NAVIGATING FUTURE OF ONLINE SHOPPING

SQL and Python Approach

PRESENTED BY MOHI GUPTA

DATASET OVERVIEW

01

CUSTOMERS.CSV: CUSTOMERS DEMOGRAPHIC

02

SELLERS.CSV : SELLERS INFORMATION

03

ORDER_ITEMS : ORDER ITEMS DETAILS

04

GEOLOCATION.CSV : GEOLOCATION DETAILS

05

PAYMENTS.CSV : PAYMENTS DETAILS

06

ORDERS.CSV : ORDER ISTORY AND DETAILS

07

PRODUCTS.CSV : PRODUCTS DETAILS

Goal: Analyze and visualize business metrics using SQL and Python

Problem Statement

In the fast-growing e-commerce and retail industry, understanding customer behavior and sales patterns is crucial for optimizing business operations and improving customer retention. This project aims to extract and analyze key business metrics from an e-commerce dataset, which includes customer orders, payments, products, and other transactional details.

- The primary challenge lies in deriving actionable insights such as:
- How do customers behave over time in terms of spending and order frequency?
- What trends can be identified for year-over-year sales growth?
- How can customer retention be measured and improved?
- Which customers contribute the most to the total revenue, and how can they be targeted for personalized marketing?

Objective

THE OBJECTIVE OF THIS PROJECT IS TO:

- Analyze the dataset using SQL for complex queries and Python for data manipulation and visualization.
- Identify strategic insights like customer retention rates, moving averages, and year-over-year growth in total sales.
- Generate customer-centric insights, such as identifying top spenders and calculating retention rates based on repeated purchases within a specific time frame.
- Create visualizations to communicate trends, correlations, and key findings clearly.

Basic Problems

OBJECTIVE: EXTRACT FUNDAMENTAL INSIGHTS FROM THE DATASET

- **List all unique cities where customers are located.**

```sql
SELECT DISTINCT customer_city
FROM customers
WHERE customer_city IS NOT NULL
ORDER BY customer_city;
```

Result Grid | Filter Rows:

customer_city
abadia dos dourados
abadiania
abaete
abaetetuba
abaiara
abaira
abare
abatia
abdon batista
abelardo luz
abrantes
abre campo
abreu e lima

```python
customer_cities = (
    final_df['customer_city']
    .dropna()
    .str.strip()
    .str.lower()
    .unique())
customer_cities = sorted(customer_cities)
for city in customer_cities:
    print(city)
```

```
abaete
abaetetuba
abaiara
abelardo luz
abrantes
abre campo
abreu e lima
acaiaca
acailandia
acopiara
acreuna
acucena
adamantina
adolfo
adustina
afogados da ingazeira
afonso claudio
afranio
```

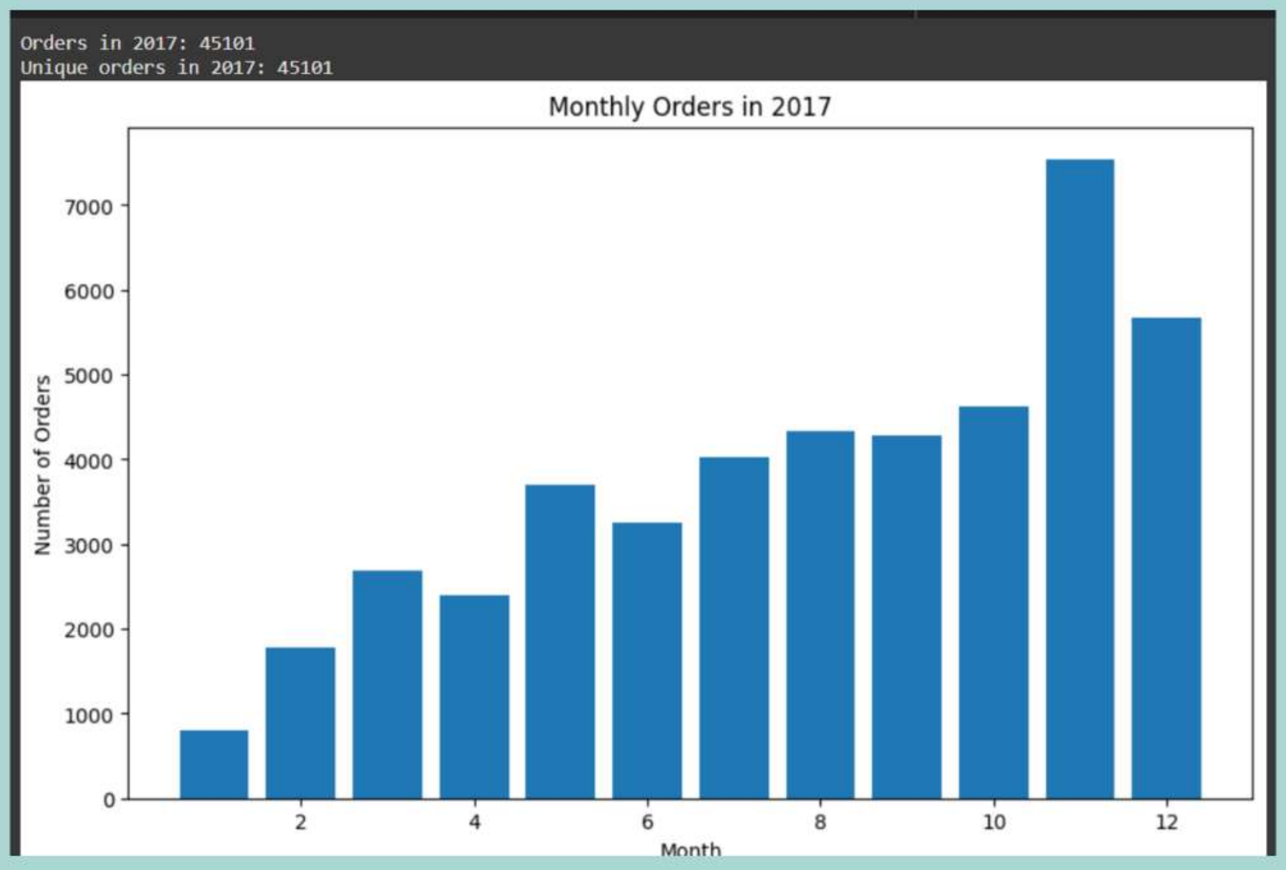
- **COUNT THE NUMBER OF ORDERS PLACED IN 2017.**

```sql
SELECT COUNT(*) AS order_count
FROM orders
WHERE YEAR(order_purchase_timestamp) = 2017
  AND order_purchase_timestamp IS NOT NULL;
```

Result Grid | Filter Rows:

order_count
45101

```python
import builtins
print = builtins.print

df_orders['order_purchase_timestamp'] = pd.to_datetime(
    df_orders['order_purchase_timestamp'], format='%d-%m-%Y %H:%M', errors='coerce'
)
orders_count = len(orders_2017)
unique_orders_count = orders_2017['order_id'].nunique()
print("Orders in 2017:", orders_count)
print("Unique orders in 2017:", unique_orders_count)

# Plot
orders_2017['month'] = orders_2017['order_purchase_timestamp'].dt.month
monthly_counts = orders_2017.groupby('month')['order_id'].nunique()
plt.figure(figsize=(10, 6))
plt.bar(monthly_counts.index, monthly_counts.values)
plt.xlabel('Month')
plt.ylabel('Number of Orders')
plt.title('Monthly Orders in 2017')
plt.show()
```



