Rutgers University

Brazilian E-Commerce Public Dataset

by Olist

*A Study on 100,000 Orders with product, customer and reviews info*

Vishal Pawar, Tiffany Fahmy, Akshita Agrawal, Chakradhar Baswaraj, Rushabh Udani

Business Data Management - Final Project

*Raw Data to SQL database for Informed Managerial Decisions*

Professor [MENG QU](https://rutgers.instructure.com/courses/250094/users/18905)

12/20/2023

Submissions:

PDF file containing the group's accountability contract, ER diagram, all SQL codes, and a 15-minute recording of the presentation. A PowerPoint presentation (optional) can also be included.

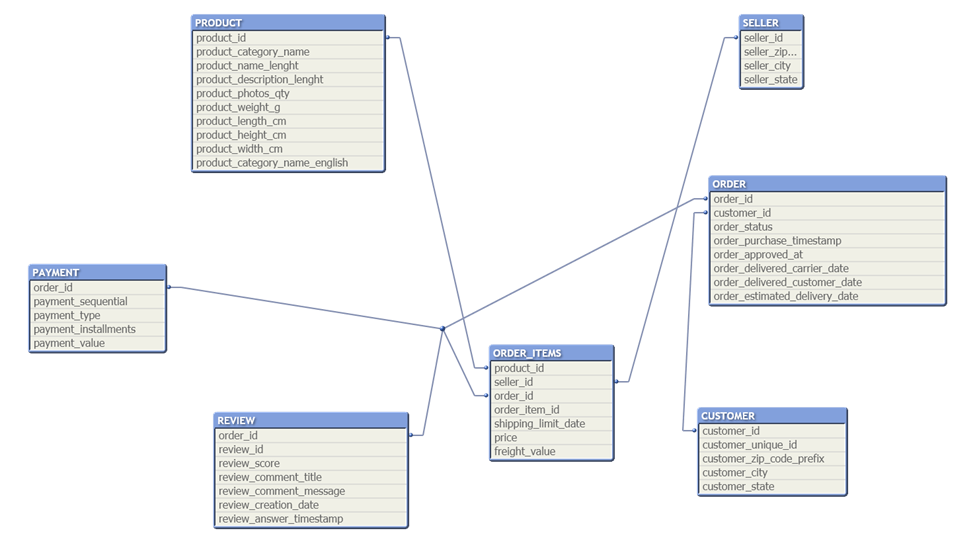
**BDM Final Project (Fall 2023)**

**Tasks and Accountability Contract:**

| **Work Breakdown** | Tiffany Fahmy | Vishal Pawar | Akshita Agrawal | Chakradhar Baswaraj | Rushabh Udani |
| --- | --- | --- | --- | --- | --- |
| Search and create datasets |  | ✔ |  |  | ✔ |
| ER Diagram |  | ✔ | ✔ | ✔ |  |
| Identify Tables and define attributes | ✔ | ✔ | ✔ |  |  |
| Design primary and foreign keys for all tables to ensure data integrity and consistency. | ✔ | ✔ |  |  |  |
| Define the relationships |  | ✔ |  |  |  |
| Provide a many-to-many relationship in the original dataset and break it by adding a bridge entity | ✔ |  |  |  |  |
| Create SQL code and import datasets to database |  | ✔ |  | ✔ |  |
| Dataset cleaning with SQL code |  | ✔ |  |  | ✔ |
| Design your own SQL question and write SQL queries | ✔ | ✔ |  |  |  |
| Presentation | ✔ | ✔ | ✔ | ✔ | ✔ |
| Visualizations (Optional) | ✔ |  |  |  |  |

**ER diagram:**

1. Drew the ER diagram to visually represent database structure.



1. List all tables and their attributes' names.

Sellers Dataset

* **seller\_id**: Unique identifier for the seller.
* **seller\_zip\_code\_prefix**: Zip code for the seller's location.
* **seller\_city**: City where the seller is located.
* **seller\_state**: State where the seller is located.

Product Category Name Translation Dataset

* **product\_category\_name**: The original name of the product category in Portuguese.
* **product\_category\_name\_english**: The English translation of the product category name.

Orders Dataset

* **order\_id**: Unique identifier for the order.
* **customer\_id**: Unique identifier for the customer.
* **order\_status**: Status of the order.
* **order\_purchase\_timestamp**: Timestamp of when the purchase was made.
* **order\_approved\_at**: Timestamp of when the order was approved.
* **order\_delivered\_carrier\_date**: Timestamp of when the order was handed off to the carrier.
* **order\_delivered\_customer\_date**: Timestamp of when the order was delivered to the customer.
* **order\_estimated\_delivery\_date**: Estimated delivery date of the order.

Order Items Dataset

* **order\_id**: Unique identifier for the order.
* **order\_item\_id**: Sequential number identifying the number of items in the same order.
* **product\_id**: Unique identifier for the product.
* **seller\_id**: Unique identifier for the seller.
* **shipping\_limit\_date**: The seller's deadline to handle the order over to the logistic partner.
* **price**: The item's price.
* **freight\_value**: The item's freight value in Brazilian Reals.

Customers Dataset

* **customer\_id**: Unique identifier for the customer.
* **customer\_unique\_id**: Unique identifier for each customer across all orders.
* **customer\_zip\_code\_prefix**: Zip code for the customer's location.
* **customer\_city**: City where the customer is located.
* **customer\_state**: State where the customer is located.

Geolocation Dataset

* **geolocation\_zip\_code\_prefix**: Zip code for the location.
* **geolocation\_lat**: Latitude of the location.
* **geolocation\_lng**: Longitude of the location.
* **geolocation\_city**: City of the location.
* **geolocation\_state**: State of the location.

Order Payments Dataset

* **order\_id**: Unique identifier for the order.
* **payment\_sequential**: A numerical identifier for each payment per order.
* **payment\_type**: Method of payment.
* **payment\_installments**: Number of installments of the payment.
* **payment\_value**: Transaction value.

Order Reviews Dataset

* **review\_id**: Unique identifier for the review.
* **order\_id**: Unique identifier for the order.
* **review\_score**: Score given by the customer.
* **review\_comment\_title**: Optional comment title.
* **review\_comment\_message**: Optional comment message.
* **review\_creation\_date**: Date of review creation.
* **review\_answer\_timestamp**: Timestamp of the review reply.

Products Dataset

* **product\_id**: Unique identifier for the product.
* **product\_category\_name**: Category name in Portuguese.
* **product\_name\_lenght**: Length of the product name.
* **product\_description\_lenght**: Length of the product description.
* **product\_photos\_qty**: Quantity of product photos available.
* **product\_weight\_g**: Product weight in grams.
* **product\_length\_cm**: Product length in centimeters.
* **product\_height\_cm**: Product height in centimeters.
* **product\_width\_cm**: Product width in centimeters.

1. Design primary keys and foreign keys for all tables to ensure data integrity and consistency.

**Customers**:

* Primary Key:
  + customer\_id
* Foreign Key:
  + customer\_unique\_id (to link multiple orders to the same customer if needed)
* Attributes:
  + customer\_zip\_code\_prefix, customer\_city, customer\_state

**Orders**:

* Primary Key:
  + order\_id
* Foreign Key:
  + customer\_id (links to Customers)
* Attributes:
  + order\_status
  + order\_purchase\_timestamp
  + order\_approved\_at
  + order\_delivered\_carrier\_date
  + order\_delivered\_customer\_date
  + order\_estimated\_delivery\_date

**Order Items**:

* Composite Primary Key:
  + order\_id
  + order\_item\_id
* Foreign Keys:
  + order\_id (links to Orders),
  + product\_id (links to Products),
  + seller\_id (links to Sellers)
* Attributes:
  + shipping\_limit\_date, price
  + freight\_value

**Products**:

* Primary Key:
  + product\_id
* Foreign Key:
  + product\_category\_name (links to Product Category)
* Attributes:
  + product\_name\_length
  + product\_description\_length
  + product\_photos\_qty
  + product\_weight\_g
  + product\_length\_cm
  + product\_height\_cm
  + product\_width\_cm

**Sellers**:

* Primary Key:
  + seller\_id
* Attributes:
  + seller\_zip\_code\_prefix
  + seller\_city
  + seller\_state

**Order Payments**:

* Composite Primary Key:
  + order\_id
  + payment\_sequential
* Foreign Key:
  + order\_id (links to Orders)
* Attributes:
  + payment\_type
  + payment\_installments
  + payment\_value

**Order Reviews**:

* Primary Key:
  + review\_id
* Foreign Key:
  + order\_id (links to Orders)
* Attributes:
  + review\_score
  + review\_comment\_title
  + review\_comment\_message
  + review\_creation\_date
  + review\_answer\_timestamp

**Product Category**:

* Primary Key:
  + product\_category\_name
* Attributes:
  + None additional in the dataset provided

**Product Category Name Translation**:

* Primary Key:
  + product\_category\_name
* Attributes:
  + product\_category\_name\_english

**Geolocation**:

* Primary Key:
  + geolocation\_zip\_code\_prefix
* Attributes:
  + geolocation\_lat
  + geolocation\_lng
  + geolocation\_city
  + geolocation\_state
* (This table does not have a direct foreign key but can be linked to Customers and Sellers via the zip\_code\_prefix)

1. Show all relationship types in your ER diagram, including one-to-one, one-to-many, and many-to-many relationships.

* **One-to-One**:
  + Each entry in the Product Category Name Translation corresponds to exactly one entry in the Product Category.
* **One-to-Many**:
  + **Customers to Orders**: A single Customer can place multiple Orders, but each Order is associated with only one Customer.
  + **Orders to Order Items**: An Order can contain multiple Order Items, but each Order Item belongs to one Order only.
  + **Orders to Order Payments**: One Order can have multiple Payments (split payments), but each Payment entry pertains to one Order.
  + **Orders to Order Reviews**: An Order can have multiple Reviews, but each Review is associated with one Order.
  + **Products to Order Items**: A Product can be part of multiple Order Items, but each Order Item refers to one Product.
  + **Sellers to Order Items**: A Seller can sell multiple Order Items, but one Seller supplies each Order Item.
  + **Product Category to Products**: Each Product Category can include multiple Products, but each Product is categorized into one Product Category only.
* **Many-to-Many**:
  + **Products to Orders through Order Items**: A Product can be part of many Orders, and an Order can contain many Products. This many-to-many relationship is resolved through the Order Items table.
  + **Sellers to Orders through Order Items**: A Seller can fulfill many Orders, and an Order can contain products from many Sellers. This relationship is also resolved through the Order Items table.

