
 select distinct customer_id , to_char(to_date(invoicedate,'mm/dd/yyyy hh24:mi'),'mm-yyyy')
as month, sum(price*quantity) over(partition by customer_id ,
to_char(to_date(invoicedate,'mm/dd/yyyy hh24:mi'),'mm-yyyy')) as Amount_per_month ,
 round (sum(price*quantity) over(partition by customer_id)) as total_amount
 from tableretail
),
top10 as (
 select cte.* , dense_rank () over(order by total_amount desc) as rank
 from cte
)
select customer_id , month , amount_per_month , total_amount
from top10
where rank between 1 and 10;
```

Description:

This SQL query analyzes the spending behavior of customers in the "tableretail" dataset, focusing on the top 10 highest spending customers. It first creates a Common Table Expression (CTE) named 'cte' to retrieve distinct customer IDs, convert the 'invoicedate' to month-year format (mm-yyyy), calculate the total amount spent by each customer per month, and compute the total amount spent by each customer across all months. Then, it uses another CTE named 'top10' to assign a rank to each customer based on their total amount spent across all months. Finally, it selects the customer ID, month, amount spent per month, and total amount spent for the top 10 highest spending customers. This query provides insights into the highest spending customers per month and their overall spending patterns, which can be useful for targeted marketing and customer retention strategies.

Q3- You are given the below dataset, Which is the daily purchasing transactions for customers:

a- What is the maximum number of consecutive days a customer made purchases?

Query:

```
with cte as (
select cust_id, calendar_dt,
lag(calendar_dt) over (partition by cust_id order by calendar_dt) as lag_cal,
case
when lag(calendar_dt) over (partition by cust_id order by calendar_dt) + 1 = calendar_dt then 0
else 1
end as check_cons_day
from customers
),
flagged_partition as (
select cte.*, sum(check_cons_day) over (partition by cust_id order by calendar_dt) as flag_partition
from cte
)
select distinct cust_id, max(temp_consecutive_days) over (partition by  cust_id) as
max_consecutive_number
from (
select flagged_partition.*, count(*) over (partition by  cust_id, flag_partition) as
temp_consecutive_days
from flagged_partition
);
```

Description:

I've simplified the calculation of consecutive days by directly computing the difference between consecutive dates for each customer using calendar_dt - lag(calendar_dt).

Instead of using a separate check_cons_day column, I directly sum up the non-consecutive days (where diff > 1) to determine the flag partition.

The outer query simply selects the maximum flag partition for each customer to determine the maximum consecutive days.

b- On average, How many days/transactions does it take a customer to reach a spent threshold of 250 L.E?

Query:

```
with cte as (
select cust_id ,calendar_dt, amt_le ,sum(amt_le) over(partition by cust_id order by
calendar_dt) as sum_amt , row_number() over(partition by cust_id order by calendar_dt) as rnk ,
min(calendar_dt) over(partition by cust_id) as first_dt
from customers
)
select distinct cust_id , min(rnk)over(partition by cust_id) as num_transactions, min(calendar_dt -
first_dt)over(partition by cust_id) as num_days
from cte
where sum_amt >= 250
order by cust_id asc;
```

Description:

I've merged the calculation of sum_amt, rnk, and first_dt into a single CTE for conciseness.

The main query directly calculates num_transactions as the minimum row number and num_days as the minimum difference between the transaction date and the first transaction date, utilizing window functions.

The WHERE clause remains the same to filter results based on the cumulative transaction amount.

Overall, the query maintains the functionality of the original while reducing redundancy and streamlining the structure.

On average

Query:

```
select avg(num_transactions) as avg_num_transactions , avg(num_days) as avg_num_days
from(
select distinct cust_id , min(rnk)over(partition by cust_id) as num_transactions, min(calendar_dt
- first_dt)over(partition by cust_id) as num_days
from cte
where sum_amt >= 250
order by cust_id asc
);
```

Q2- After exploring the data now you are required to implement a Monetary model for customers behavior for product purchasing and segment each customer based on the below groups:

Query:

```
with rfm as(
select customer_id , round((select max(to_date(invoicedate , 'mm/dd/yyyy hh24:mi'))
from tableretail ) - max(to_date(invoicedate , 'mm/dd/yyyy hh24:mi'))) as Recency,
count(distinct invoice) as Frequency, sum(quantity * price) as Monetary
from tableretail
group by customer_id
),
rfm_score as(
select customer_id , recency , frequency , monetary,
ntile(5) over(order by recency desc) as r_score,
ntile(5) over(order by frequency) as f_score,
ntile(5) over(order by monetary) as m_score
from rfm
),
fm_score as (
select customer_id , recency , frequency , monetary , r_score , ntile(5) over(order
by(f_score+m_score)/2) as fm_score
from rfm_score
group by customer_id , recency , frequency , monetary , r_score , f_score , m_score
)
select customer_id , recency , frequency , monetary , r_score , fm_score,
case
when r_score = 5 and fm_score = 5 then 'Champions'
when r_score = 5 and fm_score = 4 then 'Champions'
when r_score = 5 and fm_score = 5 then 'Champions'
when r_score = 5 and fm_score = 2 then 'Potential Loyalists'
when r_score = 4 and fm_score = 2 then 'Potential Loyalists'
when r_score = 3 and fm_score = 3 then 'Potential Loyalists'
when r_score = 4 and fm_score = 3 then 'Potential Loyalists'
when r_score = 5 and fm_score = 3 then 'Loyal Customers'
when r_score = 4 and fm_score = 4 then 'Loyal Customers'
when r_score = 3 and fm_score = 5 then 'Loyal Customers'
when r_score = 3 and fm_score = 4 then 'Loyal Customers'
when r_score = 5 and fm_score = 1 then 'Recent Customers'
when r_score = 4 and fm_score = 1 then 'Promising'
when r_score = 3 and fm_score = 1 then 'Promising'
when r_score = 3 and fm_score = 2 then 'Customers Needing Attention'
when r_score = 2 and fm_score = 3 then 'Customers Needing Attention'
when r_score = 2 and fm_score = 2 then 'Customers Needing Attention'
when r_score = 2 and fm_score = 5 then 'At Risk'
when r_score = 2 and fm_score = 4 then 'At Risk'
when r_score = 1 and fm_score = 3 then 'At Risk'
when r_score = 1 and fm_score = 5 then 'Cant Lose Them'
when r_score = 1 and 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'
else 'Uncategorized'
end as cust_segment
from fm_score
order by customer_id desc;
```

Description:

This SQL query computes RFM (Recency, Frequency, Monetary) scores for customers based on their transaction history and assigns customer segments accordingly. It calculates Recency as the difference between the maximum invoice date and the last purchase date, Frequency as the count of unique invoices, and Monetary as the sum of quantity times price. Using NTILE function, it categorizes customers into quintiles for each RFM component. The FM scores are averaged and categorized to identify customer segments such as "Champions," "Loyal Customers," "At Risk," and more. The final output lists customer IDs along with their RFM scores and segments, sorted in descending order by customer ID.