**AmazonDB: Mini E-Commerce Database for Scalable Sales**

**Team Members & Responsibilities:**

**Mohini Patil (UMID: 32423513)**

**Role:**

SQL Query Development & Index Optimization

**Responsibilities:**

• Creating the python file for data cleaning and handling missing values

• Write and optimize SQL queries for business use cases

• Analyze query performance using EXPLAIN and profiling tools

• Recommend and implement indexing strategies

• Ensure query correctness and efficiency

• Report generation and plot visual generation

• Document findings and improvement suggestions

**Venkata Vyshnavi Nemani (UMID: 04611110)**

**Role:** ER Modelling & Schema Design

**Responsibilities:**

**•** Design the Entity-Relationship (ER) diagram

**•** Identify relationships (1:N, M:N) and set up foreign key constraints

**•** Ensure referential integrity across the schema

**•** Collaborate on schema normalization

**Nikhil Reddy Kandadi (UMID: 28962694)**

**Role:** Database Implementation & Data Insertion Scripts

**Responsibilities:**

**•** Implement SQL DDL statements to create tables

**•** Write scripts to populate tables with test/sample data

**•** Handle data consistency and type validations

**•** Assist in foreign key testing and debugging

**Owesh** **Chaiwala (UMID: 23827191)**

**Role:** Reporting & Performance Evaluation

**Responsibilities:**

• Test business queries for correctness and completeness

• Generate reports based on query outputs

• Conduct performance evaluations pre- and post-indexing

**Project Description**:

This project focuses on designing and analyzing a database using a real-world Amazon sales dataset containing 128,976 records. Each record includes details such as order ID, shipping info, product category, size, order amount, courier status, and fulfilment type. With handling missing values, removing duplicates and cleaning the data we created a **structured clean data with a count of 120350 records**.

The goal is to apply key database concepts from the course to model the data, write insightful queries, and optimize performance.

This project will give us the opportunity to test the **following concepts learned in class:**

* ER Modelling & Relational Design - Create an ER diagram and convert it into a relational schema for a real world dataset.
* Normalization - Normalize the data and implement the schema in SQL.
* SQL Querying - Write SQL queries to extract key insights
* Indexing - Use indexing to optimize query performance
* Analytical Insights from Queries - Analyze patterns in sales, fulfillment, and cancellations
* Ability to measure the performance of a SQL query / index

**Attached Files**

Raw amazon sales data (csv format): Amazon Sale Report.csv

Data transformation/cleaning/duplicates : e\_commerce\_database.py

Transformed dataset (csv format): cleaned\_data folder

DDL statements: DDL\_STATEMENTS.sql, Indexes.sql

DML statements: DML\_STATEMENTS.sql

SQL queries + code for experiments: E-Commerce\_Database.sql

**Dataset Used :**

<https://drive.google.com/file/d/1iALGhi7eRRWLAVLjgjSb3JVD4Rn0FY1J/view?usp=sharing>

Dataset Attributes Include:-

ORDERS - order\_id, order\_date,status,fulfilment,sales\_channel,ship\_service\_level,sku,qty,

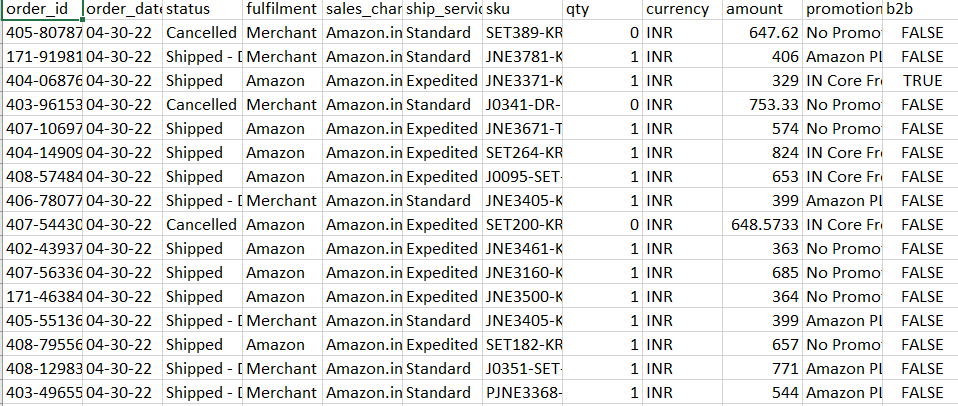
currency,amount,promotion\_ids,b2b

PRODUCTS - sku, style, category,size,asin

SHIPPING - order\_id, courier\_status, ship\_city,ship\_state,ship\_postal\_code,

ship\_country,fulfilled\_by

Below is a snapshot of the data from relation “Ordes”



**Dataset Transformation**

In order to make the data set compatible with the SQL copy command , in the notebook **E\_Commerce\_Database.ipynb**, data preprocessing begins with loading the dataset into a pandas DataFrame, renaming columns for consistency (e.g., changing 'Sales Channel' to 'sales\_channel').

Null value handling is done using both inspection and imputation. The notebook checks for missing values using isnull().sum() and handles them accordingly.

Specifically, **missing values** in columns like 'Unnamed’ are addressed by either dropping rows or filling them with default values such as the column mean or median, depending on the context.

Columns like promotion-ids, fulfilled-by are filled with appropriate values. Currency is filled with mode value. Also rows with missing currency or amount are dropped as this column is necessary.

Date columns such as 'order\_date' and 'delivery\_date' are converted to datetime objects to support time-based analysis.

For EDA (Exploratory Data Analysis), the notebook employs seaborn and matplotlib to visualize distributions of categorical features like 'status', 'category', 'region', and 'state' - Order Status Distribution, Top 10 Product Categories

The cleaned and enriched dataset is then used to generate insights on sales performance, order status trends, top categories, and regional sales behavior.

All the code for transforming and cleaning the dataset is included in “e\_commerce\_database.py”

