# Analytical SQL Project

**Prepared by :**

**Ibrahim El-Nwasany**

## 1st Query

```
select distinct invoicedate, sum(quantity) over (partition by invoicedate) as
dailysales
from tableretail
order by invoicedate;
```

| INVOICEDATE | DAILYSALES |
|---|---|
| 1/11/2011 16:24 | 38 |
| 1/12/2011 14:43 | 64 |
| 1/12/2011 15:25 | 168 |
| 1/13/2011 10:33 | 135 |
| 1/14/2011 11:50 | 36 |
| 1/16/2011 11:49 | 17 |
| 1/16/2011 13:04 | 7 |
| 1/17/2011 10:52 | 116 |
| 1/17/2011 12:34 | 146 |
| 1/18/2011 10:15 | 49 |
| 1/19/2011 12:38 | 133 |
| 1/19/2011 12:52 | 266 |
| 1/20/2011 14:01 | 88 |
| 1/21/2011 14:04 | 6 |
| 1/21/2011 14:21 | 2 |
| 1/23/2011 10:43 | 325 |

4: 1 Row 1 of 714 total rows HR@XE Modified

1. **Calculate Daily Sales Quantity:** This query helps in understanding the daily sales volume over time. It provides insights into the pattern of sales fluctuations, allowing the business to identify peak sales days and potential factors driving them.

## 2nd Query

```
select stockcode,
       totalquantity,
       rank() over (order by totalquantity desc) as rank
```

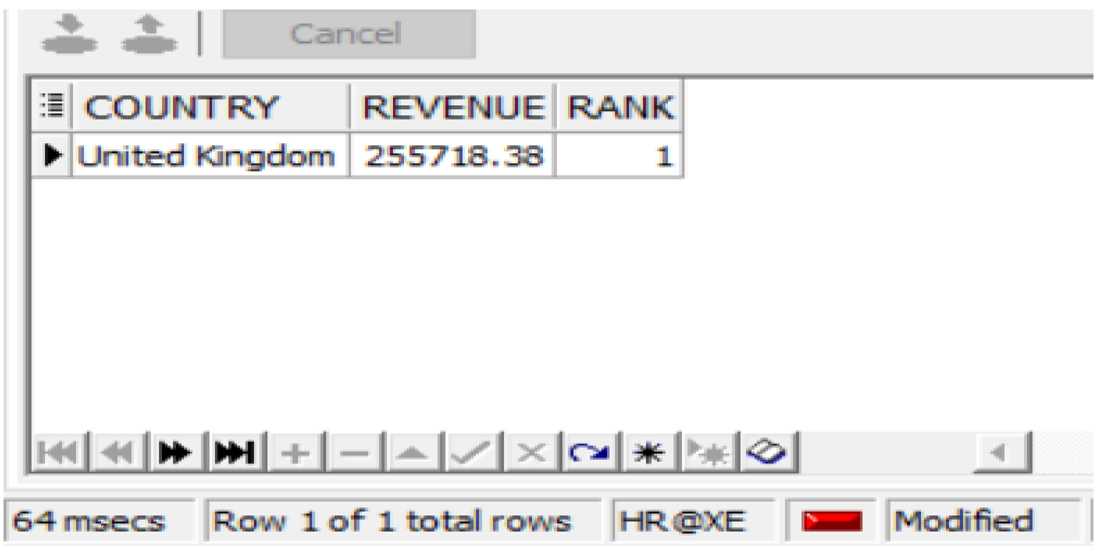
```sql
from (select stockcode,
            sum(quantity) as totalquantity
      from tableretail
      group by stockcode
      order by sum(quantity) desc
     ) subquery
where rownum <= 10;
```

| STOCKCODE | TOTALQUANTITY | RANK |
|---|---|---|
| 84077 | 7824 | 1 |
| 84879 | 6117 | 2 |
| 22197 | 5918 | 3 |
| 21787 | 5075 | 4 |
| 21977 | 4691 | 5 |
| 21703 | 2996 | 6 |
| 17096 | 2019 | 7 |
| 15036 | 1920 | 8 |
| 23203 | 1803 | 9 |
| 21790 | 1579 | 10 |

**2.Identify Top Selling Products by Quantity:** By ranking products based on their sales quantity, this query helps in identifying the best-performing products. It enables the business to focus on stocking and promoting high-demand products to maximize sales and profitability.

## 3rd Query

```sql
select country,
       sum(quantity * price) as revenue,
       dense_rank() over (order by sum(quantity * price) desc) as
rank
from tableretail
group by country
order by revenue desc;
```



| COUNTRY | REVENUE | RANK |
|---|---|---|
| United Kingdom | 255718.38 | 1 |



**Identify Our only (UK) Country with its Highest Revenue:** This query identifies the country contributing the most to revenue. It helps in prioritizing marketing efforts, understanding international market dynamics, and making informed decisions regarding expansion or localization strategies.

## 4th Query

```
with customerspending as (
   select customer_id, sum(quantity * price) as totalspending
   from tableretail
   group by customer_id
)
select customer_id, totalspending,
     rank() over (order by totalspending desc) as spendingrank
from customerspending;
```