1. Provide a many-to-many relationship in your original dataset and break it by adding a bridge entity.

To resolve the many-to-many relationships, we use the Order Items table as a junction table (also known as a bridge entity), which contains foreign keys to both Orders and Products, as well as Orders and Sellers.

* A many-to-many relationship exists between the Products and Orders tables in the datasets provided. This relationship is because:
  + A single product can be part of multiple orders (when different customers order the same product).
  + A single order can contain multiple products (when a customer orders several items).

This many-to-many relationship is managed by the Order Items table, which acts as a bridge entity (also known as an associative or junction table). The Order Items table contains foreign keys referencing the primary keys of the Products and Orders tables.

Here is how the Order Items bridge entity breaks the many-to-many relationship:

Order Items (Bridge Entity):

* Composite Primary Key:
  + order\_id
  + order\_item\_id (unique for each item in an order)
* Foreign Keys:
  + order\_id (references Orders.order\_id)
  + product\_id (references Products.product\_id)
  + seller\_id (references Sellers.seller\_id)
* Attributes:
  + shipping\_limit\_date
  + price
  + freight\_value

The Order Items table allows us to track each instance of an ordered product without duplicating information in the Products or Orders tables.

Each entry in Order Items represents a unique occurrence of a product included in an order, along with the quantity ordered, the price at the time, the seller, and other relevant details.

This bridge entity resolves the many-to-many relationship by creating two one-to-many relationships:

* One-to-Many: Products to Order Items
* One-to-Many: Orders to Order Items

With this structure, we can query the database for complex questions like "Which products are most frequently sold together?" or "What is the most common combination of products in orders?" by joining these tables on their foreign keys.

**SQL code:**

1. Create SQL code to import your data, including creating tables, designing primary keys and foreign keys, and setting up NOT NULL and unique constraints for primary keys.

[INSERT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/insert.html) INTO `product\_category\_name\_translation` (`product\_category\_name`, `product\_category\_name\_english`)

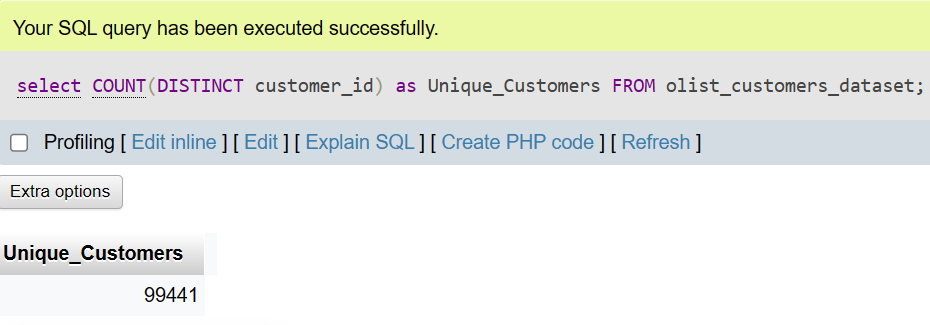
1. Ask and answer easy SQL questions using SQL code to demonstrate your understanding of basic SQL syntax and functions. For example, you could ask, "How many companies are in my dataset?" and answer it using SQL code such as "SELECT COUNT(\*) FROM company".

Here are a few basic SQL queries:

* + **Question**: How many unique customers are there in the dataset?
  + **SQL Code**:

SELECT COUNT(DISTINCT customer\_id)

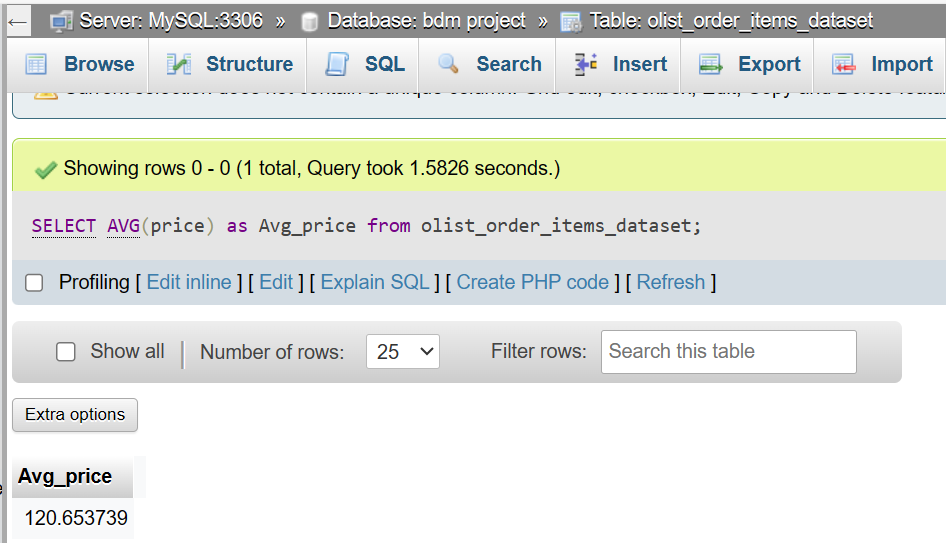
FROM Customers;



* + **Question**: What is the average price of products ordered?
  + **SQL Code**:

SELECT AVG(price)

FROM Order\_Items;



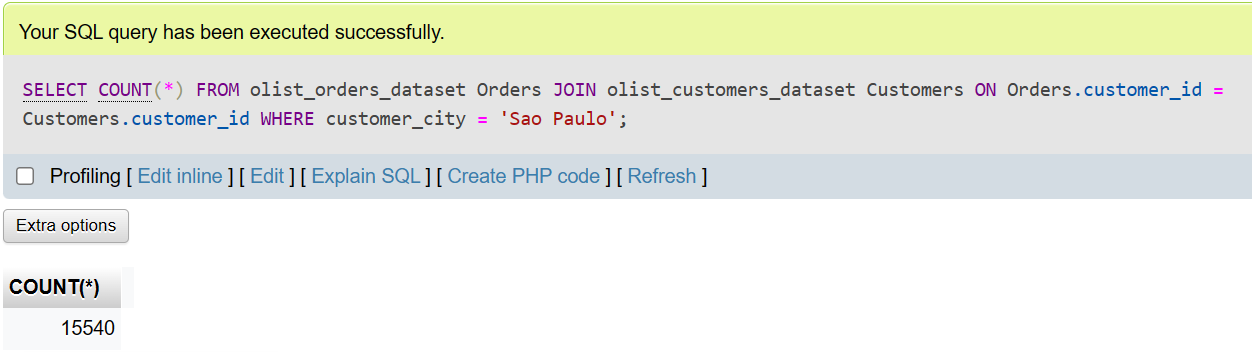
* + **Question**: How many orders were made in the city of 'Sao Paulo'?
  + **SQL Code**:

SELECT COUNT(\*) FROM Orders

JOIN Customers

ON Orders.customer\_id = Customers.customer\_id

WHERE customer\_city = 'Sao Paulo';



* + **Question**: Which product category has the highest number of products?
  + **SQL Code**:

SELECT product\_category\_name,

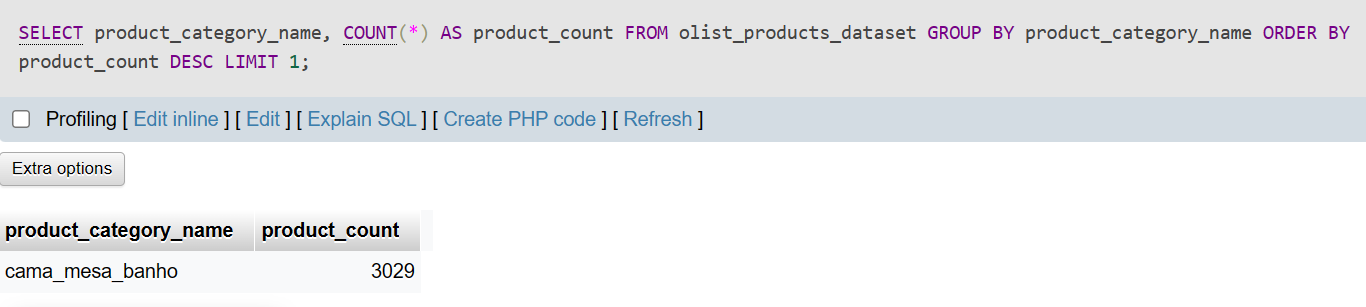
COUNT(\*) AS product\_count

FROM Products

GROUP BY product\_category\_name

ORDER BY product\_count

DESC FETCH FIRST ROW ONLY;

