Digital Sales Insights and Optimization

Advanced and Complex SQL Queries in PostgreSQL

Author

SMIT RANA

linkedin.com/in/smit98rana github.com/gentallman

Introduction

- In the rapidly evolving landscape of e-commerce, businesses continuously seek ways to optimize their sales strategies, enhance customer satisfaction, and improve inventory management. This project focuses on analyzing Amazon's e-commerce sales performance by leveraging a relational database model constructed in PostgreSQL. By utilizing complex SQL queries, we aim to extract valuable insights from various interconnected tables that represent customers, sellers, orders, shipping, payments, products, categories, order items, and inventory.

- The analysis encompasses a range of performance metrics, including top-selling products, revenue by category, customer lifetime value (CLTV), and inventory stock alerts. By addressing these critical aspects, the project aims to provide actionable insights that can help stakeholders make informed decisions to drive sales growth and improve operational efficiency.

Problem Statement

Despite the wealth of data available within Amazon's e-commerce platform, extracting actionable insights requires sophisticated data analysis techniques. The primary challenges addressed in this project include:

- **Sales Performance Metrics:** Understanding which products and categories contribute most significantly to overall sales, and identifying trends in customer purchasing behavior over time.
- **Customer Insights:** Evaluating customer activity to identify potential opportunities for engagement and retention, such as recognizing inactive customers and distinguishing between new and returning customers based on their purchasing patterns.
- **Inventory Management:** Monitoring stock levels and identifying potential inventory shortages, which could lead to lost sales opportunities, while also understanding how inventory levels relate to sales trends.
- **Shipping and Payment Efficiency:** Analyzing shipping performance and payment success rates to identify bottlenecks in the order fulfillment process and improve overall customer satisfaction.
- **Seller Performance Evaluation:** Assessing seller performance based on total sales and order success rates to identify top performers and areas for improvement.

Through a series of complex SQL queries, this project will provide in-depth insights into the above challenges, enabling stakeholders to make data-driven decisions that enhance the overall performance of Amazon's e-commerce platform.

Objectives

1. Develop a robust PostgreSQL database schema that accurately represents the various entities and their relationships within Amazon's e-commerce platform.
2. Implement advanced and complex SQL queries to derive actionable insights related to sales performance, customer behavior, inventory management, shipping efficiency, and seller performance.
3. Present the findings in a comprehensive report, highlighting key metrics, trends, and recommendations for improving Amazon's e-commerce sales strategies.

Entity Relationship Diagram

Fictitious data prepared for practicing and understanding advanced, complex SQL queries

Parent Table
Child Table
Primary Key
Foreign Key

order_items

order_item_id	Primary Key	INT
order_id	**Foreign Key**	**INT**
product_id	Foreign Key	INT
quantity		INT
price_per_unit		FLOAT

inventory

inventory_id	Primary Key	INT
product_id	**Foreign Key**	**INT**
stock		INT
warehouse_id		INT
last_stock_date_id		date

products

product_id	Primary Key	INT
product_name		VARCHAR(50)
price		FLOAT
cogs		FLOAT
category_id	**Foreign Key**	**INT**

category

category_id	Primary Key	INT
category_name		VARCHAR(25)

customers

customer_id	Primary Key	INT
first_name		VARCHAR(20)
last_name		VARCHAR(20)
state		VARCHAR(20)
address		VARCHAR(5)

orders

order_id	Primary Key	INT
order_date		date
customer_id	**Foreign Key**	**INT**
seller_id	**Foreign Key**	**INT**
order_status		VARCHAR(15)

sellers

seller_id	Primary Key	INT
seller_name		VARCHAR(25)
origin		VARCHAR(10)

shipping

shipping_id	Primary Key	INT
order_id	**Foreign Key**	**INT**
shipping_date		date
return_date		date
shipping_providers		VARCHAR(15)
delivery_status		VARCHAR(15)

payments

payment_id	Primary Key	INT
order_id	**Foreign Key**	**INT**
payment_date		date
payment_status		VARCHAR(20)

SMIT RANA

```sql
—— Schema Structure
—— Parent Table Creation :

-- Category
CREATE TABLE category (
    category_id INT PRIMARY KEY,
    category_name VARCHAR(20)
);

-- Customers
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    first_name VARCHAR(20),
    last_name VARCHAR(20),
    state VARCHAR(20),
    address VARCHAR(5) DEFAULT ('xxxx')
);

-- Sellers
CREATE TABLE sellers(
    seller_id INT PRIMARY KEY,
    seller_name VARCHAR(25),
    origin VARCHAR(10)
);

-- Updating datatype of Sellers table
ALTER TABLE sellers
    ALTER COLUMN origin TYPE VARCHAR(10);

—— Child Table Creation :

-- Product
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(50),
    price FLOAT,
    cogs FLOAT,
    category_id INT, --FK
    CONSTRAINT products_fk_category FOREIGN KEY(category_id) REFERENCES category(category_id)
);

-- orders
CREATE TABLE orders(
    order_id INT PRIMARY KEY,
    order_date DATE,
    customer_id INT, --FK
    seller_id INT, --FK
    order_status VARCHAR(15),
    CONSTRAINT orders_fk_customers FOREIGN KEY(customer_id) REFERENCES customers(customer_id),
    CONSTRAINT orders_fk_seller FOREIGN KEY(seller_id) REFERENCES sellers(seller_id)
);

-- order_items
CREATE TABLE order_items(
    order_item_id INT PRIMARY KEY,
    order_id INT, --FK
    product_id INT, --FK
    quantity INT,
    price_per_unit FLOAT,
    CONSTRAINT order_items_fk_orders FOREIGN KEY(order_id) REFERENCES orders(order_id),
    CONSTRAINT order_items_fk_products FOREIGN KEY(product_id) REFERENCES products(product_id)
);

-- payments
CREATE TABLE payments(
    payment_id INT PRIMARY KEY,
    order_id INT, --FK
    payment_date DATE,
    payment_status VARCHAR(20),
    CONSTRAINT payments_fk_orders FOREIGN KEY(order_id) REFERENCES orders(order_id)
);
```

