**PYTHON & SQL**

**Question 1**:

List all customers along with their corresponding city and state.

**Objective**:

To display each customer’s name, city, and state with the help of **Customers** table.

**SQL QUERY**:

SELECT customer\_id, customer\_city, customer\_state

FROM dbo.Customers

ORDER BY customer\_id

**SQL EXPLANATION**:

Clause Explanation

SELECT Defines which columns to show.

FROM dbo.Customers Fetches data from the Customers table.

ORDER BY Sorts the result in ascending order by customer ID.

**PYTHON EQUIVALENT (PANDAS)**:

customers\_df = pd.read\_sql(query, conn)

# Show top few rows

print(customers\_df.head())

print(f"Total customers: {len(customers\_df)}")

**OPTIONAL VISUALIZATION**:

city\_counts = customers\_df['customer\_city'].value\_counts().reset\_index()

city\_counts.columns = ['city', 'num\_customers']

plt.figure(figsize=(10,6))

sns.barplot(data=city\_counts.head(10), x='num\_customers', y='city')

plt.title('Top 10 Cities by Number of Customers')

plt.xlabel('Number of Customers')

plt.ylabel('City')

plt.tight\_layout()

plt.show()

**INTERPRETATION**:

Since customer names are not stored, this query uses **IDs** as unique identifiers and extracts **city and state** directly from the Customers table.  
It helps understand customer geographic distribution and can later be joined with Orders or Payments for deeper analysis.

**\*\*\*Imp Point**: Run your Query in the below format

import pyodbc

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

# Create connection string (update with your actual credentials)

conn = pyodbc.connect(

"DRIVER={ODBC Driver 18 for SQL Server};"

"SERVER=localhost;"

"DATABASE=your\_database\_name;"

"Trusted\_Connection=yes;"

)

# Write your SQL query

query = """

SELECT customer\_id, customer\_city, customer\_state

FROM dbo.Customers

ORDER BY customer\_id;

"""

# Execute SQL and load results into a pandas DataFrame

customers\_df = pd.read\_sql(query, conn)

# Show top few rows

print(customers\_df.head())

**Question 2**:

Display all product categories with their prices.

**Objective**:

Retrieve a list of all product categories along with their price

**SQL QUERY**:

SELECT DISTINCT p.product\_category, o.price

FROM products p

INNER JOIN order\_items o ON p.product\_id = o.product\_id

ORDER BY p.product\_category, o.price DESC;

**SQL EXPLANATION**:

Clause: Explanation

SELECT: Defines which columns to show.

FROM products p: Start from the Products table and give it alias p for simplicity.

JOIN order\_items o: Perform the JOIN and specified it as “Inner”. By default join in SQL will be ‘Inner’. Joined the above table with “order\_items” table.

ON p.product\_id = o.product\_id: It should join on particular column name present in both the tables. In our case it is product\_id.

ORDER BY: Sorts the result in ascending order by p.product\_category and o.price.

**PYTHON EQUIVALENT (PANDAS)**:

# Read both tables

df\_products = pd.read\_sql("SELECT \* FROM products", conn)

df\_order\_items = pd.read\_sql("SELECT \* FROM order\_items", conn)

# Merge and get distinct combinations

df\_category\_prices = df\_products.merge(df\_order\_items, on='product\_id')[['product\_category', 'price']].drop\_duplicates()

df\_category\_prices = df\_category\_prices.sort\_values(['product\_category', 'price'], ascending=[True, False])

print(df\_category\_prices.head(10))

**OPTIONAL VISUALIZATION**:

# Distribution plot to show price distribution by category

plt.figure(figsize=(14, 8))

top\_categories = df\_category\_prices['product\_category'].value\_counts().head(10).index

df\_top = df\_category\_prices[df\_category\_prices['product\_category'].isin(top\_categories)]

sns.displot(data=df\_top, x='product\_category', y='price', palette='Set2')

plt.title('Price Distribution by Product Category (Top 10)', fontsize=16, fontweight='bold')

plt.xlabel('Product Category', fontsize=12)

plt.ylabel('Price', fontsize=12)

plt.xticks(rotation=45, ha='right')

plt.tight\_layout()

plt.show()

**SUMMARY**:

This analysis reveals pricing strategies across product categories. Categories with wider price ranges offer diverse product tiers, while narrow ranges suggest standardized pricing. Identifying premium categories (high average prices) versus volume categories (low prices) helps in inventory and marketing prioritization.

**Question 3**:

Display all unique payment methods used by customers.

**Objective**:

Identify all distinct payment types to understand customer payment preferences and payment gateway requirements.

**SQL QUERY**:

SELECT DISTINCT payment\_type,

COUNT(\*) OVER (PARTITION BY payment\_type) as usage\_count

FROM payments

ORDER BY usage\_count DESC;

**SQL EXPLANATION**:

Clause: Explanation

SELECT: Defines which columns to show.

DISTINCT payment\_type: Retrieves unique payment methods

COUNT(\*) OVER (PARTITION BY payment\_type): Window function that counts occurrences of each payment type

FROM payments: Source table containing payment transaction data

ORDER BY usage\_count DESC: Sorts by popularity (most used first), then alphabetically

**PYTHON EQUIVALENT (PANDAS)**:

# Read payments table

df\_payments = pd.read\_sql("SELECT \* FROM payments", conn)

# Get unique payment types with counts

payment\_summary = df\_payments['payment\_type'].value\_counts().reset\_index()

payment\_summary.columns = ['payment\_type', 'usage\_count']

print("Unique Payment Methods:")

print(payment\_summary)

**SUMMARY**:

Understanding payment method preferences is crucial for optimizing checkout processes and reducing cart abandonment. Dominant payment methods should receive priority support and integration, while underutilized methods may indicate areas for improvement or potential removal to simplify the customer experience.

**Question 4**: Find the total number of sellers operating in each state.

**Objective**: Count the number of unique sellers in each state to understand seller distribution and marketplace coverage.

**SQL QUERY**:

SELECT seller\_state, COUNT(DISTINCT seller\_id) AS total\_sellers

FROM sellers

GROUP BY seller\_state

ORDER BY total\_sellers DESC;

**SQL EXPLANATION**:

Clause: Explanation

SELECT: Defines which columns to show.

COUNT(DISTINCT seller\_id) AS total\_sellers: Counts unique sellers (handles potential duplicates)

FROM sellers: Source table “sellers”

GROUP BY seller\_state: Aggregates data by state

ORDER BY total\_sellers DESC: Shows states with most sellers first

**PYTHON EQUIVALENT (PANDAS)**:

# Read sellers table

df\_sellers = pd.read\_sql("SELECT \* FROM sellers", conn)

# Group by state and count unique sellers

seller\_distribution = df\_sellers.groupby('seller\_state')['seller\_id'].nunique().reset\_index()

seller\_distribution.columns = ['seller\_state', 'total\_sellers']

seller\_distribution = seller\_distribution.sort\_values('total\_sellers', ascending=False)

print("Sellers by State:")

print(seller\_distribution)

**OPTIONAL VISUALIZATION**:

# Horizontal bar chart for top states

plt.figure(figsize=(12, 8))

top\_states = seller\_distribution.head(5)

sns.barplot(data=top\_states, y='seller\_state', x='total\_sellers', palette='rocket')

plt.title('Top 5 States by Number of Sellers', fontsize=16, fontweight='bold')

plt.xlabel('Number of Sellers', fontsize=12)

plt.ylabel('State', fontsize=12)

plt.tight\_layout()

plt.show()

**SUMMARY**:

Seller concentration by state reveals marketplace maturity and potential expansion opportunities. States with high seller density indicate competitive markets with established infrastructure, while states with few sellers represent growth opportunities. This data informs seller acquisition strategies and regional marketing investments.

**Question 5**: List all orders placed in the last quarter of 2017.

**Objective**: Retrieve all orders from Q4 2017 (October, November, December) for quarterly performance analysis and seasonal trend identification.