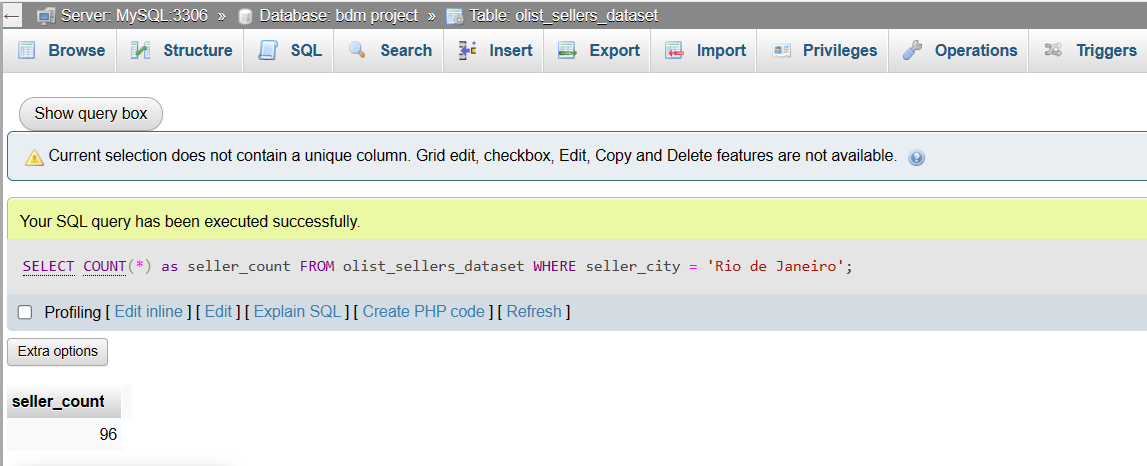




* + **Question**: How many sellers are from the state of 'Rio de Janeiro'?
  + **SQL Code**:

SELECT COUNT(\*) FROM Sellers

WHERE seller\_state = 'Rio de Janeiro';



1. Ask and answer complex SQL questions to earn more points. You can design easy questions with easy SQL codes first (such as SELECT, WHERE, DISTINCT, ORDER BY, CONCAT, LEN, LEFT, Wildcard, INNER SELECT, IS NULL, IS NOT NULL, INNER JOIN, LEFT JOIN, RIGHT JOIN), and then move on to questions that require aggregate functions (such as SUM, COUNT, MAX, MIN, AVG, GROUP BY, HAVING), date and time functions (such as MONTH, YEAR, DATE, ADDDATE, CURRENTDATE, SYSDATE), and complex SQL code (such as SELF JOIN).

SQL queries with more complexity:

* + **Basic SQL Functions**:
    1. **Question**: List the first 5 products with their prices in ascending order.
    2. **SQL Code**:

select distinct x.product\_category\_name,y.product\_category\_name\_english, x.product\_id, x.price from

(SELECT p.product\_category\_name, o.product\_id, o.price

FROM

olist\_order\_items\_dataset o

join

olist\_products\_dataset p

on o.product\_id=p.product\_id

) x join product\_category\_name\_translation y

on x.product\_category\_name=y.product\_category\_name

ORDER BY x.price ASC

limit 5;



* + **Aggregate Functions**:
    1. **Question**: What is the total revenue per seller city and number of sellers in the city to observe market?
    2. **SQL Code**:

select x.seller\_city,sum(total\_revenue) as total\_revenue,count(seller\_id) as total\_sellers from

(SELECT o.seller\_id, s.seller\_city,

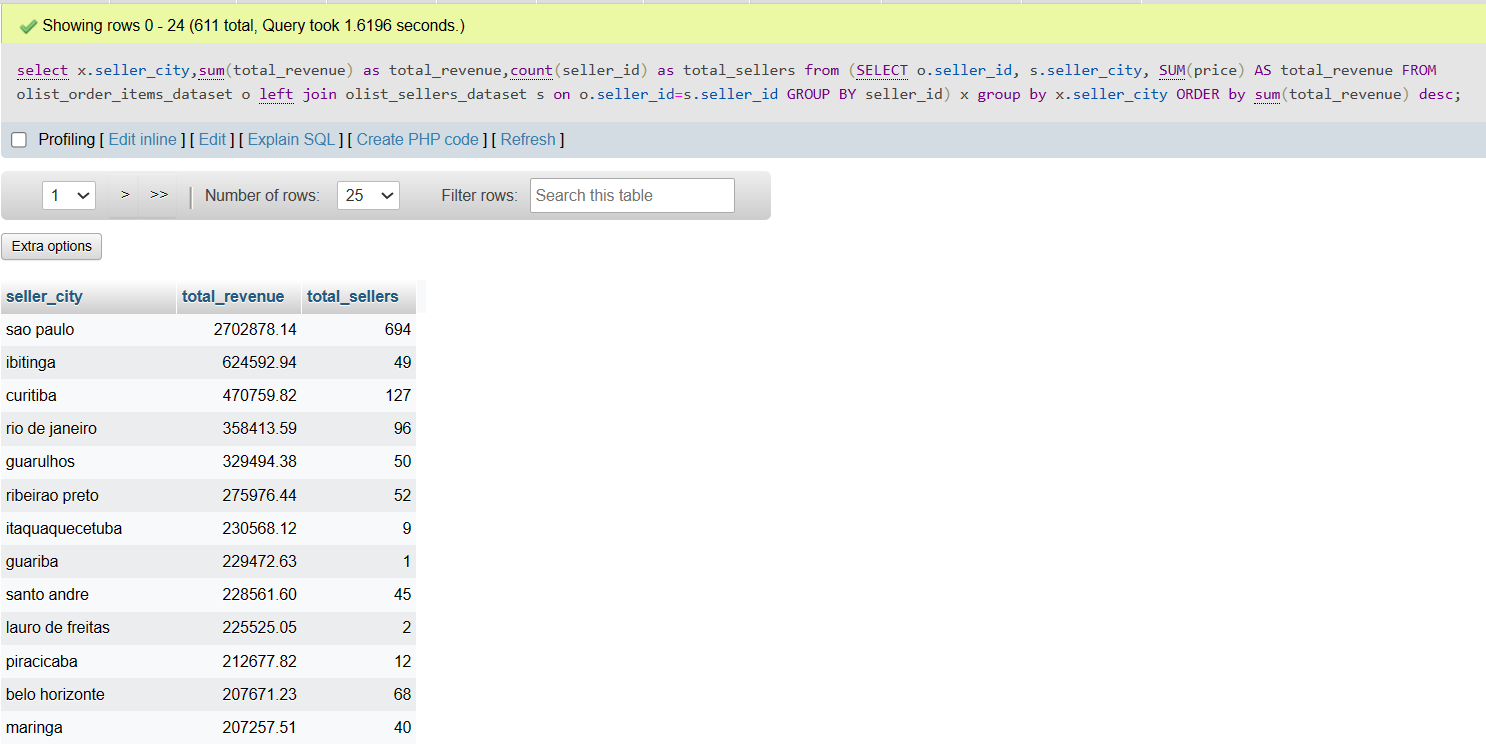
SUM(price) AS total\_revenue

FROM olist\_order\_items\_dataset o left join olist\_sellers\_dataset s on o.seller\_id=s.seller\_id

GROUP BY seller\_id) x

group by x.seller\_city

ORDER by sum(total\_revenue) desc;



* + **Date Functions**:
    1. **Question**: How many orders were placed in each month of 2018 per month?
    2. **SQL Code**:

SELECT MONTH(order\_purchase\_timestamp) AS month,

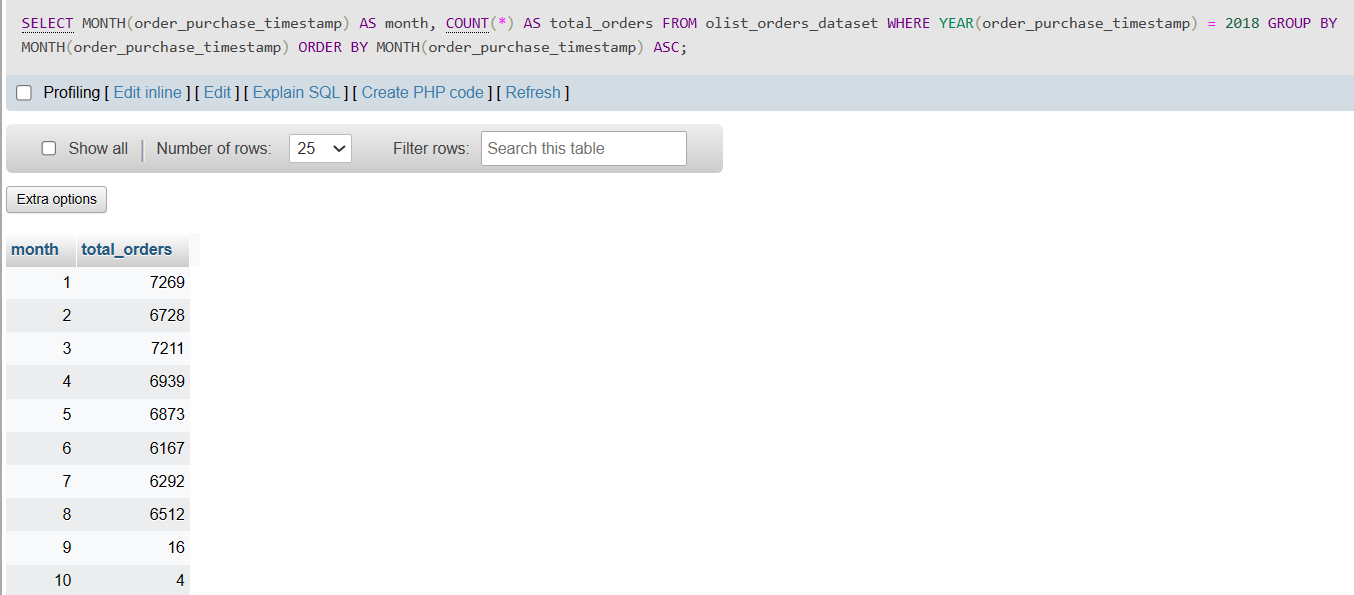
COUNT(\*) AS total\_orders

FROM olist\_orders\_dataset

WHERE YEAR(order\_purchase\_timestamp) = 2018

GROUP BY MONTH(order\_purchase\_timestamp)