-- CONTINUE --

```sql
-- shipping
CREATE TABLE shipping(
    shipping_id INT PRIMARY KEY,
    order_id INT, -- FK
    shipping_date DATE,
    return_date DATE,
    shipping_providers VARCHAR(15),
    delivery_status VARCHAR(15),
    CONSTRAINT shipping_fk_orders FOREIGN KEY (order_id) REFERENCES orders(order_id)
);

-- inventory
CREATE TABLE inventory(
    inventory_id INT,
    product_id INT, --FK
    stock INT,
    warehouse_id INT,
    last_stock_date DATE,
    CONSTRAINT inventory_fk_products FOREIGN KEY (product_id) REFERENCES products(product_id)
);

                                        -- END --
```

```sql
/*
1. Top Selling Products
Query the top 10 products by total sales value.
Challenge: Include product name, total quantity sold, and total sales value.
*/

--  Altering table by adding new column

ALTER TABLE order_items
ADD COLUMN total_sales FLOAT;

-- Updating column with values (quantity * price_per_unit)

UPDATE order_items
SET total_sales = quantity * price_per_unit;

SELECT
    oi.product_id as "Product_ID",
    p.product_name as "Product",
    ROUND(SUM(oi.total_sales)::numeric, 2) AS "Total Sales",
    COUNT(o.order_id) AS "Total Orders"
FROM
    order_items AS oi
JOIN
    orders AS o
ON
    oi.order_id = o.order_id
JOIN
    products AS p
ON
    oi.product_id = p.product_id
GROUP BY
    oi.product_id, p.product_name
ORDER BY
    "Total Sales" DESC
LIMIT 10;
```

```
Approach :

• Altered the `order_items` table by adding a new column, `total_sales`, to store the calculated sales value for each product.
• Updated the `total_sales` column by multiplying the quantity of the product by its `price_per_unit`.
• Constructed a query that joined the `order_items` table with the `orders` and `products` tables to retrieve details such as product ID, name, total sales, and total orders.
• Grouped the data by product to calculate the total sales value and the number of orders per product.
• Sorted the results in descending order of total sales to rank products by their sales performance.
• Limited the results to the top 10 best•selling products to gain insights into the most successful items.
```

```sql
/*
2. Revenue by Category
Calculate total revenue generated by each product category.
Challenge: Include the percentage contribution of each category to total revenue.
*/

SELECT
    c.category_id AS "Category_ID",
    c.category_name AS "Category",
    ROUND(SUM(oi.total_sales)::numeric,2) AS "Total Sales",
    ROUND((SUM(oi.total_sales) / (SELECT SUM(total_sales) FROM order_items) * 100)::numeric, 2)
    AS "Total Contribution (%)"
FROM
    order_items AS oi
JOIN
    products AS p
ON
    oi.product_id = p.product_id
LEFT JOIN
    category AS c
ON
    p.category_id = c.category_id
GROUP BY
    c.category_id, c.category_name
ORDER BY
    "Total Sales" DESC;
```

```
Approach :

• Retrieved the `category_id` and `category_name` to identify each product category.
• Computed the total sales for each category using the `SUM(oi.total_sales)` function and rounded it to two decimal places.
• Calculated the percentage contribution of each category to total revenue by dividing each category's total sales by the overall total sales across all products, multiplied by 100. The result was rounded to two decimal places.
• Joined the `order_items` table with the `products` table to link each product to its respective category.
• Used a `LEFT JOIN` between the `products` and `category` tables to retrieve the category details.
• Grouped the results by `category_id` and `category_name` to calculate total sales for each category.
• Ordered the results in descending order of total sales, ranking categories by revenue contribution.
```

```sql
/*
3. Average Order Value (AOV)
Compute the average order value for each customer.
Challenge: Include only customers with more than 5 orders.
*/

SELECT
    c.customer_id as "Customer_ID",
    CONCAT(c.first_name, ' ', c.last_name) AS "Name",
    COUNT(o.order_id) AS "Orders",
    ROUND((SUM(oi.total_sales) / COUNT(o.order_id))::numeric,2) AS "AOV"
FROM
    customers AS c
JOIN
    orders AS o
ON
    c.customer_id = o.customer_id
JOIN
    order_items AS oi
ON
    o.order_id = oi.order_id
GROUP BY
    c.customer_id, c.first_name, c.last_name
HAVING
    COUNT(o.order_id) > 5
ORDER BY
    "AOV" DESC;
```

```
Approach :

• Retrieved the `customer_id`, concatenated the first and last names of the customer, counted the number of orders, and computed the average order value (AOV) by dividing the total sales value by the number of orders for each customer.
• Joined the `customers` table with the `orders` table to associate customers with their respective orders.
• Joined the `order_items` table to access the total sales value for each order.
• Grouped the results by `customer_id` and customer name to calculate order counts and total sales per customer.
• Filtered the results to include only customers who had placed more than 5 orders.
• Ordered the results by AOV in descending order to rank customers based on their average order value.
```

```sql
/*
4. Monthly Sales Trend
Query monthly total sales over the past year.
Challenge: Display the sales trend, grouping by month, return current_month sale, last month sale!
*/

SELECT
    "Year",
    "Month",
    "Total Sales" AS "Current Month Sale",
    LAG("Total Sales", 1) OVER (ORDER BY "Year", "Month") AS "Last Month Sale"
FROM
(
    SELECT
        EXTRACT(MONTH FROM o.order_date) AS "Month",
        EXTRACT(YEAR FROM o.order_date) AS "Year",
        ROUND(SUM(oi.total_sales)::NUMERIC, 2) AS "Total Sales"
    FROM
        orders o
    JOIN
        order_items oi
    ON
        oi.order_id = o.order_id
    WHERE
        o.order_date ≥ CURRENT_DATE - INTERVAL '1 year' -- Get data from the last year
    GROUP BY
        "Year", "Month"
    ORDER BY
        "Year", "Month"
) AS "Sales Summary";
```

```
Approach :

• Extracted the year and month from the `order_date` to group sales by month.
• Calculated the total sales for each month by summing the `total_sales` from the `order_items` table.
• Used the `LAG()` window function to retrieve the total sales for the previous month alongside the current month's sales.
• Joined the `orders` table with the `order_items` table to obtain the sales data for each order.
• Filtered the results to include only data from the past year by comparing the `order_date` with the current date minus one year.
• Grouped the data by year and month to calculate the total sales for each month.
• Ordered the results by year and month to maintain the correct sequence for the monthly sales trend.
• The `LAG()` function retrieved the previous month's sales, creating a comparison between the current and last month's sales.
```

```sql
/*
5. Customers with No Purchases