**SQL QUERY**:

SELECT order\_status, order\_purchase\_timestamp,

MONTH(order\_purchase\_timestamp) AS purchase\_month,

DAY(order\_purchase\_timestamp) AS purchase\_day

FROM orders

WHERE order\_purchase\_timestamp >= '2017-10-01'

AND order\_purchase\_timestamp < '2018-01-01'

AND order\_status NOT IN ('canceled', 'unavailable')

ORDER BY order\_purchase\_timestamp;

**SQL EXPLANATION**:

Clause: Explanation

SELECT: Defines which columns to show.

MONTH(order\_purchase\_timestamp) AS purchase\_month: Extracts month for easy filtering in reports

DAY(order\_purchase\_timestamp) AS purchase\_day: Extracts day for daily trend analysis

FROM orders: Source table “orders”

WHERE Clause: Usage of where clause for filtering and using logical operators to filter the data for Q4 2017

ORDER BY order\_purchase\_timestamp: for Chronological ordering

**PYTHON EQUIVALENT (PANDAS)**:

# Read orders table

df\_orders = pd.read\_sql("SELECT \* FROM orders", conn)

# Convert to datetime

df\_orders['order\_purchase\_timestamp'] = pd.to\_datetime(df\_orders['order\_purchase\_timestamp'])

# Filter Q4 2017

q4\_2017 = df\_orders[(df\_orders['order\_purchase\_timestamp'] >= '2017-10-01') &

(df\_orders['order\_purchase\_timestamp'] < '2018-01-01') &

(~df\_orders['order\_status'].isin(['canceled', 'unavailable']))]

# Add month and day columns

q4\_2017['purchase\_month'] = q4\_2017['order\_purchase\_timestamp'].dt.month

q4\_2017['purchase\_day'] = q4\_2017['order\_purchase\_timestamp'].dt.day

print(f"Total Q4 2017 Orders placed were: {len(q4\_2017)}")

print(q4\_2017[['order\_status', 'order\_purchase\_timestamp']])

**SUMMARY**:

Q4 analysis is crucial for understanding holiday shopping patterns and year-end performance. This period typically shows increased order volumes due to Black Friday, Cyber Monday, and Christmas shopping. Comparing Q4 performance against other quarters reveals seasonality and helps in inventory planning and staffing for peak periods.

**Question 6**: Identify the top 5 cities with the most registered customers.

**Objective**: Find the five cities with the highest customer concentration to prioritize marketing efforts and logistics infrastructure.

**SQL QUERY**:

SELECT TOP 5

customer\_city,

COUNT(customer\_id) AS customer\_count,

CAST(COUNT(customer\_id) \* 100.0 / SUM(COUNT(customer\_id)) OVER () AS DECIMAL(5,2)) AS percentage\_of\_total

FROM customers

GROUP BY customer\_city

ORDER BY customer\_count DESC;

**SQL EXPLANATION**:

Clause: Explanation

SELECT TOP 5: Limits results to top 5 cities (SQL Server syntax)

COUNT(customer\_id) AS customer\_count: Counts customers per city

CAST(COUNT(customer\_id) \* 100.0 / SUM(COUNT(customer\_id)): Calculates percentage of total customers

OVER (): Window function to get total customer count across all cities

FROM customers: Source table “customers”

GROUP BY customer\_city: Aggregates by city

ORDER BY customer\_count DESC: Ranks by customer count

**PYTHON EQUIVALENT (PANDAS)**:

# Read customers table

df\_customers = pd.read\_sql("SELECT \* FROM customers", conn)

# Group by city and state

city\_distribution = df\_customers.groupby(['customer\_city']).size().reset\_index(name='customer\_count')

# Calculate percentage

city\_distribution['percentage\_of\_total'] = (city\_distribution['customer\_count'] / city\_distribution['customer\_count'].sum() \* 100).round(2)

# Get top 5

top\_5\_cities = city\_distribution.sort\_values('customer\_count', ascending=False).head(5)

print("Top 5 Cities by Customer Count:")

print(top\_5\_cities)

**OPTIONAL VISUALIZATION**:

# Bar chart with percentages

fig, ax = plt.subplots(figsize=(12, 6))

cities\_labels = top\_5\_cities['customer\_city']

bars = ax.bar(range(len(top\_5\_cities)), top\_5\_cities['customer\_count'], color='coral', edgecolor='black')

# Add percentage labels on bars

for i, (count, pct) in enumerate(zip(top\_5\_cities['customer\_count'], top\_5\_cities['percentage\_of\_total'])):

ax.text(i, count, f'{count}\n({pct}%)', ha='center', va='bottom', fontweight='bold')

ax.set\_xticks(range(len(top\_5\_cities)))

ax.set\_xticklabels(cities\_labels, rotation=45, ha='right')

ax.set\_ylabel('Number of Customers', fontsize=12)

ax.set\_title('Top 5 Cities by Customer Count', fontsize=16, fontweight='bold')

plt.tight\_layout()

plt.show()

**SUMMARY**:

Top customer cities represent strategic markets requiring focused attention. These cities justify dedicated distribution centers, local marketing campaigns, and faster delivery options. The percentage distribution reveals market concentration—high concentration (e.g., top 5 cities = 50%+ of customers) suggests market dependency risk, while low concentration indicates distributed customer base.

**Question 7**: Display the total quantity of each product sold.

**Objective**: Calculate the total number of units sold for each product to identify bestsellers and slow-moving inventory.

**SQL QUERY**:

SELECT

p.product\_category,

COUNT(oi.order\_item\_id) AS total\_quantity\_sold,

SUM(oi.price) AS total\_revenue

FROM order\_items oi

LEFT JOIN products p ON oi.product\_id = p.product\_id

GROUP BY p.product\_category

ORDER BY total\_quantity\_sold DESC;

**SQL EXPLANATION**:

Clause: Explanation

SELECT: Defines which columns to show

COUNT(oi.order\_item\_id) AS total\_quantity\_sold: Each order\_item\_id represents one unit sold

SUM(oi.price) AS total\_revenue: Total revenue generated by the product

OVER (): Window function to get total customer count across all cities

FROM order\_items oi: Main table with sales transactions

LEFT JOIN products p: Adds product details (LEFT to include products with missing category)

GROUP BY p.product\_category: Aggregates by product

ORDER BY total\_quantity\_sold DESC: Bestsellers first

**PYTHON EQUIVALENT (PANDAS)**:

# Read tables

df\_order\_items = pd.read\_sql("SELECT \* FROM order\_items", conn)

df\_products = pd.read\_sql("SELECT product\_id, product\_category FROM products", conn)

# Merge and aggregate

product\_sales = df\_order\_items.groupby('product\_id').agg(

total\_quantity\_sold=('order\_item\_id', 'count'),

total\_revenue=('price', 'sum')

).reset\_index()

# Add category information

product\_sales = product\_sales.merge(df\_products, on='product\_id', how='left')

# Sort by quantity

product\_sales = product\_sales.sort\_values('total\_quantity\_sold', ascending=False)

print("Product Sales Summary:")

print(product\_sales.head(20))

**SUMMARY**:

Product sales quantity analysis reveals the classic 80/20 rule—typically 20% of products generate 80% of sales. High-volume products require consistent inventory levels and optimized supply chains, while low-volume products may need promotional support or consideration for discontinuation. This data drives inventory investment decisions and SKU rationalization strategies.

**Question 8**: Determine which payment type contributes the highest total revenue.

**Objective**: Identify the payment method (e.g., credit card, debit card, etc.) that generates the highest total revenue across all customer orders.

**SQL QUERY**:

SELECT pmt.payment\_type,

ROUND(SUM(pmt.payment\_value), 2) AS total\_revenue

FROM payments AS pmt

GROUP BY pmt.payment\_type

ORDER BY total\_revenue DESC;

**SQL EXPLANATION**:

* FROM payments AS pmt — retrieves all payment records.
* SUM(pmt.payment\_value) — sums up the total payment amount per payment type.
* GROUP BY pmt.payment\_type — aggregates revenue by each distinct payment method.
* ROUND(..., 2) — ensures consistency with two decimal precision.
* ORDER BY total\_revenue DESC — sorts to show the top revenue-contributing payment type first.