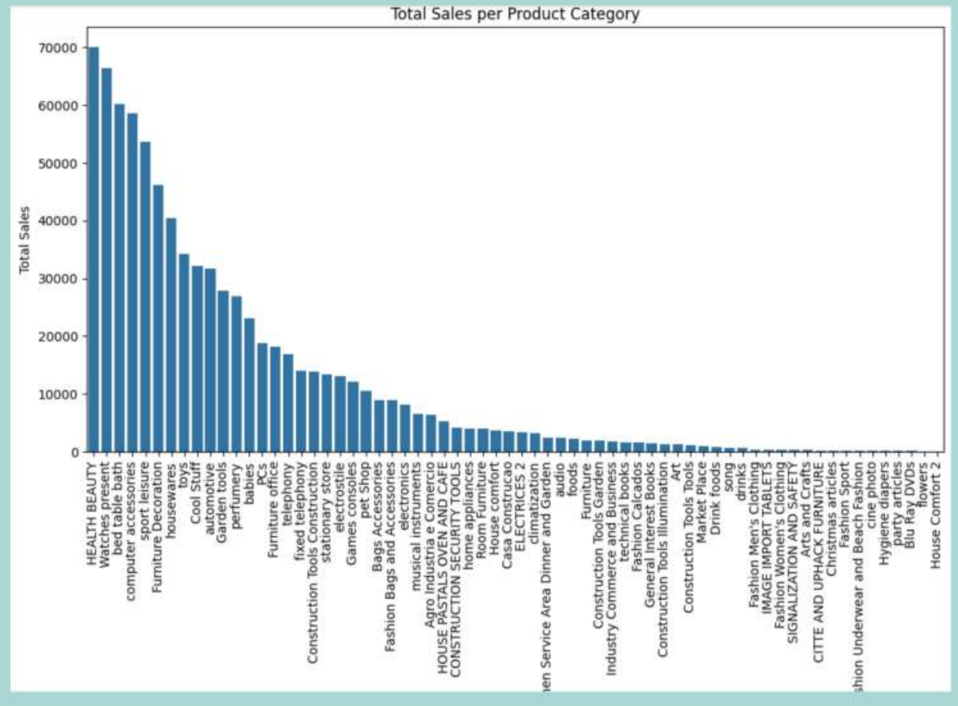
- **FIND THE TOTAL SALES PER CATEGORY.**

```sql
SELECT
    products.`product category`,
    SUM(order_items.price) AS total_sales
FROM
    order_items
JOIN
    products ON order_items.product_id = products.product_id
GROUP BY
    products.`product category`
ORDER BY
    total_sales DESC;
```

Result Grid | Filter Rows:

product category	total_sales
HEALTH BEAUTY	1258681.34
Watches present	1205005.68
bed table bath	1036988.68
sport leisure	988048.97
computer accessories	911954.32
Furniture Decoration	729762.49
Cool Stuff	635290.85
housewares	632248.66
automotive	592720.11

```python
[ ]  merged = df_order_items.merge(df_products[['product_id', 'product category']],
                                   on='product_id', how='left')

     # Check for missing categories after merge
     print("Missing product categories after merge:", merged['product category'].isnull().sum())

     # Group and sum (just like SQL)
     total_sales_per_category = merged.groupby('product category')['price'].sum().sort_values(ascending=False)

     # Print only categories you see in SQL for visual comparison
     print(total_sales_per_category)

     #Plot
     plt.figure(figsize=(12, 6))
     sns.barplot(x=total_sales_per_category.index, y=total_sales_per_category.values)
     plt.xlabel('Product Category')
     plt.ylabel('Total Sales')
     plt.title('Total Sales per Product Category')
     plt.xticks(rotation=90)
     plt.show()
```




```
Missing product categories after merge: 1603
product category
HEALTH BEAUTY                    1258681.34
Watches present                  1205005.68
bed table bath                   1036988.68
sport leisure                     988048.97
computer accessories              911954.32
                                     ...
flowers                             1110.04
House Comfort 2                      760.27
cds music dvds                       730.00
Fashion Children's Clothing          569.85
insurance and services               283.29
Name: price, Length: 73, dtype: float64
```

- **Calculate the percentage of orders that were paid in installments.**

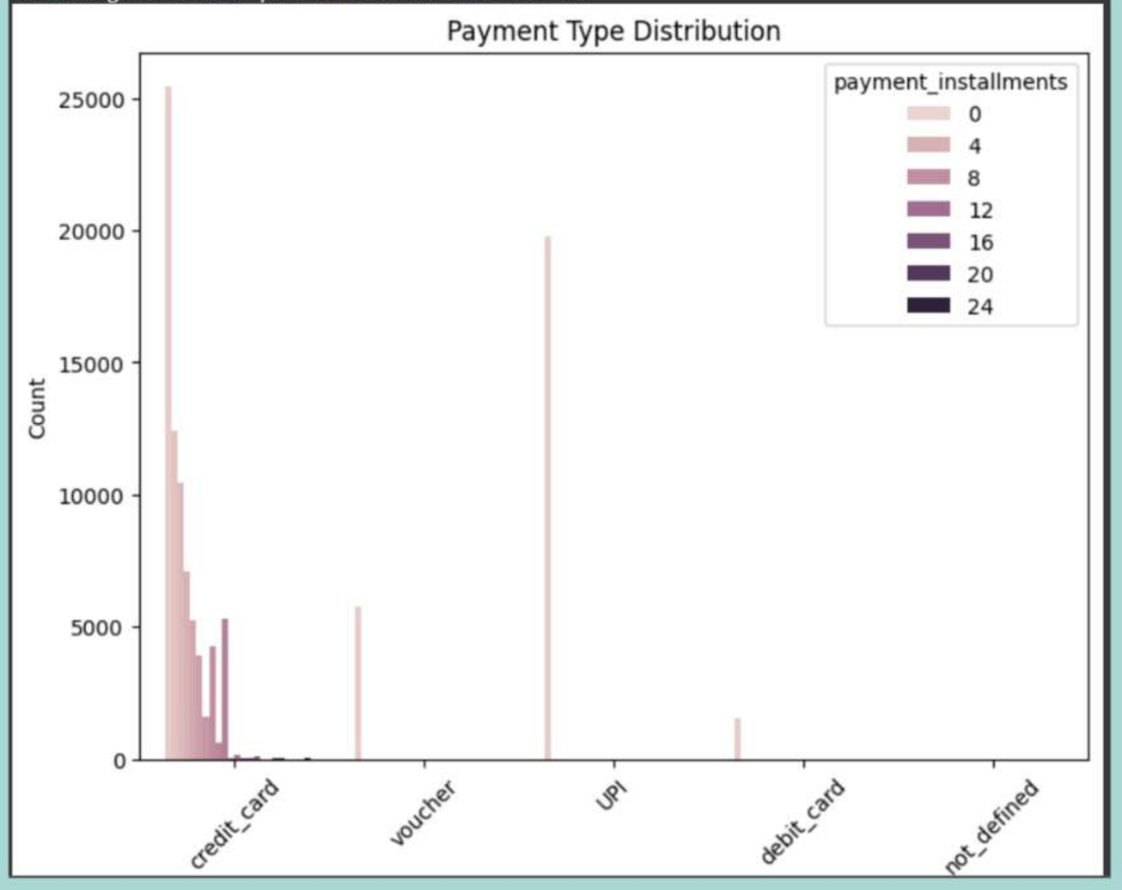
```sql
-- 4. Calculate the percentage of orders that were paid in installments.
select 100 * count(case when payment_installments > 1 then 1 end ) / count(*) as percent_installments
from payments;
```

Result Grid | Filter Rows

	percent_installments
▶	49.4176

```python
# Count the total number of orders
total_orders = orders_payments['order_id'].nunique()
installments_orders = orders_payments[orders_payments['payment_installments'] == 1]

# Count the total number of installments orders
total_installments_orders = installments_orders['order_id'].nunique()

# Calculate the percentage
percentage_installments = (total_installments_orders / total_orders) * 100
print(f"Percentage of orders paid in installments: {percentage_installments:.2f}%")

#PLot
plt.figure(figsize=(8, 6))
sns.countplot(data=orders_payments, x='payment_type', hue='payment_installments')
plt.xlabel('Payment Type')
plt.ylabel('Count')
plt.title('Payment Type Distribution')
plt.xticks(rotation=45)
plt.show()
```

```
Percentage of orders paid in installments: 49.34%
```