Find customers who have registered but never placed an order.
Challenge: List customer details and the time since their registration.
*/

-- Approach 1: Using a subquery to exclude customers with orders
SELECT *
FROM
    customers
WHERE
    customer_id NOT IN (
        SELECT DISTINCT customer_id
        FROM orders
    );

-- Approach 2: Using a LEFT JOIN to find customers without orders
SELECT *
FROM
    customers AS c
LEFT JOIN
    orders AS o
ON
    o.customer_id = c.customer_id
WHERE
    o.order_date IS NULL;
```

```
Approach 1: Subquery Method
• Selection: Retrieves all columns from the `customers` table.
• Filtering: Uses a subquery to find distinct `customer_id`s from the `orders` table.
• Condition: The main query filters out any customers whose `customer_id` is present in the list
generated by the subquery, effectively excluding customers who have made orders.

Approach 2: LEFT JOIN Method
• Selection: Retrieves all columns from the `customers` table and includes data from the `orders`
table.
• Join Operation: Performs a `LEFT JOIN` between `customers` and `orders` on `customer_id`.
• Filtering: The query checks for rows where `order_date` is `NULL`, indicating that there are no
matching orders for the customer.

Benefits of Each Approach:
• Subquery Method:
  • Straightforward and easy to understand.
  • Effectively identifies customers without orders using a simple filter.

• LEFT JOIN Method:
  • Provides a more intuitive understanding of the relationship between customers and orders.
  • May be more efficient for large datasets as it eliminates the need for a distinct subquery.
```

```sql
/*
6. Least-Selling Categories by State
Identify the least-selling product category for each state.
Challenge: Include the total sales for that category within each state.
*/

WITH Ranking_Table AS (
    SELECT
        c.state AS "State",
        ct.category_name AS "Category",
        ROUND(SUM(oi.total_sales)::NUMERIC, 2) AS "Total Sales",
        RANK() OVER (PARTITION BY c.state ORDER BY ROUND(SUM(oi.total_sales)::NUMERIC, 2) ASC)
        AS "Rank"
    FROM
        customers c
    JOIN
        orders o ON c.customer_id = o.customer_id
    JOIN
        order_items oi ON o.order_id = oi.order_id
    JOIN
        products p ON oi.product_id = p.product_id
    JOIN
        category ct ON p.category_id = ct.category_id
    GROUP BY
        c.state, ct.category_name
)

SELECT "State", "Category", "Total Sales"
FROM Ranking_Table
WHERE "Rank" = 1;
```

```
Approach :

• Identify the least-selling product category for each state, including total sales for that category.
• Use a Common Table Expression (CTE) named `Ranking_Table` to facilitate the ranking process.
• Select `state` from the `customers` table and `category_name` from the `category` table.
• Calculate total sales for each category by summing `oi.total_sales`.
• Apply the `RANK()` window function to assign a rank to each category based on total sales within
each state.
• Partition the ranking by `state` and order by total sales in ascending order.
• Group results by `state` and `category_name` to calculate total sales per category within each
state.
• In the final selection, choose `State`, `Category`, and `Total Sales` from the `Ranking_Table` CTE.
• Filter results to include only those where the rank is 1, indicating the least-selling category for
each state.
```

```sql
/*
7. Customer Lifetime Value (CLTV)
Calculate the total value of orders placed by each customer over their lifetime.
Challenge: Rank customers based on their CLTV.
*/

SELECT
    c.customer_id AS "Customer_ID",
    CONCAT(c.first_name, ' ', c.last_name) AS "Name",
    ROUND(SUM(oi.total_sales)::NUMERIC, 2) AS "CLTV",
    DENSE_RANK() OVER (ORDER BY ROUND(SUM(oi.total_sales)::NUMERIC, 2) DESC) AS "Rank"
FROM
    customers c
JOIN
    orders o ON c.customer_id = o.customer_id
JOIN
    order_items oi ON o.order_id = oi.order_id
GROUP BY
    c.customer_id, c.first_name, c.last_name;
```

```
Approach :

• Calculate the total value of orders placed by each customer over their lifetime, known as Customer Lifetime Value (CLTV).
• Select the `customer_id` from the `customers` table and concatenate the `first_name` and `last_name` to display the full name.
• Compute the CLTV by summing `oi.total_sales` for each customer.
• Round the calculated CLTV to two decimal places for clarity.
• Apply the `DENSE_RANK()` window function to rank customers based on their CLTV in descending order.
• Join the `customers`, `orders`, and `order_items` tables using appropriate keys to aggregate sales data.
• Group the results by `customer_id`, `first_name`, and `last_name` to ensure accurate CLTV calculations for each customer.
• The final output includes the customer ID, name, calculated CLTV, and their corresponding rank.
```

```sql
/*
8. Inventory Stock Alerts
Query products with stock levels below a certain threshold (e.g., less than 10 units).
Challenge: Include last restock date and warehouse information.
*/

SELECT
    i.inventory_id AS "Inventory ID",
    p.product_name AS "Product",
    i.stock AS "Current Stock Left",
    i.last_stock_date AS "Last Stock Date",
    i.warehouse_id AS "Warehouse ID"
FROM
    inventory i
JOIN
    products p ON i.product_id = p.product_id
WHERE
    i.stock < 10
ORDER BY
    i.stock DESC;
```

```
Approach :

• Query products with stock levels below a specified threshold, such as less than 10 units.
• Select the `inventory_id` from the `inventory` table for unique identification of inventory items.
• Retrieve the `product_name` from the `products` table to display the names of products with low stock.
• Include the `stock` level from the `inventory` table to indicate the current stock left for each product.
• Extract the `last_stock_date` from the `inventory` table to show when the product was last restocked.
• Obtain the `warehouse_id` from the `inventory` table to provide information about the warehouse holding the product.
• Join the `inventory` and `products` tables using the `product_id` to link stock information with product details.
• Filter results to include only those products where the stock level is below the specified threshold.
• Order the results by `stock` in descending order to prioritize the items with the least stock left.
```

```sql
/*
9. Shipping Delays
Identify orders where the shipping date is later than 3 days after the order date.
Challenge: Include customer, order details, and delivery provider.
*/

SELECT
    c.*,
    o.*,
    sh.shipping_providers,
    sh.shipping_date - o.order_date AS "Delivery Days"
FROM
    customers c
JOIN
    orders o ON c.customer_id = o.customer_id
JOIN
    shipping sh ON o.order_id = sh.order_id
WHERE
    sh.shipping_date - o.order_date > 3;
```