ORDER BY MONTH(order\_purchase\_timestamp) ASC;



* + **Complex Query with Joins and Subqueries**:
    1. **Question**: Find the names of the top 5 cities in terms of number of orders.
    2. **SQL Code**:

SELECT customer\_city,

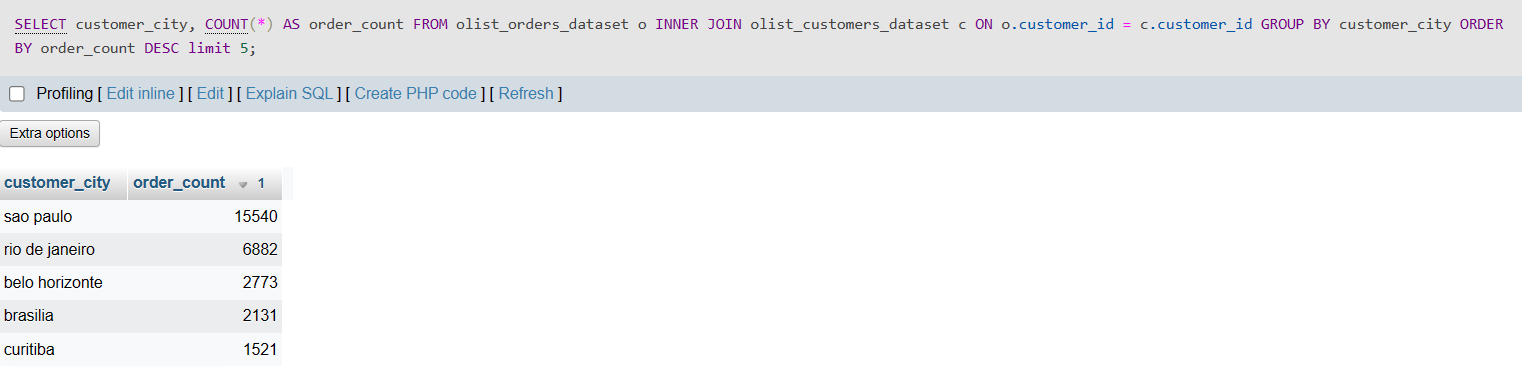
COUNT(\*) AS order\_count FROM Orders

INNER JOIN Customers ON Orders.customer\_id = Customers.customer\_id

GROUP BY customer\_city

ORDER BY order\_count

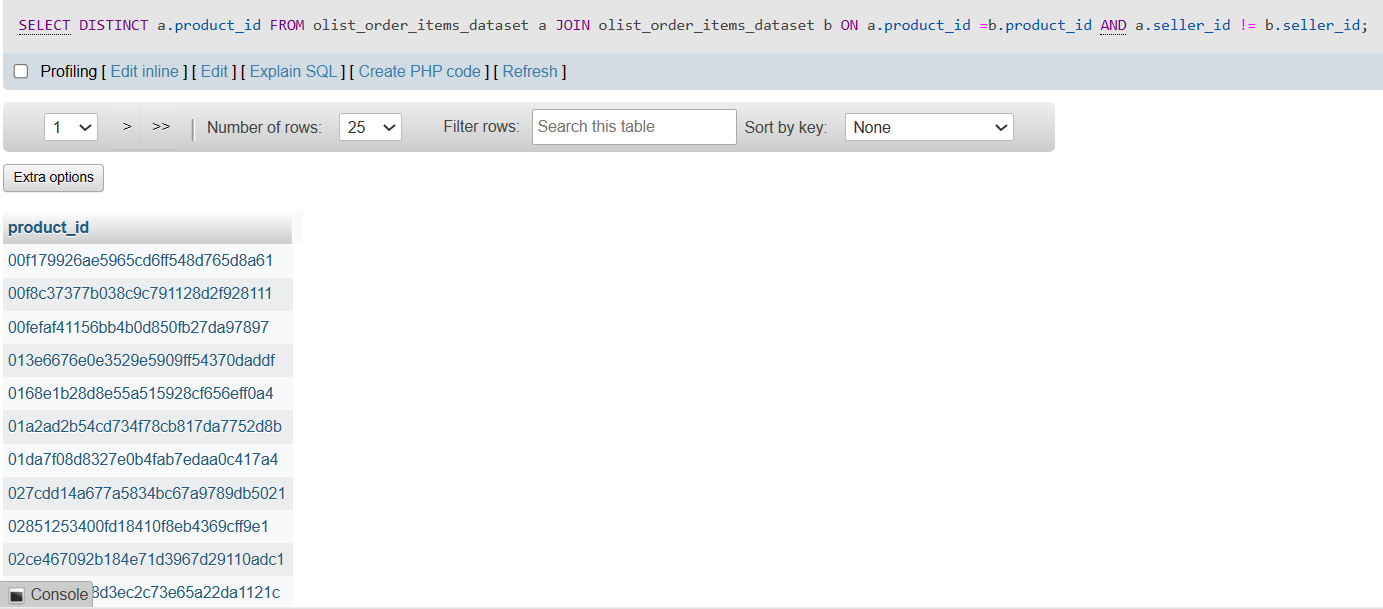
DESC FETCH FIRST 5 ROWS ONLY;



* + **Complex Query with Self Join**:
    1. **Question**: Identify products that more than one seller sold.
    2. **SQL Code**:

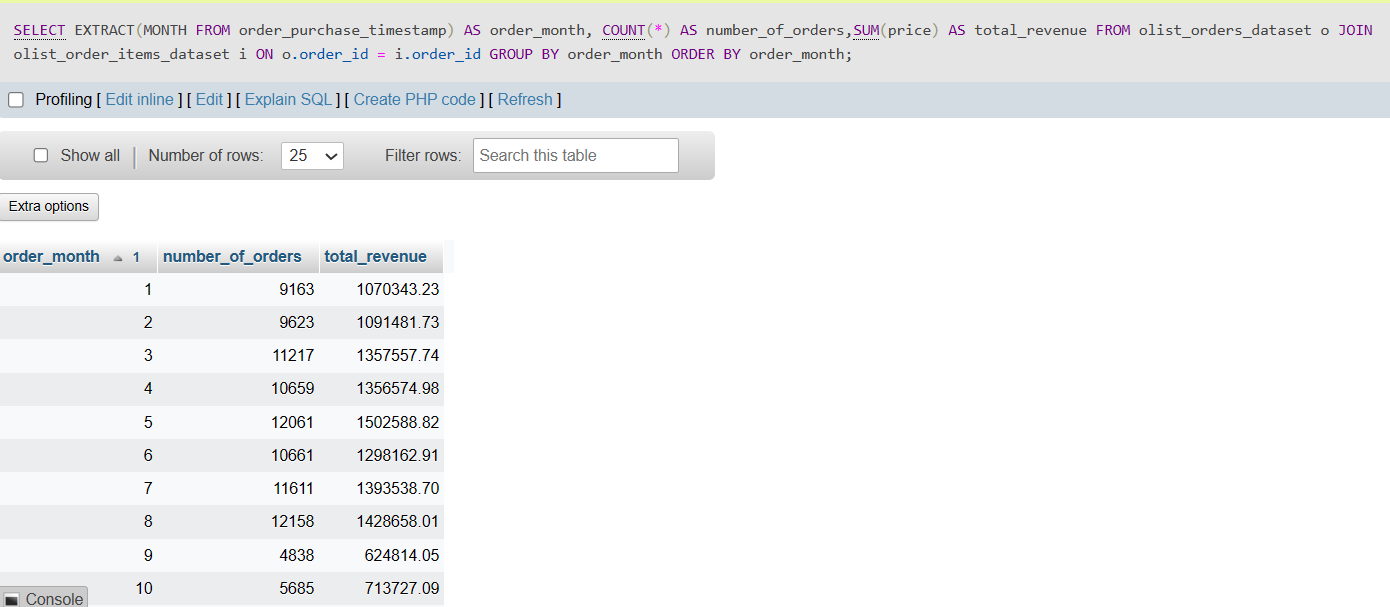
SELECT DISTINCT a.product\_id FROM olist\_order\_items\_dataset a

JOIN olist\_order\_items\_dataset b ON a.product\_id =b.product\_id AND a.seller\_id != b.seller\_id;