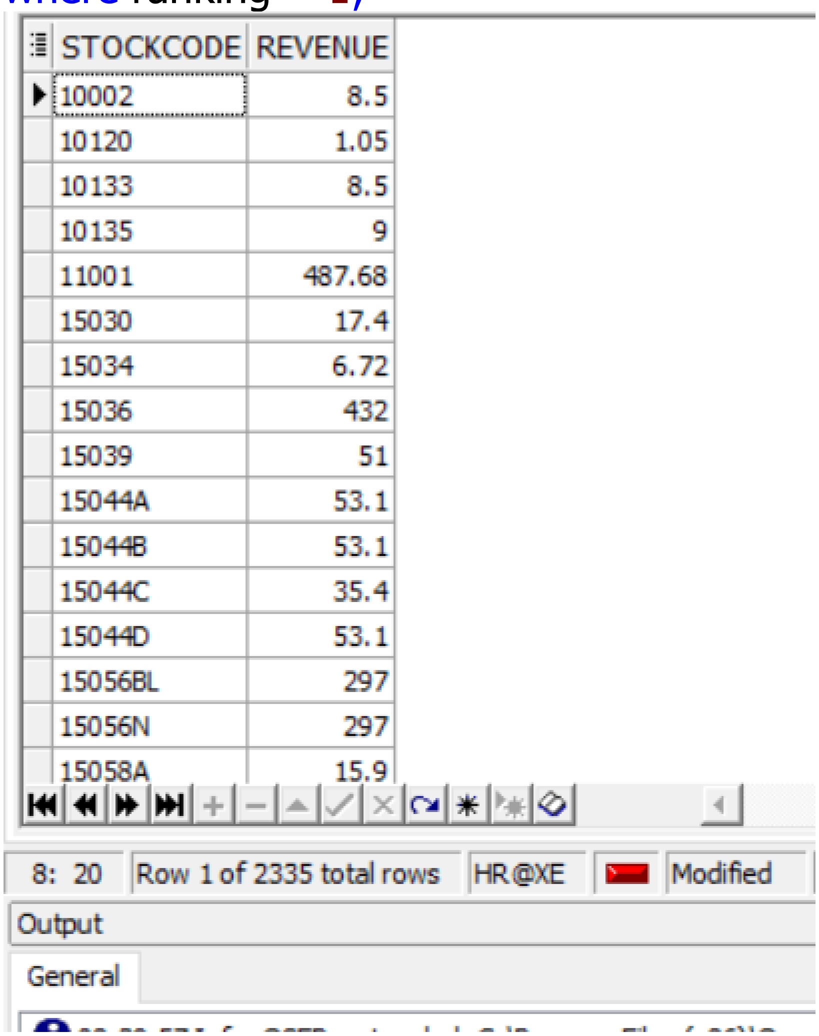
| CUSTOMER_ID | TOTALSPENDING | SPENDINGRANK |
|---|---|---|
| 12931 | 42055.96 | 1 |
| 12748 | 33719.73 | 2 |
| 12901 | 17654.54 | 3 |
| 12921 | 16587.09 | 4 |
| 12939 | 11581.8 | 5 |
| 12830 | 6814.64 | 6 |
| 12839 | 5591.42 | 7 |
| 12971 | 5190.74 | 8 |
| 12955 | 4757.16 | 9 |
| 12747 | 4196.01 | 10 |
| 12949 | 4167.22 | 11 |
| 12749 | 4090.88 | 12 |
| 12867 | 4036.82 | 13 |
| 12841 | 4022.35 | 14 |
| 12957 | 4017.54 | 15 |
| 12910 | 3075.04 | 16 |

66 msecs | Row 1 of 110 total rows | HR@XE | Modified

**4. Rank Customers by Total Spending:** This query ranks customers based on their total spending. It calculates the total spending for each customer and assigns a rank based on their total spending, with the highest spender receiving rank 1.

## 5th Query

```sql
with productrevenue as (
    select stockcode,
           quantity * price as revenue,
           row_number() over (partition by stockcode order by (quantity * price) desc) as ranking
    from tableretail
)
select stockcode, revenue
from productrevenue
where ranking = 1;
```

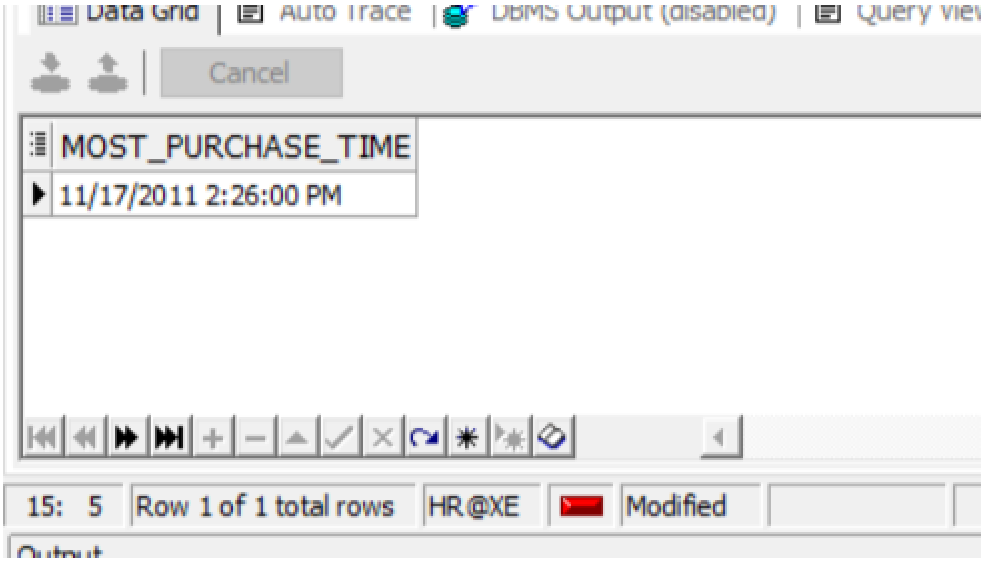


| STOCKCODE | REVENUE |
| --- | --- |
| 10002 | 8.5 |
| 10120 | 1.05 |
| 10133 | 8.5 |
| 10135 | 9 |
| 11001 | 487.68 |
| 15030 | 17.4 |
| 15034 | 6.72 |
| 15036 | 432 |
| 15039 | 51 |
| 15044A | 53.1 |
| 15044B | 53.1 |
| 15044C | 35.4 |
| 15044D | 53.1 |
| 15056BL | 297 |
| 15056N | 297 |
| 15058A | 15.9 |

**5.Calculate the Total Revenue for Each Product**: This query calculates the total revenue generated by each product using a common table expression (CTE) to calculate revenue for each transaction and a window function to rank the revenue within each product. It selects the top revenue for each product.

## 6th Query

```
with purchasetimes as (
    select
        to_date(invoicedate, 'MM/DD/YYYY HH24:MI') as most_purchase_time,
        count(*) as purchasecount,
        rank() over (order by count(*) desc) as rankbypurchasecount
    from
        tableretail
    where
        country = 'United Kingdom'
    group by
        to_date(invoicedate, 'MM/DD/YYYY HH24:MI')
)
select
    most_purchase_time
from
    purchasetimes
where
    rankbypurchasecount = 1;
```




**6.calculate swarm time purchase**: This query identifies the most frequent purchase time in the United Kingdom by counting the number of transactions for each unique purchase timestamp. It utilizes the `tableretail` dataset, filtering for transactions made in the United Kingdom. By grouping the transactions by their purchase timestamp and calculating the count of purchases for each timestamp, it determines the most common purchase time. The `rankbypurchasecount` column assigns a rank to each purchase time based on the frequency of purchases, with the most frequent purchase time receiving a rank of 1. Finally, the query selects the most common purchase time with the highest rank.