- **COUNT THE NUMBER OF CUSTOMERS FROM EACH STATE.**

```sql
-- 5. Count the number of customers from each state.
Select customer_state ,
Count(*) AS total_count
from customers group by customer_state;
```

Result Grid | Filter Rows:

customer_state	total_count
SP	41746
MG	11635
ES	2033
RJ	12852
RS	5466
BA	3380
CE	1336
PR	5045
MS	715

Result 14

```python
total_customers = df_customers['customer_state'].value_counts()
print(total_customers)
#plot
plt.figure(figsize=(12, 6))
sns.barplot(x=total_customers.index, y=total_customers.values)
plt.xlabel('State')
plt.ylabel('Number of Customers')
plt.title('Number of Customers per State')
plt.xticks(rotation=90)
plt.show()
```

```
customer_state
SP    41746
RJ    12852
MG    11635
RS     5466
PR     5045
SC     3637
BA     3380
DF     2140
ES     2033
GO     2020
PE     1652
CE     1336
PA      975
MT      907
MA      747
MS      715
PB      536
```

INTERMEDIATE PROBLEMS

OBJECTIVE: DIVE DEEPER INTO SALES AND ORDER TRENDS

- **Calculate the number of orders per month in 2018**

```
-- INTERMEDIATE PROBLEMS
-- Objective: Dive deeper into sales and order trends.
-- 1.Calculate the number of orders per month in 2018.
SELECT MONTH(order_purchase_timestamp) as month ,
count(*)as order_count
from orders
where year(order_purchase_timestamp)=2018
group by MONTH(order_purchase_timestamp)
order by month;
```

Result Grid | Filter Rows:

month	order_count
1	7269
2	6728
3	7211
4	6939
5	6873
6	6167
7	6292
8	6512
9	16

```
df_orders['order_purchase_timestamp'] = pd.to_datetime(df_orders['order_purchase_timestamp'], errors='coerce')
df_orders = df_orders.dropna(subset=['order_purchase_timestamp'])  # Drop NaT

# Filter for 2018
orders_2018 = df_orders[df_orders['order_purchase_timestamp'].dt.year == 2018]

# Group and count
orders_per_month = orders_2018.groupby(orders_2018['order_purchase_timestamp'].dt.month).size().sort_index()
print(orders_per_month)
```

```
order_purchase_timestamp
1     7269
2     6728
3     7211
4     6939
5     6873
6     6167
7     6292
8     6512
9       16
10       4
dtype: int64
```

- **Find the average number of products per order, grouped by customer city.**

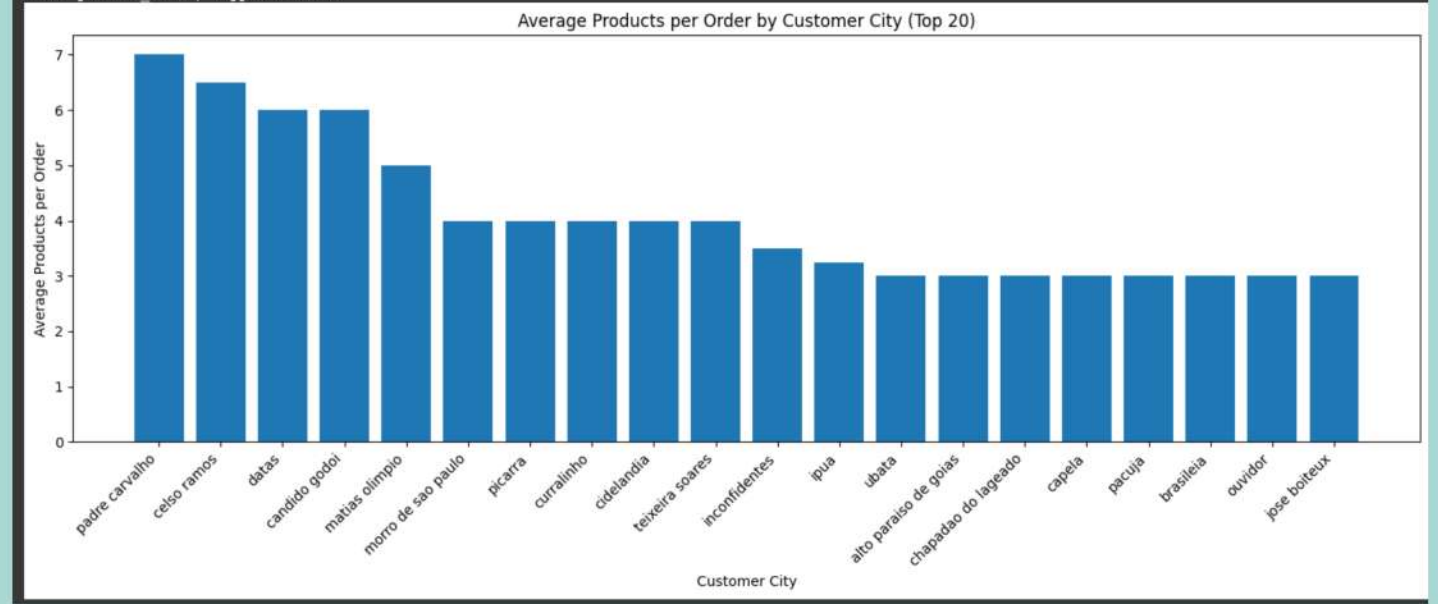
```sql
-- 2.Find the average number of products per order, grouped by customer city.
SELECT
    customers.customer_city,
    AVG(order_count.products_in_order) AS avg_order
FROM
    (
        SELECT
            order_items.order_id,
            COUNT(order_items.product_id) AS products_in_order
        FROM
            order_items
        GROUP BY
            order_items.order_id
    ) AS order_count
JOIN orders ON order_count.order_id = orders.order_id
JOIN customers ON orders.customer_id = customers.customer_id
GROUP BY
    customers.customer_city
ORDER BY
    avg_order;
```

Result Grid | Filter Rows:

customer_city	avg_order
alvares machado	1.0000
santa fe do sul	1.0000
penaforte	1.0000
mampituba	1.0000
pouso novo	1.0000
nova america da colina	1.0000
cerro grande do sul	1.0000
bora	1.0000

```python
products_per_order = df_order_items.groupby('order_id')['product_id'].count().reset_index(name='product_count')

# Now products_per_order is a DataFrame with columns 'order_id' and 'product_count'
products_per_order_per_city = products_per_order.merge(orders_customers[['order_id', 'customer_city']], on='order_id', how='left')

# We now group by 'customer_city' and take the mean of the 'product_count' column
average_products_per_city = products_per_order_per_city.groupby('customer_city')['product_count'].mean()

print("Average number of products per order, grouped by customer city:")
print(average_products_per_city.head())

#plot
top_n = 20
top_cities = average_products_per_city.sort_values(ascending=False).head(top_n)
plt.figure(figsize=(14, 6))
plt.bar(top_cities.index, top_cities.values)
plt.xlabel('Customer City')
plt.ylabel('Average Products per Order')
plt.title('Average Products per Order by Customer City (Top 20)')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```

```
Average number of products per order, grouped by customer city:
customer_city
abadia dos dourados    1.000000
abadiania              1.000000
abaete                 1.000000
abaetetuba             1.272727
abaiara                1.000000
Name: product_count, dtype: float64
```



- **Calculate the percentage of total revenue contributed by each product category**