**Conceptual Design**

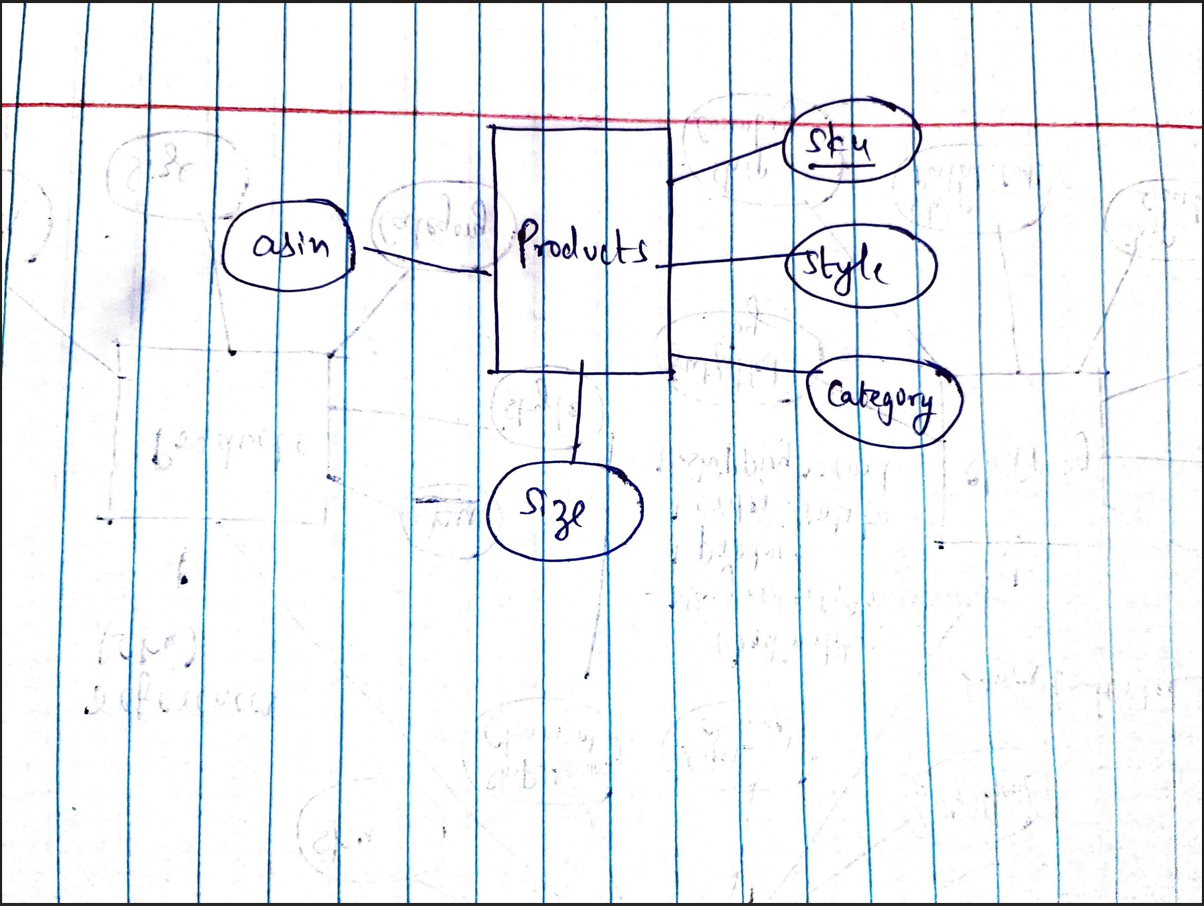
**ER Diagram:**

**Below is the Entity Relationship Diagrams for the Database we are using:**

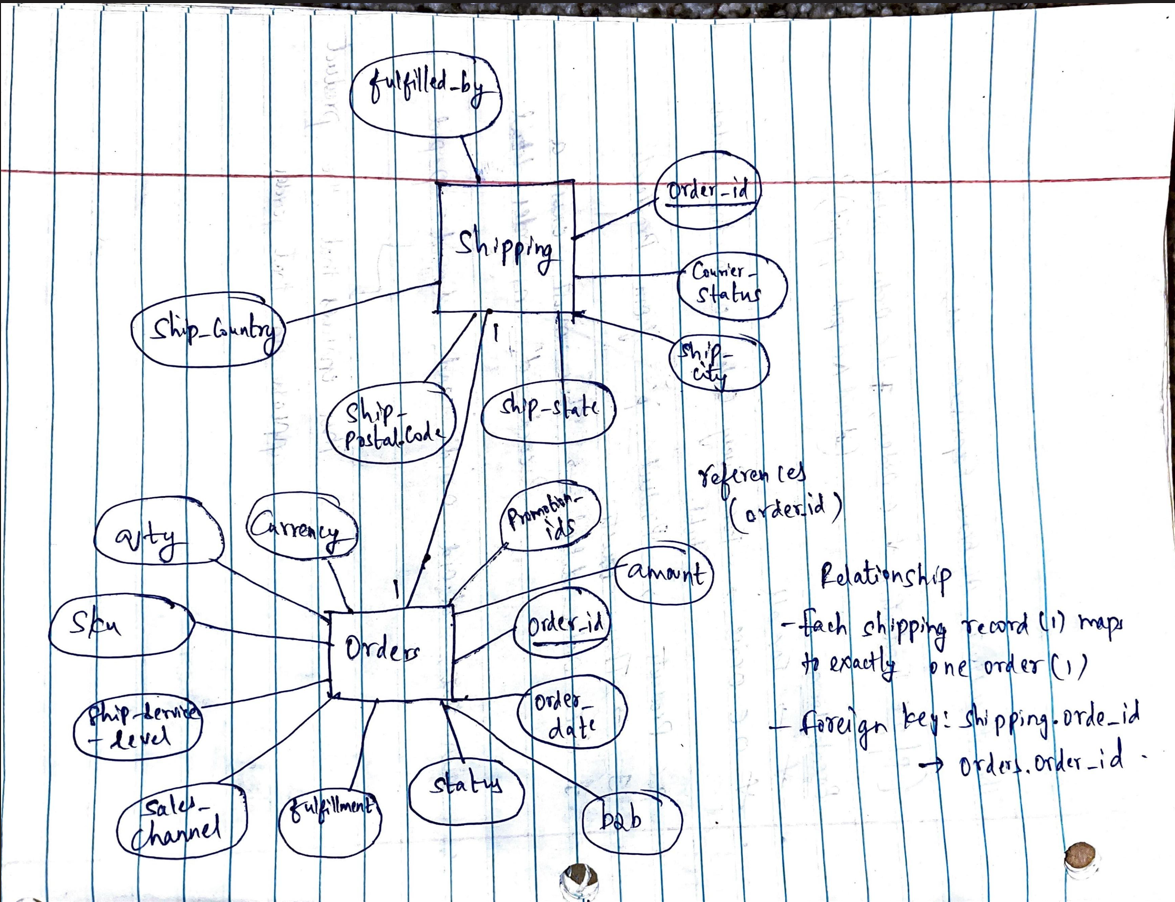
**ER DIAGRAM OF THE TABLE “ORDERS”:**

****

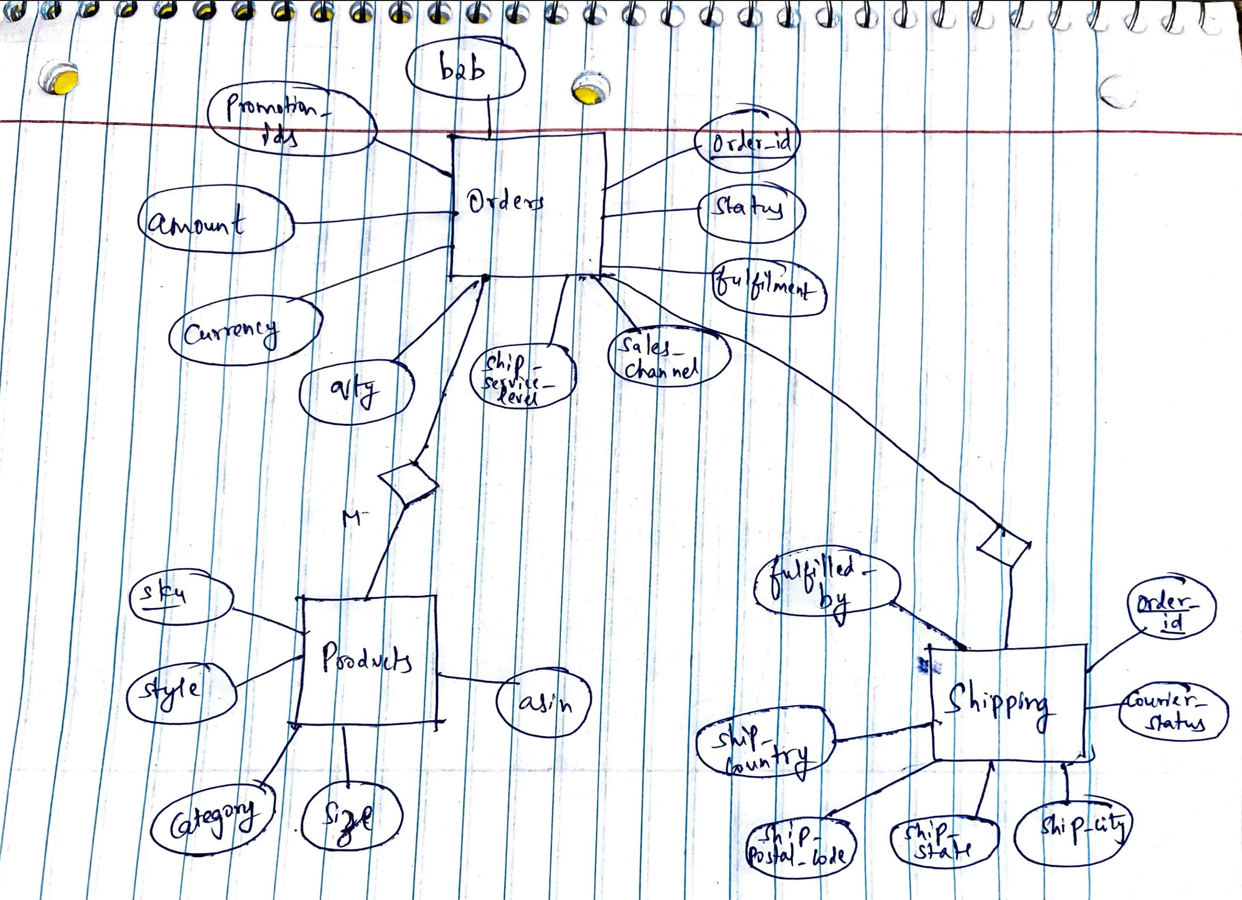
**ER DIAGRAM OF THE TABLE “PRODUCTS”:**

****

**ER DIAGRAM OF THE TABLE “SHIPPING”:**

****

**ER DIAGRAM OF THE OVERALL TABLES COMBINED:**

****

**Database Schema/DDL Statements**

DB NAME -Amazon\_Sales\_Database

SCEHMA NAME – sales

TABLES – orders,products,shipping

We converted the above conceptual design into the following SQL schema:

DROP TABLE IF EXISTS sales.orders;

############# ORDERS ######################

CREATE TABLE IF NOT EXISTS sales.orders

(

order\_id text COLLATE pg\_catalog."default" NOT NULL,

order\_date date,

status text COLLATE pg\_catalog."default",

fulfilment text COLLATE pg\_catalog."default",

sales\_channel text COLLATE pg\_catalog."default",

ship\_service\_level text COLLATE pg\_catalog."default",

sku text COLLATE pg\_catalog."default",

qty integer,

currency text COLLATE pg\_catalog."default",

amount numeric,

promotion\_ids text COLLATE pg\_catalog."default",

b2b boolean,

CONSTRAINT orders\_pkey PRIMARY KEY (order\_id)

)

############# PRODUCTS ######################

DROP TABLE IF EXISTS sales.products;

CREATE TABLE IF NOT EXISTS sales.products

(

sku text COLLATE pg\_catalog."default",

style text COLLATE pg\_catalog."default",

category text COLLATE pg\_catalog."default",

size text COLLATE pg\_catalog."default",

