Global Superstore

*Note: Sub-titles are not captured in Xplore and should not be used

1st Akanksh Gatla
akankshg)
(University at Buffalo -Suny.)
Buffalo, US
email address: akankshg@buffalo.edu

2nd Rakshitha Shivaraj
rshivara)
University at Buffalo -Suny.)
Buffalo, US
email address: rshivara@buffalo.edu

3rd Venkata Subramanyam
vupputur)
University at Buffalo -Suny.)
Buffalo, US
email address: vupputur

I. Introduction

The retail industry has witnessed a significant increase in large global stores that operate across multiple countries and offer a wide variety of products to customers under one roof. These superstores have become popular among consumers due to their convenience, competitive pricing, and diverse product offerings. This report aims to analyze the database management system (DBMS) used by a global superstore, with a focus on PostgreSQL, an open-source relational DBMS widely used in the industry. We will explore the various features of PostgreSQL that make it a suitable choice for managing the vast amount of data generated by a global superstore. The report will also provide insights into how this DBMS can improve operational efficiency and decision-making capabilities. The report aims to offer a comprehensive understanding of the importance of DBMS in the retail industry, specifically in the context of global superstores. It highlights the significance of selecting the appropriate DBMS to achieve business success in a highly competitive and constantly evolving industry.

II. Ease of Use

A. Who will use your database.

- The target users of a superstore database could include:
- E-commerce store owners: They need the shopping cart database to manage their online store and track customer orders, payment details, and shipping information.
- Marketers: They can use the shopping cart database to analyze customer behavior, purchasing patterns, and shopping preferences to create targeted marketing campaigns.
- Customer service teams: They need access to the shopping cart database to track customer orders, update order status, and address customer concerns related to shipping, payments, and returns.
- Inventory managers: They need the shopping cart database to manage product inventory, track sales data, and ensure the availability of popular products.
- Data analysts: They can use the shopping cart database to generate reports, perform data analysis, and identify trends and patterns that can be used to optimize the online store's performance.
- Developers: They need the shopping cart database to create custom applications, develop integrations with third-party services, and build features that enhance the shopping experience for customers.
- In our project we are trying to be most of the roles that are represented above, To track the data, analyze them get the insight's with complex queries in PostgreSQL, Would be building up a Superstore Website which helps to shop and store the information of the user and track every customers information for future use and choices of the customer can be generated by the number of times that they are ordering the product. We will try to post the complex queries on the website based on the Database which helps in computation of faster loading time for a scenario than a usual time running.

B. Who will administer the database?

- Depending on the organization or firm that operates the e-commerce website, the administrator of a shopping cart database may differ. An administrator is typically a member of the IT team, such as a database administrator or a system administrator.
- With smaller organizations or firms, the administrator may be the owner or a member of the marketing team in charge of the e-commerce website. In larger enterprises, the shopping cart database and its operations may be overseen by a dedicated team of administrators.
- The administrator's major task, regardless of who they are, is to guarantee that the shopping cart database is properly maintained, secure, and working optimally. This includes activities like database backup and recovery, user administration, and troubleshooting, technical issues, and managing integrations with third-party services.

C. Real Life Scenario:

- A Superstore database in action be an e-commerce website that offers electronics to accessories. Customers can explore products, add them to their shopping cart, and then proceed to checkout to complete their purchase.

Identify applicable funding agency here. If none, delete this.

- All the customer's order details, including product information, pricing, shipping address, and payment details, would be stored in the shopping cart database. The database would also record account information for the customer, such as their email address, username, and order history.
- When a customer adds a product to their shopping cart, the shopping cart database is automatically updated to reflect the addition of the item. The database would continue to update as the customer browsed and added more things to their cart. Where are the data is stored in the database.
- The database would process the payment and update the database with the final order details, including the transaction ID and order status, once the customer had completed their order and went to checkout.
- The administrator of the e-commerce website might use the superstore database to track inventory levels, evaluate consumer activity, and generate sales performance statistics. The database could also be used by the customer service team to address any customer concerns about orders, shipping, and payments.
- Examples: Amazon, Walmart, Costco, etc.

ER - Diagram

Fig. 1. ER Diagram

- Relation Schema
  - customer_details:
    customer_details(customer_id : int, customer_name : string, country : string, state : string, region : string, city : string, phone_number : int, email_id : string)
  - shipping_details:
    shipping_details(order_id : int, order_date : date, order_priority : string, ship_date : date, ship_mode : string, country : string, state : string, region : string)
  - billing_details:
    billing_details(order_id : int, quantity : int, unit_price : double, sales : int, discount : double, shipping_cost : double)
  - product_details:
    product_details(product_id : int, product_name : string, category : string, sub_category : string, unit_price : double, sales : int, row_id : int)
  - order_customer:
    order_customer(order_id : int, order_date : date, customer_id : int, ship_mode : string, order_priority : string)
  - order_details:
    order_details(order_id : int, order_date : date, customer_id : int, ship_mode : string, product_id : int, quantity : int, category : string, order_priority : string)
  - order_product:
    order_product(order_id : int, product_id : int, order_date : date, ship_mode : string, category : string, order_priority : string)

A: We have used real dataset

B: Usage of Data

- The target users of a superstore database could include:
- The target users of a superstore database could include:
- E-commerce store owners: They need the shopping cart database to manage their online store and track customer orders, payment details, and shipping information.
- Marketers: They can use the shopping cart database to analyze customer behavior, purchasing patterns, and shopping preferences to create targeted marketing campaigns.
- Customer service teams: They need access to the shopping cart database to track customer orders, update order status, and address customer concerns related to shipping, payments, and returns.
- Inventory managers: They need the shopping cart database to manage product inventory, track sales data, and ensure the availability of popular products.
- Data analysts: They can use the shopping cart database to generate reports, perform data analysis, and identify trends and patterns that can be used to optimize the online store's performance.
- Developers: They need the shopping cart database to create custom applications, develop integrations with third-party services, and build features that enhance the shopping experience for customers.
- ...In our project we are trying to be most of the roles that are represented above, To track the data, analyze them get the insight's with complex queries in PostgreSQL, Would be building up a Superstore Website which helps to shop and store the information of the user and track

every customers information for future use and choices of the customer can be generated by the number of times that they are ordering the product. We will try to post the complex queries on the website based on the Database which helps in computation of faster loading time for a scenario than a usual time running.