Approach :

- Identify orders where the shipping date is more than 3 days later than the order date.
- Select all customer details from the `customers` table.
- Retrieve all order details from the `orders` table to provide complete information about the orders.
- Include the `shipping_providers` from the `shipping` table to indicate which delivery provider was used for each order.
- Calculate the difference between the `shipping_date` and `order_date` to determine the number of delivery days.
- Join the `customers`, `orders`, and `shipping` tables using appropriate keys (e.g., `customer_id` and `order_id`) to associate customers with their respective orders and shipping details.
- Filter the results to include only those orders that have a shipping delay of more than 3 days.
- The final output includes customer information, order details, shipping provider, and the calculated delivery days to highlight any significant delays.

```sql
/*
10. Payment Success Rate
Calculate the percentage of successful payments across all orders.
Challenge: Include breakdowns by payment status (e.g., failed, pending).
*/

SELECT
    p.payment_status AS "Payment Status",
    COUNT(*) AS "Count",
    ROUND((COUNT(*)::NUMERIC / (SELECT COUNT(*) FROM payments)::NUMERIC) * 100, 2)
    AS "Breakdown (%)"
FROM
    orders o
JOIN
    payments p ON o.order_id = p.order_id
GROUP BY
    p.payment_status;
```

```
Approach :

• Calculate the percentage of successful payments across all orders, with breakdowns by payment status
(e.g., failed, pending).
• Select the `payment_status` from the `payments` table to categorize the payments.
• Count the total number of payments for each payment status to provide a breakdown of payment counts.
• Compute the percentage breakdown for each payment status by dividing the count of payments by the
total number of payments and multiplying by 100.
• Round the calculated percentage to two decimal places for clarity.
• Join the `orders` and `payments` tables using the `order_id` to link each payment with its
corresponding order.
• Group the results by `payment_status` to ensure distinct categories for successful, failed, and
pending payments.
• The final output includes payment status, count of payments, and percentage breakdowns, offering
insights into overall payment success rates.
```

```sql
/*
11. Top Performing Sellers
Find the top 5 sellers based on total sales value.
Challenge: Include both successful and failed orders, and display their percentage of successful
orders.
*/

WITH top_sellers AS
(
    SELECT
        s.seller_id,
        s.seller_name,
        ROUND(SUM(oi.total_sales)::numeric,2) AS "Total Sales"
    FROM
        orders AS o
    JOIN
        sellers AS s ON o.seller_id = s.seller_id
    JOIN
        order_items AS oi ON oi.order_id = o.order_id
    GROUP BY
        s.seller_id, s.seller_name
    ORDER BY
        "Total Sales" DESC
    LIMIT 5
),
seller_order_status AS
(
    SELECT
        o.seller_id,
        ts.seller_name,
        o.order_status,
        COUNT(*) AS "Total Orders"
    FROM
        orders AS o
    JOIN
        top_sellers AS ts ON ts.seller_id = o.seller_id
    WHERE
        o.order_status NOT IN ('Inprogress', 'Returned')
    GROUP BY
        o.seller_id, o.order_status, ts.seller_name
)

SELECT
    seller_id as "Seller ID",
    seller_name as "Name",
    SUM(CASE WHEN order_status = 'Completed' THEN "Total Orders" ELSE 0 END) AS "Completed Orders",
    SUM(CASE WHEN order_status = 'Cancelled' THEN "Total Orders" ELSE 0 END) AS "Cancelled Orders",
    SUM("Total Orders") AS "Total Orders",
    ROUND(SUM(CASE WHEN order_status = 'Completed' THEN "Total Orders" ELSE 0 END)::NUMERIC /
          NULLIF(SUM("Total Orders")::NUMERIC, 0) * 100, 2) AS "Order Completion Rate"
FROM
    seller_order_status
GROUP BY
    "Seller ID", "Name"
ORDER BY
    "Seller ID", "Name";
```

```
Approach :

• Identify the top 5 sellers based on total sales value.
• Use a Common Table Expression (CTE) named `top_sellers` to calculate the total sales for each seller
by summing the `total_sales` from the `order_items` table.
• Join the `orders`, `sellers`, and `order_items` tables using appropriate keys (e.g., `seller_id` and
`order_id`) to associate sales with their respective sellers.
• Group the results by `seller_id` and `seller_name`, ordering by total sales in descending order, and
limit the results to the top 5 sellers.
• Create a second CTE named `seller_order_status` to analyze the order status for these top sellers.
• Count the total orders for each seller while excluding specific order statuses ('In progress' and
'Returned') to focus on meaningful sales data.
• Select the seller ID and name, along with counts for completed and cancelled orders, and the total
number of orders.
• Calculate the order completion rate as a percentage of completed orders to total orders, using the
`NULLIF` function to avoid division by zero.
• Group the final results by seller ID and name, and order the output for clarity, providing a
comprehensive overview of seller performance, including both successful and failed orders.
```