**PYTHON EQUIVALENT (PANDAS)**:

# Load payments data

payments = pd.read\_sql("SELECT \* FROM payments", conn)

# Calculate total revenue per payment type

revenue\_by\_payment = (

payments.groupby("payment\_type")["payment\_value"]

.sum()

.round(2)

.reset\_index()

.sort\_values(by="payment\_value", ascending=False)

)

print(revenue\_by\_payment)

**OPTIONAL VISUALIZATION**:

plt.figure(figsize=(8,5))

total\_payments = revenue\_by\_payment

sns.barplot(x="payment\_value", y="payment\_type", data=total\_payments, palette="crest")

plt.title("Total Revenue by Payment Type")

plt.xlabel("Total Revenue")

plt.ylabel("Payment Type")

plt.tight\_layout()

plt.show()

**SUMMARY**:

This analysis reveals which payment methods customers prefer and which drive the most revenue. Typically, credit card payments dominate, reflecting convenience and installment flexibility.

**INTERMEDIATE LEVEL QUERIES**

**Question 1**: Calculate the total revenue per product category.

**Objective**: Determine which product categories generate the most revenue to prioritize inventory investment and marketing focus.

**SQL QUERY**:

SELECT

p.product\_category,

ROUND(SUM(oi.price + oi.freight\_value), 2) AS total\_revenue

FROM order\_items AS oi

JOIN products AS p

ON oi.product\_id = p.product\_id

GROUP BY p.product\_category

ORDER BY total\_revenue DESC;

**SQL EXPLANATION**:

Clause: Explanation

SELECT: Defines which columns to show

ROUND(SUM(oi.price + oi.freight\_value), 2) AS total\_revenue: Total revenue generated by the product category includes freight values

FROM order\_items oi: Main table with sales transactions

JOIN products p: Adds product details

GROUP BY p.product\_category: Aggregates metrics by category

ORDER BY total\_revenue DESC: Shows highest revenue categories first

**PYTHON EQUIVALENT (PANDAS)**:

# Read tables

order\_items = pd.read\_sql("SELECT \* FROM order\_items", conn)

products = pd.read\_sql("SELECT \* FROM products", conn)

# Merge and aggregate

merged\_df = pd.merge(order\_items, products, on="product\_id")

merged\_df["total\_value"] = merged\_df["price"] + merged\_df["freight\_value"]

revenue\_per\_category = (

merged\_df.groupby("product\_category")["total\_value"]

.sum()

.round(2)

.reset\_index()

.sort\_values(by="total\_value", ascending=False)

)

print(revenue\_per\_category.head(5))

**SUMMARY**:

Category revenue analysis reveals strategic product portfolio priorities. High-revenue categories warrant premium shelf space and marketing investment, while low-revenue categories may need repositioning or elimination. The relationship between average price and units sold identifies whether categories compete on volume (low price, high volume) or premium positioning (high price, lower volume).

**Question 2**: Find the average order value per customer.

**Objective**: To calculate the **average order value (AOV)** for each customer by determining their total order amount and averaging it across all their orders. This metric helps measure customer spending behavior and overall purchase value trends.

**SQL QUERY**:

SELECT TOP 10 c.customer\_id,

ROUND(AVG(order\_total), 2) AS avg\_order\_value

FROM

(

SELECT o.order\_id, o.customer\_id,

SUM(oi.price + oi.freight\_value) AS order\_total

FROM orders AS o

JOIN order\_items AS oi

ON o.order\_id = oi.order\_id

GROUP BY o.order\_id, o.customer\_id

) AS os

JOIN customers AS c

ON os.customer\_id = c.customer\_id

GROUP BY c.customer\_id

ORDER BY avg\_order\_value DESC;

**SQL EXPLANATION**:

Clause: Explanation

SELECT TOP 10: Returns only the top 10 customers with the highest average order values.

ROUND(AVG(order\_total), 2): Calculates the average order value for each customer and rounds it to two decimal places.

Subquery (AS os): Computes total value of each order (price + freight) by joining orders and order\_items

SUM(oi.price + oi.freight\_value): Aggregates product price and freight cost to determine total order value.

GROUP BY o.order\_id, o.customer\_id: Ensures one total per order per customer before averaging.

JOIN customers AS c: Links computed order totals back to the customers table to fetch customer-level information.

GROUP BY c.customer\_id: Groups all order totals by customer to compute their average order value.

ORDER BY avg\_order\_value DESC: Sorts results so highest-spending customers appear first.

**PYTHON EQUIVALENT (PANDAS)**:

orders = pd.read\_sql("SELECT \* FROM orders", conn)

order\_items = pd.read\_sql("SELECT \* FROM order\_items", conn)

customers = pd.read\_sql("SELECT \* FROM customers", conn)

# Step 1: Calculate total value per order

order\_totals = order\_items.groupby(["order\_id"])[["price", "freight\_value"]].sum()

order\_totals["order\_total"] = order\_totals["price"] + order\_totals["freight\_value"]

order\_totals = order\_totals.reset\_index()[["order\_id", "order\_total"]]

# Step 2: Merge with orders and customers

merged = orders.merge(order\_totals, on="order\_id").merge(customers, on="customer\_id")

# Step 3: Calculate average order value per customer

avg\_order\_value = (

merged.groupby("customer\_id")["order\_total"]

.mean()

.round(2)

.reset\_index()

.sort\_values(by="order\_total", ascending=False)

)

print(avg\_order\_value.head(10))

**OPTIONAL VISUALIZATION**:

top\_customers = avg\_order\_value.head(10)

plt.figure(figsize=(10,6))

sns.barplot(x="order\_total", y="customer\_id", data=top\_customers)

plt.title("Top 10 Customers by Average Order Value")

plt.xlabel("Average Order Value")

plt.ylabel("Customer ID")

plt.tight\_layout()

plt.show()

**SUMMARY**:

This query identifies which customers have the highest average order value, giving insight into **customer profitability** and **spending habits**.  
The AOV metric is essential for revenue analysis, helping businesses identify their most valuable customers and tailor loyalty or upselling strategies accordingly.

**Question 3**: Show the top 10 best-selling products in year 2017.

**Objective**: To identify the **top 10 products** with the highest number of units sold during **2017**. This helps evaluate product popularity, demand trends, and supports inventory or marketing planning for future sales cycles.

**SQL QUERY**:

SELECT TOP 10

p.product\_id,

p.product\_category,

sales\_per\_product.total\_units\_sold

FROM products AS p

INNER JOIN (

SELECT

oi.product\_id,

COUNT(oi.order\_item\_id) AS total\_units\_sold

FROM order\_items AS oi

INNER JOIN orders AS o

ON oi.order\_id = o.order\_id

WHERE YEAR(o.order\_purchase\_timestamp) = 2017

GROUP BY oi.product\_id

) AS sales\_per\_product

ON p.product\_id = sales\_per\_product.product\_id

ORDER BY

sales\_per\_product.total\_units\_sold DESC;

**SQL EXPLANATION**:

Clause: Explanation

SELECT TOP 10: Limits output to the top 10 best-selling products based on total units sold.

p.product\_id, p.product\_category: Displays product identifiers and their respective categories for clarity.

Subquery (sales\_per\_product): Calculates how many units of each product were sold in 2017.

COUNT(oi.order\_item\_id): Counts each line item sold — one per product per order — giving total quantity sold.

INNER JOIN orders AS o ON oi.order\_id = o.order\_id: Combines order\_items with orders to filter by purchase date.

WHERE YEAR(o.order\_purchase\_timestamp) = 2017: Filters only those orders that occurred during the year 2017.

GROUP BY oi.product\_id: Groups results by product so the count reflects total units sold per product.