asin text COLLATE pg\_catalog."default"

)

############# SHIPPING ######################

DROP TABLE IF EXISTS sales.shipping;

CREATE TABLE IF NOT EXISTS sales.shipping

(

order\_id text COLLATE pg\_catalog."default" NOT NULL,

courier\_status text COLLATE pg\_catalog."default",

ship\_city text COLLATE pg\_catalog."default",

ship\_state text COLLATE pg\_catalog."default",

ship\_postal\_code text COLLATE pg\_catalog."default",

ship\_country text COLLATE pg\_catalog."default",

fulfilled\_by text COLLATE pg\_catalog."default",

CONSTRAINT shipping\_pkey PRIMARY KEY (order\_id),

CONSTRAINT shipping\_order\_id\_fkey FOREIGN KEY (order\_id)

REFERENCES sales.orders (order\_id) MATCH SIMPLE

ON UPDATE NO ACTION

ON DELETE NO ACTION

)

**DML Statements**

We populated our schema with the following DML statements:

--command " "\\copy sales.orders (order\_id, order\_date, status, fulfilment, sales\_channel, ship\_service\_level, sku, qty, currency, amount, promotion\_ids, b2b) FROM 'C:/Users/mohin/Desktop/UMD/DB\_SYS~1/PROJEC~1/PROJEC~1/orders.csv' DELIMITER ',' CSV HEADER QUOTE '\"' ESCAPE '''';""

--command " "\\copy sales.products (sku, style, category, size, asin) FROM 'C:/Users/mohin/Desktop/UMD/DB\_SYS~1/PROJEC~1/PROJEC~1/products.csv' DELIMITER ',' CSV HEADER QUOTE '\"' ESCAPE '''';""

--command " "\\copy sales.shipping (order\_id, courier\_status, ship\_city, ship\_state, ship\_postal\_code, ship\_country, fulfilled\_by) FROM 'C:/Users/mohin/Desktop/UMD/DB\_SYS~1/PROJEC~1/PROJEC~1/shipping.csv' DELIMITER ',' CSV HEADER QUOTE '\"' ESCAPE '''';""