## 7th Query

```sql
select
    customer_id,
    invoicedate,
    total_sales,
    avg(total_sales) over (
        partition by customer_id
        order by to_date(invoicedate, 'MM/DD/YYYY HH24:MI')
        rows between 90 preceding and current row
    ) as moving_avg_sales
from (
```

```
    select
        customer_id,
        invoicedate,
        total_sales
    from (
        select
            customer_id,
            invoicedate,
            sum(quantity * price) over (partition by customer_id order by
to_date(invoicedate, 'MM/DD/YYYY HH24:MI')) as total_sales,
            row_number() over (partition by customer_id order by
to_date(invoicedate, 'MM/DD/YYYY HH24:MI')) as row_num
        from
            tableretail
    ) salesdata
    where row_num = 1
) earlysalesdata;
```

| CUSTOMER_ID | INVOICEDATE | TOTAL_SALES | MOVING_AVG_SALES |
|---|---|---|---|
| 12747 | 12/5/2010 15:38 | 358.56 | 358.56 |
| 12748 | 12/1/2010 12:48 | 4.95 | 4.95 |
| 12749 | 5/10/2011 15:25 | 859.1 | 859.1 |
| 12820 | 1/17/2011 12:34 | 170.46 | 170.46 |
| 12821 | 5/9/2011 15:51 | 92.72 | 92.72 |
| 12822 | 9/13/2011 13:46 | 690.9 | 690.9 |
| 12823 | 2/16/2011 12:15 | 306 | 306 |
| 12824 | 10/11/2011 12:49 | 397.12 | 397.12 |
| 12826 | 12/9/2010 15:21 | 155 | 155 |
| 12827 | 10/26/2011 15:44 | 217.75 | 217.75 |
| 12828 | 8/1/2011 16:16 | 227 | 227 |
| 12829 | 12/14/2010 14:54 | 85.75 | 85.75 |
| 12830 | 6/21/2011 10:53 | 2221.84 | 2221.84 |
| 12831 | 3/22/2011 13:02 | 215.05 | 215.05 |
| 12832 | 9/2/2011 13:48 | 267.8 | 267.8 |

21: 13 | Row 1 of 110 total rows | HR@XE | Modified

**7.Average Sales Rolling**: This query calculates the moving average sales for each customer over a rolling window of 90 days. It first computes the total sales for each customer on each transactional date by summing the product of quantity and price. Then, it assigns a row number for each transaction within each customer's data, ensuring only the first transaction of each customer is considered. Finally, it calculates the moving average sales using a window function, partitioning by customer_id and ordering by the transactional date. The moving average is computed over a window that includes the current transaction and the 90 preceding transactions, providing insights into each customer's sales trend over time.

## 8th Query

```
with lag_lead as (
   select t.*,
         lag(quantity) over (partition by invoice order by to_date(invoicedate,
'MM/DD/YYYY HH24:MI')) as lag_quantity,
         lead(quantity) over (partition by invoice order by
to_date(invoicedate, 'MM/DD/YYYY HH24:MI')) as lead_quantity
   from tableretail t
),
stats as (
   select stockcode,
         count(distinct price) as distinct_prices,
         avg(price) as avg_price,
         case when count(distinct price) <= 1 then 0 else stddev_pop(price)
end as std_dev_price,
         case when count(distinct price) <= 1 then 0 else variance(price) end
as variance_price
   from tableretail
   group by stockcode
)
select
   s.stockcode,
```

```sql
 s.distinct_prices,
    round(s.avg_price, 2) as avg_price,
    round(s.std_dev_price, 2) as std_dev_price,
    round(s.variance_price, 2) as variance_price,
    round(covar_pop(ll.quantity, ll.price), 2) as covariance,
    round(corr(ll.quantity, ll.price), 2) as correlation
from stats s
left join lag_lead ll on s.stockcode = ll.stockcode
group by s.stockcode, s.distinct_prices, s.avg_price, s.std_dev_price,
s.variance_price;
```

| STOCKCODE | DISTINCT_PRICES | AVG_PRICE | STD_DEV_PRICE | VARIANCE_PRICE | COVARIANCE | CORRELATION |
|---|---|---|---|---|---|---|
| 23222 | 1 | 0.83 | 0 | 0 | 0 | |
| 21944 | 1 | 0.39 | 0 | 0 | 0 | |
| 21749 | 1 | 2.1 | 0 | 0 | 0 | |
| 37495 | 1 | 3.75 | 0 | 0 | 0 | |
| 22801 | 2 | 3.65 | 0.16 | 0.03 | -0.88 | -0.99 |
| 22861 | 1 | 1.65 | 0 | 0 | 0 | |
| 23147 | 1 | 1.45 | 0 | 0 | 0 | |
| 21458 | 1 | 1.25 | 0 | 0 | 0 | |
| 37479P | 1 | 0.39 | 0 | 0 | 0 | |
| 22779 | 1 | 4.25 | 0 | 0 | 0 | |
| 22572 | 1 | 0.85 | 0 | 0 | 0 | |
| 23307 | 1 | 0.55 | 0 | 0 | 0 | |
| 23583 | 1 | 1.65 | 0 | 0 | 0 | |
| 23528 | 1 | 3.75 | 0 | 0 | 0 | |
| 21391 | 1 | 0.75 | 0 | 0 | 0 | |

27: 1 | Row 1 of 2335 total rows | HR@XE | Modified

Output

**8.Stastistical analysis** : This SQL query calculates various statistical measures related to the prices and quantities of products in the `tableretail` dataset. It utilizes common table expressions (CTEs) to first compute lag and lead quantities for each invoice, allowing for comparisons with the previous and subsequent transaction quantities. Then, it calculates statistics such as distinct prices, average price, standard deviation of price, variance of price, covariance between quantity and price, and correlation between quantity and price for each unique stock code. These statistical measures provide insights into the pricing patterns and relationships between quantities and prices for different products in the dataset, aiding in better understanding the dynamics of sales and pricing.

# Monetary model for customers behavior for product purchasing and segment each customer

```
with maxinvoicedate as (
    select max(to_timestamp(invoicedate, 'MM/DD/YYYY HH24:MI')) as
max_date
    from tableretail
),
customerrfm as (
    select
        customer_id,
        ceil(extract(day from (select max_date from maxinvoicedate) -
max(to_timestamp(invoicedate, 'MM/DD/YYYY HH24:MI')))) as recency,
        count(distinct invoice) as frequency,
        round(sum(quantity * price) / 1000, 2) as monetary
    from
        tableretail
    group by
        customer_id
),
rfmscores as (
    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,
        round((ntile(5) over(order by frequency) + ntile(5) over(order by
monetary))/2,0)  as fm_score
    from
        customerrfm
),
customersegment as (
    select
        customer_id,
        recency,
        frequency,
        monetary,
        r_score,
        fm_score,
        case
            when (r_score >= 5 and fm_score >= 5)
                or (r_score >= 5 and fm_score = 4)
                or (r_score = 4 and fm_score >= 5) then 'champions'
            when (r_score >= 5 and fm_score = 2)
                or (r_score = 4 and fm_score = 2)
                or (r_score = 3 and fm_score = 3)
                or (r_score = 4 and fm_score >= 3) then 'potential loyalists'
            when (r_score >= 5 and fm_score = 3)
                or (r_score = 4 and fm_score = 4)
                or (r_score = 3 and fm_score >= 5)
                or (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)
                or (r_score = 3 and fm_score = 1) then 'promising'
            when (r_score = 3 and fm_score = 2)
                or (r_score = 2 and fm_score = 3)
                or (r_score = 2 and fm_score = 2) then 'customers needing
attention'
            when (r_score = 2 and fm_score >= 5)
```

```sql
                or (r_score = 2 and fm_score = 4)
                or (r_score = 1 and fm_score = 3) then 'at risk'
            when (r_score = 1 and fm_score >= 5)
                or (r_score = 1 and fm_score = 4) then 'cant lose them'
            when (r_score = 1 and fm_score = 2)
                or (r_score = 2 and fm_score = 1) then 'hibernating'
            when r_score = 1 and fm_score <= 1 then 'lost'
            else 'other'
        end as cust_segment
    from
        rfmscores
)
select
    customer_id,
    recency,
    frequency,
    monetary,
    r_score,
    fm_score,
    cust_segment
from
    customersegment
order by
    customer_id;
```