INNER JOIN products AS p: Links sales data with the product details (category, etc.).

ORDER BY total\_units\_sold DESC: Sorts from highest to lowest sales volume to get best-sellers first.

**PYTHON EQUIVALENT (PANDAS)**:

orders\_df = pd.read\_sql("SELECT \* FROM orders", conn)

order\_items\_df = pd.read\_sql("SELECT \* FROM order\_items", conn)

products\_df = pd.read\_sql("SELECT \* FROM products", conn)

orders\_df['order\_purchase\_timestamp'] = pd.to\_datetime(

orders\_df['order\_purchase\_timestamp'])

orders\_2017\_df = orders\_df[

orders\_df['order\_purchase\_timestamp'].dt.year == 2017

][['order\_id', 'customer\_id']]

order\_product\_2017\_df = pd.merge(

orders\_2017\_df,

order\_items\_df[['order\_id', 'product\_id', 'order\_item\_id']],

on='order\_id',

how='inner'

)

product\_sales\_2017 = order\_product\_2017\_df.groupby('product\_id').agg(

Total\_Units\_Sold\_2017=('order\_item\_id', 'count')

).reset\_index()

final\_sales\_df = pd.merge(

product\_sales\_2017,

products\_df[['product\_id', 'product\_category']],

on='product\_id',

how='inner'

)

top\_10\_selling\_products\_2017 = final\_sales\_df.sort\_values(

by='Total\_Units\_Sold\_2017',

ascending=False

).head(10)

print(top\_10\_selling\_products\_2017[['product\_category', 'Total\_Units\_Sold\_2017']])

**SUMMARY**:

This query determines which products sold the most units in 2017. It helps identify **high-performing items**, guiding **inventory management**, **promotion planning**, and **product forecasting**.  
The results provide a clear view of customer preferences and market demand within that year.

**Question 4**: Find the number of orders handled by each seller and rank them.

**Objective**: To calculate the total number of unique orders managed by each seller and assign a rank based on their order-handling volume. This analysis identifies **top-performing sellers** and helps evaluate workload distribution and seller performance efficiency.

**SQL QUERY**:

SELECT

seller\_orders.seller\_id,

seller\_orders.total\_orders,

RANK() OVER (ORDER BY seller\_orders.total\_orders DESC) AS seller\_rank

FROM (

SELECT

oi.seller\_id,

COUNT(DISTINCT oi.order\_id) AS total\_orders

FROM order\_items AS oi

GROUP BY oi.seller\_id

) AS seller\_orders

JOIN sellers AS s

ON seller\_orders.seller\_id = s.seller\_id

ORDER BY

seller\_orders.total\_orders DESC;

**SQL EXPLANATION**:

Clause: Explanation

Subquery (seller\_orders): Calculates the number of unique orders per seller from the order\_items table.

COUNT(DISTINCT oi.order\_id): Counts distinct orders, ensuring each order is only counted once even if it includes multiple items.

GROUP BY oi.seller\_id: Groups the data by seller to compute order counts per seller.

JOIN sellers AS s: Connects aggregated seller data with the sellers table to validate or retrieve additional seller details.

RANK() OVER (ORDER BY seller\_orders.total\_orders DESC): Assigns a rank to each seller based on their total orders, ranking the seller with most orders as 1.

ORDER BY seller\_orders.total\_orders DESC: Displays sellers in descending order of total orders handled for easy identification of top performers.

**PYTHON EQUIVALENT (PANDAS)**:

# Load CSVs

order\_items = pd.read\_sql("SELECT \* FROM order\_items", conn)

sellers = pd.read\_sql("SELECT seller\_id FROM sellers", conn)

# Count distinct orders handled by each seller

seller\_orders = (

order\_items.groupby('seller\_id')['order\_id']

.nunique()

.reset\_index(name='total\_orders')

)

# Merge with sellers data for city/state info

seller\_summary = pd.merge(seller\_orders, sellers, on='seller\_id', how='left')

# Rank sellers by total orders handled

seller\_summary['seller\_rank'] = seller\_summary['total\_orders'].rank(method='dense', ascending=False).astype(int)

# Sort and display

seller\_summary = seller\_summary.sort\_values(by='total\_orders', ascending=False)

print(seller\_summary.head(10))

**SUMMARY**:

This query highlights how many unique orders each seller processed and ranks them accordingly. It provides valuable insights into **seller productivity** and **sales contribution**, helping management identify **key partners**, optimize logistics, and set performance benchmarks across the seller network.

**Question 5**: Calculate the monthly sales trend for the year 2018.

**Objective**: To analyze **monthly sales trends** for 2018 by calculating total sales (including freight) per month. This helps identify seasonal patterns, sales peaks, and months with low performance, aiding in forecasting and strategic planning.

**SQL QUERY**:

SELECT

YEAR(o.order\_purchase\_timestamp) AS sales\_year,

MONTH(o.order\_purchase\_timestamp) AS sales\_month,

SUM(oi.price + oi.freight\_value) AS total\_sales

FROM

orders AS o

INNER JOIN

order\_items AS oi

ON o.order\_id = oi.order\_id

WHERE

YEAR(o.order\_purchase\_timestamp) = 2018

GROUP BY

YEAR(o.order\_purchase\_timestamp),

MONTH(o.order\_purchase\_timestamp)

ORDER BY MONTH(o.order\_purchase\_timestamp);

**SQL EXPLANATION**:

Clause: Explanation

SELECT YEAR(o.order\_purchase\_timestamp): Extracts the year from the order date for reference.

MONTH(o.order\_purchase\_timestamp): Extracts the month to group and analyze sales on a monthly basis.

SUM(oi.price + oi.freight\_value): Calculates total sales per order, combining product price and shipping value.

FROM orders AS o: Specifies the main table containing order-level details.

INNER JOIN order\_items AS oi ON o.order\_id = oi.order\_id: Merges each order with its respective items to access sales and freight data.

WHERE YEAR(o.order\_purchase\_timestamp) = 2018: Filters results to include only orders placed in 2018.

GROUP BY YEAR(o.order\_purchase\_timestamp), MONTH(o.order\_purchase\_timestamp): Groups data by year and month to compute monthly totals.

ORDER BY MONTH(o.order\_purchase\_timestamp): Sorts results chronologically from January to December.

**PYTHON EQUIVALENT (PANDAS)**:

# Load data

orders = pd.read\_sql("SELECT \* FROM orders", conn, parse\_dates=['order\_purchase\_timestamp'])

order\_items = pd.read\_sql("SELECT \* FROM order\_items", conn)

# Filter for 2018 orders

orders\_2018 = orders[orders['order\_purchase\_timestamp'].dt.year == 2018]

# Merge orders with order\_items

merged\_df = pd.merge(order\_items, orders\_2018, on='order\_id', how='inner')

# Create a month column

merged\_df['sales\_month'] = merged\_df['order\_purchase\_timestamp'].dt.month

# Calculate total monthly sales

monthly\_sales = (

merged\_df.groupby('sales\_month')

.apply(lambda x: (x['price'] + x['freight\_value']).sum())

.reset\_index(name='total\_sales')

.sort\_values('sales\_month')

)

print(monthly\_sales)

**OPTIONAL VISUALIZATION**:

plt.plot(monthly\_sales['sales\_month'], monthly\_sales['total\_sales'], marker='o')

plt.title('Monthly Sales Trend for 2018')

plt.xlabel('Month')

plt.ylabel('Total Sales')

plt.grid(True)

plt.show()

**SUMMARY**:

This query produces the **month-wise total sales** for 2018, revealing seasonal sales trends.  
It helps identify high-performing months such as festive seasons and assists in **forecasting demand** and **budget allocation** for future sales planning.  
Visualizing this data in Python as a line or bar chart can further highlight month-to-month performance variations.