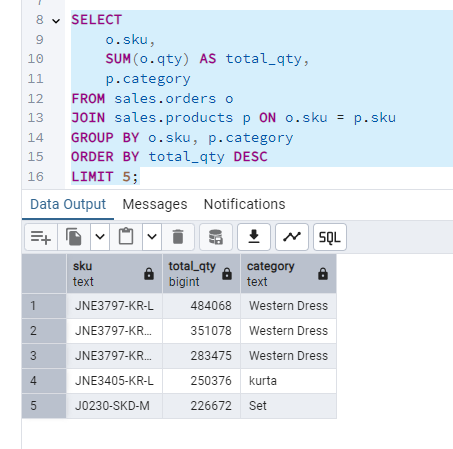
**Analysis Done based on the Concepts taught in Class**

Use Cases based on the three tables

1)Total Revenue by Month



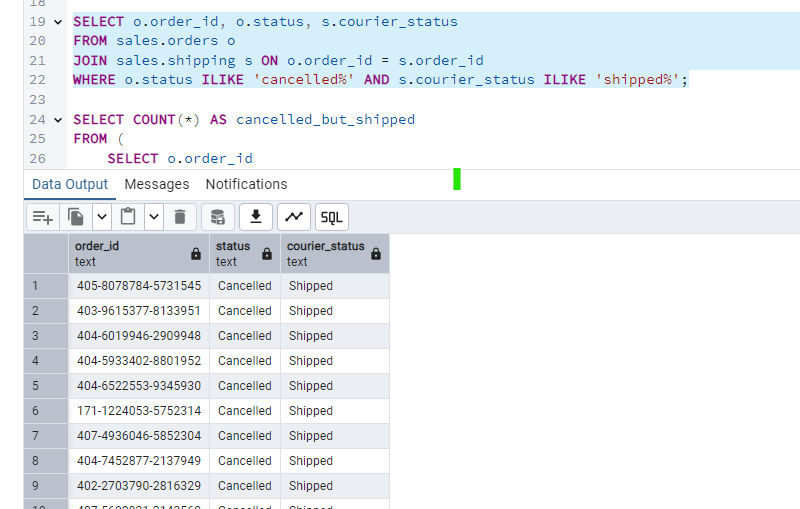
2)Top 5 Best-Selling Products by Quantity and Category



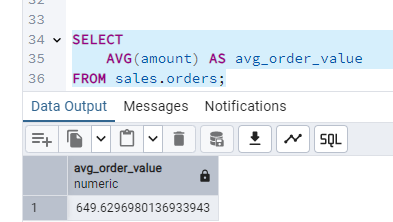
3)Count of Cancelled But Shipped Orders



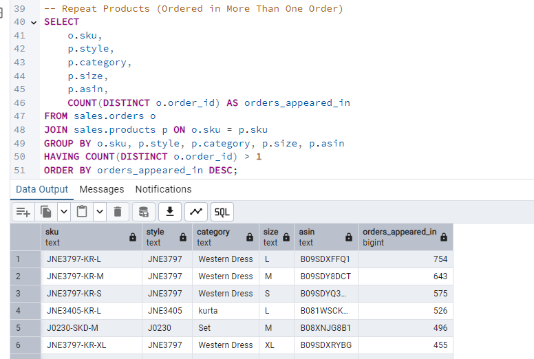
4)Orders That Were Cancelled But Shipped



5)Average Order Value (AOV)



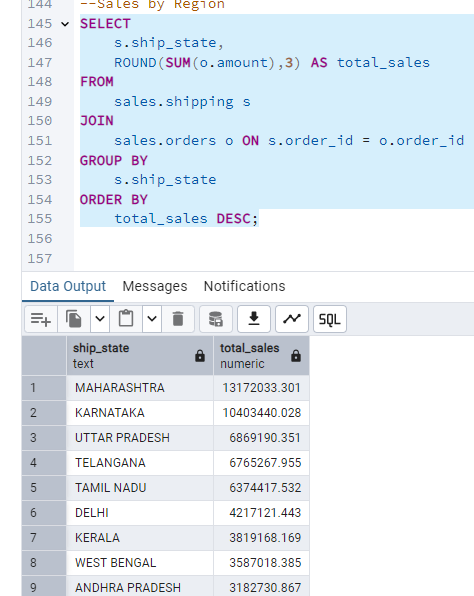
6) Repeat Products (Ordered in More Than One Order)



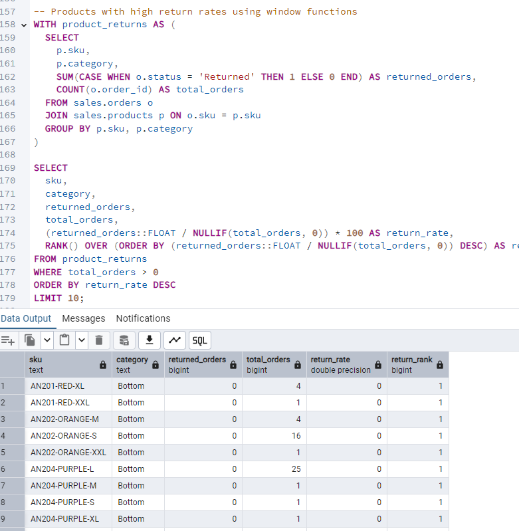
7)Sales by Category

****

8)Sales by Region

****

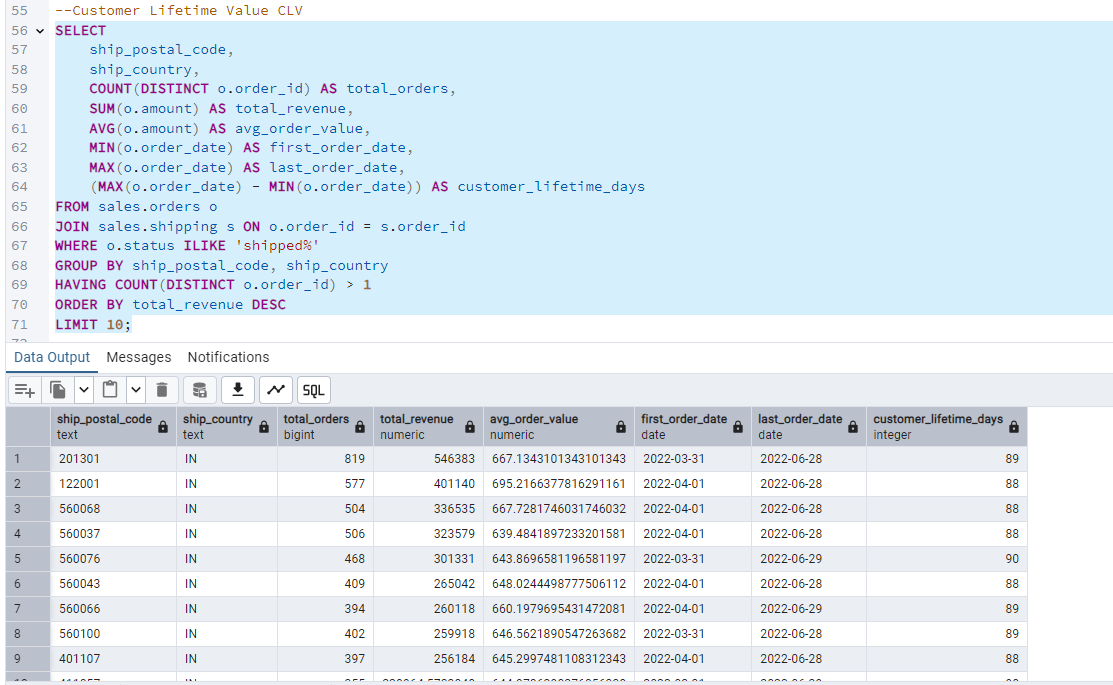
**9) Products with high return rates using window functions**