| CUSTOMER_ID | RECENCY | FREQUENCY | MONETARY | R_SCORE | FM_SCORE | CUST_SEGMENT |
|---|---|---|---|---|---|---|
| 12747 | 1 | 11 | 4.2 | 5 | 5 | champions |
| 12748 | 0 | 210 | 33.72 | 5 | 5 | champions |
| 12749 | 3 | 5 | 4.09 | 5 | 5 | champions |
| 12820 | 2 | 4 | 0.94 | 5 | 3 | loyal customers |
| 12821 | 213 | 1 | 0.09 | 1 | 2 | hibernating |
| 12822 | 70 | 2 | 0.95 | 3 | 3 | potential loyalists |
| 12823 | 74 | 5 | 1.76 | 2 | 4 | at risk |
| 12824 | 58 | 1 | 0.4 | 3 | 2 | customers needing attention |
| 12826 | 2 | 7 | 1.47 | 5 | 5 | champions |
| 12827 | 5 | 3 | 0.43 | 5 | 3 | loyal customers |
| 12828 | 2 | 6 | 1.02 | 5 | 4 | champions |
| 12829 | 336 | 2 | 0.29 | 1 | 2 | hibernating |
| 12830 | 37 | 6 | 6.81 | 3 | 5 | loyal customers |
| 12831 | 261 | 1 | 0.22 | 1 | 1 | lost |
| 12832 | 31 | 2 | 0.38 | 3 | 2 | customers needing attention |
| 12833 | 144 | 1 | 0.42 | 2 | 2 | customers needing attention |
| 12834 | 282 | 1 | 0.31 | 1 | 1 | lost |

108 msecs | Row 1 of 110 total rows | HR@XE | Modified

## Analysis Description

The provided SQL code performs a comprehensive analysis of customer behavior based on Recency, Frequency, and Monetary (RFM) metrics. RFM analysis is a widely used technique in marketing and customer relationship management (CRM) to segment customers based on their purchasing behavior. The analysis involves several steps, each contributing to the creation of meaningful customer segments.

### *Step 1: Defining MaxInvoiceDate*

- A Common Table Expression (CTE) named **`maxinvoicedate`** is created to determine the maximum invoice date across the dataset (**`tableretail`**). This CTE retrieves the maximum date among all the invoice dates.

### *Step 2: Calculating RFM Metrics*

- Another CTE named `customerrfm` is created to calculate the RFM metrics for each customer:
  - **Recency (R):** Calculated as the difference between the maximum invoice date and the invoice date for each customer, rounded up to the nearest day.
  - **Frequency (F):** Represents the count of distinct invoices for each customer.
  - **Monetary (M):** Indicates the total monetary value of purchases made by each customer, expressed as a percentage of the total monetary value across all customers.

### *Step 3: Assigning RFM Scores*

- The `rfmscores` CTE assigns RFM scores to each customer based on their calculated RFM metrics. The `ntile()` function is used to divide customers into quintiles (5 groups) based on their recency, frequency, and monetary scores separately.

### Step 4: Segmenting Customers

- In the `customersegment` CTE, customers are segmented into different categories based on their RFM scores:
  - Segments like "champions," "loyal customers," "at risk," etc., are defined based on combinations of recency and frequency/monetary scores.
  - Each segment represents a distinct group of customers exhibiting similar purchasing patterns and behaviors.

### *Step 5: Final Output*

- The final SQL query selects customer ID along with their recency, frequency, monetary, RFM scores, and assigned customer segments.
- The results are ordered by customer ID.

### Conclusion

This SQL code efficiently analyzes customer behavior using RFM metrics and segments customers into meaningful categories based on their purchasing patterns. By understanding these segments, businesses can tailor their marketing strategies and customer engagement efforts to better meet the needs of different customer groups, ultimately leading to improved customer satisfaction and retention.

# Q3: Daily purchasing transactions for customers?

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

```
with ranked_transactions as (
    select
        cust_id,
        to_date(calendar_dt, 'YYYY-MM-DD') as calendar_dt,
        row_number() over (partition by cust_id order by
to_date(calendar_dt, 'YYYY-MM-DD')) as rn
    from
        transactions
),
consecutive_days as (
    select
        cust_id,
        calendar_dt,
        calendar_dt - rn as grp
    from
        ranked_transactions
)
select
```

```sql
    cust_id,
    max(count_consecutive_days) as max_consecutive_days
from (
    select
        cust_id,
        count(*) as count_consecutive_days
    from
        consecutive_days
    group by
        cust_id,
        grp
) max_consecutive_days_per_group
group by
    cust_id
order by
    cust_id;
```