SMIT RANA

```sql
/*
12. Product Profit Margin
Calculate the profit margin for each product (difference between price and cost of goods sold).
Challenge: Rank products by their profit margin, showing highest to lowest.
*/

SELECT
    product_id,
    product_name,
    "Profit Margin",
    DENSE_RANK() OVER(ORDER BY "Profit Margin" DESC) AS "Product Rank"
FROM
(
    SELECT
        p.product_id,
        p.product_name,
        ROUND((SUM(oi.total_sales - (p.cogs * oi.quantity)) /
               NULLIF(SUM(oi.total_sales), 0))::numeric * 100, 2) AS "Profit Margin"
    FROM
        products AS p
    JOIN
        order_items AS oi ON p.product_id = oi.product_id
    GROUP BY
        p.product_id,
        p.product_name
) AS subquery;
```

```
Approach :

• Calculate the profit margin for each product, defined as the difference between total sales and the cost of goods sold (COGS) expressed as a percentage of total sales.
• Use a subquery to aggregate sales data, joining the `products` and `order_items` tables on `product_id` to relate product details to their sales performance.
• In the subquery, compute total sales for each product by summing the `total_sales` from `order_items`.
• Calculate total cost for each product using the formula: Total_Cost = COGS × Quantity.
• Derive the profit margin by subtracting total cost from total sales, dividing the result by total sales, and multiplying by 100 to express it as a percentage.
• Round the profit margin to two decimal places for clarity.
• In the outer query, select the `product_id`, `product_name`, and calculated "Profit Margin".
• Use the `DENSE_RANK()` window function to rank products based on their profit margins, ordering from highest to lowest.
• This structured approach enables the identification of products with the highest profitability, informing business decisions regarding product offerings and pricing strategies.
```

```sql
/*
13. Most Returned Products
Query the top 10 products by the number of returns.
Challenge: Display the return rate as a percentage of total units sold for each product.
*/

SELECT
    p.product_id,
    p.product_name,
    COUNT(*) AS "Total Units Sold",
    SUM(CASE WHEN o.order_status = 'Returned' THEN 1 ELSE 0 END) AS "Total Returned",
    ROUND((SUM(CASE WHEN o.order_status = 'Returned' THEN 1 ELSE 0 END)::numeric /
           COUNT(*)::numeric * 100), 2
         ) AS "Returned (%)"
FROM
    order_items AS oi
JOIN
    products AS p ON oi.product_id = p.product_id
JOIN
    orders AS o ON o.order_id = oi.order_id
GROUP BY
    p.product_id,
    p.product_name
ORDER BY
    "Returned (%)" DESC
LIMIT 10;
```

```
Approach :

• Query the top 10 products with the highest number of returns based on their order status.
• Join the `order_items`, `products`, and `orders` tables using appropriate keys to gather relevant data.
• Count the total units sold for each product by counting all records in the `order_items` table.
• Use a conditional statement to count the total number of returns by summing the cases where the order status is 'Returned'.
• Calculate the return rate as a percentage of total units sold by dividing the total returns by the total units sold, multiplying by 100 to express it as a percentage.
• Round the return rate to two decimal places for better readability.
• Group the results by `product_id` and `product_name` to aggregate the data for each product.
• Order the final results by the return rate in descending order to highlight the products with the highest return rates.
• Limit the output to the top 10 products to focus on the most returned items.
```

```sql
/*
14. Orders Pending Shipment
Find orders that have been paid but are still pending shipment.
Challenge: Include order details, payment date, and customer information.
*/

SELECT
    c.customer_id AS "Customer ID",
    o.order_id AS "Order ID",
    CONCAT(c.first_name, ' ', c.last_name) AS "Name",
    c.state AS "State",
    o.order_date AS "Order Date",
    o.order_status AS "Order Status"
FROM
    customers AS c
JOIN
    orders AS o ON c.customer_id = o.customer_id
JOIN
    payments AS p ON o.order_id = p.order_id
WHERE
    o.order_status = 'Inprogress'
    AND p.payment_status = 'Payment Successed';
```