****

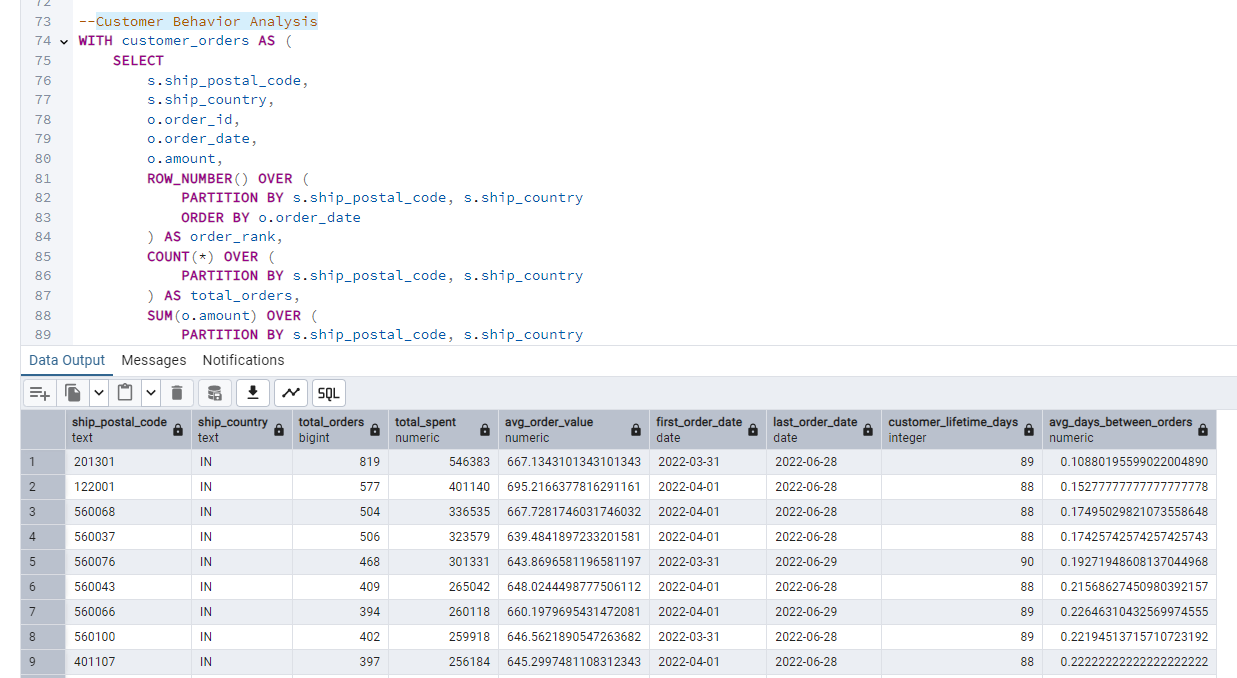
**10)Customer Lifetime Value (CLV)**

To calculate CLV, we’ll need to use the data that we have i.e order\_ids which are customer-specific

* order\_id is customer-specific, OR
* We treat ship\_postal\_code + ship\_country as a proxy for customer identity



**11) Customer Behavior Analysis**



Insights drawn from this query where we have used a **CTE and window functions**

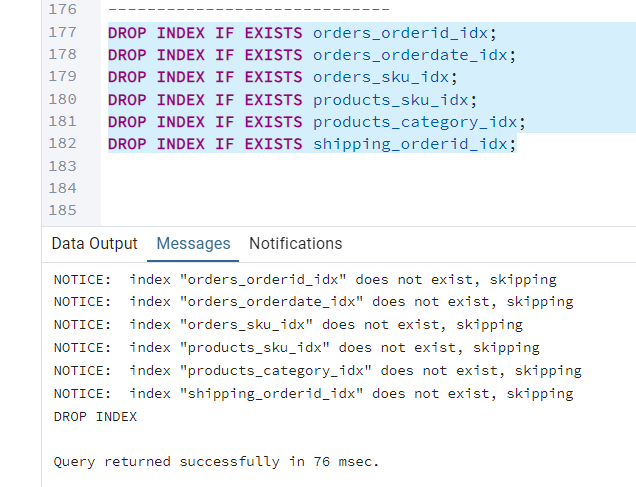
| **Metric** | **What it tells you** |
| --- | --- |
| total\_orders | Purchase frequency |
| total\_spent | CLV (Customer Lifetime Value) |
| avg\_order\_value | Spend per order |
| customer\_lifetime\_days | Duration of active purchasing |
| avg\_days\_between\_orders | Time between orders (buying cycle) |

This query analyzes customer behavior using shipping data as a proxy for customer identity. It leverages CTEs and window functions to calculate metrics like total orders, total spend, average order value, customer lifetime, and average days between purchases. The result helps identify high-value and repeat customers for better targeting and retention strategies.

**Metodology**

To test its execution time, we used the command EXPLAIN ANALYZE <query>.  
We run Q3 in three different situations:

**Scenario 1: Before any statistics collection or indexing:**

****

**Query used:**

EXPLAIN ANALYZE

WITH product\_returns AS (

SELECT

p.sku,

p.category,

SUM(CASE WHEN o.status = 'Returned' THEN 1 ELSE 0 END) AS returned\_orders,

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

FROM sales.orders o

JOIN sales.products p ON o.sku = p.sku

GROUP BY p.sku, p.category

)