**Question 6**: Determine the correlation between product price and number of items sold (using Python libraries).

**Objective**: To measure the **relationship between product price and sales volume** by calculating the correlation coefficient. This analysis helps understand whether higher-priced products tend to sell more or less, guiding pricing and marketing strategies.

**PYTHON EQUIVALENT (PANDAS)**:

# Load data

order\_items = pd.read\_sql("SELECT \* FROM order\_items", conn)

products = pd.read\_sql("SELECT \* FROM products", conn)

# Merge product and order item data

merged\_df = pd.merge(order\_items, products, on='product\_id', how='left')

# Calculate total items sold per product

product\_sales = merged\_df.groupby('product\_id')['order\_item\_id'].count().reset\_index(name='total\_items\_sold')

# Get average price per product

avg\_price = merged\_df.groupby('product\_id')['price'].mean().reset\_index(name='avg\_price')

# Merge sales count and average price

price\_sales\_corr = pd.merge(product\_sales, avg\_price, on='product\_id', how='inner')

# Compute correlation coefficient

correlation\_value = price\_sales\_corr['avg\_price'].corr(price\_sales\_corr['total\_items\_sold'])

print(f"Correlation between product price and total items sold: {correlation\_value:.4f}")

**SUMMARY**:

Using Python’s pandas and seaborn, we can compute and visualize the correlation between **product price** and **total quantity sold**.  
A **positive correlation** indicates that higher-priced products sell more, whereas a **negative correlation** suggests that lower-priced items dominate sales.  
This insight supports **pricing optimization** and **product positioning** decisions in sales strategy.

**ADVANCED LEVEL QUERIES**

**Question 1**: Calculate the customer lifetime value (CLV) based on total purchases.

**Objective**: To compute the **Customer Lifetime Value (CLV)** by aggregating each customer’s total spending across all their purchases, including product prices and freight charges. This helps identify high-value customers who contribute the most revenue over time.

**SQL QUERY**:

WITH customer\_revenue AS (

SELECT

o.customer\_id,

SUM(oi.price + oi.freight\_value) AS customer\_lifetime\_value

FROM

orders AS o

INNER JOIN

order\_items AS oi

ON o.order\_id = oi.order\_id

GROUP BY

o.customer\_id

)

SELECT

customer\_id,

customer\_lifetime\_value

FROM

customer\_revenue

ORDER BY

customer\_lifetime\_value DESC;

**SQL EXPLANATION**:

Clause: Explanation

WITH customer\_revenue AS (...): Defines a **Common Table Expression (CTE)** to first calculate total revenue per customer.

SUM(oi.price + oi.freight\_value): Computes the total amount spent by each customer (product + freight).

INNER JOIN orders AS o: Links orders and order\_items to combine order details with product-level values.

GROUP BY o.customer\_id: Groups all transactions belonging to the same customer to aggregate lifetime purchases.

SELECT customer\_id, customer\_lifetime\_value: Retrieves each customer with their calculated lifetime value.

ORDER BY customer\_lifetime\_value DESC: Sorts the output so that top-spending customers appear first.

**PYTHON EQUIVALENT (PANDAS)**:

# Load CSVs

orders = pd.read\_sql("SELECT \* FROM orders", conn)

order\_items = pd.read\_sql("SELECT \* FROM order\_items", conn)

# Merge orders with order\_items

merged\_df = pd.merge(order\_items, orders, on='order\_id', how='inner')

# Calculate CLV per customer

customer\_revenue = (

merged\_df.groupby('customer\_id')

.apply(lambda x: (x['price'] + x['freight\_value']).sum())

.reset\_index(name='customer\_lifetime\_value')

)

# Sort by CLV descending

customer\_revenue = customer\_revenue.sort\_values(by='customer\_lifetime\_value', ascending=False)

print(customer\_revenue.head(10))

**SUMMARY**:

This query calculates **Customer Lifetime Value (CLV)** — a key metric showing how much revenue each customer has generated.  
It helps businesses identify their **most profitable customers**, focus on **retention strategies**, and design **targeted loyalty or upselling programs** to maximize long-term profitability.

**Question 2**: Analyze repeat customer behavior — how many customers made more than one purchase.

**Objective**: To identify **repeat customers** by counting how many customers placed **more than one order**. This analysis helps measure customer loyalty and engagement, showing how often buyers return for additional purchases.

**SQL QUERY**:

SELECT

COUNT(\*) AS repeat\_customer\_count

FROM (

SELECT

customer\_id

FROM orders

GROUP BY customer\_id

HAVING COUNT(order\_id) > 1

) AS repeat\_customers;

**SQL EXPLANATION**:

Clause: Explanation

Subquery (repeat\_customers): Identifies customers who made multiple purchases by grouping data by customer\_id.

GROUP BY customer\_id: Groups all orders under each unique customer.

HAVING COUNT(order\_id) > 1: Filters only those customers with more than one order — i.e., repeat buyers.

SELECT COUNT(\*): Counts the total number of such repeat customers.

AS repeat\_customer\_count: Assigns a readable alias to the output column.

**PYTHON EQUIVALENT (PANDAS)**:

# Load Queries

orders = pd.read\_sql("SELECT \* FROM orders", conn)

# Step 1: Count number of orders per customer

customer\_order\_count = orders.groupby('customer\_id')['order\_id'].count().reset\_index(name='order\_count')

# Step 2: Filter customers with more than one purchase

repeat\_customers = customer\_order\_count[customer\_order\_count['order\_count'] > 1]

# Step 3: Get count of repeat customers

repeat\_customer\_count = repeat\_customers.shape[0]

print("Number of repeat customers:", repeat\_customer\_count)

**SUMMARY**:

This query reveals the total number of **repeat customers**, providing a direct measure of **customer retention** and satisfaction.  
A higher repeat rate indicates stronger customer loyalty, while a lower rate suggests the need for better post-purchase engagement or loyalty programs.

**Question 3**: Calculate the moving average of monthly sales.

**Objective**: To compute a **3-month moving average** of total monthly sales, smoothing out short-term fluctuations and highlighting longer-term sales trends. This helps in identifying consistent growth, seasonal dips, and forecasting future performance.

**SQL QUERY**:

WITH MonthlySales AS (

SELECT

YEAR(o.order\_purchase\_timestamp) AS order\_year,

MONTH(o.order\_purchase\_timestamp) AS order\_month,

SUM(oi.price + oi.freight\_value) AS total\_monthly\_sales

FROM orders o

INNER JOIN Order\_Items oi ON o.order\_id = oi.order\_id

GROUP BY YEAR(order\_purchase\_timestamp), MONTH(order\_purchase\_timestamp)

)

SELECT

order\_year,

order\_month,

total\_monthly\_sales,

ROUND(AVG(total\_monthly\_sales) OVER (

ORDER BY order\_year, order\_month

ROWS BETWEEN 2 PRECEDING AND CURRENT ROW

), 2) AS moving\_avg\_3month

FROM MonthlySales

ORDER BY order\_year, order\_month;

**SQL EXPLANATION**:

Clause: Explanation

CTE (MonthlySales): Calculates total monthly sales before applying the moving average.

SUM(oi.price + oi.freight\_value): Computes total monthly revenue (product + freight).

GROUP BY YEAR(), MONTH(): Groups results by month and year to summarize sales.

AVG(total\_monthly\_sales) OVER (...): Uses a **window function** to compute the moving average dynamically across rows.

ORDER BY order\_year, order\_month: Ensures the average is calculated in chronological order.

ROWS BETWEEN 2 PRECEDING AND CURRENT ROW: Defines a 3-month rolling window (current month + 2 previous months).

ROUND(..., 2): Rounds the result to two decimal places for readability.

ORDER BY order\_year, order\_month: Displays results sequentially by time period.