What kind of queries do you want to ask? (Real life scenario)

- A Superstore database in action be an e-commerce website that offers electronics to accessories. Customers can explore products, add them to their shopping cart, and then proceed to checkout to complete their purchase.
- All the customer's order details, including product information, pricing, shipping address, and payment details, would be stored in the shopping cart database. The database would also record account information for the customer, such as their email address, username, and order history.
- When a customer adds a product to their shopping cart, the shopping cart database is automatically updated to reflect the addition of the item. The database would continue to update as the customer browsed and added more things to their cart. Where are the data is stored in the database.
- The database would process the payment and update the database with the final order details, including the transaction ID and order status, once the customer had completed their order and went to checkout.
- The administrator of the e-commerce website might use the superstore database to track inventory levels, evaluate consumer activity, and generate sales performance statistics. The database could also be used by the customer service team to address any customer concerns about orders, shipping, and payments. Examples: Amazon, Walmart, Costco, etc.

III. Database

There are 7 tables

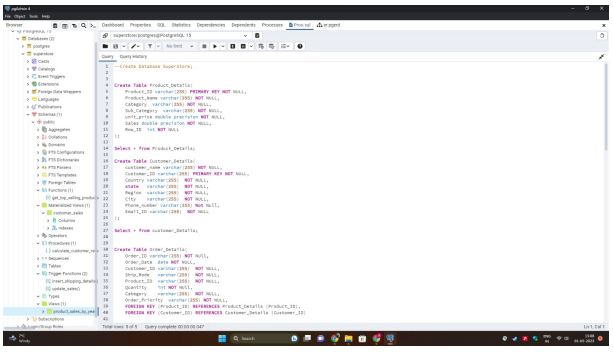

Fig. 2. Create table

- Product_Details: creates a table named "Product_Details" with columns for Product_ID, Product_Name, Category, Sub_Category, unit_price, Sales, and Row_ID. Product_ID is set as the primary key and cannot be null. All other columns cannot be null. The data types for unit_price, Sales, and Row_ID are double precision, and int respectively.
- Customer_Details: a table named "Customer_Details" with columns for customer_name, Customer_ID, Country, state, Region, City, Phone_number, and Email_ID. Customer_ID is set as the primary key and cannot be null. All other columns cannot be null. The data type for Phone_number is varchar(255), and the data types for all other columns are also varchar(255).
- Order_Details: creates a table named "Order_Details" with columns for Order_ID, Order_Date, Customer_ID, Ship_Mode, Product_ID, Quantity, Category, and Order_Priority. Order_ID, Order_Date, Customer_ID, Ship_Mode, Productvv_ID, Quantity, Category, and Order_Priority cannot be null. The table also includes two foreign keys to link the Product_Details and Customer_Details tables with the Product_ID and Customer_ID columns respectively.
- Billing_Details: The statement creates a table named "Billing_Details" with columns for Order_ID, Quantity, Unit_Price, Sales, Discount, and Shipping_Cost. Order_ID is set as the primary key and cannot be null. All other columns cannot be null. The data types for Quantity and Discount are int and double precision respectively, and the data types for Unit_Price, Sales, and

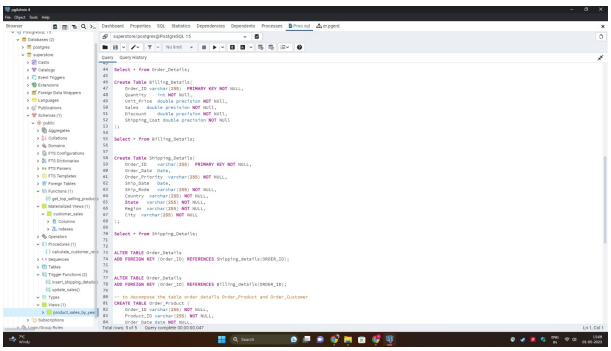

Fig. 3. Create table

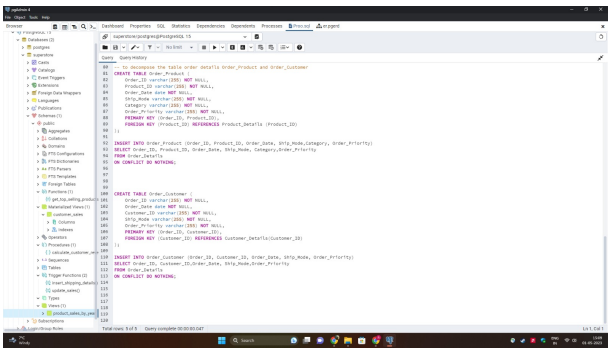

Fig. 4. Create table

Shipping_Cost are double precision.

- Shipping_Details: The statement creates a table named "Shipping_Details" with columns for Order_ID, Order_Date, Order_Priority, Ship_Date, Ship_Mode, Country, State, Region, and City. Order_ID is set as the primary key and cannot be null. Order_Date, Order_Priority, Ship_Mode, Country, State, Region, and City cannot be null. The data types for Order_Date, Ship_Date are Date, and the data types for all other columns are varchar(255).
- Order_Product: creates a table named "Order_Product" with columns for Order_ID, Product_ID, Order_Date, Ship_Mode, Category, and Order_Priority. Order_ID, Product_ID, Order_Date, Ship_Mode, Category, and Order_Priority cannot be null. The table also includes a composite primary key on Order_ID and Product_ID and a foreign key to link the Product_Details table with the Product_ID column.
- Order_Customer: creates a table named "Order_Customer" with columns for Order_ID, Order_Date, Customer_ID, Ship_Mode, and Order_Priority. Order_ID, Order_Date, Customer_ID, Ship_Mode, and Order_Priority cannot be null. The table also includes a composite primary key on Order_ID and Customer_ID and a foreign key to link the Customer_Details table with the Customer_ID column.

Insert queries:

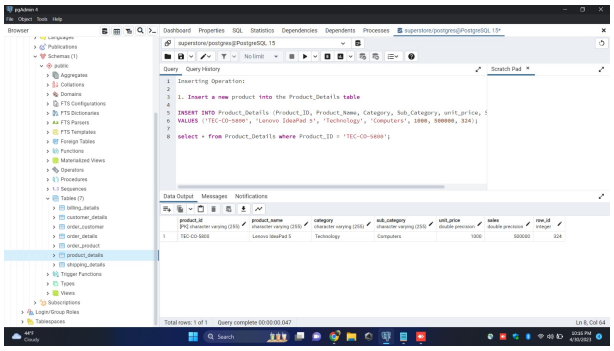

Fig. 5. Insert1

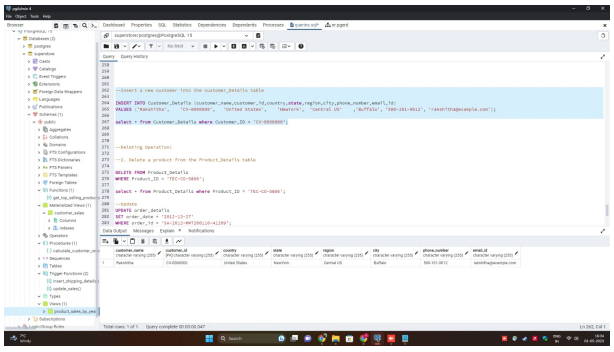

Fig. 6. Insert2

- statement is an INSERT query that inserts a new row into the Product_Details table with values for Product_ID, Product_Name, Category, Sub_Category, unit_price, Sales, and Rov_ID. The second statement is a SELECT query that retrieves the row from the Product_Details table where the Product_ID is 'TEC-CO-5800'. This query will return the row that was inserted in the previous INSERT statement, if it exists. In summary, the INSERT statement adds a new row to the Product_Details table, and the SELECT statement retrieves the newly inserted row based on its Product_ID.
- is an INSERT query that inserts rows into the Order_Product table by selecting data from the Order_Details table. It inserts values for Order_ID, Product_ID, Order_Date, Ship_Mode, Category, and Order_Priority. The ON CONFLICT DO NOTHING clause tells the database to do nothing if there is a conflict during the insertion process. In summary, this query is used to populate the Order_Product table with data from the Order_Details table while avoiding conflicts that might occur during the insertion process.
- is an INSERT query that inserts rows into the Order_Customer table by selecting data from the Order_Details table. It inserts values for Order_ID, Customer_ID, Order_Date, Ship_Mode, and Order_Priority. The ON CONFLICT DO NOTHING clause tells the database to do nothing if there is a conflict during the insertion process. In summary, this query is used to populate the Order_Customer table with data from the Order_Details table while avoiding conflicts that might occur during the insertion process.

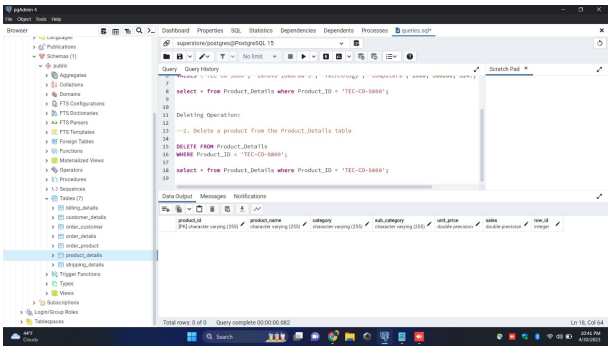

Fig. 7. delete

- statement is a DELETE query that deletes the row from the Product_Details table where the Product_ID is 'TEC-CO-5800'. The second statement is a SELECT query that retrieves all rows from the Product_Details table where the Product_ID is 'TEC-CO-5800'. As the row with Product_ID 'TEC-CO-5800' has been deleted in the previous query, the SELECT statement will not return any rows.
- statement is a DELETE query that deletes the row from the Customer_Details table where the Customer_ID is 'CV-0000000'. The second statement is a SELECT query that retrieves all rows from the Customer_Details table

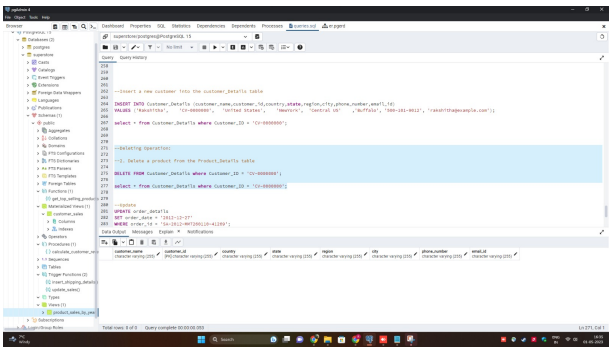

Fig. 8. delete

where the Customer_ID is 'CV-0000000'. As the row with Customer_ID 'CV-0000000' has been deleted in the previous query, the SELECT statement will not return any rows.

Update Query:

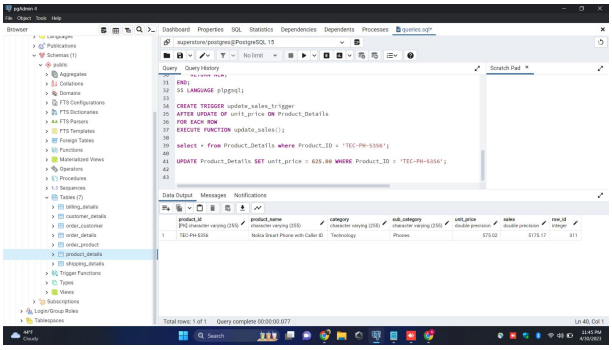

Fig. 9. Update

- creates a PostgreSQL trigger function that updates the "Sales" column in the "Product_Details" table whenever the "unit_price" column is updated. The "update_sales" function calculates the total sales based on the quantity of product ordered and the new unit price, and updates the "Sales" column accordingly. The "update_sales_trigger" trigger is then attached to the "Product_Details" table to execute the "update_sales" function after any updates to the "unit_price" column. Finally, an update statement is executed to update the unit price of the product with ID 'TEC-PH-5356', which triggers the "update_sales_trigger" and updates the "Sales" column accordingly. The "select" statement at the end verifies that the update was successful.

Alter Query:

- statements are adding foreign key constraints to the Order_Details table. The first statement adds a foreign key referencing the ORDER_ID column in the Shipping_details table, while the second statement adds a foreign key referencing the ORDER_ID column in the Billing_details table. These foreign keys enforce referential integrity between the Order_Details table and the referenced tables.

IV. BCNF

- To determine if the initial schema is in BCNF, we need to check if all the functional dependencies in the schema satisfy the condition that the left-hand side of every dependency is a superkey of the relation. If any functional dependency violates this condition, we need to decompose the relation to satisfy BCNF.

Let's start by analyzing each relation and identifying the functional dependencies:
Product_Details:
Product_ID → Product_Name, Category, Sub_Category, unit_price, Sales, Row_ID The functional dependency is valid because Product_ID is the primary key of the table. Therefore, Product_Details is already in BCNF.

Customer_Details:
Customer_ID → customer_name, Country, state, Region, City, Phone_number, Email_ID The functional dependency is valid because Customer_ID is the primary key of the table. Therefore, Customer_Details is already in BCNF.

Billing_Details:
Order_ID → Quantity, Unit_Price, Sales, Discount, Shipping_Cost
The functional dependency is valid because Order_ID is the primary key of the table. Therefore, Billing_Details is already in BCNF.

Order_Details:
Order_ID, Product_ID → Order_Date, Customer_ID, Ship_Mode, Category, Order_Priority

Product_ID → Category, Sub_Category, unit_price, Sales, Row_ID

Customer_ID → customer_name, Country, state, Region, City, Phone_number, Email_ID

The first functional dependency is not in BCNF because neither Order_ID , Product_ID,customer_ID is a superkey of the table. To satisfy BCNF, we need to decompose the table into two relations:

Order_Product:
Order_ID,Product_ID,order_Date,ship_mode, category,order_prority.

Order_Customer:
Order_ID,order_Date,customer_id,ship_mode,order_prority.