```sql
SELECT
    products.`product category`,
    SUM(order_items.price) AS category_revenue,
    ROUND(100 * SUM(order_items.price) /
        (SELECT SUM(order_items.price)
         FROM order_items
         JOIN products ON order_items.product_id = products.product_id
        ), 2) AS percent_of_total
FROM order_items
JOIN products ON order_items.product_id = products.product_id
GROUP BY products.`product category`
ORDER BY percent_of_total DESC;
```

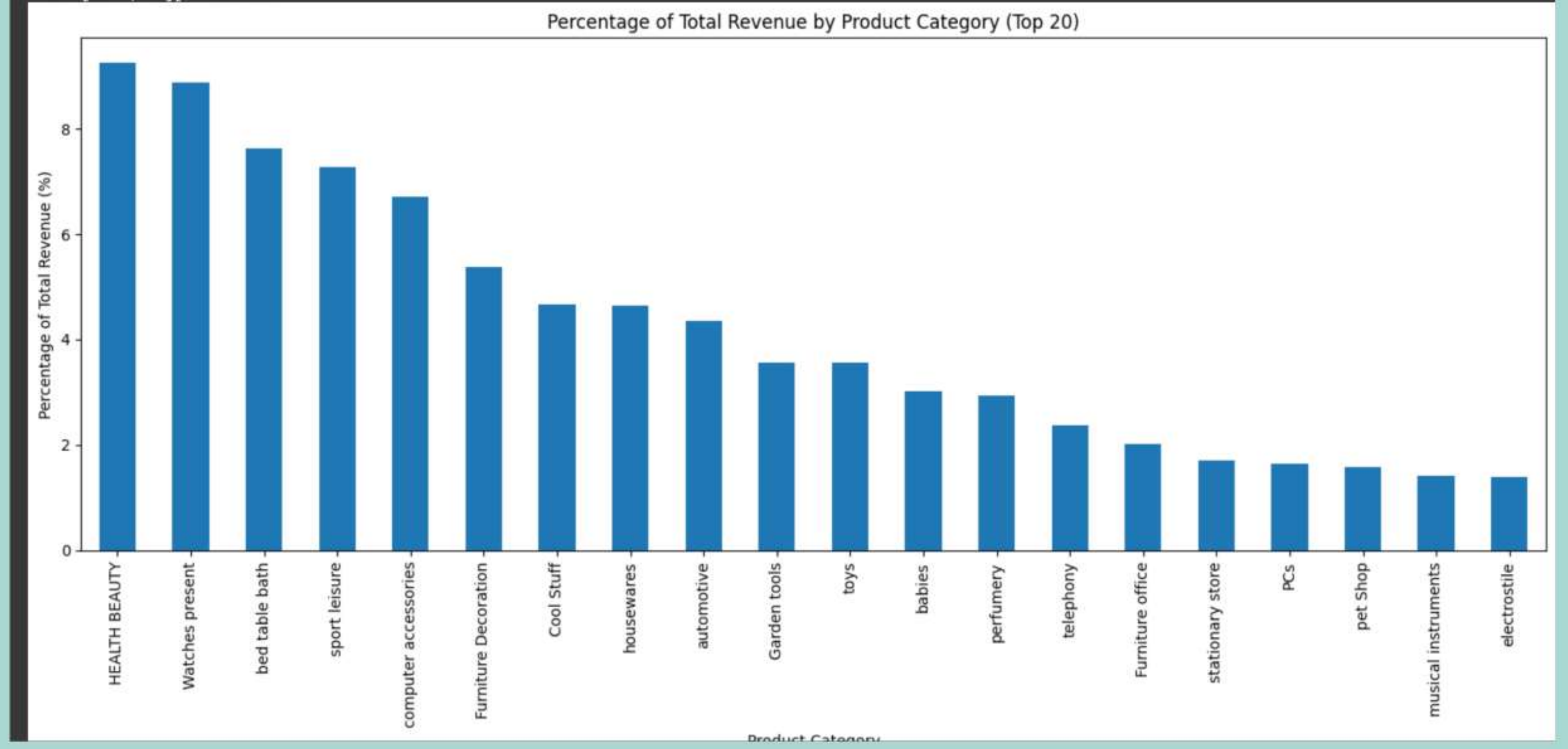
Result Grid | Filter Rows: | Export:

product category	category_revenue	percent_of_tota
HEALTH BEAUTY	1258681.34	9.26
Watches present	1205005.68	8.87
bed table bath	1036988.68	7.63
sport leisure	988048.97	7.27
computer accessories	911954.32	6.71
Furniture Decoration	729762.49	5.37
Cool Stuff	635290.85	4.67
housewares	632248.66	4.65
automotive	592720.11	4.36

Result 19 ×

```python
# Merge order_items with products on product_id
order_items_products = df_order_items.merge(df_products[['product_id', 'product category']],on='product_id',how='inner')

# Clean product category
order_items_products['product category'] = (order_items_products['product category'].astype(str).str.strip())
order_items_products = order_items_products[order_items_products['product category'] != '']

# Group by product category and sum the price
revenue_by_category = order_items_products.groupby('product category')['price'].sum()

# Calculate total revenue from joined data (same as SQL logic)
total_revenue = order_items_products['price'].sum()

# Calculate percentage
percentage_revenue_by_category = ((revenue_by_category / total_revenue) * 100).round(2)
percentage_revenue_by_category = percentage_revenue_by_category.sort_values(ascending=False)

# Show the top 10
print("Category Percentages (to match SQL):")
print(percentage_revenue_by_category.head(10))

# Plotting
plt.figure(figsize=(14, 7))
percentage_revenue_by_category.head(20).plot(kind='bar')
plt.title('Percentage of Total Revenue by Product Category (Top 20)')
plt.xlabel('Product Category')
plt.ylabel('Percentage of Total Revenue (%)')
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()
```