SELECT

sku,

category,

returned\_orders,

total\_orders,

(returned\_orders::FLOAT / NULLIF(total\_orders, 0)) \* 100 AS return\_rate,

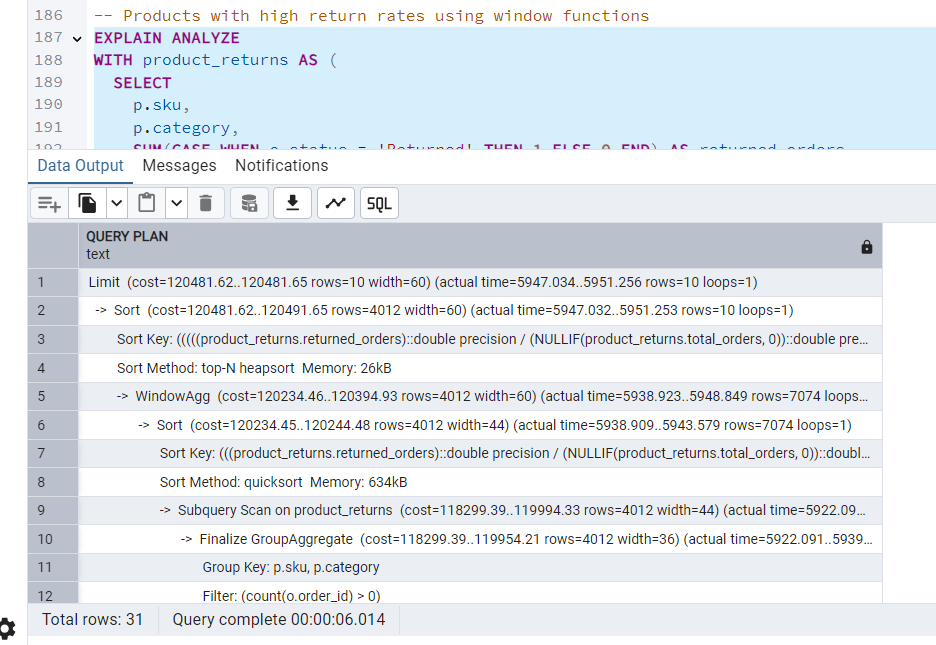
RANK() OVER (ORDER BY (returned\_orders::FLOAT / NULLIF(total\_orders, 0)) DESC) AS return\_rank

FROM product\_returns

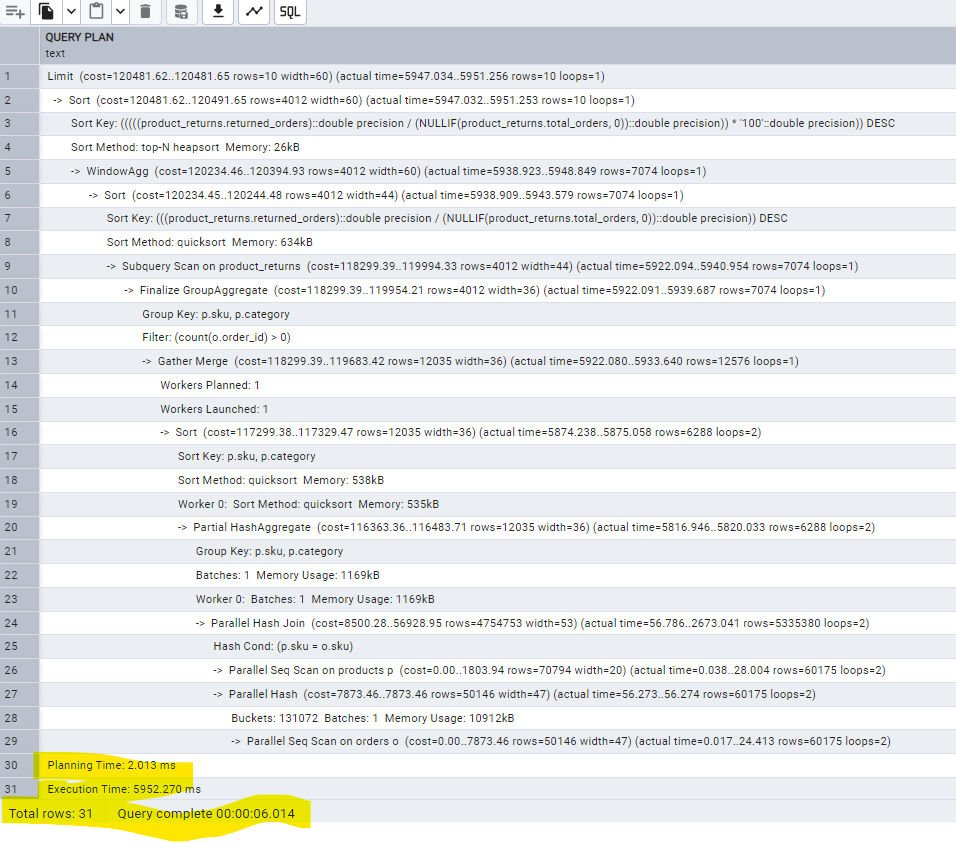
WHERE total\_orders > 0

ORDER BY return\_rate DESC

LIMIT 10;