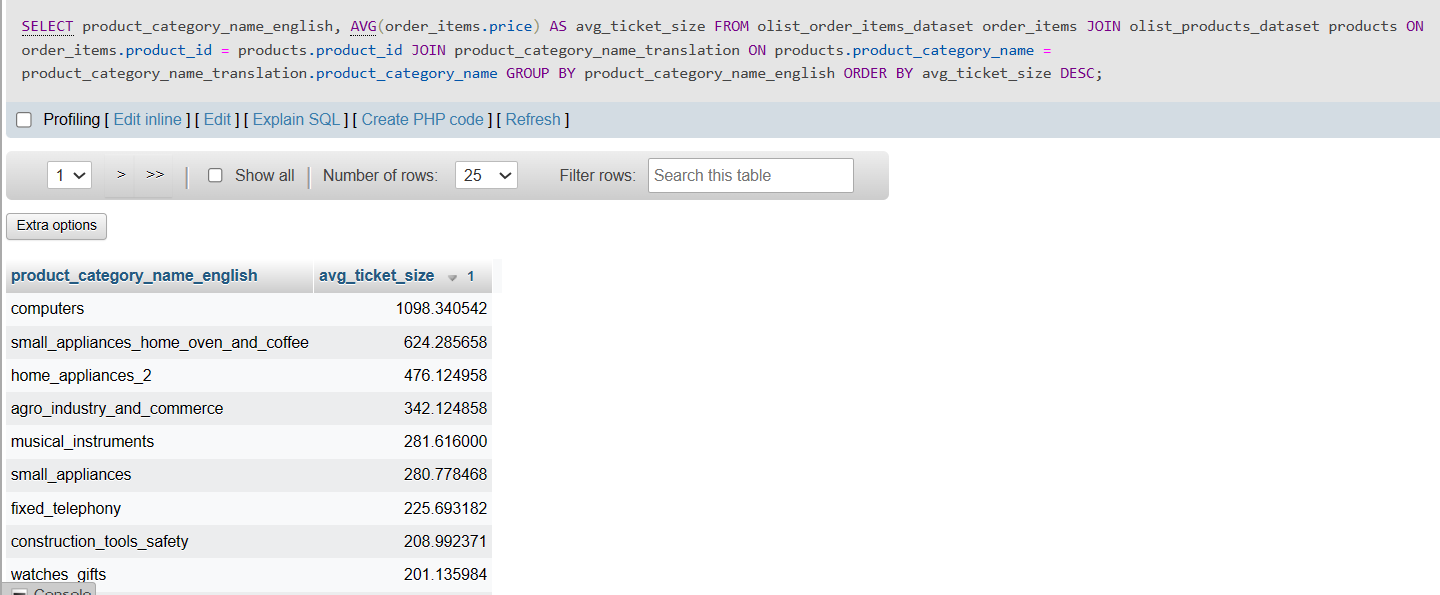
* + **Complex Query**:
    1. **Question**: Find the number of orders and total revenue for each month
    2. **SQL Code**:

SELECT EXTRACT(MONTH FROM order\_purchase\_timestamp) AS order\_month, COUNT(\*) AS number\_of\_orders,SUM(price) AS total\_revenue FROM olist\_orders\_dataset o JOIN olist\_order\_items\_dataset i ON o.order\_id = i.order\_id GROUP BY order\_month ORDER BY order\_month;



* + **Complex Query**:
    1. **Question**: Determine the average ticket size per category.
    2. **SQL Code**:

SELECT product\_category\_name\_english, AVG(order\_items.price) AS avg\_ticket\_size FROM olist\_order\_items\_dataset order\_items JOIN olist\_products\_dataset products ON order\_items.product\_id = products.product\_id JOIN product\_category\_name\_translation ON products.product\_category\_name = product\_category\_name\_translation.product\_category\_name GROUP BY product\_category\_name\_english ORDER BY avg\_ticket\_size DESC;

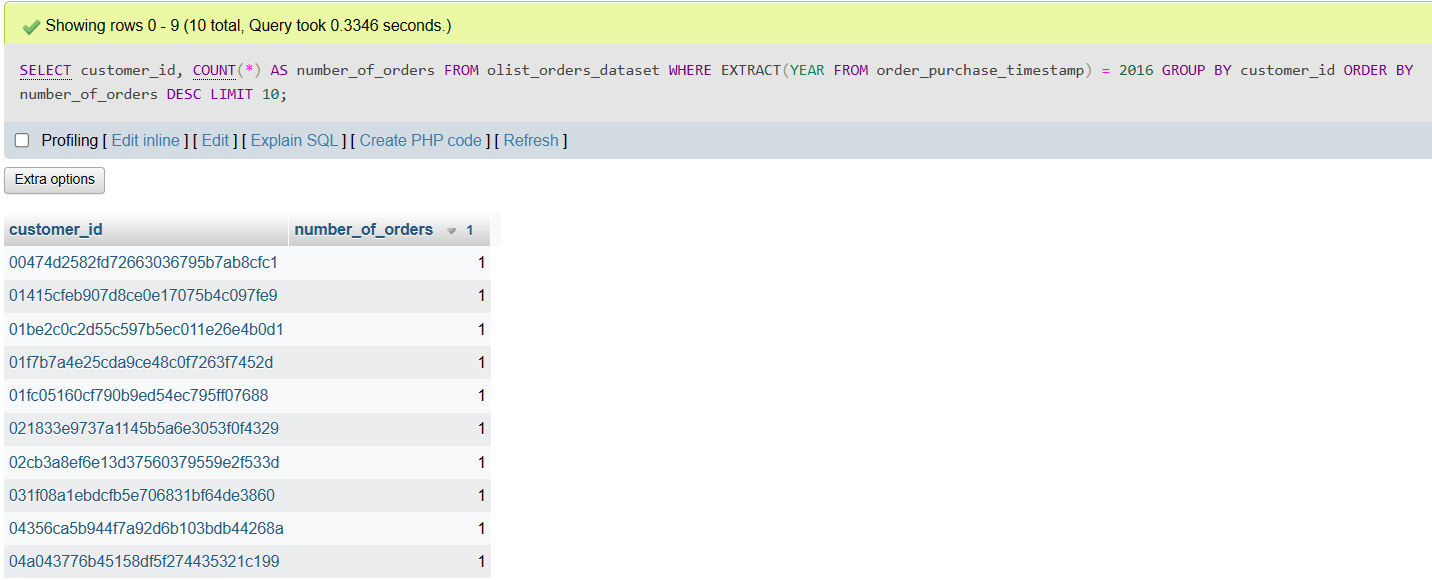


* + **Complex Query**:
    1. **Question**: Find the seller with the highest average review score.
    2. **SQL Code**:

SELECT product\_category\_name\_english, AVG(order\_items.price) AS avg\_ticket\_size FROM olist\_order\_items\_dataset order\_items JOIN olist\_products\_dataset products ON order\_items.product\_id = products.product\_id JOIN product\_category\_name\_translation ON products.product\_category\_name = product\_category\_name\_translation.product\_category\_name GROUP BY product\_category\_name\_english ORDER BY avg\_ticket\_size DESC;

* + **Complex Query**:
    1. **Question**: Identify customers who made the most orders in a given year.
    2. **SQL Code**:

SELECT customer\_id, COUNT(\*) AS number\_of\_orders FROM olist\_orders\_dataset WHERE EXTRACT(YEAR FROM order\_purchase\_timestamp) = 2016 GROUP BY customer\_id ORDER BY number\_of\_orders DESC LIMIT 10;



* + **Complex Query with CTE Functions**:
    1. **Question**: Analyze customer spending habits, seller performance, and product popularity all in one go. The final result set will provide a list of customers, how much they’ve spent, information about the sellers of the products they’ve bought (provided they are in the top 10), and details about the products they’ve bought if those products are the most popular in their category.
    2. **SQL Code**:

WITH CustomerSpending AS (

SELECT

o.customer\_id,

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

FROM olist\_orders\_dataset o

INNER JOIN olist\_order\_items\_dataset oi ON o.order\_id = oi.order\_id

GROUP BY o.customer\_id

),

TopSellers AS (

SELECT

oi.seller\_id,

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

RANK() OVER (ORDER BY COUNT(DISTINCT o.order\_id) DESC) AS seller\_rank

FROM olist\_orders\_dataset o

INNER JOIN olist\_order\_items\_dataset oi ON o.order\_id = oi.order\_id

GROUP BY oi.seller\_id

),

ProductPopularity AS (

SELECT

p.product\_id,

p.product\_category\_name,

COUNT(oi.order\_id) AS number\_of\_orders,

SUM(oi.price) AS total\_revenue,

DENSE\_RANK() OVER (PARTITION BY p.product\_category\_name ORDER BY COUNT(oi.order\_id) DESC) AS category\_rank

FROM olist\_products\_dataset p

INNER JOIN olist\_order\_items\_dataset oi ON p.product\_id = oi.product\_id

GROUP BY p.product\_id, p.product\_category\_name

)