Approach :

- Identify orders that are currently marked as 'Inprogress' (indicating they are pending shipment) but have successfully been paid for.
- Join the `customers`, `orders`, and `payments` tables to collect necessary details for each order.
- Select relevant columns, including the customer ID, order ID, customer name (concatenating first and last names), customer state, order date, and order status.
- Filter the results to include only those orders that have a payment status of 'Payment Successed', ensuring that only successfully paid orders are considered.
- This query provides a clear overview of orders that are awaiting shipment despite having been paid, along with associated customer information for better tracking and management.

```sql
/*
15. Inactive Sellers
Identify sellers who haven't made any sales in the last 6 months.
Challenge: Show the last sale date and total sales from those sellers.
*/

WITH seller_not_sale_6_month AS (
    -- Sellers who have not made any sales in the last 6 months
    SELECT seller_id, seller_name
    FROM sellers
    WHERE seller_id NOT IN (
        SELECT DISTINCT seller_id
        FROM orders
        WHERE order_date ≥ CURRENT_DATE - INTERVAL '6 months'
    )
)

SELECT
    s.seller_id "Seller ID",
    s.seller_name as "Name",
    MAX(o.order_date) AS "Last Sale Date",
    COALESCE(SUM(oi.total_sales), 0) AS "Total Sales"
FROM
    seller_not_sale_6_month AS s
LEFT JOIN
    orders AS o ON s.seller_id = o.seller_id
LEFT JOIN
    order_items AS oi ON o.order_id = oi.order_id
GROUP BY
    "Seller ID", "Name";
```

```
Approach :

• Identify sellers who have not made any sales in the last six months by utilizing a Common Table Expression (CTE) named `seller_not_sale_6_month`.
• Within this CTE, select sellers from the `sellers` table whose `seller_id` does not appear in the `orders` table for any orders placed within the last six months.
• In the final query, select the seller ID and name from the CTE.
• Use `LEFT JOIN` to connect the CTE with the `orders` and `order_items` tables, allowing you to access any related sales data for these sellers.
• Retrieve the maximum order date from the `orders` table as the "Last Sale Date" and calculate the total sales from the `order_items` table using the `SUM` function, ensuring to use `COALESCE` to return 0 if there are no sales.
• Group the results by seller ID and name to present a summarized view of inactive sellers, including their last sale date (which would be `NULL` if they have no sales) and total sales (which will be 0).
```

```sql
/*
16. IDENTITY customers into returning or new
if the customer has done more than 5 return categorize them as returning otherwise new
Challenge: List customers id, name, total orders, total returns
*/

SELECT
    "Name",
    "Total Orders",
    "Total Returns",
    CASE
        WHEN "Total Returns" > 5 THEN 'Returning'
        ELSE 'New'
    END AS "Category"
FROM (
    SELECT
        CONCAT(c.first_name, ' ', c.last_name) AS "Name",
        COUNT(o.order_id) AS "Total Orders",
        SUM(CASE
            WHEN o.order_status = 'Returned' THEN 1
            ELSE 0
        END) AS "Total Returns"
    FROM
        customers AS c
    JOIN
        orders AS o ON o.customer_id = c.customer_id
    GROUP BY
        c.first_name, c.last_name
) AS subquery
ORDER BY
    "Total Returns" DESC;
```

```
Approach :

• Categorize customers as either "Returning" or "New" based on the number of returns they have made, specifically identifying those with more than 5 returns as "Returning."
• The final query selects customer names, total orders, and total returns, alongside their assigned category.
• An inner subquery retrieves data by joining the `customers` table with the `orders` table.
• In the subquery, concatenate the first and last names of customers to create a full name.
• Use the `COUNT` function to tally total orders placed by each customer.
• For total returns, employ a `SUM` function with a `CASE` statement to count only those orders where the order status is marked as "Returned."
• Group the results by customer name to obtain aggregate counts.
• In the outer query, use a `CASE` statement to classify customers into "Returning" or "New" based on their total returns.
• Finally, sort the results by total returns in descending order to prioritize customers with the highest return counts.
```

```sql
/*
17. Top 5 Customers by Orders in Each State
Identify the top 5 customers with the highest number of orders for each state.
Challenge: Include the number of orders and total sales for each customer.
*/

SELECT
    CONCAT(c.first_name, ' ', c.last_name) AS "Name",
    c.state AS "State",
    COUNT(o.order_id) AS "No. of Orders",
    ROUND(SUM(oi.total_sales)::numeric, 2) AS "Total Sales",
    DENSE_RANK() OVER(PARTITION BY c.state ORDER BY COUNT(o.order_id) DESC) AS "Rank"
FROM
    orders AS o
JOIN