**PYTHON EQUIVALENT (PANDAS)**:

orders\_df = pd.read\_sql("SELECT \* FROM orders", conn)

order\_items\_df = pd.read\_sql("SELECT \* FROM order\_items", conn)

# Step 1: Data Preprocessing and Joining

# Ensure the timestamp is in datetime format

orders\_df['order\_purchase\_timestamp'] = pd.to\_datetime(

orders\_df['order\_purchase\_timestamp']

)

# Join the necessary tables: orders and order\_items

sales\_df = pd.merge(

orders\_df[['order\_id', 'order\_purchase\_timestamp']],

order\_items\_df[['order\_id', 'price', 'freight\_value']],

on='order\_id',

how='inner'

)

# Step 2: Calculate Total Monthly Revenue (Equivalent to the MonthlySales CTE)

# Create a 'YearMonth' column for easy grouping and sorting

sales\_df['YearMonth'] = sales\_df['order\_purchase\_timestamp'].dt.to\_period('M')

# Calculate total revenue (price + freight)

sales\_df['Total\_Revenue'] = sales\_df['price'] + sales\_df['freight\_value']

monthly\_revenue\_df = sales\_df.groupby('YearMonth').agg(

Total\_Monthly\_Revenue=('Total\_Revenue', 'sum')

).reset\_index()

# Convert 'YearMonth' back to a sortable datetime for the rolling window

# This ensures correct chronological ordering, crucial for time series analysis

monthly\_revenue\_df['YearMonth'] = monthly\_revenue\_df['YearMonth'].dt.to\_timestamp()

# Step 3: Calculate the 3-Month Moving Average

# Use the .rolling() method on the revenue column

# window=3 specifies a 3-month window (current row + 2 preceding rows)

monthly\_revenue\_df['Moving\_Avg\_3\_Month'] = (

monthly\_revenue\_df['Total\_Monthly\_Revenue']

.rolling(window=3, min\_periods=1) # min\_periods=1 allows calculation for the first month

.mean()

)

# Sort the final result chronologically

monthly\_revenue\_df = monthly\_revenue\_df.sort\_values(by='YearMonth').reset\_index(drop=True)

# Display the final result

print(monthly\_revenue\_df)

**SUMMARY**:

This query provides a **3-month moving average** of total sales, revealing smoother sales patterns over time.  
It helps in understanding **sales momentum**, detecting seasonal effects, and predicting upcoming trends. Businesses can use these insights for **inventory planning**, **budgeting**, and **marketing campaigns** based on consistent performance trends.

**Question 4**: Determine year-over-year sales growth by category.

**Objective**: Calculate how much sales have increased or decreased each year for every product category to identify growth trends and category performance over time.

**SQL QUERY**:

WITH CategoryYearSales AS (

SELECT

p.product\_category,

YEAR(o.order\_purchase\_timestamp) AS order\_year,

SUM(oi.price + oi.freight\_value) AS total\_sales

FROM order\_items oi

JOIN orders o

ON oi.order\_id = o.order\_id

JOIN products p

ON oi.product\_id = p.product\_id

GROUP BY p.product\_category, YEAR(o.order\_purchase\_timestamp)

)

SELECT

product\_category,

order\_year,

total\_sales,

LAG(total\_sales) OVER (PARTITION BY product\_category ORDER BY order\_year) AS prev\_year\_sales,

ROUND(

((total\_sales - LAG(total\_sales) OVER (PARTITION BY product\_category ORDER BY order\_year))

/ NULLIF(LAG(total\_sales) OVER (PARTITION BY product\_category ORDER BY order\_year), 0)) \* 100, 2

) AS yoy\_growth\_percent

FROM CategoryYearSales

ORDER BY product\_category, order\_year;

**SQL EXPLANATION**:

Clause: Explanation

WITH CategoryYearSales AS (...): Creates a **CTE** (Common Table Expression) summarizing yearly sales per category before the main query.

SELECT p.product\_category, YEAR(o.order\_purchase\_timestamp): Extracts each category and order year for grouping.

SUM(oi.price + oi.freight\_value): Calculates total sales (including product price + freight cost).

GROUP BY p.product\_category, YEAR(o.order\_purchase\_timestamp): Aggregates total sales by category and year.

LAG(total\_sales) OVER (...): Retrieves **previous year's sales** for each category using a window function.

ROUND(((total\_sales - LAG(...)) / NULLIF(LAG(...), 0)) \* 100, 2): Computes **YoY growth percentage** while handling division by zero using NULLIF

ORDER BY product\_category, order\_year: Orders results chronologically for easier trend visualization.

**PYTHON EQUIVALENT (PANDAS)**:

orders = pd.read\_sql("SELECT \* FROM orders", conn)

order\_items = pd.read\_sql("SELECT \* FROM order\_items", conn)

products = pd.read\_sql("SELECT \* FROM products", conn)

# Step 1: Merge datasets

merged\_df = (

order\_items.merge(orders, on='order\_id')

.merge(products, on='product\_id')

)

# Step 2: Extract year from order\_purchase\_timestamp

merged\_df['order\_year'] = pd.to\_datetime(merged\_df['order\_purchase\_timestamp']).dt.year

# Step 3: Calculate total annual sales by category (price + freight\_value)

merged\_df['total\_amount'] = merged\_df['price'] + merged\_df['freight\_value']

category\_sales = (

merged\_df.groupby(['product\_category', 'order\_year'])['total\_amount']

.sum()

.reset\_index()

.rename(columns={'total\_amount': 'total\_sales'})

)

# Step 4: Calculate year-over-year growth (sorted for order-by behavior)

category\_sales = category\_sales.sort\_values(['product\_category', 'order\_year'])

category\_sales['prev\_year\_sales'] = (

category\_sales.groupby('product\_category')['total\_sales']

.shift(1)

)

category\_sales['yoy\_growth\_percent'] = (

((category\_sales['total\_sales'] - category\_sales['prev\_year\_sales'])

/ category\_sales['prev\_year\_sales']) \* 100

).round(2)

# Step 5: Display results

print(category\_sales.head(15))

**SUMMARY**:

This query measures how each category’s total sales changed compared to the previous year. The YoY growth percentage reveals which categories are **expanding**, **stable**, or **declining**. Business teams can use this insight for forecasting, marketing strategy, and inventory alignment based on performance trends.

**Question 5**: Identify the top 5 customers who contributed the most revenue in each year.

**Objective**: Find the top 5 highest revenue-generating customers per year to understand key contributors to annual sales performance and customer concentration.

**SQL QUERY**:

WITH CustomerYearlyRevenue AS (

SELECT

o.customer\_id,

DATEPART(YEAR, o.order\_purchase\_timestamp) AS Sale\_Year,

SUM(oi.price + oi.freight\_value) AS Total\_Revenue

FROM orders o

INNER JOIN

order\_items oi ON o.order\_id = oi.order\_id

GROUP BY o.customer\_id, DATEPART(YEAR, o.order\_purchase\_timestamp)

),

RankedCustomers AS (

SELECT

Sale\_Year,

customer\_id,

Total\_Revenue,

RANK() OVER (PARTITION BY Sale\_Year ORDER BY Total\_Revenue DESC) AS Revenue\_Rank

FROM CustomerYearlyRevenue

)

SELECT Sale\_Year, Revenue\_Rank, customer\_id, Total\_Revenue

FROM RankedCustomers

WHERE Revenue\_Rank <= 5 -- Filter for the top 5 ranks

ORDER BY Sale\_Year, Revenue\_Rank;

**SQL EXPLANATION**:

Clause: Explanation

WITH CustomerYearlyRevenue AS (...): A **CTE** that calculates total revenue per customer for each year.

DATEPART(YEAR, o.order\_purchase\_timestamp): Extracts the year from each order’s purchase timestamp.

SUM(oi.price + oi.freight\_value): Adds up product and freight costs to get total revenue.

GROUP BY o.customer\_id, DATEPART(YEAR, ...): Aggregates revenue per customer per year.

RANK() OVER (PARTITION BY Sale\_Year ORDER BY Total\_Revenue DESC): Assigns ranks to customers within each year based on their total revenue, highest first.

WHERE Revenue\_Rank <= 5: Filters to show only the **top 5 customers** in each year.