```
Category Percentages (to match SQL):
product category
HEALTH BEAUTY           9.26
Watches present         8.87
bed table bath          7.63
sport leisure           7.27
computer accessories    6.71
Furniture Decoration    5.37
Cool Stuff              4.67
housewares              4.65
automotive              4.36
Garden tools            3.57
Name: price, dtype: float64
```



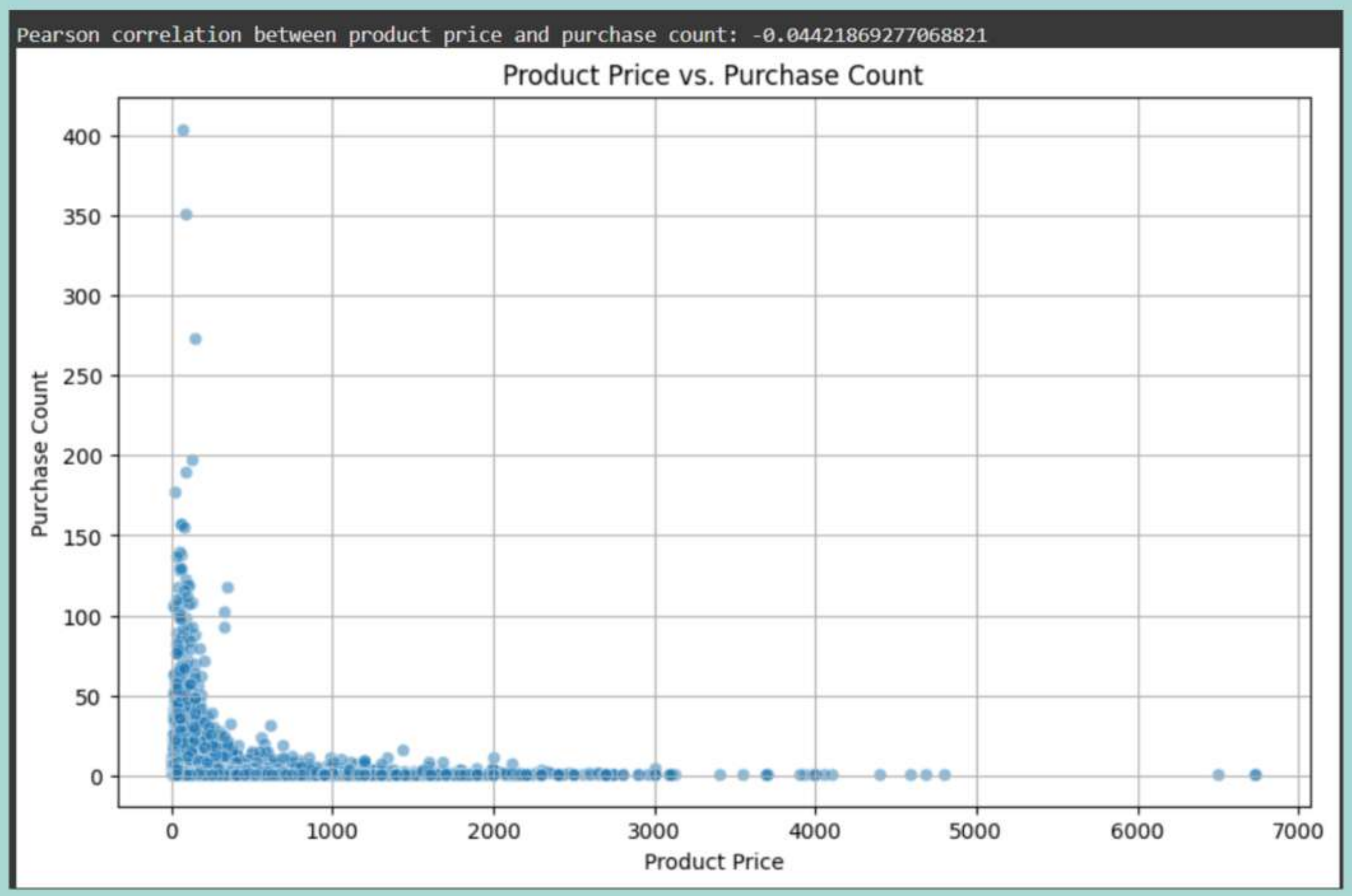
- **Identify the correlation between product price and the number of times a product has been purchased**

```sql
WITH product_sales AS (
    SELECT
        oi.product_id,
        oi.price,
        COUNT(oi.order_id) AS purchase_count
    FROM
        order_items oi
    GROUP BY
        oi.product_id, oi.price
)
SELECT
  (
    COUNT(*) * SUM(price * purchase_count) - SUM(price) * SUM(purchase_count)
  ) /
  (
    SQRT(COUNT(*) * SUM(POWER(price, 2)) - POWER(SUM(price), 2)) *
    SQRT(COUNT(*) * SUM(POWER(purchase_count, 2)) - POWER(SUM(purchase_count), 2))
  ) AS correlation
FROM
    product_sales;
```

Result Grid | Filter Rows:

correlation
-0.0442186927706809

```python
# Group by both product_id and price, count purchases
product_sales = (
    df_order_items
    .groupby(['product_id', 'price'])
    .agg(purchase_count=('order_id', 'count'))
    .reset_index()
)

# Calculate Pearson correlation (same formula as SQL)
correlation = product_sales['price'].corr(product_sales['purchase_count'])
print(f"Pearson correlation between product price and purchase count: {correlation}")

# Plot
plt.figure(figsize=(10, 6))
sns.scatterplot(x='price', y='purchase_count', data=product_sales, alpha=0.5)
plt.xlabel('Product Price')
plt.ylabel('Purchase Count')
plt.title('Product Price vs. Purchase Count')
plt.grid(True)
plt.show()
```



- **Calculate the total revenue generated by each seller, and rank them by revenue**

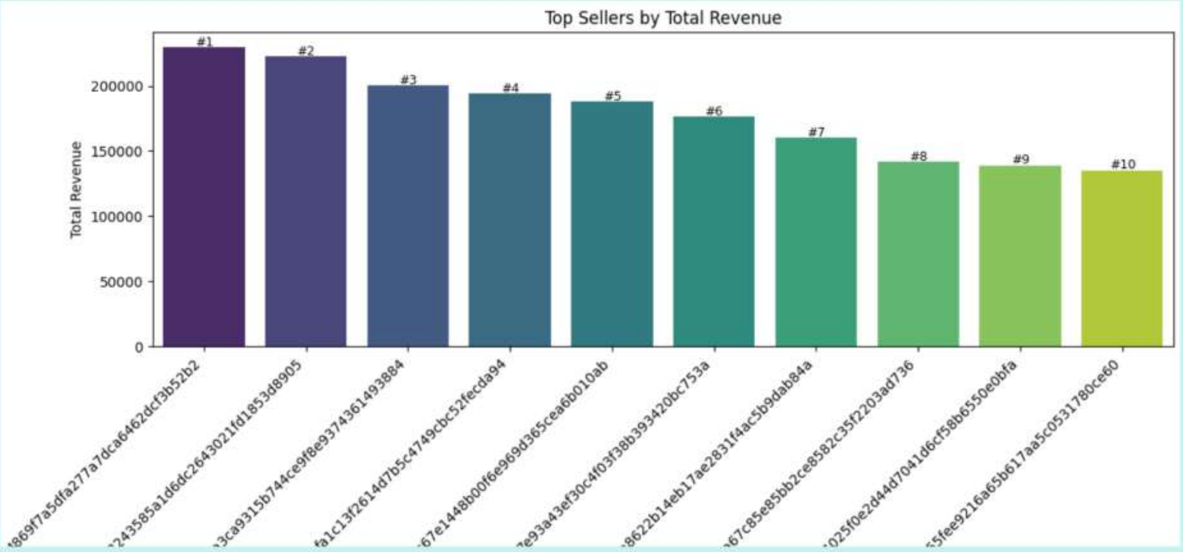
```sql
SELECT
    seller_id,
    total_revenue,
    RANK() OVER (ORDER BY total_revenue DESC) AS revenue_rank
from(SELECT
    order_items.seller_id,
    SUM(order_items.price) AS total_revenue
FROM
    order_items
group by order_items.seller_id)
as seller_revenue
order by revenue_rank;
```

Result Grid | Filter Rows: | Export: | Wrap Cell

seller_id	total_revenue	revenue_rank
4869f7a5dfa277a7dca6462dcf3b52b2	229472.63	1
53243585a1d6dc2643021fd1853d8905	222776.05	2
4a3ca9315b744ce9f8e9374361493884	200472.92	3
fa1c13f2614d7b5c4749cbc52fecda94	194042.03	4
7c67e1448b00f6e969d365cea6b010ab	187923.89	5

```python
# Calculate total revenue per seller
seller_revenue = (df_order_items
    .groupby('seller_id', as_index=False)['price']
    .sum()
    .rename(columns={'price': 'total_revenue'}))

# Apply RANK
seller_revenue['revenue_rank'] = seller_revenue['total_revenue'].rank(
    method='dense', ascending=False).astype(int)

# Order by revenue_rank
seller_revenue = seller_revenue.sort_values('revenue_rank')
print(seller_revenue)

# Limit to top 10 or 20 for visualization (otherwise chart is too big)
top_sellers = seller_revenue.head(10)
plt.figure(figsize=(12, 6))
sns.barplot(x='seller_id', y='total_revenue', data=top_sellers, palette='viridis')
plt.title('Top Sellers by Total Revenue')
plt.xlabel('Seller ID')
plt.ylabel('Total Revenue')
plt.xticks(rotation=45, ha='right')
for i, row in top_sellers.reset_index(drop=True).iterrows():
    plt.text(i, row.total_revenue + 1000, f"#{row.revenue_rank}", ha='center', fontsize=9)
plt.tight_layout()
plt.show()
```

	seller_id	total_revenue	revenue_rank
857	4869f7a5dfa277a7dca6462dcf3b52b2	229472.63	1
1013	53243585a1d6dc2643021fd1853d8905	222776.05	2
881	4a3ca9315b744ce9f8e9374361493884	200472.92	3
3024	fa1c13f2614d7b5c4749cbc52fecda94	194042.03	4
1535	7c67e1448b00f6e969d365cea6b010ab	187923.89	5



ADVANCED PROBLEMS

OBJECTIVE: GENERATE STRATEGIC AND CUSTOMER-CENTRIC INSIGHTS.

- **Calculate the moving average of order values for each customer over their order history**