    customers AS c ON c.customer_id = o.customer_id
JOIN
    order_items AS oi ON o.order_id = oi.order_id
GROUP BY
    c.first_name, c.last_name, c.state
ORDER BY
    "State", "Rank";
```

```
Approach :

• Identify the top 5 customers with the highest number of orders in each state.
• The final query combines customer names, their respective states, the number of orders, total sales, and their ranking based on order count.
• It begins by selecting customer names, states, order counts, and total sales through appropriate aggregations.
• The customer's first and last names are concatenated to create a full name for better readability.
• To count the number of orders per customer, the `COUNT` function is used, while `SUM` calculates the total sales, rounding to two decimal places for precision.
• The `DENSE_RANK()` window function assigns a rank to each customer within their respective state, ordered by the count of orders in descending order.
• The query groups results by customer name and state, ensuring aggregation works correctly.
• Finally, results are ordered by state and rank to clearly present the top customers per state.
```

```sql
/*
18. Revenue by Shipping Provider
Calculate the total revenue handled by each shipping provider.
Challenge: Include the total number of orders handled and the average delivery time for each provider.
*/

SELECT
    s.shipping_providers AS "Shipment Providers",
    COUNT(o.order_id) AS "Orders Handled",
    ROUND(SUM(oi.total_sales)::numeric, 2) AS "Total Sales",
    -- Delivery date should fall somewhere between the shipping date and return date
    -- (s.shipping_date + (s.return_date - s.shipping_date)) AS "Delivery Date",
    COALESCE(ROUND(AVG((s.shipping_date + (s.return_date - s.shipping_date) / 2) -
s.shipping_date)::numeric, 2), 0) AS "Avg Delivery Time"
FROM
    order_items AS oi
JOIN
    orders AS o ON oi.order_id = o.order_id
JOIN
    shipping AS s ON s.order_id = o.order_id
GROUP BY
    s.shipping_providers;
```

```
Approach :

• Calculate the total revenue handled by each shipping provider.
• This query gathers data on the revenue generated, the number of orders processed, and the average
delivery time for each shipping provider.
• It begins by selecting the name of each shipping provider from the shipping table.
• The `COUNT` function counts the total number of orders associated with each provider.
• The `SUM` function calculates the total sales for each provider by summing up the sales values from
the order items.
• To determine the average delivery time, the average of the difference between the shipping date and
the average return date (calculated as half the difference between the shipping date and the return
date) is computed.
• The `COALESCE` function ensures that if there are no delivery times, the average will return as 0
instead of NULL.
• Finally, the results are grouped by the shipping provider to aggregate the data correctly for each
one.
```

```sql
/*
19. Top 10 product with highest decreasing revenue ratio compare to last year(2022) and
current_year(2023)
Challenge: Return product_id, product_name, category_name, 2022 revenue and 2023 revenue decrease ratio
at end Round the result
Note: Decrease ratio = cr-ls/ls* 100 (cs = current_year ls=last_year)
*/

WITH last_year_sale AS (
    SELECT
        p.product_id,
        p.product_name,
        ROUND(SUM(oi.total_sales)::numeric, 2) AS revenue
    FROM
        products AS p
    JOIN
        order_items AS oi ON p.product_id = oi.product_id
    JOIN
        orders AS o ON oi.order_id = o.order_id
    WHERE
        EXTRACT(YEAR FROM o.order_date) = 2022
    GROUP BY
        p.product_id, p.product_name
),
current_year_sale AS (
    SELECT
        p.product_id,
        p.product_name,
        ROUND(SUM(oi.total_sales)::numeric, 2) AS revenue
    FROM
        products AS p
    JOIN
        order_items AS oi ON p.product_id = oi.product_id
    JOIN
        orders AS o ON oi.order_id = o.order_id
    WHERE
        EXTRACT(YEAR FROM o.order_date) = 2023
    GROUP BY
        p.product_id, p.product_name
)

SELECT
    cs.product_id AS "Product ID",
    cs.product_name AS "Name",
    ls.revenue AS "Last Year Revenue",
    cs.revenue AS "Current Year Revenue",
    (ls.revenue - cs.revenue) AS "Revenue Difference",
    ROUND(((ls.revenue - cs.revenue) / ls.revenue) * 100, 2) AS "Revenue Decrease Ratio"
FROM
    last_year_sale AS ls
JOIN
    current_year_sale AS cs ON ls.product_id = cs.product_id
WHERE
    ls.revenue > cs.revenue
ORDER BY
    "Revenue Decrease Ratio" DESC
LIMIT 10;
```

Approach :

1. Common Table Expressions (CTEs):

- `last_year_sale` calculates the total revenue for each product for the year 2022. It joins the `products`, `order_items`, and `orders` tables, filters for the year 2022, and sums up the sales.
- `current_year_sale`performs a similar calculation for the year 2023, summing up the total sales for each product.