ORDER BY Sale\_Year, Revenue\_Rank: Sorts results chronologically and by rank for clarity.

**PYTHON EQUIVALENT (PANDAS)**:

orders\_df = pd.read\_sql("SELECT \* FROM orders", conn)

order\_items\_df = pd.read\_sql("SELECT \* FROM order\_items", conn)

# Step 1: Join and Prepare Data

# ---------------------------------------------------------------------

# Ensure timestamp is datetime and calculate total revenue

orders\_df['order\_purchase\_timestamp'] = pd.to\_datetime(orders\_df['order\_purchase\_timestamp'])

order\_items\_df['Total\_Revenue'] = order\_items\_df['price'] + order\_items\_df['freight\_value']

# Join orders (for customer\_id and date) and order\_items (for revenue)

customer\_sales\_detail = pd.merge(

orders\_df[['order\_id', 'customer\_id', 'order\_purchase\_timestamp']],

order\_items\_df[['order\_id', 'Total\_Revenue']],

on='order\_id',

how='inner'

)

# Extract the Year

customer\_sales\_detail['Sale\_Year'] = customer\_sales\_detail['order\_purchase\_timestamp'].dt.year

# Step 2: Aggregate to get Yearly Customer Revenue

# ---------------------------------------------------------------------

# Group by both year and customer\_id

customer\_yearly\_revenue = customer\_sales\_detail.groupby(['Sale\_Year', 'customer\_id']).agg(

Total\_Revenue=('Total\_Revenue', 'sum')

).reset\_index()

# Step 3: Identify the Top 5 Customers in each Year (Top N Per Group)

# ---------------------------------------------------------------------

# The nlargest(5) function is applied to the 'Total\_Revenue' column within each 'Sale\_Year' group.

def get\_top\_n\_customers(group):

"""Returns the top 5 customers from a single year group based on Total\_Revenue."""

return group.nlargest(5, 'Total\_Revenue')

top\_5\_customers\_yearly = customer\_yearly\_revenue.groupby('Sale\_Year').apply(

get\_top\_n\_customers

).reset\_index(drop=True)

# Step 4: Add Rank and Final Display

# ---------------------------------------------------------------------

# Calculate the rank within the final result set for clear presentation

top\_5\_customers\_yearly['Revenue\_Rank'] = (

top\_5\_customers\_yearly.groupby('Sale\_Year')['Total\_Revenue']

.rank(method='dense', ascending=False)

.astype(int)

)

# Select and display the final result columns

final\_result = top\_5\_customers\_yearly[['Sale\_Year', 'Revenue\_Rank', 'customer\_id', 'Total\_Revenue']]

print(final\_result.sort\_values(by=['Sale\_Year', 'Revenue\_Rank']))

**SUMMARY**:

This query identifies each year’s top 5 revenue-contributing customers, highlighting who drives the most sales. These insights support **customer retention strategies**, **VIP segmentation**, and **targeted marketing efforts** to sustain or grow key relationships.

**Question 6**: Perform a seller performance analysis — revenue, number of orders, and average order value (visualize using Seaborn).

**Objective**: Evaluate seller performance based on total revenue, number of orders, and average order value to identify top-performing sellers.

**SQL QUERY**:

SELECT

oi.seller\_id,

COUNT(DISTINCT oi.order\_id) AS total\_orders,

SUM(oi.price + oi.freight\_value) AS total\_revenue,

Round((SUM(oi.price + oi.freight\_value) / COUNT(DISTINCT oi.order\_id)),2) AS avg\_order\_value

FROM order\_items oi

JOIN orders o

ON oi.order\_id = o.order\_id

GROUP BY oi.seller\_id

ORDER BY total\_revenue DESC;

**SQL EXPLANATION**:

Clause: Explanation

SELECT oi.seller\_id: Identifies each seller for performance comparison.

COUNT(DISTINCT oi.order\_id): Counts unique orders handled by each seller.

SUM(oi.price + oi.freight\_value): Calculates total revenue earned (sales + freight).

ROUND(...,2): Computes average order value, rounded to two decimals.

FROM order\_items oi JOIN orders o: Combines order details and order timestamps.

GROUP BY oi.seller\_id: Aggregates metrics at the seller level.

ORDER BY total\_revenue DESC: Sorts sellers by highest revenue first.

**PYTHON EQUIVALENT (PANDAS)**:

orders = pd.read\_sql("SELECT \* FROM orders", conn)

order\_items = pd.read\_sql("SELECT \* FROM order\_items", conn)

# Step 1: Merge necessary datasets

merged\_df = order\_items.merge(orders, on='order\_id')

# Step 2: Calculate total revenue per seller

seller\_perf = (

merged\_df.groupby('seller\_id')

.agg(

total\_orders=('order\_id', 'nunique'),

total\_price=('price', 'sum'),

total\_freight=('freight\_value', 'sum')

)

.reset\_index()

)

# Step 3: Derive total revenue and average order value

seller\_perf['total\_revenue'] = seller\_perf['total\_price'] + seller\_perf['total\_freight']

seller\_perf['avg\_order\_value'] = (seller\_perf['total\_revenue'] / seller\_perf['total\_orders']).round(2)

# Step 4: Sort by revenue (Top 15 sellers)

top\_sellers = seller\_perf.sort\_values(by='total\_revenue', ascending=False).head(15)

**VISUALIZATION**:

# Step 5: Visualization 1 — Revenue by Seller

plt.figure(figsize=(12, 6))

sns.barplot(data=top\_sellers, x='seller\_id', y='total\_revenue', palette='viridis')

plt.title('Top 15 Sellers by Total Revenue', fontsize=15)

plt.xlabel('Seller ID')

plt.ylabel('Total Revenue')

plt.xticks(rotation=45)

plt.tight\_layout()

plt.show()

# Step 6: Visualization 2 — Average Order Value vs Total Orders

plt.figure(figsize=(10, 6))

sns.scatterplot(

data=top\_sellers,

x='total\_orders',

y='avg\_order\_value',

size='total\_revenue',

hue='total\_revenue',

palette='coolwarm',

alpha=0.8

)

plt.title('Seller Performance: Orders vs Average Order Value', fontsize=15)

plt.xlabel('Total Orders')

plt.ylabel('Average Order Value (AOV)')

plt.legend(title='Revenue', loc='upper right')

plt.tight\_layout()

plt.show()

**SUMMARY**:

This query highlights the most profitable sellers by evaluating total sales, order counts, and average order value. It helps businesses identify top performers and strategize seller engagement, incentives, or training accordingly.

**Question 7(a)**: Build a revenue heatmap showing monthly revenue by product category.

**Objective**: Visualize how revenue varies across months and product categories to identify seasonal trends and high-performing product lines.

**SQL QUERY**:

SELECT

p.product\_category,

YEAR(o.order\_purchase\_timestamp) AS order\_year,

MONTH(o.order\_purchase\_timestamp) AS order\_month,

round(SUM(oi.price + oi.freight\_value), 2) AS total\_revenue

FROM order\_items oi

JOIN orders o

ON oi.order\_id = o.order\_id

JOIN products p

ON oi.product\_id = p.product\_id

GROUP BY

p.product\_category,

YEAR(o.order\_purchase\_timestamp),

MONTH(o.order\_purchase\_timestamp)

ORDER BY

p.product\_category,

order\_year,

order\_month;

**SQL EXPLANATION**:

Clause: Explanation

SELECT p.product\_category: Groups data by product category for comparison.

YEAR(o.order\_purchase\_timestamp): Extracts order year for time-based grouping.

MONTH(o.order\_purchase\_timestamp): Extracts order month to form a monthly trend.

ROUND(SUM(oi.price + oi.freight\_value), 2): Computes total monthly revenue, rounded for clarity.

FROM order\_items ... JOIN orders ... JOIN products: Merges sales, order, and product data for a unified dataset.

GROUP BY ...: Aggregates total revenue by category, year, and month.