```sql
SELECT
    customer_id,
    order_purchase_timestamp,
    order_value,
    AVG(order_value) OVER (
        PARTITION BY customer_id
        ORDER BY order_purchase_timestamp
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS moving_avg_3
FROM (
    SELECT
        o.customer_id,
        o.order_purchase_timestamp,
        SUM(oi.price) AS order_value
    FROM
        orders o
    JOIN
        order_items oi ON o.order_id = oi.order_id
    GROUP BY
        o.customer_id, o.order_purchase_timestamp, o.order_id
) AS order_history
ORDER BY
    customer_id, order_purchase_timestamp;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

customer_id	order_purchase_timestamp	order_value	moving_avg_3
00012a2ce6f8dcda20d059ce98491703	2017-11-14 16:08:00	89.80	89.800000
000161a058600d5901f007fab4c27140	2017-07-16 09:40:00	54.90	54.900000
0001fd6190edaaf884bcaf3d49edf079	2017-02-28 11:06:00	179.99	179.990000
0002414f95344307404f0ace7a26f1d5	2017-08-16 13:09:00	149.90	149.900000
000379cdec625522490c315e70c7a9fb	2018-04-02 13:42:00	93.00	93.000000
0004164d20a9e969af783496f3408652	2017-04-12 08:35:00	59.99	59.990000
000419c5494106c306a97b5635748086	2018-03-02 17:47:00	34.30	34.300000
00046a560d407e99b969756e0b10f282	2017-12-18 11:08:00	120.90	120.900000

```python
#Calculate order_value for each order
order_value_per_order = df_order_items.groupby('order_id')['price'].sum().reset_index(name='order_value')
order_history = pd.merge(
    df_orders[['order_id', 'customer_id', 'order_purchase_timestamp']],
    order_value_per_order, on='order_id', how='inner')
order_history['order_purchase_timestamp'] = pd.to_datetime(order_history['order_purchase_timestamp'])
order_history = order_history.sort_values(['customer_id', 'order_purchase_timestamp'])

# Calculate the moving average
order_history['moving_avg_3'] = (order_history.groupby('customer_id')['order_value'].rolling(window=3, min_periods=1).mean().reset_index(level=0, drop=True))

# Display the first 10 rows of the table
top_10_orders = order_history[['customer_id', 'order_purchase_timestamp', 'order_value', 'moving_avg_3']].head(10)
print(top_10_orders.to_string(index=False))

#Plot
plt.figure(figsize=(12, 6))
sns.lineplot(data=order_history, x='order_purchase_timestamp', y='moving_avg_3', color='red', linestyle='--', label="Moving Average")
plt.xlabel('Order Purchase Timestamp')
plt.ylabel('Order Value')
plt.title('Order Values and Moving Average by Customer')
plt.xticks(rotation=45)
plt.legend()
plt.tight_layout()
plt.show()
```

	customer_id	order_purchase_timestamp	order_value	moving_avg_3
68055	00012a2ce6f8dcda20d059ce98491703	2017-11-14 16:08:00	89.80	89.80
9936	000161a058600d5901f007fab4c27140	2017-07-16 09:40:00	54.90	54.90
65380	0001fd6190edaaf884bcaf3d49edf079	2017-02-28 11:06:00	179.99	179.99
42834	0002414f95344307404f0ace7a26f1d5	2017-08-16 13:09:00	149.90	149.90
5843	000379cdec625522490c315e70c7a9fb	2018-04-02 13:42:00	93.00	93.00
73081	0004164d20a9e969af783496f3408652	2017-04-12 08:35:00	59.99	59.99
45797	000419c5494106c306a97b5635748086	2018-03-02 17:47:00	34.30	34.30
59517	00046a560d407e99b969756e0b10f282	2017-12-18 11:08:00	120.90	120.90

- **CALCULATE THE CUMULATIVE SALES PER MONTH FOR EACH YEAR**

```sql
select
year(order_purchase_timestamp),
month(order_purchase_timestamp),
sum(order_value) as monthly_sales,
sum(sum(order_value))over(order by year(order_purchase_timestamp), month(order_purchase_timestamp)) as cummalat
from (select
o.order_id,
o.order_purchase_timestamp,
sum(oi.price) as order_value
from orders o
join order_items oi on o.order_id = oi.order_id
group by o.order_id,o.order_purchase_timestamp) as order_values
group by
year(order_purchase_timestamp),
month(order_purchase_timestamp)
ORDER BY
    YEAR(order_purchase_timestamp), MONTH(order_purchase_timestamp);
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

year(order_purchase_timestamp)	month(order_purchase_timestamp)	monthly_sales	cummalative_sales
2016	9	267.36	267.36
2016	10	49507.66	49775.02
2016	12	10.90	49785.92
2017	1	120312.87	170098.79
2017	2	247303.02	417401.81
2017	3	374344.30	791746.11
2017	4	359927.23	1151673.34
2017	5	506071.14	1657744.48
2017	6	433038.60	2090783.08

```python
#Merge order_items with orders to get order_purchase_timestamp with every item
merged = pd.merge(df_order_items, df_orders[['order_id', 'order_purchase_timestamp']], on='order_id', how='inner')
merged['order_purchase_timestamp'] = pd.to_datetime(merged['order_purchase_timestamp'])

# Calculate order_value for each order (SUM(oi.price) per order)
order_value_df = merged.groupby(['order_id', 'order_purchase_timestamp'])['price'].sum().reset_index(name='order_value')
order_value_df['year'] = order_value_df['order_purchase_timestamp'].dt.year
order_value_df['month'] = order_value_df['order_purchase_timestamp'].dt.month

# Group by year and month, and sum order_value to get monthly_sales
monthly_sales = (
    order_value_df
    .groupby(['year', 'month'])['order_value']
    .sum()
    .reset_index(name='monthly_sales')
    .sort_values(['year', 'month']))

#Calculate cumulative_sales like the SQL window function
monthly_sales['cumulative_sales'] = monthly_sales['monthly_sales'].cumsum()
print(monthly_sales)

#Plot
plt.figure(figsize=(14, 7))
sns.lineplot(data=monthly_sales, x='month', y='cumulative_sales', hue='year', palette='viridis')
plt.xlabel('Month')
plt.ylabel('Cumulative Sales')
plt.title('Cumulative Sales Over Time (by Month and Year)')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

	year	month	monthly_sales	cumulative_sales
0	2016	9	267.36	267.36
1	2016	10	49507.66	49775.02
2	2016	12	10.90	49785.92
3	2017	1	120312.87	170098.79
4	2017	2	247303.02	417401.81
5	2017	3	374344.30	791746.11
6	2017	4	359927.23	1151673.34
7	2017	5	506071.14	1657744.48
8	2017	6	433038.60	2090783.08
9	2017	7	498031.48	2588814.56