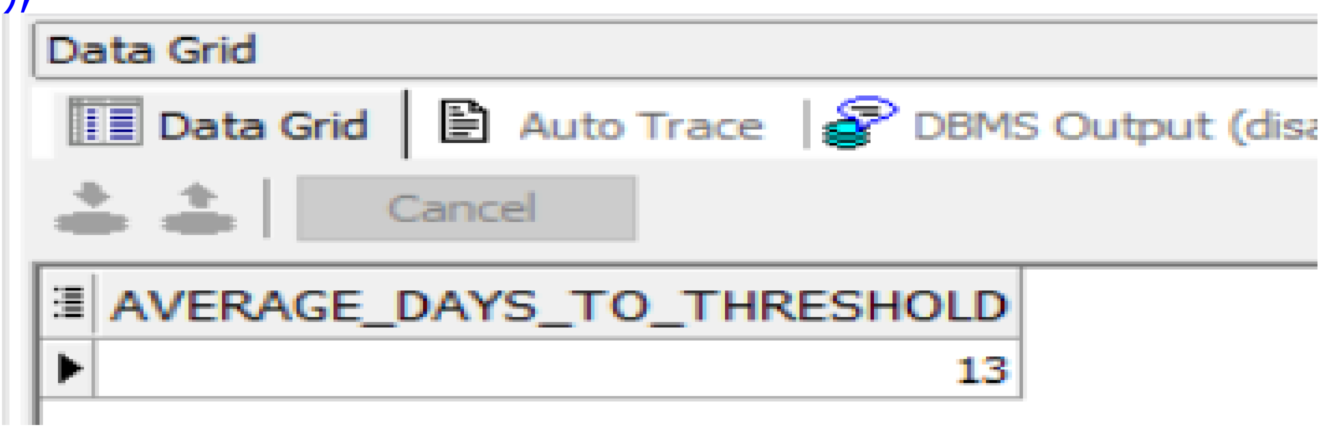
| CUST_ID | MAX_CONSECUTIVE_DAYS |
|---|---|
| 100010376 | 5 |
| 100011085 | 10 |
| 100014033 | 46 |
| 100018482 | 3 |
| 100020880 | 46 |
| 100035887 | 13 |
| 100054374 | 8 |
| 100070652 | 1 |
| 100077596 | 2 |
| 100087785 | 61 |
| 100105254 | 10 |
| 100135808 | 15 |
| 100158557 | 4 |
| 100190321 | 25 |
| 100203920 | 20 |
| 100213644 | 11 |
| 100216969 | 14 |

Description:This SQL query utilizes Common Table Expressions (CTEs) to analyze transaction data and determine the maximum number of consecutive days each customer made purchases. By assigning row numbers and calculating grouping values for consecutive days, it identifies streaks of transactions for each customer. The resulting insights into customer behavior aid in designing targeted marketing strategies, optimizing inventory management, and enhancing customer retention efforts, ultimately driving business growth and profitability.

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

```
with customer_cumulative_spending as (
    select
        cust_id,



to_date(calendar_dt, 'yyyy-mm-dd') as calendar_date,
        sum(to_number(amt_l)) over (partition by cust_id order by
to_date(calendar_dt, 'yyyy-mm-dd')) as cumulative_spending
    from
        transactions
),
threshold_dates as (
    select
        cust_id,
        min(calendar_date) as threshold_date
    from
        customer_cumulative_spending
    where
        cumulative_spending >= 250
    group by
        cust_id
```

```
)
select
    floor(avg(days_to_threshold)) as average_days_to_threshold
from (
    select
        cust_id,
        avg(days_to_threshold) as days_to_threshold
    from (
        select
            ccs.cust_id,
            ccs.calendar_date,
            td.threshold_date,
            ccs.calendar_date - td.threshold_date as days_to_threshold
        from
            customer_cumulative_spending ccs
        join
            threshold_dates td on ccs.cust_id = td.cust_id
    )



  group by
        cust_id
);
```

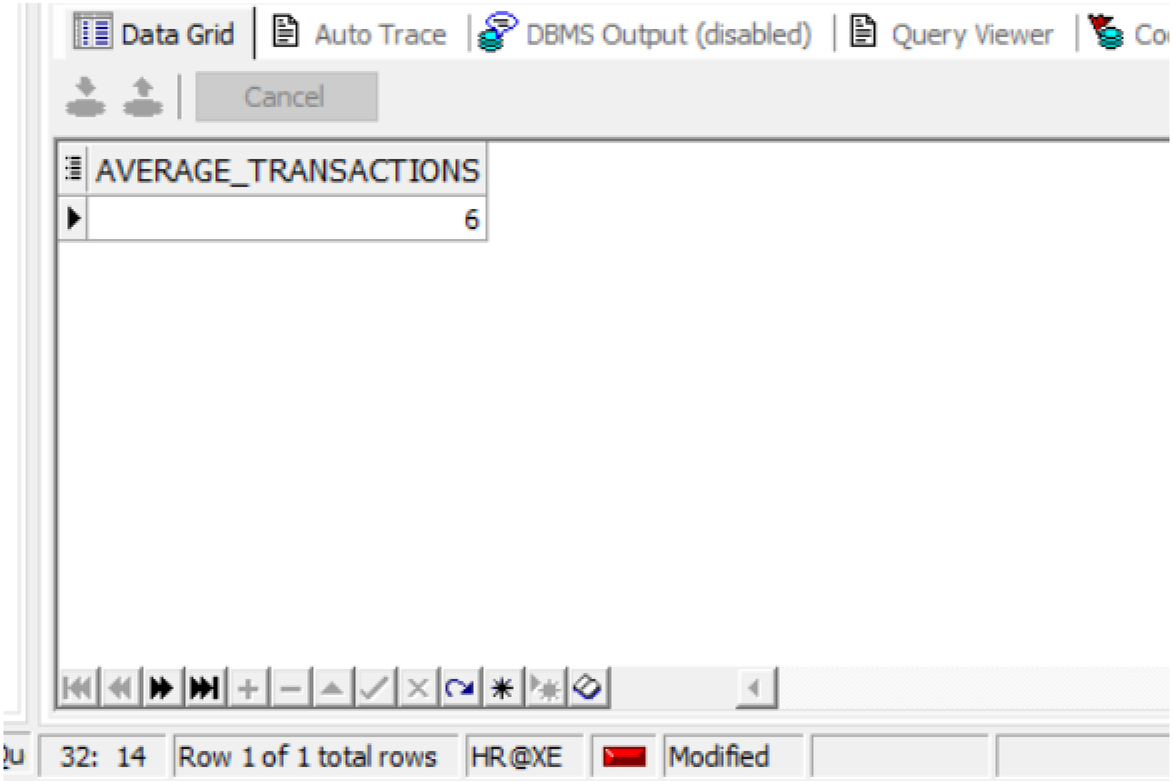



**Description**: This SQL query segments customers based on their cumulative spending reaching a threshold of $250 and calculates the average number of days it takes for customers to reach this threshold. It provides insights into customer spending behavior and helps businesses optimize marketing efforts and enhance customer engagement and loyalty over time.