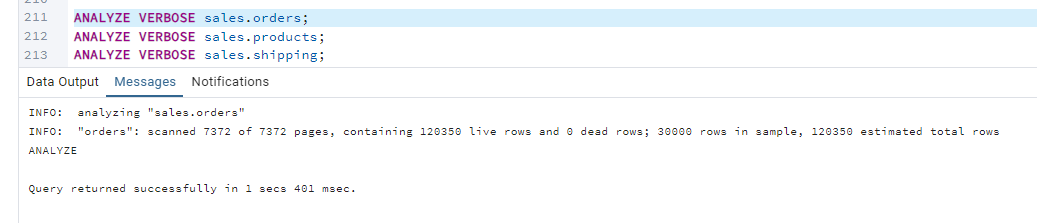
****

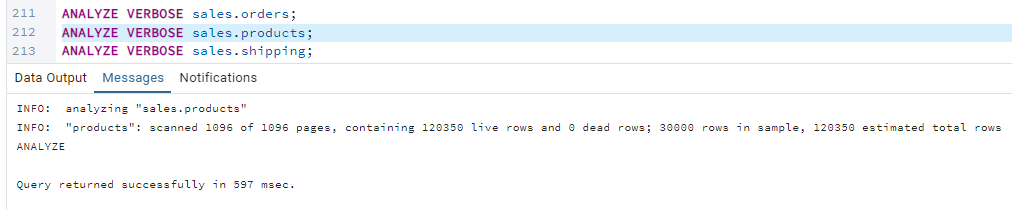
execution plan and timing

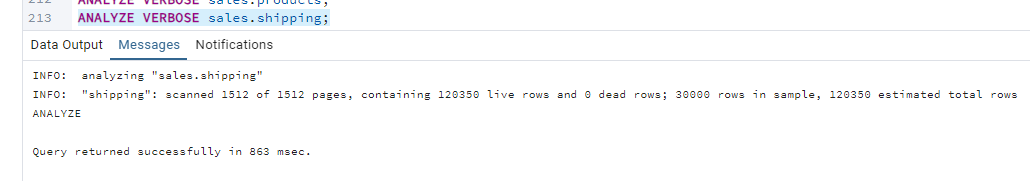
****

**Scenario 2: With Statistics but Without Indexes**

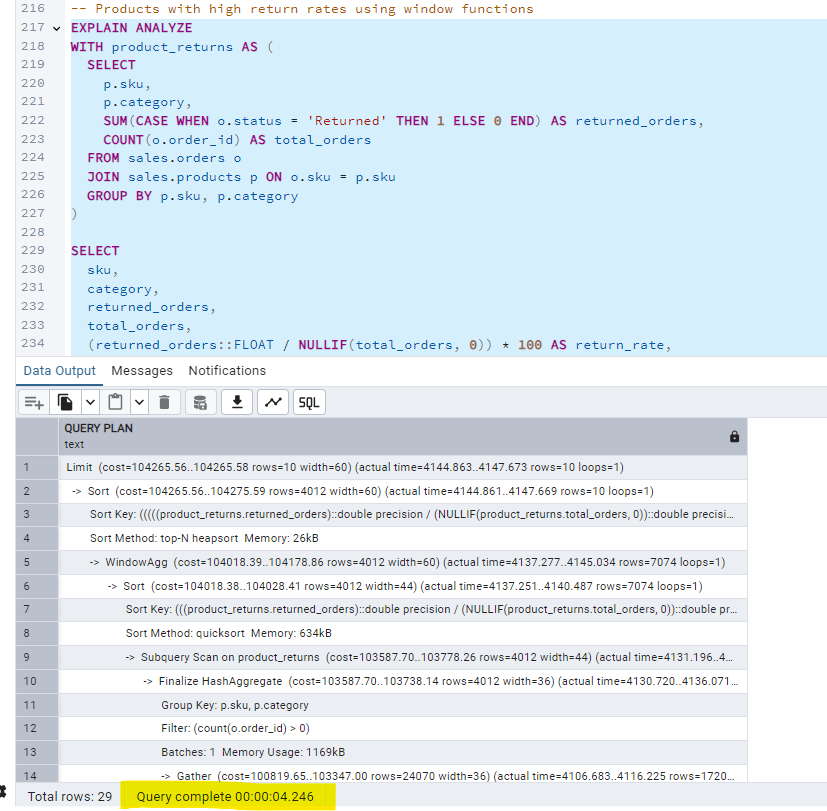
* **Still no indexes.**
* **Collect statistics:**

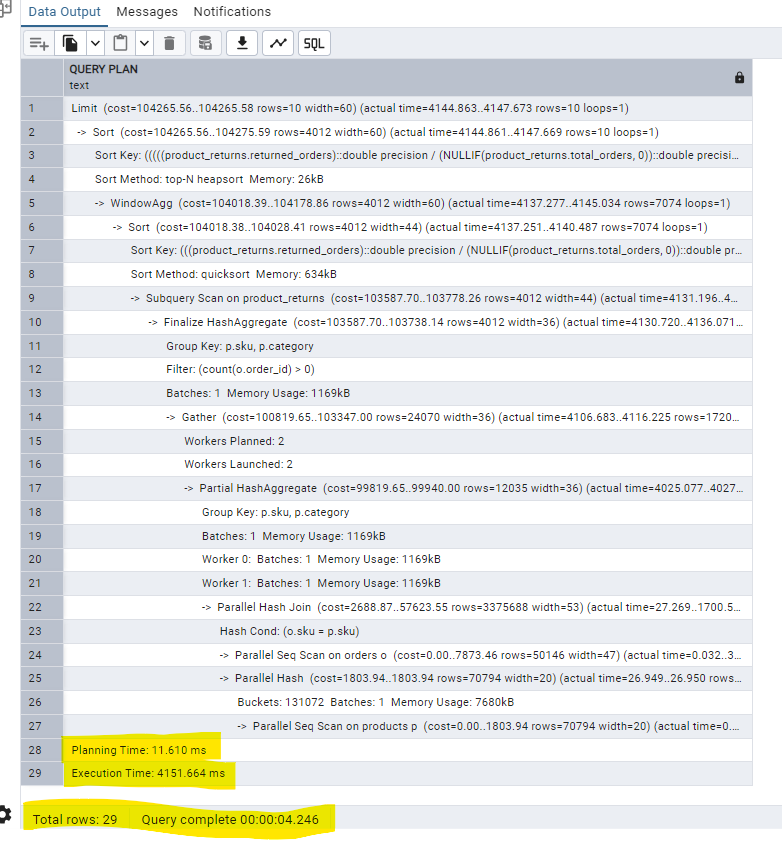
****

****

****

* Running the same query again with EXPLAIN ANALYZE.



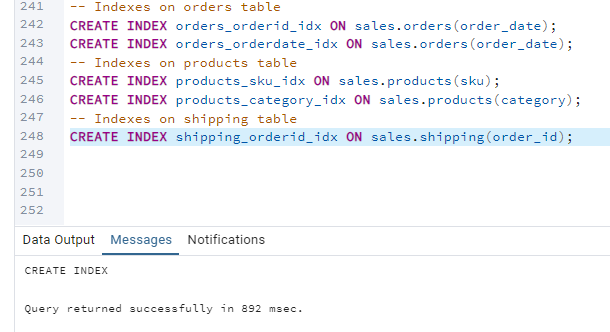


* Compared the estimated row count and execution time —seen improvements in planning decisions.

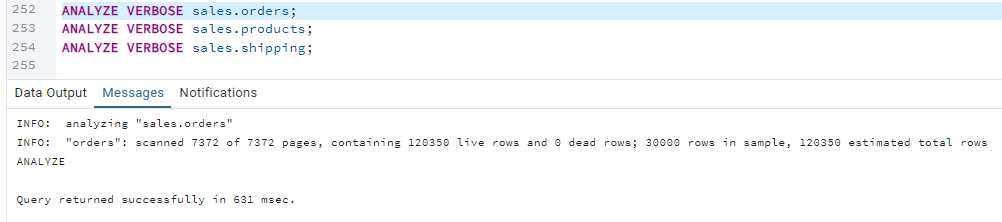
**Scenario 3: With Both Statistics and Indexes**

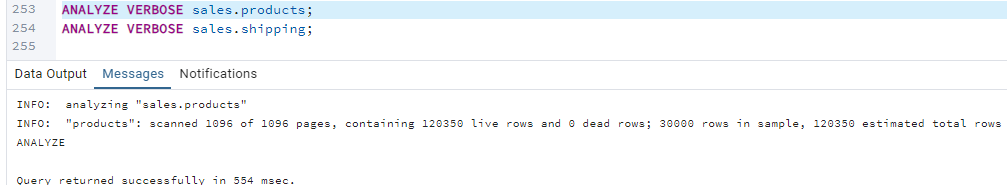
Goal: Full optimization with planner stats + indexes