- **CALCULATE THE YEAR-OVER-YEAR GROWTH RATE OF TOTAL SALES**

```sql
CREATE TEMPORARY TABLE order_sales AS
SELECT
    o.order_id,
    SUM(oi.price) AS order_total,
    o.order_purchase_timestamp
FROM
    orders o
    INNER JOIN order_items oi ON o.order_id = oi.order_id
WHERE
    o.order_purchase_timestamp IS NOT NULL
GROUP BY
    o.order_id, o.order_purchase_timestamp;
SELECT
    year,
    total_sales,
    LAG(total_sales) OVER (ORDER BY year) AS previous_year_sales,
    ROUND(
        CASE
            WHEN LAG(total_sales) OVER (ORDER BY year) IS NULL THEN 0
            ELSE ((total_sales - LAG(total_sales) OVER (ORDER BY year)) / LAG(total_sales) OVER (ORDER BY year))
        END
    ,2) AS YoY_Growth_Rate_Percent
FROM (
    SELECT
        YEAR(order_purchase_timestamp) AS year,
        SUM(order_total) AS total_sales
    FROM
        order_sales
    GROUP BY
        YEAR(order_purchase_timestamp)
) yearly
ORDER BY year;
```

Result Grid | Filter Rows: | Export: | Wrap C

year	total_sales	previous_year_sales	YoY_Growth_Rate_Percent
2016	49785.92	NULL	0.00
2017	6155806.98	49785.92	12264.55
2018	7386050.80	6155806.98	19.99

```python
df_orders['order_purchase_timestamp'] = pd.to_datetime(df_orders['order_purchase_timestamp'], errors='coerce')
df_orders = df_orders.dropna(subset=['order_purchase_timestamp'])

# Merge orders with order_items to get price information per order
orders_with_prices = pd.merge(df_orders, df_order_items[['order_id', 'price']], on='order_id', how='inner')

# Calculate total sales per order
order_sales = orders_with_prices.groupby('order_id')['price'].sum().reset_index(name='order_total')
order_sales = pd.merge(order_sales, df_orders[['order_id', 'order_purchase_timestamp']], on='order_id', how='left')
order_sales['year'] = order_sales['order_purchase_timestamp'].dt.year

# Calculate total sales per year
yearly_sales = order_sales.groupby('year')['order_total'].sum().reset_index(name='total_sales')
yearly_sales = yearly_sales.sort_values('year')
yearly_sales['previous_year_sales'] = yearly_sales['total_sales'].shift(1)

# Calculate growth rate: (Current Year Sales - Previous Year Sales) / Previous Year Sales * 100
yearly_sales['YoY_Growth_Rate (%)'] = ((yearly_sales['total_sales'] - yearly_sales['previous_year_sales']) / yearly_sales['previous_year_sales']) * 100
yearly_sales['YoY_Growth_Rate (%)'] = yearly_sales['YoY_Growth_Rate (%)'].fillna(0)

print("Year-over-Year Growth Rate of Total Sales:")
print(yearly_sales)

# Plotting the YoY Growth Rate
plt.figure(figsize=(10, 6))
sns.lineplot(x='year', y='YoY_Growth_Rate (%)', data=yearly_sales, marker='o')
plt.xlabel('Year')
plt.ylabel('YoY Growth Rate (%)')
plt.title('Year-over-Year Growth Rate of Total Sales')
plt.grid(True)
plt.show()
```

```
Year-over-Year Growth Rate of Total Sales:
   year  total_sales  previous_year_sales  YoY_Growth_Rate (%)
0  2016     41987.60                  NaN             0.000000
1  2017   5182653.34             41987.60         12243.295020
2  2018   6264888.10           5182653.34            20.881867
```

- **Calculate the retention rate of customers, defined as the percentage of customers who make another purchase within 6 months of their first purchase**

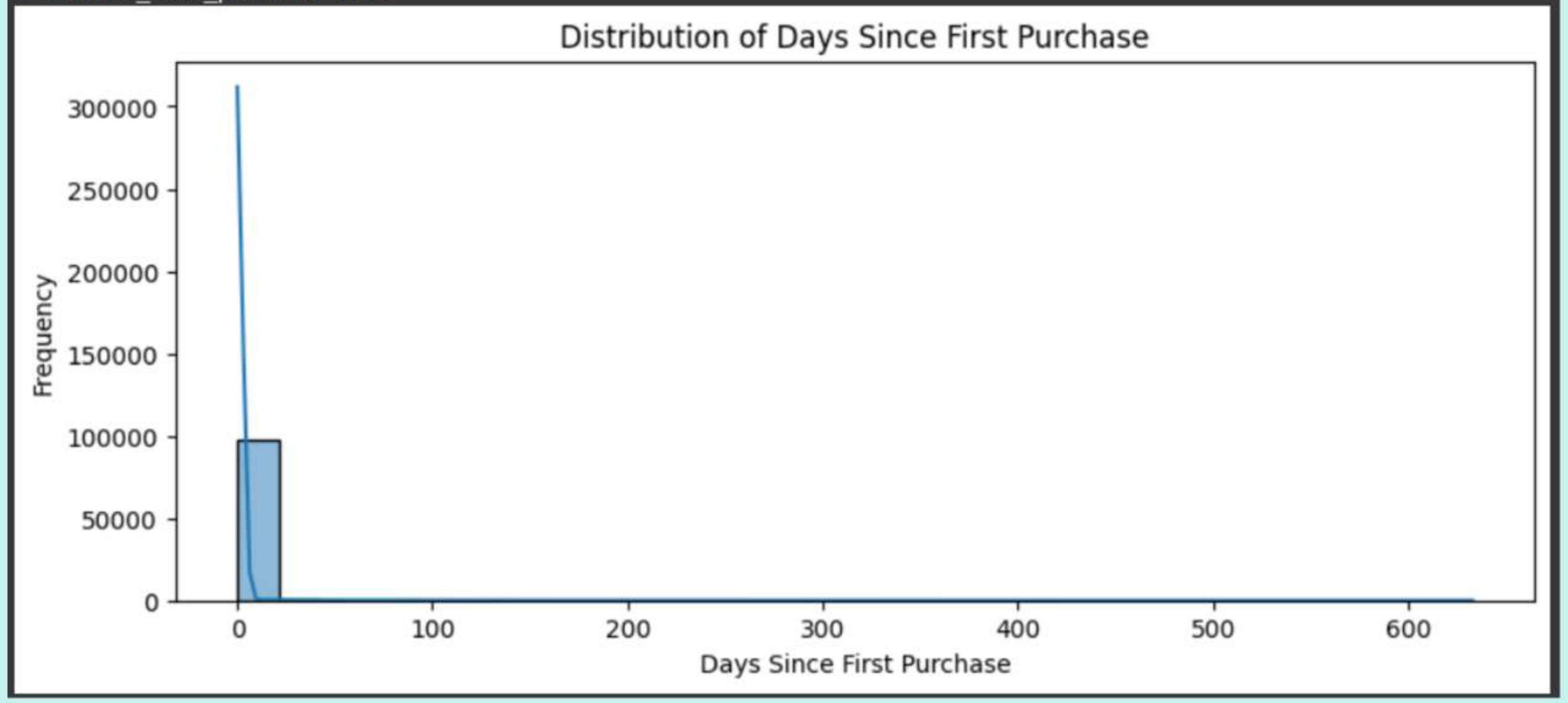
```sql
SELECT order_purchase_timestamp FROM orders LIMIT 5;
WITH first_purchase AS (
    SELECT
        c.customer_unique_id,
        MIN(o.order_purchase_timestamp) AS first_order_date
    FROM
        orders o
    JOIN
        customers c ON o.customer_id = c.customer_id
    GROUP BY
        c.customer_unique_id
),
retained_customers AS (
    SELECT DISTINCT c.customer_unique_id
    FROM orders o
    JOIN customers c ON o.customer_id = c.customer_id
    JOIN first_purchase fp ON c.customer_unique_id = fp.customer_unique_id
    WHERE
        o.order_purchase_timestamp > fp.first_order_date
        AND o.order_purchase_timestamp <= DATE_ADD(fp.first_order_date, INTERVAL 6 MONTH)
)
SELECT
    (SELECT COUNT(*) FROM retained_customers) AS retained_customers,
    (SELECT COUNT(*) FROM first_purchase) AS total_customers,
    ROUND(100.0 * (SELECT COUNT(*) FROM retained_customers) / (SELECT COUNT(*) FROM first_purchase), 2)
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

retained_customers	total_customers	retention_rate_percent
1804	96096	1.88