## Working on Transactions

```
with customer_cumulative_spending as (
    select
        cust_id,
        to_date(calendar_dt, 'yyyy-mm-dd') as calendar_date,
        sum(to_number(amt_l)) over(partition by cust_id order by
to_date(calendar_dt, 'yyyy-mm-dd')) as cumulative_spending
    from
        transactions
),
threshold_dates as (
    select
        cust_id,
        min(calendar_date) as threshold_date
    from
        customer_cumulative_spending
    where
        cumulative_spending >= 250
    group by
        cust_id
)
select
    cast(avg(transaction_count) as int) as average_transactions
from (
    select
        ccs.cust_id,
        count(*) as transaction_count
    from
        customer_cumulative_spending ccs
    join
        threshold_dates td on ccs.cust_id = td.cust_id
    where
        ccs.calendar_date < td.threshold_date
```

```
group by
    ccs.cust_id
);
```




**Description**: This SQL query segments customers based on their cumulative spending reaching a threshold of $250 and calculates the average number of transactions made by customers before reaching this spending threshold. It provides insights into customer spending behavior and helps businesses optimize marketing efforts and enhance customer engagement and loyalty by understanding the frequency of transactions required for customers to reach the spending threshold. This analysis can inform targeted marketing strategies aimed at encouraging more frequent transactions or increasing the average transaction value to expedite customers' progression towards the spending threshold.

# Some Analytics using python

Let’s Start out Amazing Trip:

## 1) Exploration and Investigation

```
In [18]: print(df.head())
```

```
   INVOICE STOCKCODE  QUANTITY       INVOICEDATE  PRICE  CUSTOMER_ID  \
0   537213     21556         1  12/5/2010 15:26   2.55        12748
1   537225    84795B         4  12/5/2010 16:41   7.95        12748
2   537225    79191B         5  12/5/2010 16:41   0.85        12748
3   537225     22927         1  12/5/2010 16:41   5.95        12748
4   537225     22926         1  12/5/2010 16:41   5.95        12748

          COUNTRY
0  United Kingdom
1  United Kingdom
2  United Kingdom
3  United Kingdom
4  United Kingdom
```

```
In [19]: # Step 2: Data Cleaning
         # Check for missing values
         print("Missing Values:")
         print(df.isnull().sum())

         # Drop duplicates
         df.drop_duplicates(inplace=True)

         # Step 3: Data Exploration
         # Display basic statistics
         print("Basic Statistics:")
         print(df.describe())

         # Display first few rows
         print("First Few Rows:")
         print(df.head())
```

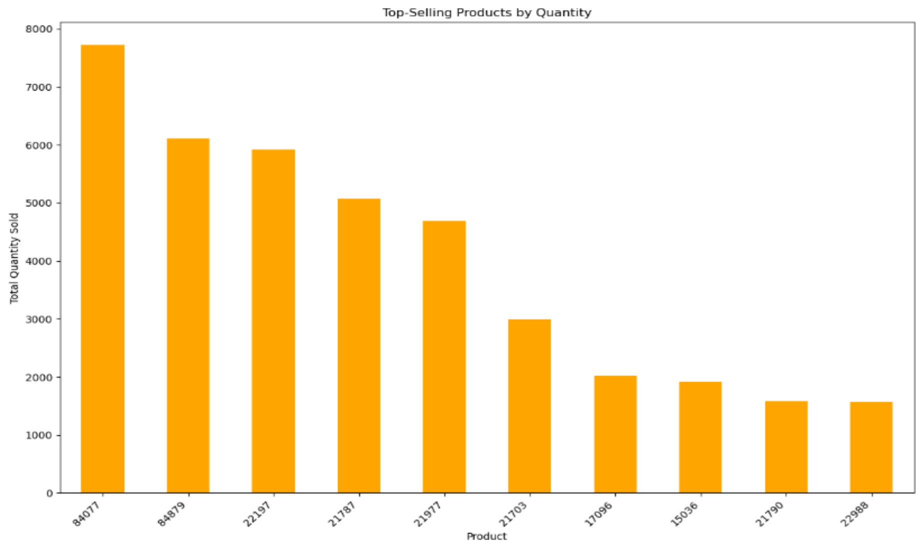
```
Missing Values:
INVOICE        0
STOCKCODE      0
QUANTITY       0
INVOICEDATE    0
PRICE          0
CUSTOMER_ID    0
COUNTRY        0
dtype: int64
Basic Statistics:
             INVOICE      QUANTITY         PRICE   CUSTOMER_ID
count   12598.000000  12598.000000  12598.000000  12598.000000
mean   562134.629306     13.943007      2.759041  12840.400857
std     13997.744908     76.446740      9.215789     80.173874
min    536415.000000      1.000000      0.000000  12747.000000
25%    550320.000000      2.000000      0.850000  12748.000000
50%    564556.000000      4.000000      1.650000  12841.000000
75%    575766.000000     12.000000      2.950000  12921.000000
max    581580.000000   4800.000000    850.500000  12971.000000
First Few Rows:
   INVOICE STOCKCODE  QUANTITY      INVOICEDATE  PRICE  CUSTOMER_ID  \
0   537213     21556         1  12/5/2010 15:26   2.55        12748
1   537225    84795B         4  12/5/2010 16:41   7.95        12748
2   537225    79191B         5  12/5/2010 16:41   0.85        12748
3   537225     22927         1  12/5/2010 16:41   5.95        12748
4   537225     22926         1  12/5/2010 16:41   5.95        12748

          COUNTRY
0  United Kingdom
1  United Kingdom
2  United Kingdom
3  United Kingdom
4  United Kingdom
```

In [25]:
```python
# Identify top-selling products
top_products = df['STOCKCODE'].value_counts().head(10)
print("Top Selling Products:")
print(top_products)

# Geographic Analysis
# Analyze sales by country
country_sales = df.groupby('COUNTRY')['PRICE'].sum().sort_values(ascending=False)
print("Sales by Country:")
print(country_sales)
```

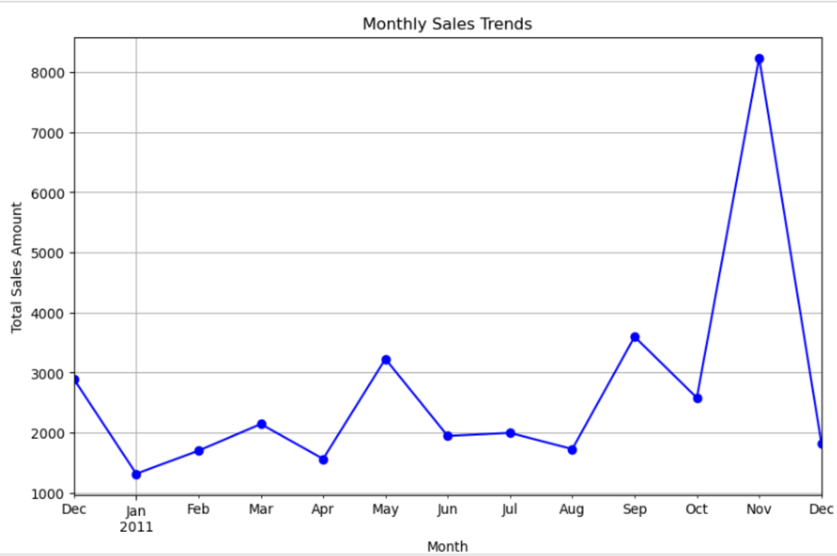
```
Top Selling Products:
84879     60
22086     56
85099B    53
22197     52
85123A    49
22457     49
47566     48
23298     47
20725     46
21034     45
Name: STOCKCODE, dtype: int64
Sales by Country:
COUNTRY
United Kingdom    34758.4
Name: PRICE, dtype: float64
```