ORDER BY ...: Orders results for chronological and categorical visualization.

**PYTHON EQUIVALENT (PANDAS)**:

orders = pd.read\_sql("SELECT \* FROM orders", conn)

order\_items = pd.read\_sql("SELECT \* FROM order\_items", conn)

products = pd.read\_sql("SELECT \* FROM products", conn)

# Step 1: Merge datasets

merged\_df = (

order\_items.merge(orders, on='order\_id')

.merge(products, on='product\_id')

)

# Step 2: Extract year and month

merged\_df['order\_year'] = pd.to\_datetime(merged\_df['order\_purchase\_timestamp']).dt.year

merged\_df['order\_month'] = pd.to\_datetime(merged\_df['order\_purchase\_timestamp']).dt.month

# Step 3: Calculate monthly revenue by category (including freight)

monthly\_revenue = (

merged\_df.groupby(['product\_category', 'order\_year', 'order\_month'])

.agg(total\_revenue=('price', 'sum'), freight\_revenue=('freight\_value', 'sum'))

.reset\_index()

)

# Combine total price and freight

monthly\_revenue['total\_revenue'] = monthly\_revenue['total\_revenue'] + monthly\_revenue['freight\_revenue']

**OPTIONAL VISUALIZATION**:

# Step 4: Pivot data for heatmap (rows: category, columns: month)

heatmap\_data = monthly\_revenue.pivot\_table(

index='product\_category',

columns='order\_month',

values='total\_revenue',

aggfunc='sum'

)

# Step 5: Plot heatmap

plt.figure(figsize=(12, 8))

sns.heatmap(heatmap\_data, cmap='YlGnBu', annot=False, linewidths=0.5)

plt.title('Monthly Revenue by Product Category', fontsize=16)

plt.xlabel('Month')

plt.ylabel('Product Category')

plt.show()

**SUMMARY**:

This query prepares data for a heatmap that visually displays monthly revenue trends per product category, helping identify peak months, product seasonality, and revenue concentration areas.

**Question 7(b)**: Build a revenue heatmap showing monthly revenue by product category.

**Objective**: Generate a pivot-style monthly revenue summary per product category to visualize seasonal performance trends across all 12 months.

**SQL QUERY**:

WITH MonthlyCategorySales AS (

-- CTE 1: Calculate Total Revenue for each Product Category per Month

SELECT

p.product\_category,

-- Extract the Month (as a number)

DATEPART(MONTH, o.order\_purchase\_timestamp) AS Sale\_Month,

-- Calculate total sales (price + freight)

SUM(oi.price + oi.freight\_value) AS Monthly\_Revenue

FROM

orders o

INNER JOIN

order\_items oi ON o.order\_id = oi.order\_id

INNER JOIN

products p ON oi.product\_id = p.product\_id

GROUP BY

p.product\_category,

DATEPART(MONTH, o.order\_purchase\_timestamp)

)

-- Final SELECT: Pivot the data to get Months as Columns

-- Note: SQL Server (SSMS) requires hardcoding the month numbers for the pivot columns.

SELECT

product\_category,

-- Pivot the Monthly\_Revenue into 12 columns

SUM(CASE WHEN Sale\_Month = 1 THEN Monthly\_Revenue ELSE 0 END) AS Month\_01\_Revenue,

SUM(CASE WHEN Sale\_Month = 2 THEN Monthly\_Revenue ELSE 0 END) AS Month\_02\_Revenue,

SUM(CASE WHEN Sale\_Month = 3 THEN Monthly\_Revenue ELSE 0 END) AS Month\_03\_Revenue,

SUM(CASE WHEN Sale\_Month = 4 THEN Monthly\_Revenue ELSE 0 END) AS Month\_04\_Revenue,

SUM(CASE WHEN Sale\_Month = 5 THEN Monthly\_Revenue ELSE 0 END) AS Month\_05\_Revenue,

SUM(CASE WHEN Sale\_Month = 6 THEN Monthly\_Revenue ELSE 0 END) AS Month\_06\_Revenue,

SUM(CASE WHEN Sale\_Month = 7 THEN Monthly\_Revenue ELSE 0 END) AS Month\_07\_Revenue,

SUM(CASE WHEN Sale\_Month = 8 THEN Monthly\_Revenue ELSE 0 END) AS Month\_08\_Revenue,

SUM(CASE WHEN Sale\_Month = 9 THEN Monthly\_Revenue ELSE 0 END) AS Month\_09\_Revenue,

SUM(CASE WHEN Sale\_Month = 10 THEN Monthly\_Revenue ELSE 0 END) AS Month\_10\_Revenue,

SUM(CASE WHEN Sale\_Month = 11 THEN Monthly\_Revenue ELSE 0 END) AS Month\_11\_Revenue,

SUM(CASE WHEN Sale\_Month = 12 THEN Monthly\_Revenue ELSE 0 END) AS Month\_12\_Revenue

FROM

MonthlyCategorySales

GROUP BY

product\_category

ORDER BY

product\_category;

**SQL EXPLANATION**:

Clause: Explanation

WITH MonthlyCategorySales AS (...): Defines a CTE to calculate total monthly revenue for each category.

DATEPART(MONTH, order\_purchase\_timestamp): Extracts month as a number for grouping.

SUM(oi.price + oi.freight\_value): Aggregates total monthly revenue.

SUM(CASE WHEN Sale\_Month = N THEN Monthly\_Revenue ELSE 0 END): Converts month values into columns to form a pivoted table.

GROUP BY product\_category: Ensures one summarized row per category.

ORDER BY product\_category: Orders the final output alphabetically by category.

**PYTHON EQUIVALENT (PANDAS)**:

orders\_df = pd.read\_sql("SELECT \* FROM orders", conn)

order\_items\_df = pd.read\_sql("SELECT \* FROM order\_items", conn)

products\_df = pd.read\_sql("SELECT \* FROM products", conn)

# --- 1. Data Loading and Preparation (Equivalent to SQL Joins) ---

# Ensure timestamp is datetime and calculate total revenue

orders\_df['order\_purchase\_timestamp'] = pd.to\_datetime(orders\_df['order\_purchase\_timestamp'])

order\_items\_df['Total\_Revenue'] = order\_items\_df['price'] + order\_items\_df['freight\_value']

# Join the necessary tables: orders, order\_items, and products

sales\_detail\_df = pd.merge(orders\_df[['order\_id', 'order\_purchase\_timestamp']],

order\_items\_df[['order\_id', 'product\_id', 'Total\_Revenue']],

on='order\_id', how='inner')

sales\_detail\_df = pd.merge(sales\_detail\_df,

products\_df[['product\_id', 'product\_category']],

on='product\_id', how='inner')

# Extract Month (as an integer)

sales\_detail\_df['Sale\_Month'] = sales\_detail\_df['order\_purchase\_timestamp'].dt.month

# --- 2. Aggregation and Pivoting (Equivalent to SQL CTE and Pivot) ---

# Aggregate total revenue by category and month

monthly\_category\_sales = sales\_detail\_df.groupby(['product\_category', 'Sale\_Month']).agg(

Monthly\_Revenue=('Total\_Revenue', 'sum')

).reset\_index()

# Pivot the table to get months as columns (This creates the matrix for the heatmap)

# The index will be 'product\_category', columns will be 'Sale\_Month' (1 to 12), and values will be 'Monthly\_Revenue'

revenue\_matrix = monthly\_category\_sales.pivot(

index='product\_category',

columns='Sale\_Month',

values='Monthly\_Revenue'

)

# Replace any missing months (NaN) with 0 for the visualization

revenue\_matrix = revenue\_matrix.fillna(0)

print("\n--- Revenue Matrix Head ---")

print(revenue\_matrix.head(10))

**SUMMARY**:

This query produces a pivoted heatmap-style dataset showing each product category’s monthly revenue, ideal for visual dashboards or Seaborn heatmaps. It highlights seasonal performance peaks and product demand trends across the year.