Both the new relations satisfy BCNF.

Shipping_Details:
Order_ID → Order_Date, Order_Priority, Ship_Date, Ship_Mode, Country, State, Region, City
The functional dependency is valid because Order_ID is the primary key of the table. Therefore, Shipping_Details is already in BCNF.

In summary, the final schema is as follows:
All tables are now in BCNF, and the original functional dependencies are preserved. Therefore, this schema is better suited to handle large datasets and reduces data redundancy.

V. SQL QUERIES

- 1. Query: SELECT * FROM Order_Details

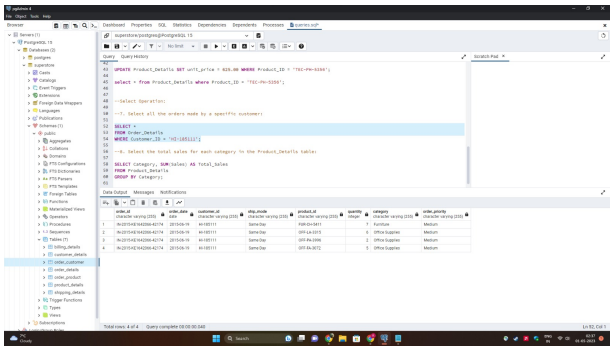

Fig. 10. Where Condition

WHERE Customer_ID = 'HI-185111';

The given SQL query selects in fig 10: all columns from the Order_Details table where the Customer_ID is equal to 'HI-185111'. The query returns all orders made by the customer with the ID 'HI-185111'.

- 2. Query: SELECT Category, SUM(Sales) AS Total_Sales FROM Product_Details GROUP BY Category;

The given SQL query selects in fig 11: all columns from the Order_Details table where the Customer_ID is equal to 'HI-185111'. The query returns all orders made by the customer with the ID 'HI-185111'.

- 3. Query:
SELECT p.Product_Name, p.Category, c.customer_name, c.Country, c.state
FROM Product_Details p
INNER JOIN Order_Details o ON p.Product_ID = o.Product_ID
INNER JOIN Customer_Details c ON o.Customer_ID = c.Customer_ID

```
55
56  --8. Select the total sales for each category in the Product_Details table:
57
58  SELECT Category, SUM(Sales) AS Total_Sales
59  FROM Product_Details
60  GROUP BY Category;
61
```

Data Output | Messages | Notifications

	category character varying (255)	total_sales double precision
1	Furniture	580975.26
2	Office Supplies	456552.28000000026
3	Technology	728823.0999999974

Fig. 11. Group By

WHERE o.Order_ID = 'IN-2012-AB1010558-41270';
GROUP BY Category;

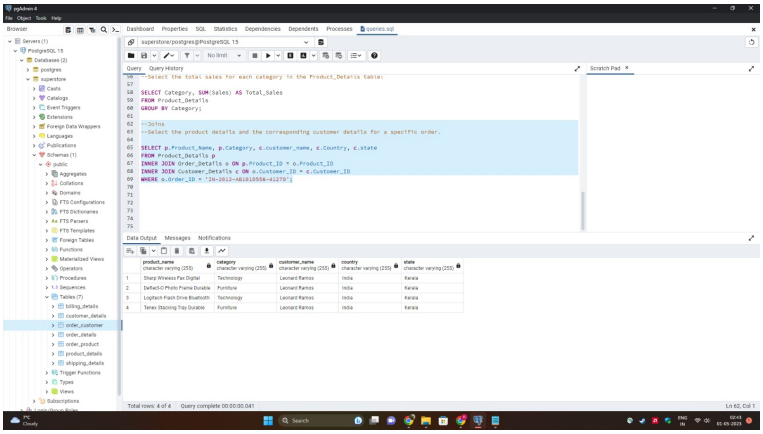

Fig. 12. Join

This SQL statement is using the INNER JOIN operator to combine the Product_Details, Order_Details, and Customer_Details tables based on their common column(s). It is selecting the Product_Name, Category, customer_name, Country, and state columns from the tables to retrieve information about a specific order (specified by the Order_ID in the WHERE clause). The join condition is matching the Product_ID column in the Product_Details table with the Product_ID column in the Order_Details table, and then matching the resulting Customer_ID column in the Order_Details table with the Customer_ID column in the Customer_Details table. Refer fig 12.

- 4. Query:

SELECT c.customer_name, c.Country, c.state,
(SELECT COUNT(*) FROM Order_Details
WHERE Customer_ID = c.Customer_ID) AS Total_Orders,
(SELECT COALESCE(SUM(pd.unit_price * od.quantity), 0)

FROM Product_Details pd INNER JOIN Order_Details od ON pd.Product_ID = od.Product_ID
WHERE od.Customer_ID = c.Customer_ID) AS Total_Sales,
(SELECT COUNT(*) FROM Shipping_Details WHERE Order_ID IN
(SELECT Order_ID FROM Order_Details WHERE Customer_ID = c.Customer_ID)) AS Total_Shipments
FROM Customer_Details c
ORDER BY Total_Sales DESC
LIMIT 10;

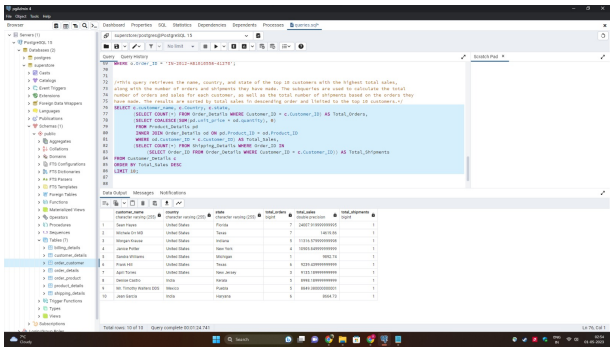

Fig. 13. Order by, Joins

This query retrieves the name, country, and state of the top 10 customers with the highest total sales, along with the number of orders and shipments they have made. The subqueries are used to calculate the total number of orders and sales for each customer, as well as the total number of shipments based on the orders they have made. The results are sorted by total sales in descending order and limited to the top 10 customers. Refer fig 13.