SELECT

cs.customer\_id,

cs.total\_spending,

ts.seller\_id,

ts.total\_orders,

ts.seller\_rank,

pp.product\_id,

pp.product\_category\_name,

pp.number\_of\_orders,

pp.total\_revenue,

pp.category\_rank

FROM CustomerSpending cs

JOIN olist\_orders\_dataset o ON cs.customer\_id = o.customer\_id

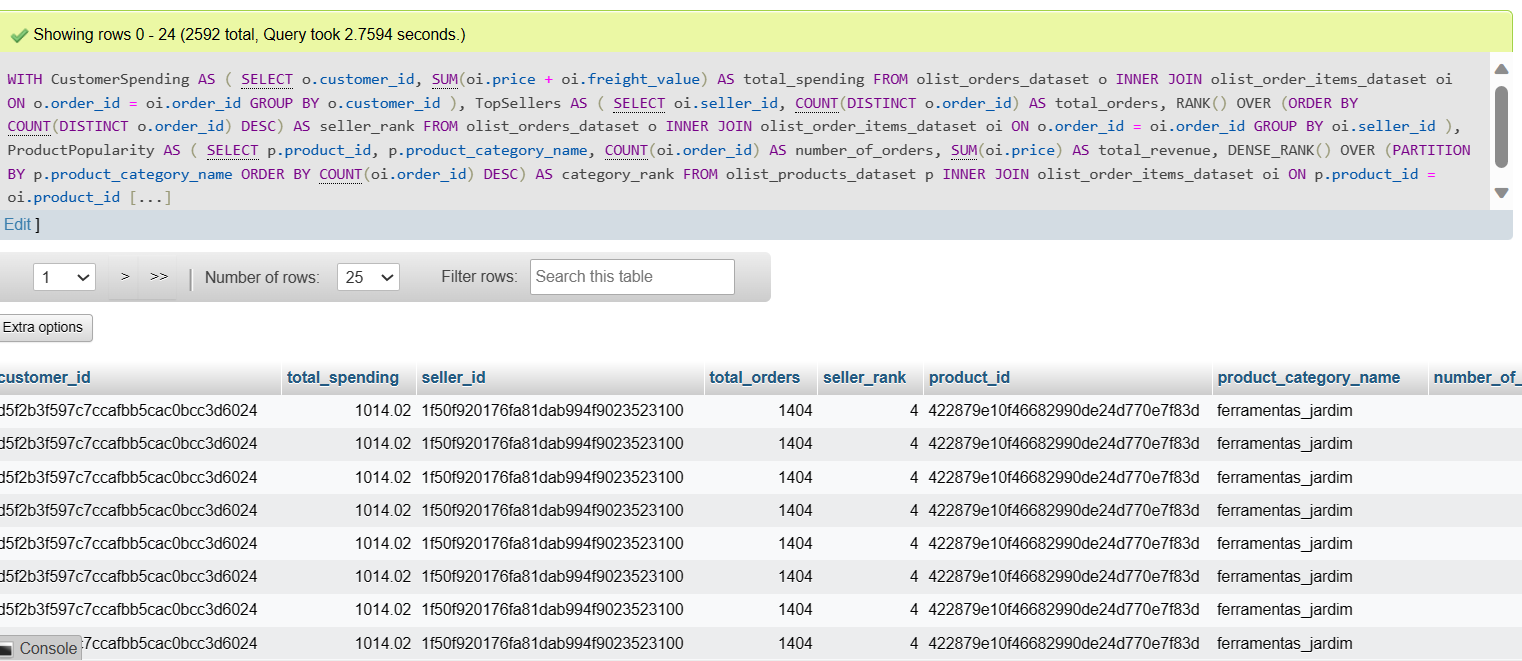
JOIN olist\_order\_items\_dataset oi ON o.order\_id = oi.order\_id

JOIN TopSellers ts ON oi.seller\_id = ts.seller\_id

JOIN ProductPopularity pp ON oi.product\_id = pp.product\_id

WHERE ts.seller\_rank <= 10 AND pp.category\_rank = 1

ORDER BY cs.total\_spending DESC, ts.seller\_rank, pp.category\_rank;



In this query, we are:

1. Calculating total customer spending in the CustomerSpending CTE.

2. Determining top sellers based on the total number of distinct orders in the TopSellers CTE and ranking them.

3. Analyzing product popularity within its category and ranking the products in the ProductPopularity CTE.

4. Bringing all this information together in the final SELECT statement to provide a comprehensive view of spending, seller performance, and product popularity.

5. Filtering to only show data for top 10 sellers and the most popular products in each category.

* + **Complex Query with CTE Functions**:
    1. **Question**: Determine the most common pairs of products sold together.
    2. **SQL Code**:

WITH PairedProducts AS (

SELECT

a.order\_id,

a.product\_id AS product\_a,

b.product\_id AS product\_b

FROM olist\_order\_items\_dataset a

INNER JOIN olist\_order\_items\_dataset b ON a.order\_id = b.order\_id AND a.product\_id < b.product\_id

),

ProductNames AS (

SELECT

pp.product\_a,

pp.product\_b,

pa.product\_category\_name AS product\_name\_a,

pb.product\_category\_name AS product\_name\_b

FROM PairedProducts pp

INNER JOIN olist\_products\_dataset pa ON pp.product\_a = pa.product\_id

INNER JOIN olist\_products\_dataset pb ON pp.product\_b = pb.product\_id

),

ProductPairsCount AS (

SELECT

product\_name\_a,

product\_name\_b,

COUNT(\*) AS times\_sold\_together

FROM ProductNames

GROUP BY product\_name\_a, product\_name\_b

)

SELECT

product\_name\_a,

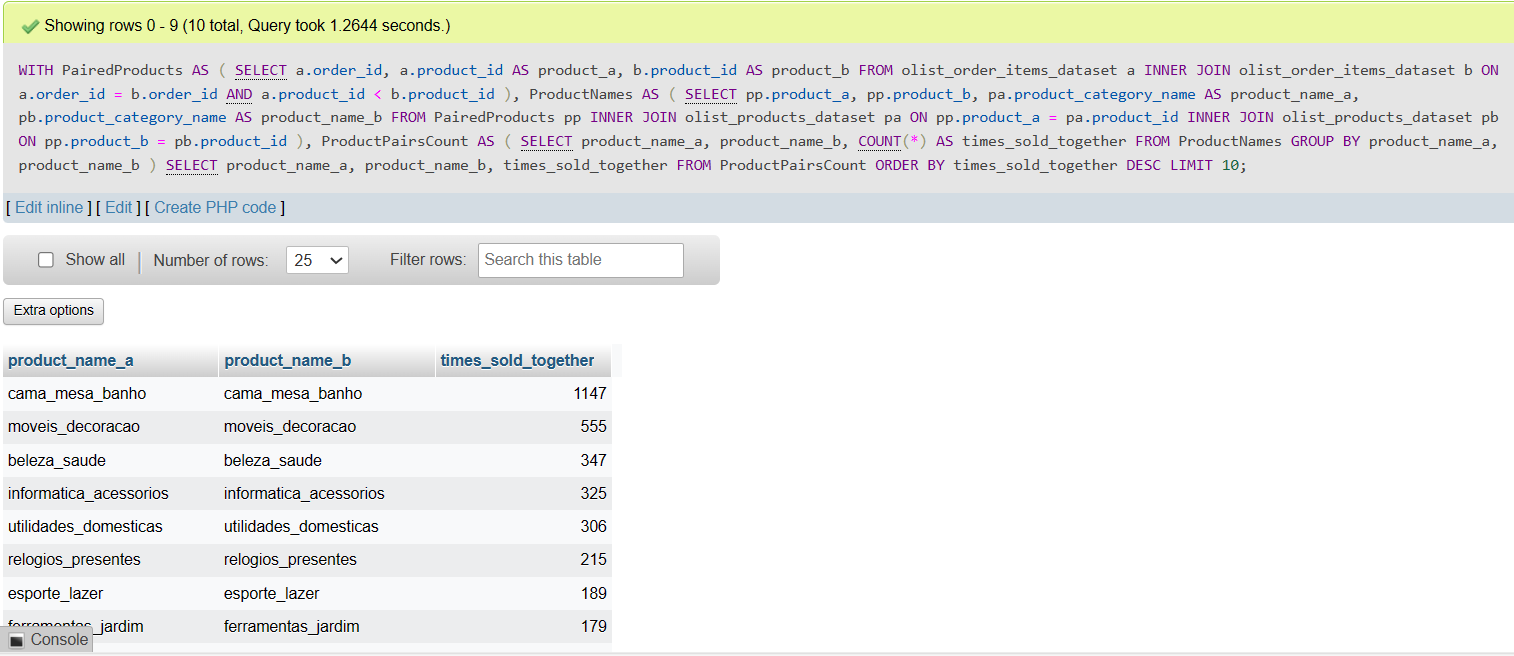
product\_name\_b,

times\_sold\_together

FROM ProductPairsCount

ORDER BY times\_sold\_together DESC

LIMIT 10;



In this query:

1. The PairedProducts CTE finds all possible pairs of products within each order. The a.product\_id < b.product\_id condition ensures each pair is unique and not duplicated with reverse order (e.g., (product1, product2) is considered the same as (product2, product1) and only the former is included).

2. The ProductNames CTE joins the paired products with the products table to get the product category names for each product in the pair.

3. The ProductPairsCount CTE counts how many times each pair of products is sold together.

4. The final SELECT statement retrieves the product names and the count of times they were sold together, ordered by the most frequently sold pairs first, limiting to the top 10 pairs.

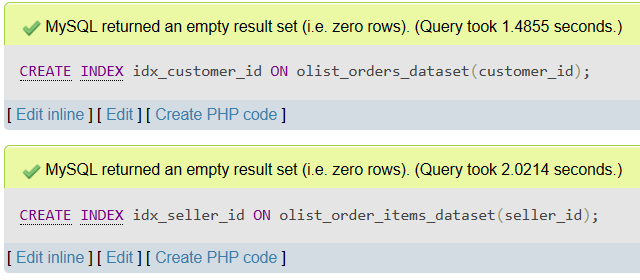
1. For advanced topics in database design (such as index, trigger, and constraints), you can choose to include them in your project if you wish to show your knowledge and skills. However, please note that this is not a requirement for the project.