# 2)Visualization

In [41]:
```python
# Calculate total quantity sold for each product
quantity_by_product = df.groupby('STOCKCODE')['QUANTITY'].sum().sort_values(ascending=False).head(10)

# Plot top-selling products by quantity
plt.figure(figsize=(12, 8))
quantity_by_product.plot(kind='bar', color='ORANGE')
plt.title('Top-Selling Products by Quantity')
plt.xlabel('Product')
plt.ylabel('Total Quantity Sold')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```




In [20]:
```python
# Sales Trends
# Convert 'INVOICEDATE' to datetime format
df['INVOICEDATE'] = pd.to_datetime(df['INVOICEDATE'])

# Aggregate sales by month
monthly_sales = df.resample('M', on='INVOICEDATE')['PRICE'].sum()

# Plot monthly sales
plt.figure(figsize=(10, 6))
monthly_sales.plot(kind='line', marker='o', color='blue')
plt.title('Monthly Sales Trends')
plt.xlabel('Month')
plt.ylabel('Total Sales Amount')
plt.grid(True)
plt.show()
```



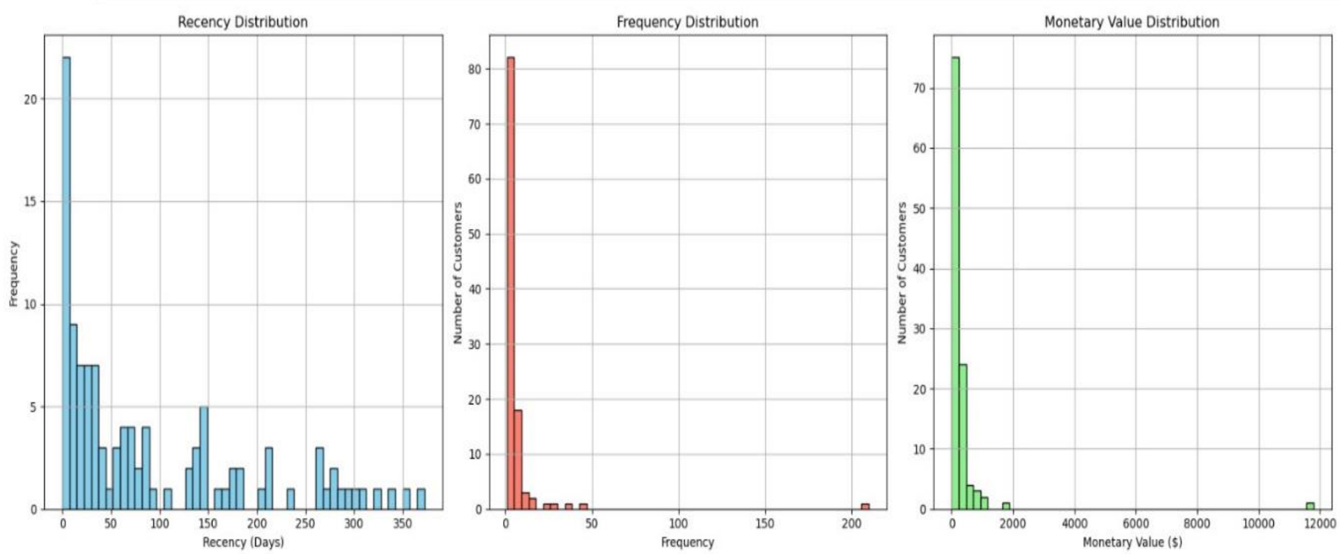
# 3)Segmentation Analysis

```python
import pandas as pd

# Step 1: Calculate RFM Metrics
# Calculate Recency: Number of days since each customer's last purchase
max_invoice_date = df['INVOICEDATE'].max()
recency_df = df.groupby('CUSTOMER_ID')['INVOICEDATE'].max().reset_index()
recency_df['RECENCY'] = (max_invoice_date - recency_df['INVOICEDATE']).dt.days
recency_df.drop(columns=['INVOICEDATE'], inplace=True)

# Calculate Frequency: Count the total number of transactions made by each customer
frequency_df = df.groupby('CUSTOMER_ID')['INVOICE'].nunique().reset_index()
frequency_df.columns = ['CUSTOMER_ID', 'FREQUENCY']

# Calculate Monetary Value: Sum the total price of all transactions for each customer
monetary_df = df.groupby('CUSTOMER_ID')['PRICE'].sum().reset_index()
monetary_df.columns = ['CUSTOMER_ID', 'MONETARY']

# Merge the RFM metrics into a single DataFrame
rfm_df = pd.merge(recency_df, frequency_df, on='CUSTOMER_ID')
rfm_df = pd.merge(rfm_df, monetary_df, on='CUSTOMER_ID')

# Step 2: Segment Customers
# Assign RFM scores based on quartiles
rfm_df['R_SCORE'] = pd.qcut(rfm_df['RECENCY'], q=5, labels=False) + 1
rfm_df['FM_SCORE'] = pd.qcut(rfm_df['FREQUENCY'] + rfm_df['MONETARY'], q=5, labels=False) + 1

# Define segments based on RFM scores
def assign_segment(row):
    r_score = row['R_SCORE']
    fm_score = row['FM_SCORE']
    if (r_score >= 5 and fm_score >= 5) or (r_score >= 5 and fm_score == 4) or (r_score == 4 and fm_score >= 5):
        return 'Champions'
    elif (r_score >= 5 and fm_score == 2) or (r_score == 4 and fm_score == 2) or (r_score == 3 and fm_score == 3) or (r_score
        return 'Potential Loyalists'
    elif (r_score >= 5 and fm_score == 3) or (r_score == 4 and fm_score == 4) or (r_score == 3 and fm_score >= 5) or (r_score
        return 'Loyal Customers'
    elif r_score >= 5 and fm_score == 1:
        return 'Recent Customers'
    elif (r_score == 4 and fm_score == 1) or (r_score == 3 and fm_score == 1):
        return 'Promising'
    elif (r_score == 3 and fm_score == 2) or (r_score == 2 and fm_score == 3) or (r_score == 2 and fm_score == 2):
        return 'Customers Needing Attention'
    elif (r_score == 2 and fm_score >= 5) or (r_score == 2 and fm_score == 4) or (r_score == 1 and fm_score == 3):
        return 'At Risk'
    elif (r_score == 1 and fm_score >= 5) or (r_score == 1 and fm_score == 4):
        return 'Can\'t Lose Them'
    elif (r_score == 1 and fm_score == 2) or (r_score == 2 and fm_score == 1):
        return 'Hibernating'
    elif r_score == 1 and fm_score <= 1:
        return 'Lost'
    else:
        return 'Other'