- 5. Query:

SELECT c.customer_name, c.Country,
c.state, SUM(b.Sales) AS Total_Sales,
MAX(s.Ship_Date) AS Last_Shipment_Date
FROM Customer_Details c
INNER JOIN Order_Details o ON c.Customer_ID = o.Customer_ID
INNER JOIN Billing_Details b ON o.Order_ID = b.Order_ID
INNER JOIN Shipping_Details s ON o.Order_ID = s.Order_ID WHERE c.Country = 'United States'
GROUP BY c.customer_name, c.Country, c.state
HAVING COUNT(DISTINCT o.Product_ID) ¿ 5
ORDER BY Total_Sales DESC;

This query retrieves the total sales and last shipment date for each customer in the USA who has ordered more than 5 different products. The results are sorted by total sales in descending order.
The query uses JOINs to combine data from the Customer_Details, Order_Details, Billing_Details, and Shipping_Details tables. It also uses subqueries to calculate

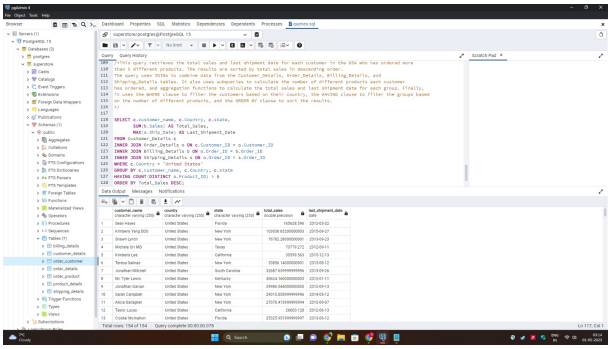

Fig. 14. Order by, Joins, having, group by

the number of different products each customer has ordered, and aggregation functions to calculate the total sales and last shipment date for each group.
Finally, it uses the WHERE clause to filter the customers based on their country, the HAVING clause to filter the groups based on the number of different products, and the ORDER BY clause to sort the results. Refer fig 14.

- 6. Query:

CREATE VIEW product_sales_by_year
AS SELECT pd.product_id, pd.product_name, DATE_TRUNC('year', od.order_date)
AS year, SUM(bd.sales) AS total_sales
FROM product_details pd
INNER JOIN order_details od ON pd.product_id = od.product_id
INNER JOIN billing_details bd ON od.order_id = bd.order_id
GROUP BY pd.product_id, pd.product_name, DATE_TRUNC('year', od.order_date);

SELECT * FROM product_sales_by_year;

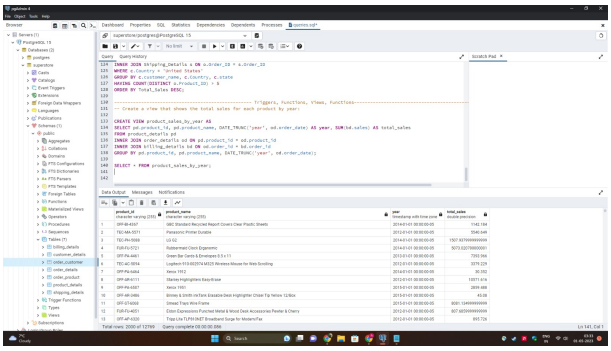

Fig. 15. View

This query retrieves the total sales and last shipment date for each customer in the USA who has ordered more than 5 different products. The results are sorted by total sales in descending order.

The first query creates a view named product_sales_by_year that shows the total sales for each product by year, by joining the product_details, order_details and billing_details tables, grouping the results by product_id, product_name and year.
The second query selects all rows from the product_sales_by_year view, displaying the total sales for each product by year. Refer fig 15 and 16.

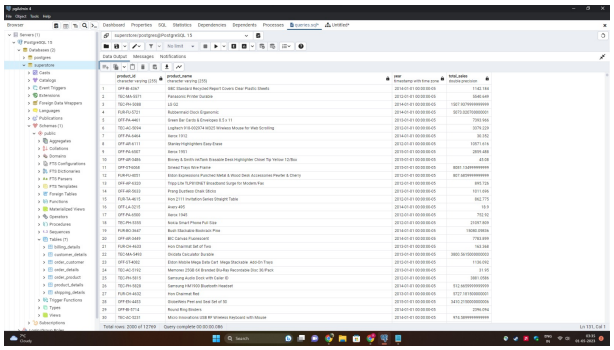

Fig. 16. Output

Triggers

- 7. Query:

– Create a trigger that automatically inserts a new record into the Shipping_Details table whenever a new order is inserted into the Order_Details table:

```
CREATE OR REPLACE FUNCTION insert_shipping_details()
RETURNS TRIGGER AS $$
BEGIN
– Try to update the existing record with the new values
UPDATE Shipping_Details
SET Order_Date = NEW.Order_Date,
Order_Priority = NEW.Order_Priority,
Ship_Mode = NEW.Ship_Mode,
Country = (SELECT Country FROM Customer_Details
WHERE Customer_ID = NEW.Customer_ID),
State = (SELECT State FROM Customer_Details
WHERE Customer_ID = NEW.Customer_ID),
Region = (SELECT Region FROM Customer_Details
WHERE Customer_ID = NEW.Customer_ID),
City = (SELECT City FROM Customer_Details
WHERE Customer_ID = NEW.Customer_ID)
WHERE Order_ID = NEW.Order_ID;

– If no record was updated, insert a new one
IF NOT FOUND THEN
INSERT INTO Shipping_Details (Order_ID, Order_Date,
Order_Priority, Ship_Mode, Country, State, Region, City)
SELECT NEW.Order_ID, NEW.Order_Date,
NEW.Order_Priority, NEW.Ship_Mode, c.Country,
c.State, c.Region, c.City
FROM Customer_Details c
WHERE c.Customer_ID = NEW.Customer_ID;
END IF;

RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_insert_shipping_details
AFTER INSERT ON Order_Details
FOR EACH ROW
EXECUTE FUNCTION insert_shipping_details();

/* trigger first tries to update the existing record in the Shipping_Details table with the new values based on the Order_ID. If no record was updated, it means that there was no existing record with that Order_ID, so it inserts a new record into the Shipping_Details table.*/

– checking purpose
select * from order_details;

INSERT INTO Order_Details (Order_ID,
Order_Date,customer_id,ship_mode,product_id
,quantity,category,order_priority)
VALUES ('SA-2012-MM7260110-41269', '2015-12-31', 'Fe-3512840', 'Second Class', 'TEC-PH-3807', 4 ,'Technology', 'Critical');

select * from shipping_details where order_id='SA-2012-MM7260110-41269';
```

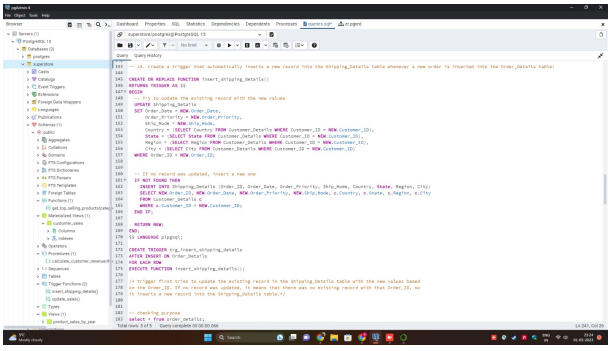

Fig. 17. Function come trigger

This query provided is used to create a trigger in PostgreSQL that automatically inserts a new record into the Shipping_Details table whenever a new order is inserted into the Order_Details table.