2. Main Query:

- It selects the product ID and name, last year's revenue, current year's revenue, and calculates the revenue difference.
- `Revenue Decrease Ratio` = (Last Year Revenue – Current Year Revenue)/Last Year Revenue × 100. This indicates the percentage decrease in revenue from 2022 to 2023.
- The `ROUND` function is used to round the revenue decrease ratio to two decimal places.
- The results are filtered to show only products where the current year’s revenue is less than last year’s.
- Finally, the results are ordered by the revenue decrease ratio in descending order, limiting the output to the top 10 products.

SMIT RANA

```sql
/*
20. Store Procedure
create a function as soon as the product is sold the the same quantity should reduced
from inventory table after adding any sales records it should update the stock in
the inventory table based on the product and qty purchased
*/

CREATE OR REPLACE PROCEDURE process_order_and_update_inventory
(
    p_order_id INT,          -- Order ID provided for the new order
    p_customer_id INT,       -- Customer ID associated with the order
    p_seller_id INT,         -- Seller ID associated with the order
    p_order_item_id INT,     -- Unique Order Item ID for each product in the order
    p_product_id INT,        -- Product ID of the item being ordered
    p_quantity INT           -- Quantity of the product being ordered
)
LANGUAGE plpgsql
AS $$

DECLARE
    -- Variables to store product details and stock check result
    v_count INT;
    v_price FLOAT;
    v_product VARCHAR(50);

BEGIN
    -- Retrieve product name and price based on the provided product ID
    SELECT
        price,
        product_name
    INTO
        v_price,
        v_product
    FROM products
    WHERE product_id = p_product_id;

    -- Check inventory for product availability with the required quantity
    SELECT
        COUNT(*)
    INTO
        v_count
    FROM inventory
    WHERE
        product_id = p_product_id   -- Check for the specified product
        AND stock ≥ p_quantity;     -- Ensure available stock is equal to or
                                    -- greater than the ordered quantity
```

-- CONTINUE --

Approach :

1. Procedure Declaration:
• The procedure is named `process_order_and_update_inventory` and accepts parameters for order ID, customer ID, seller ID, order item ID, product ID, and the quantity being purchased.

2. Variable Declarations:
• `v_count`: Used to check if sufficient stock is available for the product.
• `v_price`: Holds the price of the product.
• `v_product`: Holds the name of the product for messaging purposes.

3. Retrieving Product Information:
• A `SELECT` query fetches the price and name of the product based on the provided `product_id`. This information is stored in the declared variables.

4. Checking Inventory:
• Another `SELECT` query counts the available stock for the product to ensure enough stock is available for the order.

```
    -- If sufficient stock is available, proceed with order and update inventory
    IF v_count > 0 THEN

        -- Insert the new order details into the 'orders' table
        INSERT INTO orders(order_id, order_date, customer_id, seller_id)
        VALUES
        (p_order_id, CURRENT_DATE, p_customer_id, p_seller_id);

        -- Insert product details into the 'order_items' table, calculating total sales
        INSERT INTO order_items(order_item_id, order_id, product_id,
                                quantity, price_per_unit, total_sales)
        VALUES
        (p_order_item_id, p_order_id, p_product_id, p_quantity, v_price, v_price * p_quantity);

        -- Update the 'inventory' table to reduce stock by the ordered quantity
        UPDATE inventory
        SET stock = stock - p_quantity
        WHERE product_id = p_product_id;

        -- Display a success message indicating the product sale has been recorded
        -- and stock updated

        RAISE NOTICE 'Thank you, the product % sale has been added and
                      inventory stock has been updated.', v_product;
    ELSE
        -- If stock is insufficient, display a message indicating the product is unavailable

        RAISE NOTICE 'Unfortunately, the product: % is not available', v_product;
    END IF;

END;
$$;

-- execute the procedure
call process_order_and_update_inventory (24001, 10 , 3, 24007, 38, 11);
```

```
Approach :

5. Conditional Logic:
• If sufficient stock is available (`v_count > 0`):
  • A new record is inserted into the `orders` table.
  • Product details, including total sales, are inserted into the `order_items` table.
  • The inventory is updated to reduce the stock based on the quantity ordered.
  • A success message is raised to notify that the sale has been recorded and inventory updated.
• If the stock is insufficient:
  • A notice is raised to inform the user that the product is not available.

6. Procedure Call:
• The procedure is called with example parameters to process an order.
```

Learning Outcome

- **Joins:** Mastered joins to efficiently retrieve data from multiple related tables.
- **Subqueries and CTEs:** Learned to use subqueries and Common Table Expressions (CTEs) to simplify and organize complex queries for better readability and maintainability.
- **Aggregation Functions:** Gained proficiency in using SUM, COUNT, AVG, and other aggregation functions for summarizing data.
- **Window Functions:** Understood how to apply window functions like DENSE_RANK() to rank and calculate cumulative statistics. Understand how the LAG() window function can be used to perform comparisons between current and previous rows
- **Conditional Logic:** Implemented CASE statements to categorize and apply logic based on various conditions within queries.
- **Procedural Logic in pgSQL:** Developed stored procedures using pgSQL for reusability and handling procedural logic, including error handling and validation.
- **Date Filtering:** Applied date functions (EXTRACT(), INTERVAL) to filter data by specific time frames or periods.
- **Efficient Stored Procedures:** Created modular, reusable stored procedures for dynamic input and managing complex database transactions.
- **Query Optimization:** Learned to write optimized queries, reducing redundancy, enhancing performance, and avoiding unnecessary complexity.

THANK YOU !