```python
orders = pd.merge(df_orders, df_customers[['customer_id', 'customer_unique_id']], on='customer_id')
orders['order_purchase_timestamp'] = pd.to_datetime(orders['order_purchase_timestamp'], format='%d-%m-%Y %H:%M', errors='coerce')
orders.dropna(subset=['order_purchase_timestamp'], inplace=True)

# Find first purchase
first_order_timestamp = (orders.groupby('customer_unique_id')['order_purchase_timestamp'].min().reset_index()
.rename(columns={'order_purchase_timestamp': 'first_order_timestamp'}))
orders = pd.merge(orders, first_order_timestamp, on='customer_unique_id', how='left')

# Calculate the date exactly 6 months after the first purchase timestamp
orders['six_months_later'] = orders['first_order_timestamp'] + pd.DateOffset(months=6)
retained_orders_within_window = orders[(orders['order_purchase_timestamp']> orders['first_order_timestamp']) &
    (orders['order_purchase_timestamp'] <= orders['six_months_later'])]

# Count the number of unique customers
retained_customers = retained_orders_within_window['customer_unique_id'].nunique()
total_customers = orders['customer_unique_id'].nunique()
retention_rate = round(100 * retained_customers / total_customers, 2)
print('retained_customers:', retained_customers)
print('total_customers:', total_customers)
print('retention_rate_percent:', retention_rate)
orders['days_since_first'] = (orders['order_purchase_timestamp'] - orders['first_order_timestamp']).dt.days

# Plot
plt.figure(figsize=(10, 4))
sns.histplot(orders['days_since_first'], bins=30, kde=True)
plt.xlabel('Days Since First Purchase')
plt.ylabel('Frequency')
plt.title('Distribution of Days Since First Purchase')
plt.show()
```

```
retained_customers: 1804
total_customers: 96096
retention_rate_percent: 1.88
```



- **Identify the top 3 customers who spent the most money in each year**

```sql
SELECT *
FROM (
    SELECT
        YEAR(o.order_purchase_timestamp) AS year,
        c.customer_unique_id,
        SUM(p.payment_value) AS total_spent,
        ROW_NUMBER() OVER (PARTITION BY YEAR(o.order_purchase_timestamp) ORDER BY SUM(p.payment_value) DESC)
    FROM orders o
    JOIN customers c ON o.customer_id = c.customer_id
    JOIN payments p ON o.order_id = p.order_id
    GROUP BY year, c.customer_unique_id
) ranked
WHERE rn <= 3
ORDER BY year, total_spent DESC;
```

Result Grid | Filter Rows: | Export: | Wrap Cell

year	customer_unique_id	total_spent	rn
2016	fdaa290acb9eeacb66fa7f979baa6803	1423.55	1
2016	753bc5d6efa9e49a03e34cf521a9e124	1400.74	2
2016	b92a2e5e8a6eabcc80882c7d68b2c70b	1227.78	3
2017	0a0a92112bd4c708ca5fde585afaa872	13664.08	1
2017	da122df9eeddfedc1dc1f5349a1a690c	7571.63	2
2017	dc4802a71eae9be1dd28f5d788ceb526	6929.31	3
2018	46450c74a0d8c5ca9395da1daac6c120	9553.02	1
2018	763c8b1c9c68a0229c42c9fc6f662b93	7274.88	2
2018	459bef486812aa25204be022145caa62	6922.21	3

```python
merged = pd.merge(df_orders, df_customers[['customer_id', 'customer_unique_id']], on='customer_id', how='inner')
merged = pd.merge(merged, df_payments[['order_id', 'payment_value']], on='order_id', how='inner')

# Convert order_purchase_timestamp to datetime with the correct format
merged['order_purchase_timestamp'] = pd.to_datetime(merged['order_purchase_timestamp'],
                                                    format='%d-%m-%Y %H:%M', errors='coerce')
merged.dropna(subset=['order_purchase_timestamp'], inplace=True)
merged['year'] = merged['order_purchase_timestamp'].dt.year

# Group by year and customer, sum payment_value
customer_yearly_spend = (
    merged.groupby(['year', 'customer_unique_id'])['payment_value']
    .sum()
    .reset_index(name='total_spent'))
customer_yearly_spend = customer_yearly_spend.sort_values(['year', 'total_spent'], ascending=[True, False])
top3_each_year = customer_yearly_spend.groupby('year').head(3).reset_index(drop=True)
print(top3_each_year)
```

	year	customer_unique_id	total_spent
0	2016	fdaa290acb9eeacb66fa7f979baa6803	1423.55
1	2016	753bc5d6efa9e49a03e34cf521a9e124	1400.74
2	2016	b92a2e5e8a6eabcc80882c7d68b2c70b	1227.78
3	2017	0a0a92112bd4c708ca5fde585afaa872	13664.08
4	2017	da122df9eeddfedc1dc1f5349a1a690c	7571.63
5	2017	dc4802a71eae9be1dd28f5d788ceb526	6929.31
6	2018	46450c74a0d8c5ca9395da1daac6c120	9553.02
7	2018	763c8b1c9c68a0229c42c9fc6f662b93	7274.88
8	2018	459bef486812aa25204be022145caa62	6922.21

Key Insights

CUSTOMER RETENTION IS LOW:
Only about 1.8% of customers made a repeat purchase within 6 months of their first order, indicating a potential area for growth in customer engagement and loyalty programs.

SALES SHOW YEAR-OVER-YEAR GROWTH:
The dataset reveals a consistent increase in total sales year over year, demonstrating healthy business expansion and a growing customer base.

TOP CUSTOMERS DRIVE REVENUE:
The top 3 customers in each year accounted for a disproportionately large share of annual revenue, emphasizing the value of nurturing high-spending customers.

MONTHLY SALES FLUCTUATIONS:
Certain months saw spikes in sales, possibly due to seasonality or marketing campaigns. Identifying these peaks can help with inventory and marketing planning.

ORDER VALUE TRENDS:
The moving average analysis showed that while most customers place orders of moderate value, a few high-value purchases significantly boost overall revenue.

Challenges

Data Consistency:
Merging multiple CSV files required careful handling of inconsistent columns, missing values, and duplicate entries, especially when joining orders, payments, and customer data.

SQL vs Python Logic Differences:
Ensuring that calculations (such as moving averages and window functions) produced identical results in both SQL and Python was sometimes tricky due to subtle differences in how each tool handles dates and grouping.

Handling Time Data:
Some discrepancies in retention rates and time-based calculations arose due to variations in how time intervals (like "6 months") are treated in SQL versus Python.

Performance:
Processing large datasets with groupby and merge operations requires optimization, especially in Python, to avoid long execution times.

Conclusions

DATA-DRIVEN STRATEGY:
The analysis provided actionable insights into sales patterns, customer retention, and the impact of top customers on business growth.

GROWTH OPPORTUNITIES:
Improving customer retention even by a few percentage points could result in substantial increases in revenue.

BUSINESS FOCUS:
Efforts should focus on engaging high-value customers and converting first-time buyers into repeat purchasers through targeted campaigns.

SQL + PYTHON SYNERGY:
Combining SQL for data extraction with Python for visualization and deeper analysis enabled a comprehensive exploration of the dataset.

Thank you very much!

For More Info-

Email – mohi.workspace@gmail.com
LinkedIn Id- www.linkedin.com/in/mohi-gupta-17b8b51b6
Github-https://github.com/MohiGupta26