The trigger is defined as "trg_insert_shipping_details" and is executed after each insert operation on the Order_Details table. The trigger calls a function called "insert_shipping_details" that performs the necessary

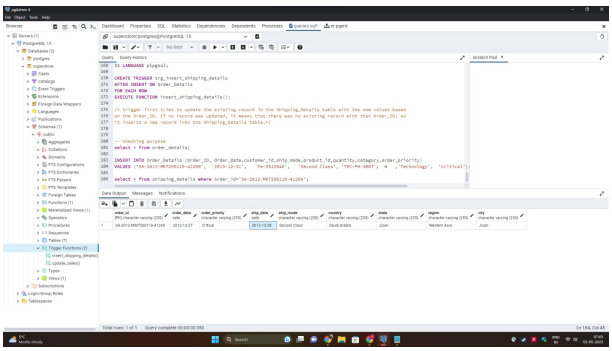

Fig. 18. Function come trigger Output

operations to insert or update the Shipping_Details table.

The function first tries to update an existing record in the Shipping_Details table with the new values based on the Order_ID. If the update operation doesn't affect any record, it means that there was no existing record with that Order_ID, so it inserts a new record into the Shipping_Details table. The values inserted in the new record are obtained from the Customer_Details table using the Customer_ID provided in the new order.

Finally, the trigger returns the new row inserted into the Order_Details table.

To check if the trigger is working properly, you can execute the select query to view the contents of the Order_Details table before and after inserting a new row using the INSERT statement. Then, you can execute the select query to view the contents of the Shipping_Details table and check if the new record was properly inserted.

- 8. Query:

CREATE OR REPLACE PROCEDURE calculate_customer_revenue(
IN customer_name varchar(255),
IN year int,
OUT revenue double precision
)
AS $$
BEGIN
SELECT SUM(bd.Sales) INTO revenue
FROM Order_Details od
JOIN Billing_Details bd ON od.Order_ID = bd.Order_ID
JOIN Customer_Details cd ON od.Customer_ID = cd.Customer_ID
WHERE cd.Customer_Name = calculate_customer_revenue.customer_name
AND extract(year from od.Order_Date) = year;
END;
$$ LANGUAGE plpgsql;

DO $$
DECLARE
v_revenue double precision;
BEGIN
CALL calculate_customer_revenue('Stacy Cruz', 2014, v_revenue);
RAISE NOTICE 'Revenue for Stacy Cruz in 2014: END $$;

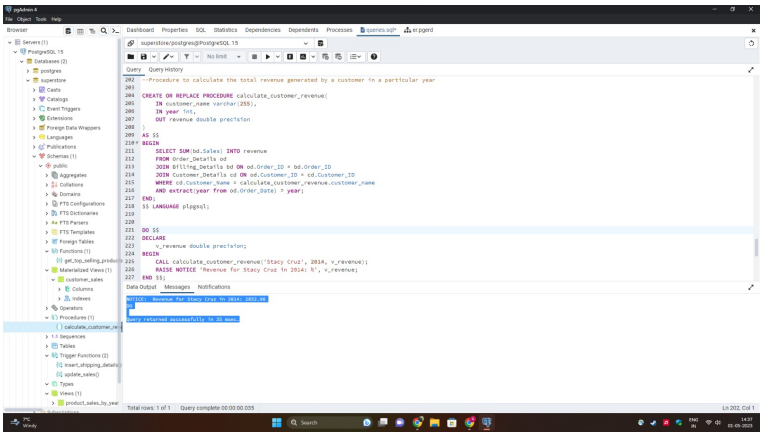

Fig. 19. Procedure

This is a PostgreSQL script that creates a stored procedure named 'calculate_customer_revenue'. The procedure takes two input parameters, customer_name and year, and one output parameter, revenue. The procedure calculates the revenue generated by the specified customer in the specified year by summing up the sales amount from the Order_Details and Billing_Details tables, and stores it in the revenue variable.

The second part of the script demonstrates how to call the stored procedure using the 'CALL' command and pass the output parameter to a variable named 'v_revenue'. Finally, the script uses the 'RAISE NOTICE' command to print the value of the 'v_revenue' variable. Fig 19

- 9. Query:
– Find the top 5 customers who have the highest number of orders, along with the average order amount:

SELECT cd.customer_name, COUNT(DISTINCT od.order_id) AS num_orders, AVG(bd.quantity * pd.unit_price) AS avg_order_amount
FROM billing_details bd
JOIN order_details od ON bd.order_id = od.order_id
JOIN customer_details cd ON od.customer_id cd.customer_id
JOIN product_details pd ON od.product_id = pd.product_id
GROUP BY cd.customer_name

ORDER BY num_orders DESC
LIMIT 5;

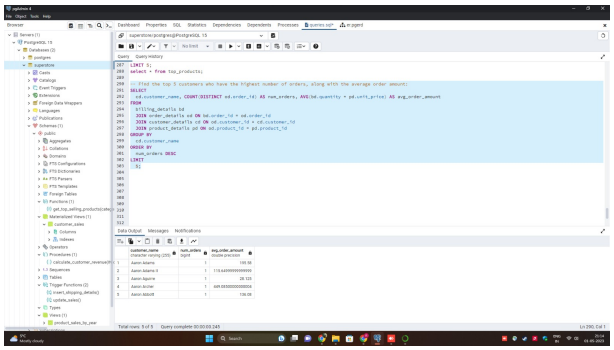

Fig. 20. Higest Number of orders

query fig 20, finds the top 5 customers who have the highest number of orders, along with the average order amount. It joins the billing_details, order_details, customer_details, and product_details tables to obtain the required information.

The SELECT statement selects the customer_name from the customer_details table, counts the distinct order_id values from the order_details table for each customer, and calculates the average order amount by multiplying the quantity and unit_price columns from the product_details table.

The GROUP BY clause groups the results by the customer_name column, and the ORDER BY clause orders the results in descending order based on the num_orders column, which represents the number of orders for each customer. The LIMIT clause limits the results to the top 5 rows.

Overall, this query provides useful information about the top customers in terms of the number of orders and their average order amounts.