SQL implementations for indexes, triggers, and constraints:

* + **Indexes:**
    1. Since customer\_id in Orders and seller\_id in Order\_Items are frequently used for lookups and joins, create indexes on these columns.
    2. **SQL Code**:

CREATE INDEX idx\_customer\_id ON Orders(customer\_id);

CREATE INDEX idx\_seller\_id ON Order\_Items(seller\_id);



* + **Triggers:**
    1. Create a trigger to automatically update the order\_status to 'Delivered' when the order\_delivered\_customer\_date is set.
    2. **SQL Code**:

CREATE OR REPLACE TRIGGER update\_order\_status

AFTER UPDATE OF order\_delivered\_customer\_date ON Orders

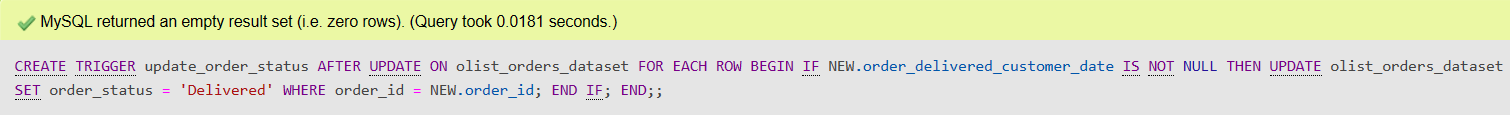
FOR EACH ROW

WHEN (NEW.order\_delivered\_customer\_date IS NOT NULL)

BEGIN

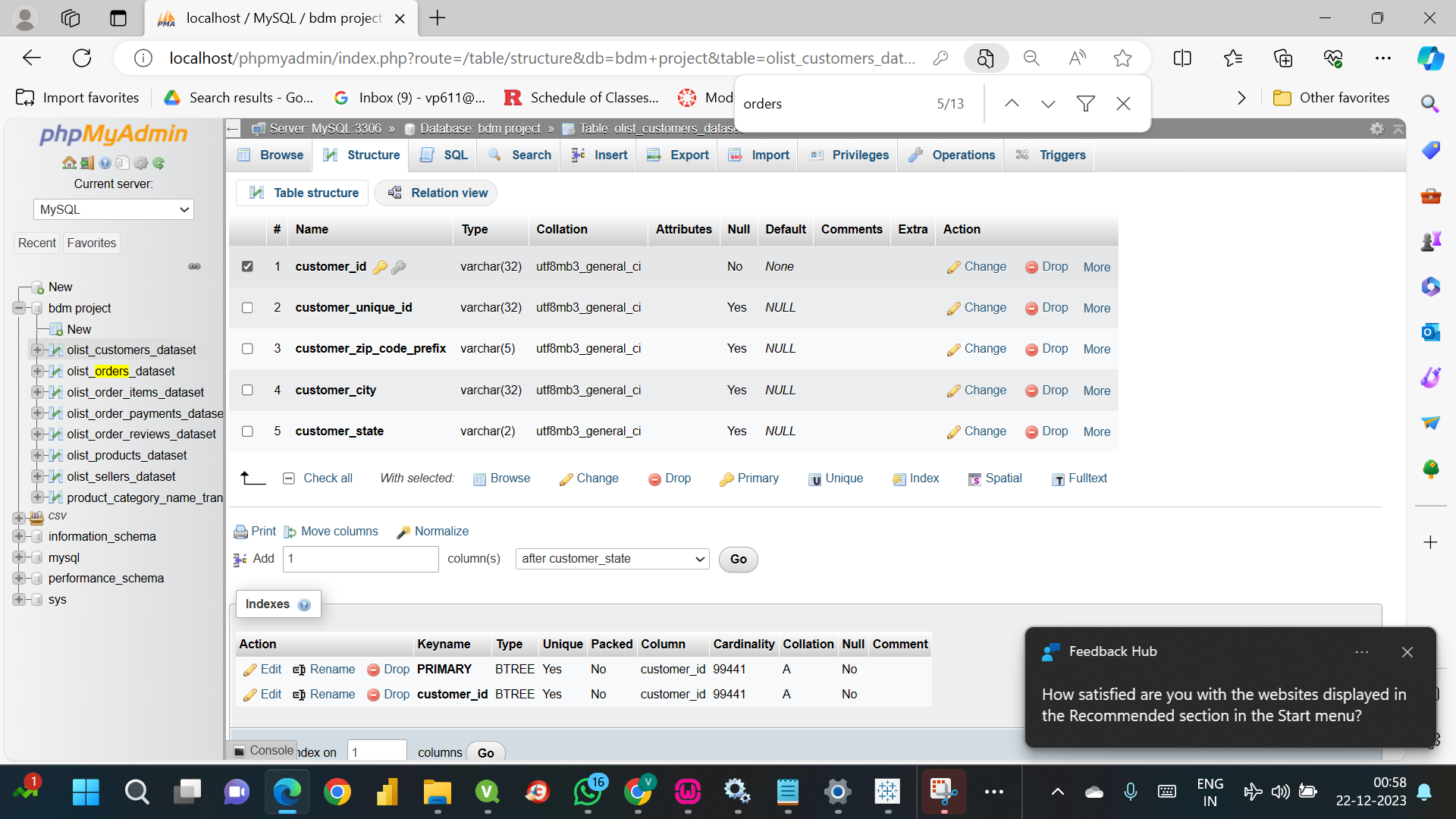
UPDATE Orders SET order\_status = 'Delivered' WHERE order\_id = :NEW.order\_id;

END;

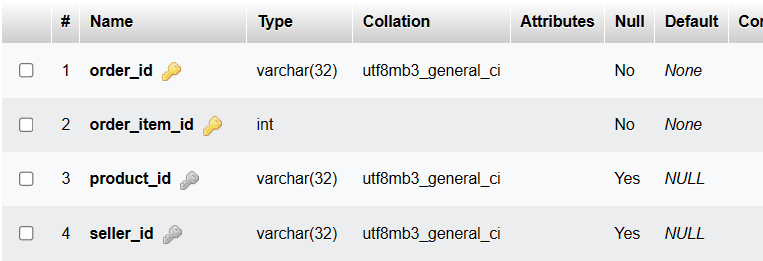


* + **Constraints:**
    1. Implement a UNIQUE constraint on customer\_unique\_id in the Customers table to prevent duplicate entries
    2. **SQL Code**:

ALTER TABLE Customers ADD CONSTRAINT unique\_customer\_unique\_id UNIQUE(customer\_unique\_id);

****

* + 1. Add NOT NULL constraints to critical fields such as order\_id and order\_item\_id in Orders and product\_id in Products.



* + 1. **SQL Code**:

ALTER TABLE olist\_orders\_dataset MODIFY order\_id VARCHAR(255) NOT NULL;

ALTER TABLE olist\_products\_dataset MODIFY product\_id VARCHAR(255) NOT NULL;

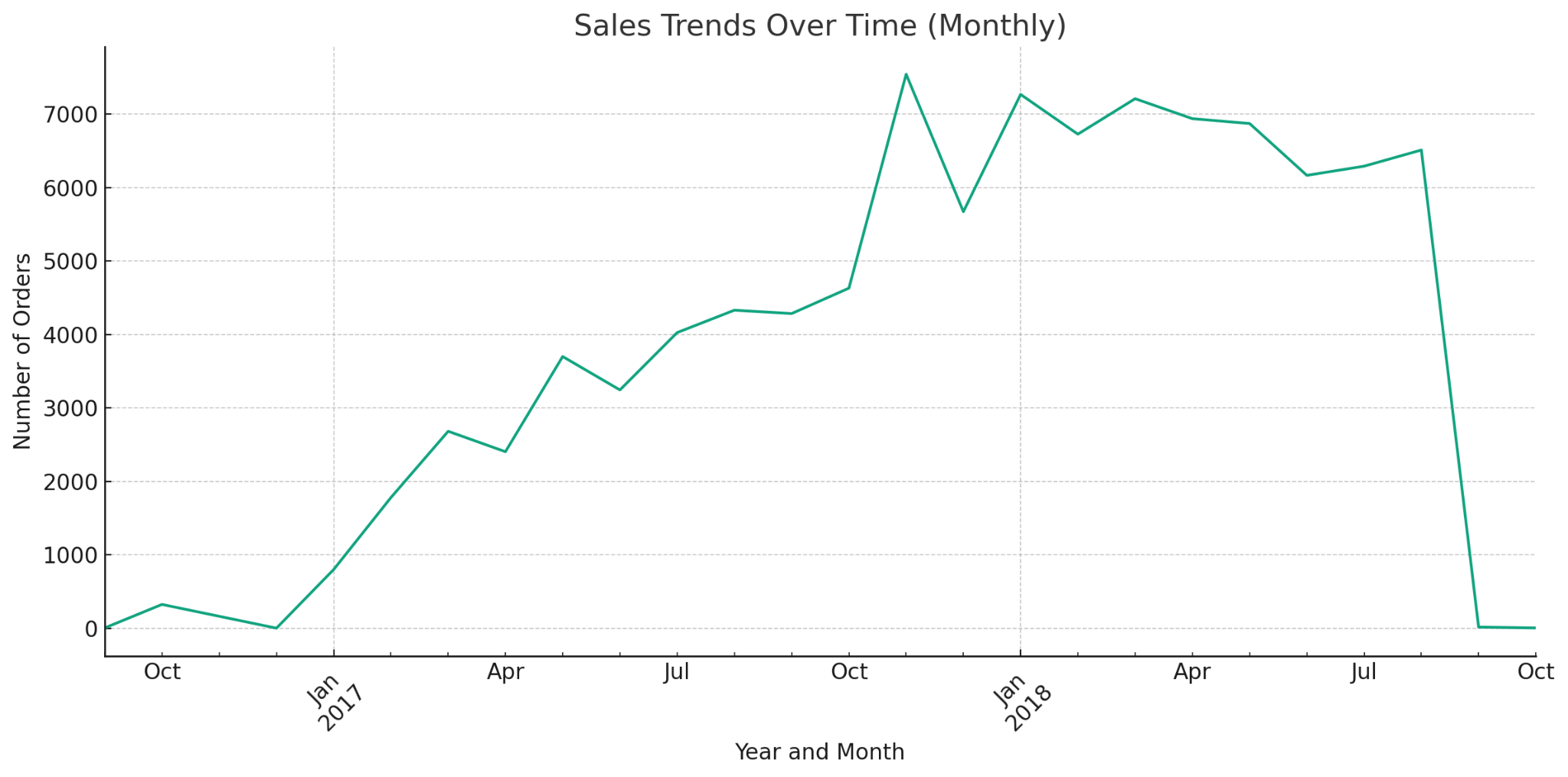


**Visualization of hidden information**

You may choose to explain interesting results from your dataset through visualizations. Charts from any software, such as Excel, Tableau, Python, R, Java, C, C++, or C#, are acceptable.