# Apply segment assignment function
rfm_df['CUST_SEGMENT'] = rfm_df.apply(assign_segment, axis=1)

# Display the RFM metrics and customer segments
print("RFM Metrics and Customer Segments:")
print(rfm_df[['CUSTOMER_ID', 'RECENCY', 'FREQUENCY', 'MONETARY', 'R_SCORE', 'FM_SCORE', 'CUST_SEGMENT']].head())
```

```
RFM Metrics and Customer Segments:
   CUSTOMER_ID  RECENCY  FREQUENCY  MONETARY  R_SCORE  FM_SCORE  \
0        12747        1         11    449.89        1         5
1        12748        0        210  11788.31        1         5
2        12749        3          5    994.99        1         5
3        12820        2          4    112.38        1         3
4        12821      213          1     14.99        5         1

      CUST_SEGMENT
0  Can't Lose Them
1  Can't Lose Them
2  Can't Lose Them
3          At Risk
4 Recent Customers
```

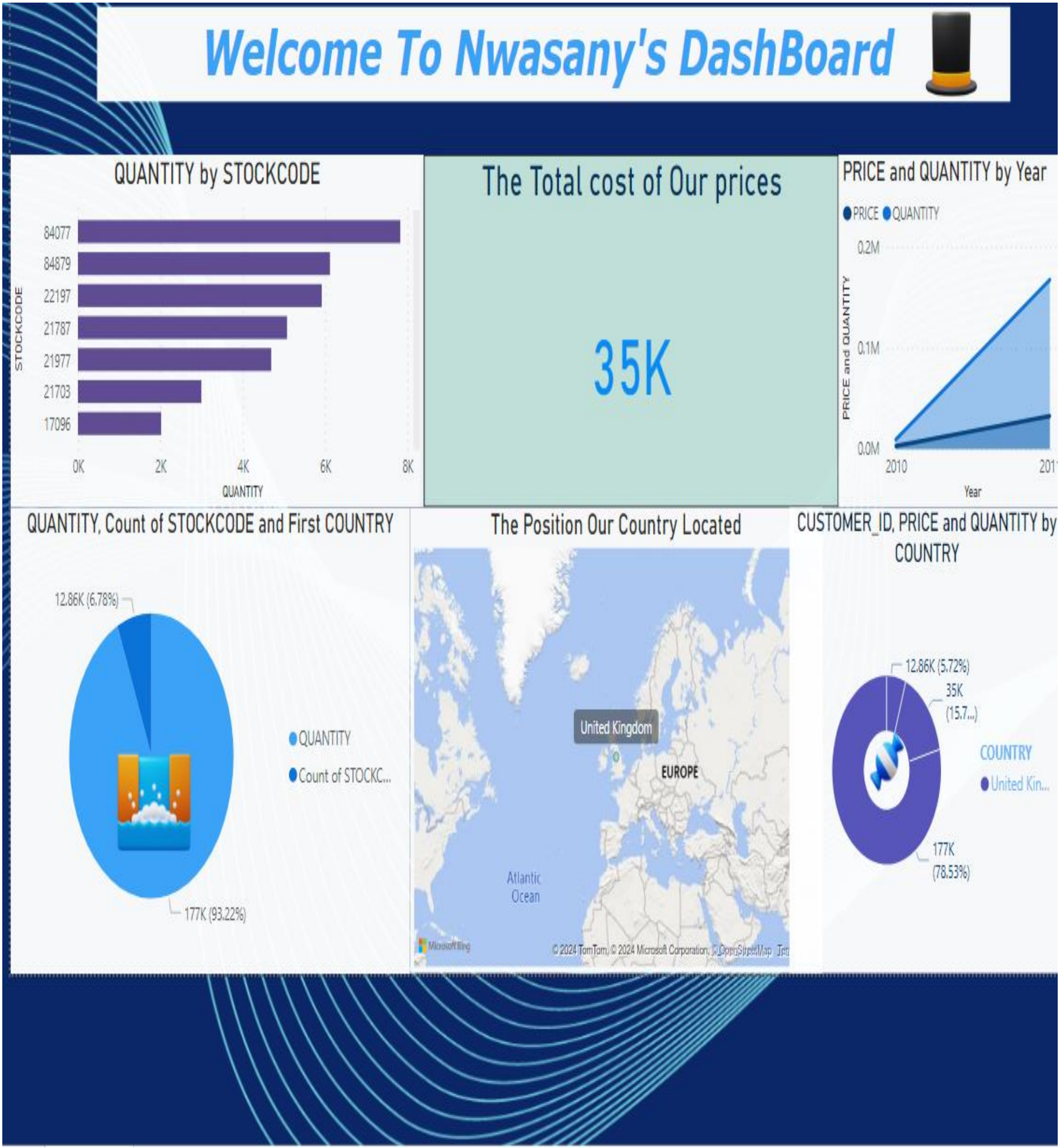
In [24]:

```python
import matplotlib.pyplot as plt

# Create a figure with subplots
fig, axs = plt.subplots(1, 3, figsize=(18, 6))

# Plot Recency distribution
axs[0].hist(rfm_df['RECENCY'], bins=50, color='skyblue', edgecolor='black')
axs[0].set_title('Recency Distribution')
axs[0].set_xlabel('Recency (Days)')
axs[0].set_ylabel('Frequency')
axs[0].grid(True)

# Plot Frequency distribution
axs[1].hist(rfm_df['FREQUENCY'], bins=50, color='salmon', edgecolor='black')
axs[1].set_title('Frequency Distribution')
axs[1].set_xlabel('Frequency')
axs[1].set_ylabel('Number of Customers')
axs[1].grid(True)

# Plot Monetary Value distribution
axs[2].hist(rfm_df['MONETARY'], bins=50, color='lightgreen', edgecolor='black')
axs[2].set_title('Monetary Value Distribution')
axs[2].set_xlabel('Monetary Value ($)')
axs[2].set_ylabel('Number of Customers')
axs[2].grid(True)

# Adjust layout to prevent overlap
plt.tight_layout()

# Show the plot
plt.show()
```



# DashBoard

I have created simple dash board using power bi that support the analysis




Thank you for reading!