- 10. Query:
Retrieve the top 5 customers who have spent the most amount of money on purchases:

SELECT customer_details.customer_name,
SUM(billing_details.sales) AS total_sales
FROM customer_details
INNER JOIN order_details ON customer_details.customer_id = order_details.customer_id
INNER JOIN billing_details
ON order_details.order_id = billing_details.order_id
GROUP BY customer_details.customer_name
HAVING SUM(billing_details.sales) ¿ 0
ORDER BY total_sales DESC
LIMIT 5;

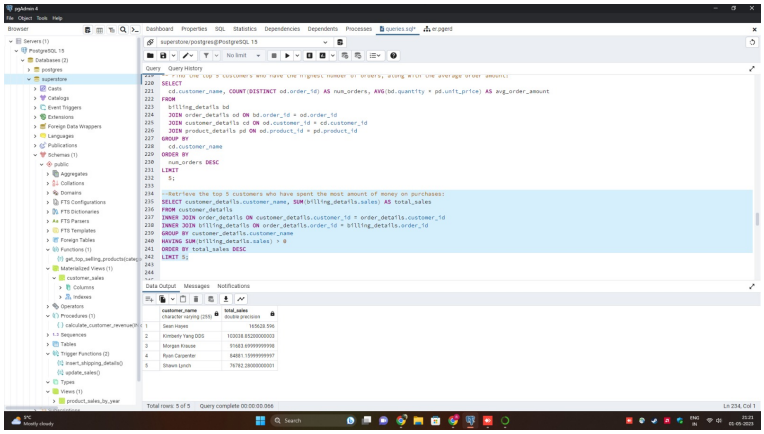

Fig. 21. Spent most amount of money

The query fig 21 provided retrieves the top 5 customers who have spent the most amount of money on purchases. It joins the customer_details, order_details, and billing_details tables to obtain the required information.
The SELECT statement selects the customer_name from the customer_details table and calculates the total sales amount by summing the sales column from the billing_details table for each customer. The GROUP BY clause groups the results by the customer_name column, and the HAVING clause filters out customers with zero sales.
The ORDER BY clause orders the results in descending order based on the total_sales column, which represents the total amount spent by each customer. The LIMIT clause limits the results to the top 5 rows.
Overall, this query provides useful information about the top customers in terms of the amount of money spent on purchases, which can be helpful for customer analysis and marketing strategies.

VI. Three problematic queries

- 1. Problematic Query 1: Before optimization
This query retrieves the name, country, and state of the top 10 customers with the highest total sales, along with the number of orders and shipments they have made. The subqueries are used to calculate the total number of orders and sales for each customer, as well as the total number of shipments based on the orders they have made. The results are sorted by total sales in descending order and limited to the top 10 customers.

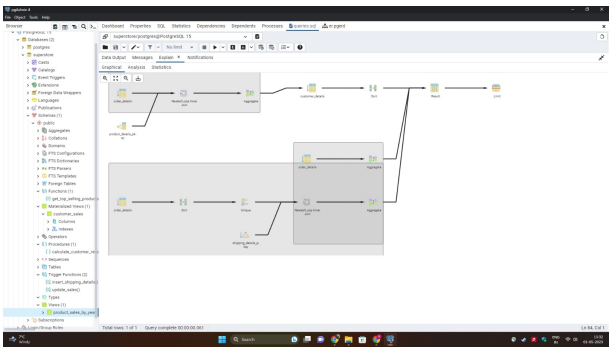

Fig. 22. Before Optimization : Problematic 1

Query:
SELECT c.customer_name, c.Country, c.state,
(SELECT COUNT(*) FROM Order_Details
WHERE Customer_ID = c.Customer_ID) AS Total_Orders,
(SELECT COALESCE(SUM(pd.unit_price * od.quantity), 0)
FROM Product_Details pd
INNER JOIN Order_Details od ON pd.Product_ID = od.Product_ID
WHERE od.Customer_ID = c.Customer_ID) AS Total_Sales,
(SELECT COUNT(*) FROM Shipping_Details
WHERE Order_ID IN
(SELECT Order_ID FROM Order_Details WHERE Customer_ID = c.Customer_ID)) AS Total_Shipments
FROM Customer_Details c
ORDER BY Total_Sales DESC
LIMIT 10;
Result: This query is a problamatic query which is taking around 1:24:741 time. Fig 22

- After Optimization:
query:
CREATE MATERIALIZED VIEW customer_sales AS
SELECT od.Customer_ID,
COALESCE(SUM(pd.unit_price * od.quantity), 0) AS Total_Sales
FROM Product_Details pd
INNER JOIN Order_Details od ON pd.Product_ID = od.Product_ID
GROUP BY od.Customer_ID;

SELECT c.customer_name, c.Country, c.state,
(SELECT COUNT(*) FROM Order_Details WHERE Customer_ID = c.Customer_ID) AS Total_Orders, cs.Total_Sales,
(SELECT COUNT(*) FROM Shipping_Details WHERE Order_ID IN
(SELECT Order_ID FROM Order_Details
WHERE Customer_ID = c.Customer_ID)) AS Total_Shipments
FROM Customer_Details c
LEFT JOIN customer_sales cs ON c.Customer_ID = cs.Customer_ID
ORDER BY cs.Total_Sales DESC
LIMIT 10;

Result: It just took around 0:00:106 time after optimization by using views. Fig 23

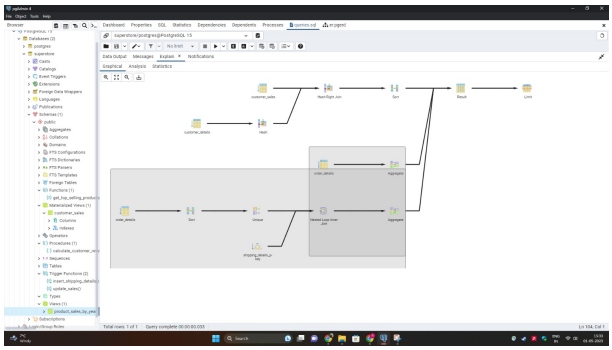

Fig. 23. After Optimization : Problematic 1

- 2. Problematic Query 2:
Before optimization:
The query you provided creates a view called ”top_products” and then selects all rows from it. The view is created by joining the product_details, order_product, order_details, and billing_details tables to obtain the total sales for each product.
Fig 24 This query may take a long time to execute because it involves multiple table joins and grouping and aggregating large amounts of data. Creating a view can also add to the execution time since it requires the query to be executed and the results stored in memory.