**Using Python:**

1. **Sales Trends Over Time**: A line chart shows monthly sales trends. This can help in understanding seasonal patterns in sales.



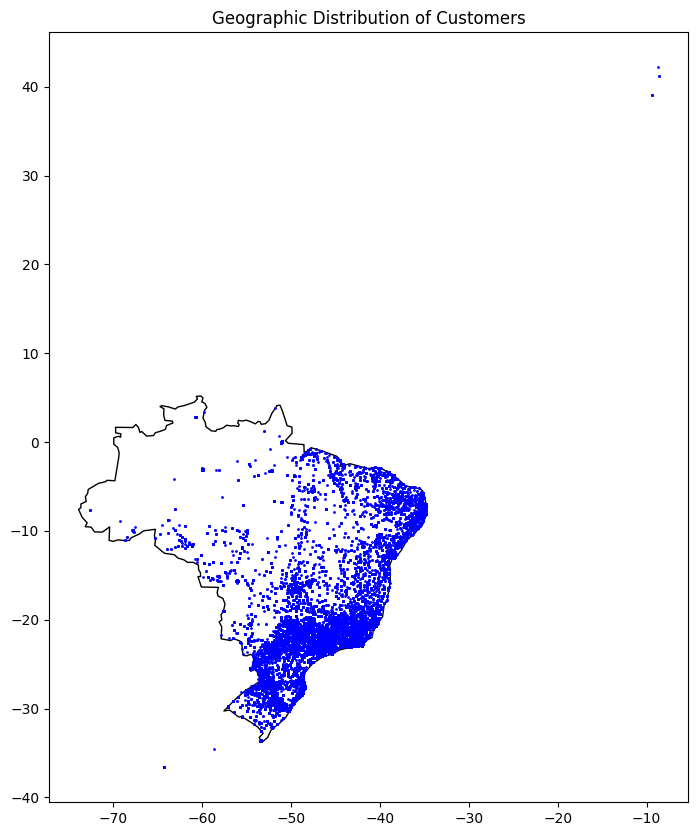
The sales trend visualization has been created, showing the number of orders per month over time.

This line chart represents the monthly sales trends in the dataset.

* It helps understand the fluctuations in order volume over time, which factors like seasonality, holidays, or promotional events could influence.
* Peaks and troughs in the chart can indicate periods of high and low demand, respectively.

1. **Geographic Distribution of Customers or Sellers**: A map visualization showing the geographic distribution of customers or sellers can provide insights into market penetration and logistics planning.

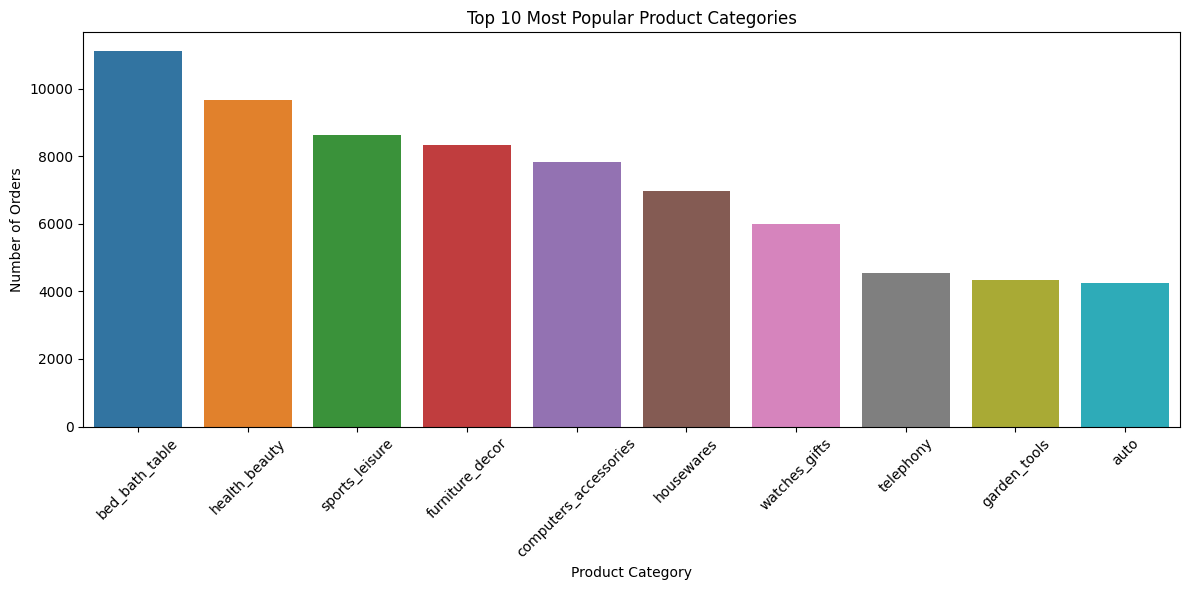
Merging the customer dataset with the geolocation dataset and then plots the geographic distribution of customers across Brazil.



This map visualization represents the geographic distribution of customers across Brazil.

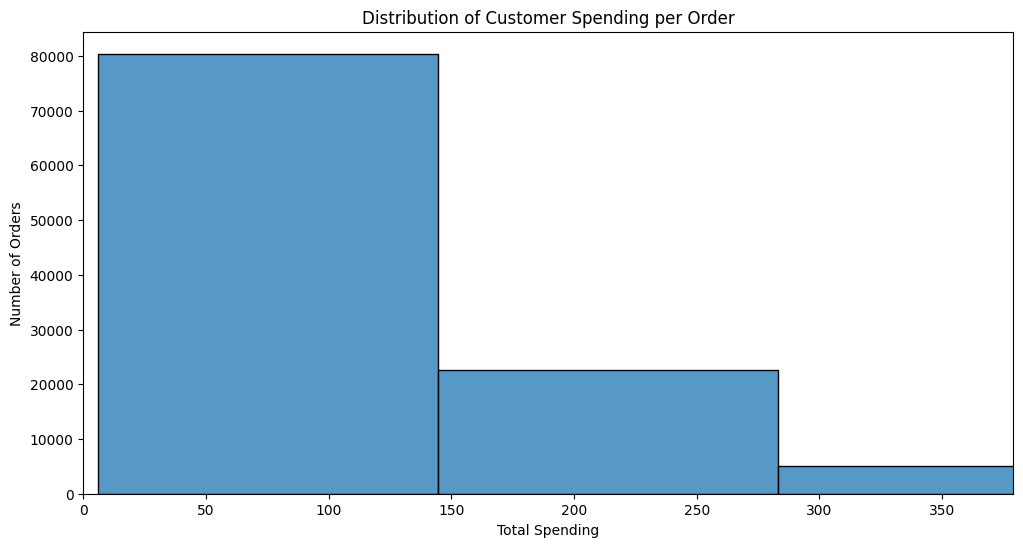
* Blue points on the map indicate customers’ locations, providing insights into market penetration and potentially highlighting areas with higher customer density.

1. **Product Category Popularity**: A bar chart displaying the number of orders per product can reveal the most popular categories.



1. **Customer Spending Analysis**: Create a histogram or box plot of customer spending to understand spending habits.

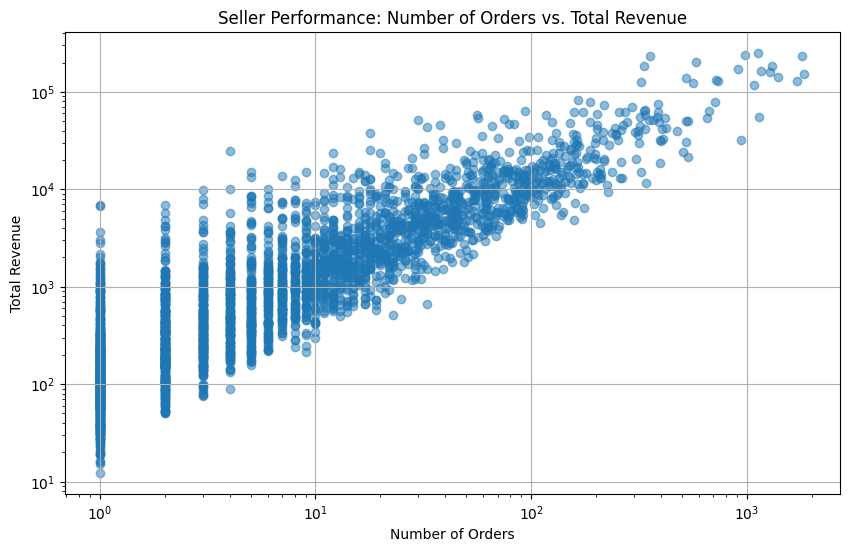
The histogram shows the distribution of customer spending per order.



Most orders fall within a lower spending range, with a significant drop-off as spending amounts increase. This is a typical pattern in consumer spending, where most customers tend to purchase lower-priced items, and fewer customers spend more significant amounts.

1. **Seller Performance**: Use a scatter plot to compare sellers regarding revenue and number of orders.

The scatter plot shows seller performance with the number of orders on the X-axis and total revenue on the Y-axis, both on a logarithmic scale. This type of visualization helps identify trends and outliers in the data.



There appears to be a positive correlation showing a general trend where sellers with more orders tend to have higher total revenue, which is what we would expect.