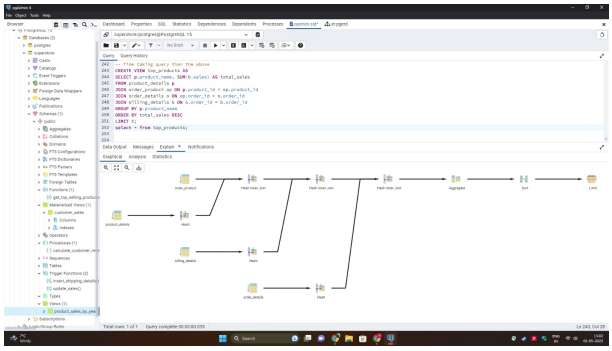

Fig. 24. Before Optimization : Problematic 2

Query:
– Time-taking query than the above
CREATE VIEW top_products AS
SELECT p.product_name, SUM(b.sales) AS total_sales
FROM product_details p
JOIN order_product op ON p.product_id = op.product_id
JOIN order_details o ON op.order_id = o.order_id
JOIN billing_details b ON o.order_id = b.order_id
GROUP BY p.product_name
ORDER BY total_sales DESC
LIMIT 5;
select * from top_products;

- After Optimization:
query:
SELECT p.product_name, s.total_sales
FROM product_details p
JOIN (
SELECT op.product_id, SUM(b.sales) AS total_sales
FROM order_product op
JOIN billing_details b ON op.order_id = b.order_id
GROUP BY op.product_id
) s ON p.product_id = s.product_id
ORDER BY s.total_sales DESC
LIMIT 5;
s
Result: It took less time

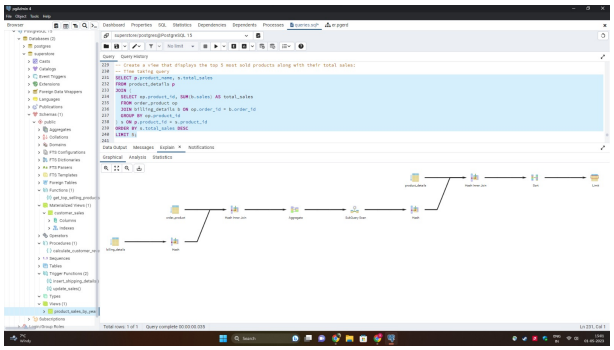

Fig. 25. After Optimization: Problematic 2

This optimized query avoids joining the order_details table, which can reduce the amount of data that needs to be processed.
It also uses a subquery to aggregate the sales data before joining with the product_details table, which can improve performance by reducing the number of rows that need to be joined.
Additionally, indexing the columns used for joining and aggregating data can further optimize the query. Fig 25

- 3. Problematic Query 3:
Before optimization:
The query is updating the Sales column of the Product_Details table with the total sales value for each product. It does so by multiplying the unit_price of the product with the sum of the quantity of all orders that contain that product. The subquery in the SET clause of the UPDATE statement returns the total quantity of orders for the product.

However, this query can be slow for large datasets, as it calculates the total sales for each product by summing up all orders that contain that product. Therefore, it may be inefficient to update the Sales column every time an order is added or updated.

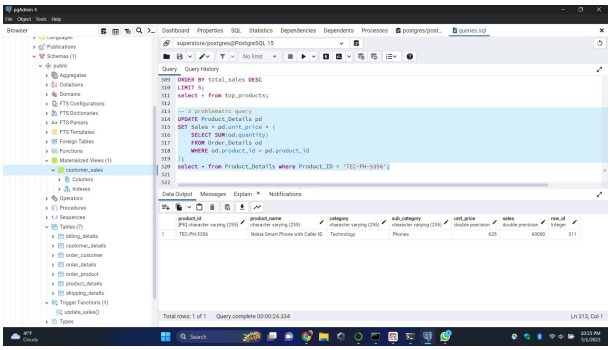

Fig. 26. Before Optimization : Problematic 3

Query:
UPDATE Product_Details pd
SET Sales = pd.unit_price * (
SELECT SUM(od.quantity)
FROM Order_Details od
WHERE od.product_id = pd.product_id
);
select * from Product_Details where Product_ID = 'TEC-PH-5356';

- After Optimization:
query:
CREATE OR REPLACE FUNCTION update_sales()
RETURNS TRIGGER AS
$$
BEGIN
UPDATE Product_Details
SET Sales = NEW.unit_price * (
SELECT SUM(od.quantity)
FROM Order_Details od
WHERE od.product_id = NEW.product_id
)
WHERE Product_ID = NEW.Product_ID;
RETURN NEW;
END;
$$
LANGUAGE plpgsql;
This is an optimized version of the code that updates the sales of a product whenever its unit price is updated. The original code had a subquery that summed the quantity of all orders containing the product and multiplied it by its unit price. This approach required scanning the entire

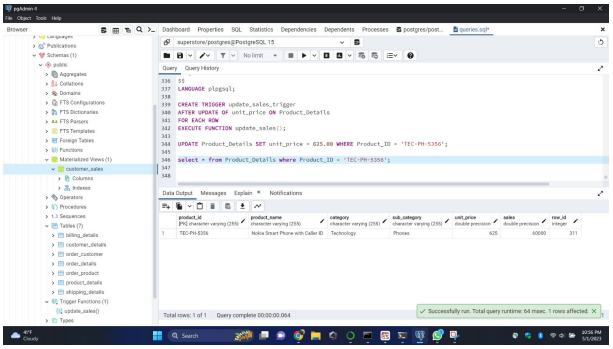

Fig. 27. After Optimization: Problematic 3

Order_Details table every time a product's unit price was updated, which could be slow if the table was large.

The optimized code uses a trigger to update the sales of a product only when its unit price changes. The trigger fires after an update on the Product_Details table's unit_price column. The trigger executes the update_sales() function, which updates the sales of the product using a subquery that sums the quantity of orders containing the product.

The optimized code is more efficient because it updates the sales of a product only when necessary, rather than scanning the entire Order_Details table every time a product's unit price is updated. This approach improves the query's performance and reduces the database's workload.

VII. Team Contribution

- Akanksh Gatla: Procedures, Views, triggers, Problematic query, ER diagram, create 7 tables, Front end, report
- Rakshitha Shivaraj: CRUD opeartaions, Triggers, Problematic query, report, order by, group by, Triggers
- Venkata Subramnayam: Problematic query, report, DML, order by, group by, Procedures report

Acknowledgment

We would like to express our sincere gratitude to everyone who has contributed to the successful completion of this SQL project report. We would like to thank our mentor for providing us with constant guidance and support throughout this project. We would also like to thank our team members for their hard work and dedication towards the completion of this project.

We would like to express our gratitude towards our institute for providing us with the necessary resources and facilities that were required to complete this project. Finally, we would like to thank our families and friends for their encouragement and support throughout this project.

Thank you all for your contributions towards this project.

References

[1] https://www.airops.com/blog/learn-sql-for-free-sql-learning-resources
[2] https://www.w3schools.com/sql/
[3] https://livesql.oracle.com/apex/f?p=590:1000
[4] https://www.oracle.com/database/sqldeveloper/
[5] https://www.youtube.com/watch?v=ObbNGhcxXJA
[6] https://www.sciencegate.app/keyword/324683
[7] https://ieeexplore.ieee.org/document/5669100
[8] https://www.researchgate.net/topic/Oracle-Database/publications
[9] https://www.ijirmf.com/wp-content/uploads/2017/01/201701023.pdf
[10] https://www.academia.edu/34568839/Huge_and_Real_Time_Database_Systems_A_Cor
[11] https://portswigger.net/web-security/sql-injection/examining-the-database/lab-listing-database-contents-oracle