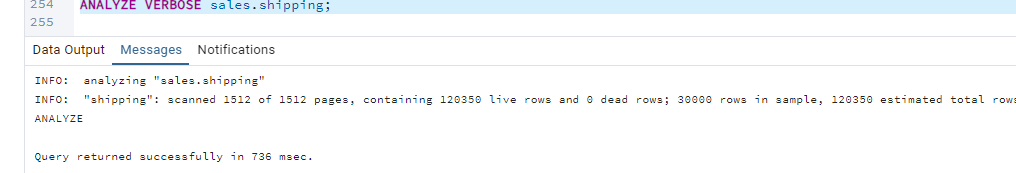
1. Indexes created

****

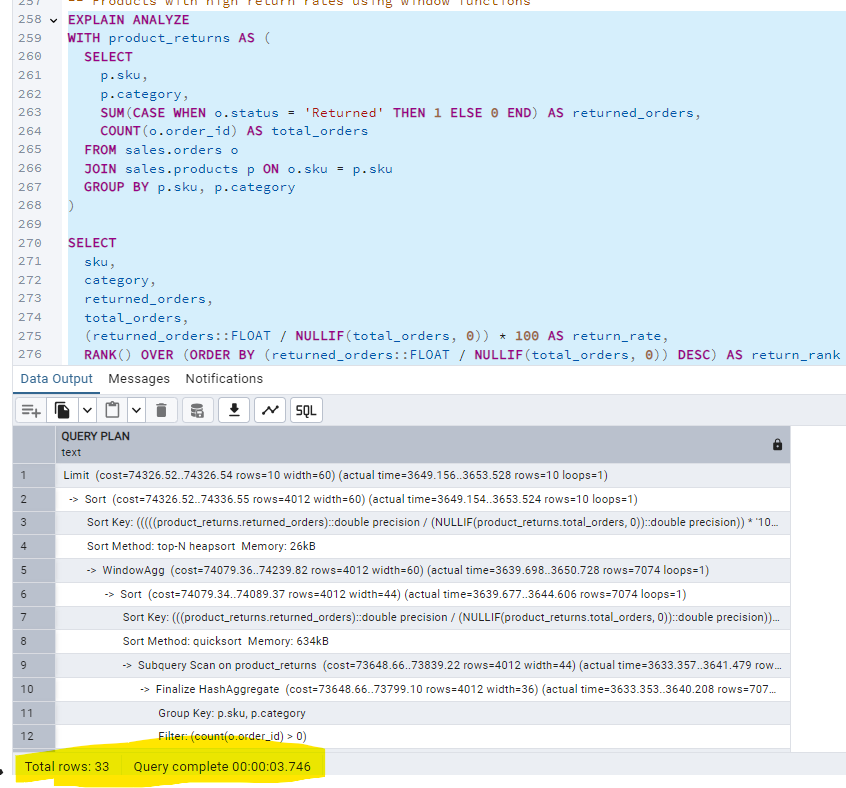
1. Ran ANALYZE again to update stat with indexes



****

****

1. Running query with EXPLAIN ANALYZE again.





**We used the following commands to collect the table statistics:**

ANALYZE VERBOSE sales.orders;

ANALYZE VERBOSE sales.products;

ANALYZE VERBOSE sales.shipping;

**We used the following indexing scheme:**

-- Indexes on orders table

CREATE INDEX orders\_orderid\_idx ON sales.orders(order\_date);

CREATE INDEX orders\_orderdate\_idx ON sales.orders(order\_date);

-- Indexes on products table

CREATE INDEX products\_sku\_idx ON sales.products(sku);

CREATE INDEX products\_category\_idx ON sales.products(category);

-- Indexes on shipping table

CREATE INDEX shipping\_orderid\_idx ON sales.shipping(order\_id);

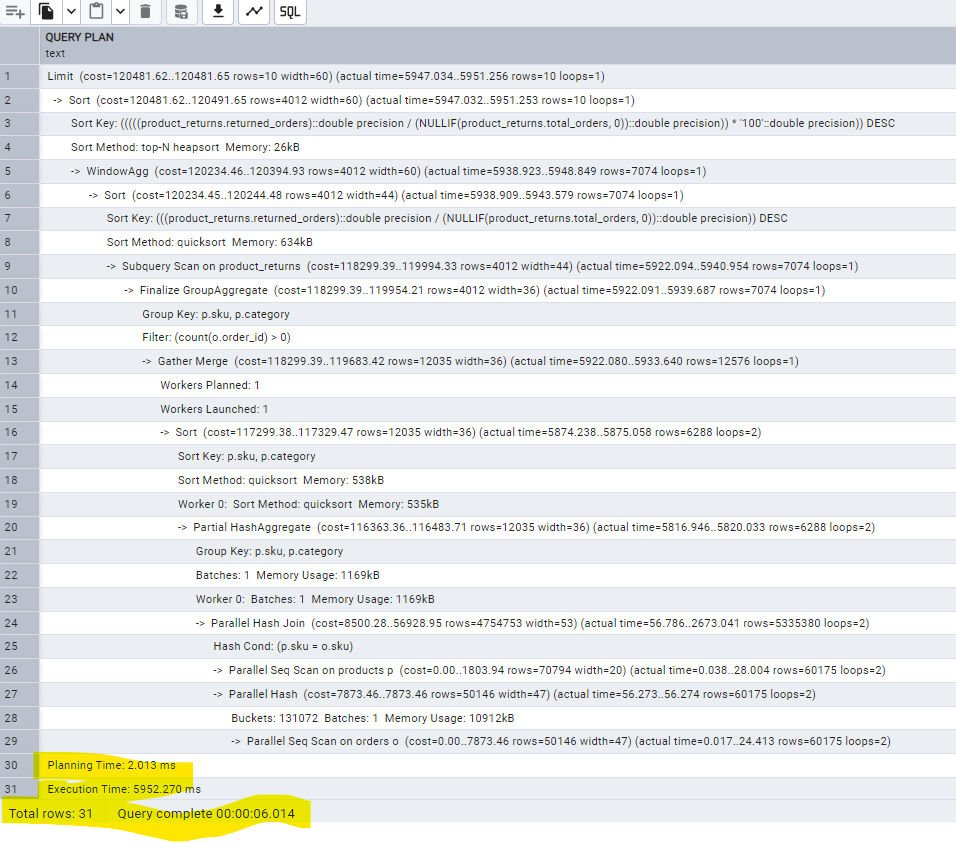
**Benchmarks:**In this section we report the observed performance for each query execution, together with the  
query plans generated by the optimizer. We run each query N times, and averaged...

**QUERY PLAN 1** — Without Indexes or ANALYZE:

Total runtime: 00:00:06.014 (Execution Time: 5952.270 ms)

Scan Type: Parallel Sequential Scan (Parallel Seq Scan)

Join Type: Parallel Hash Join

****

**QUERY PLAN 2** — With ANALYZE + Indexes:

Total runtime:  00:00:03.746 (Execution Time: ~3.746 seconds)

Scan Type: Sequential Scan (Seq Scan), no indexes used

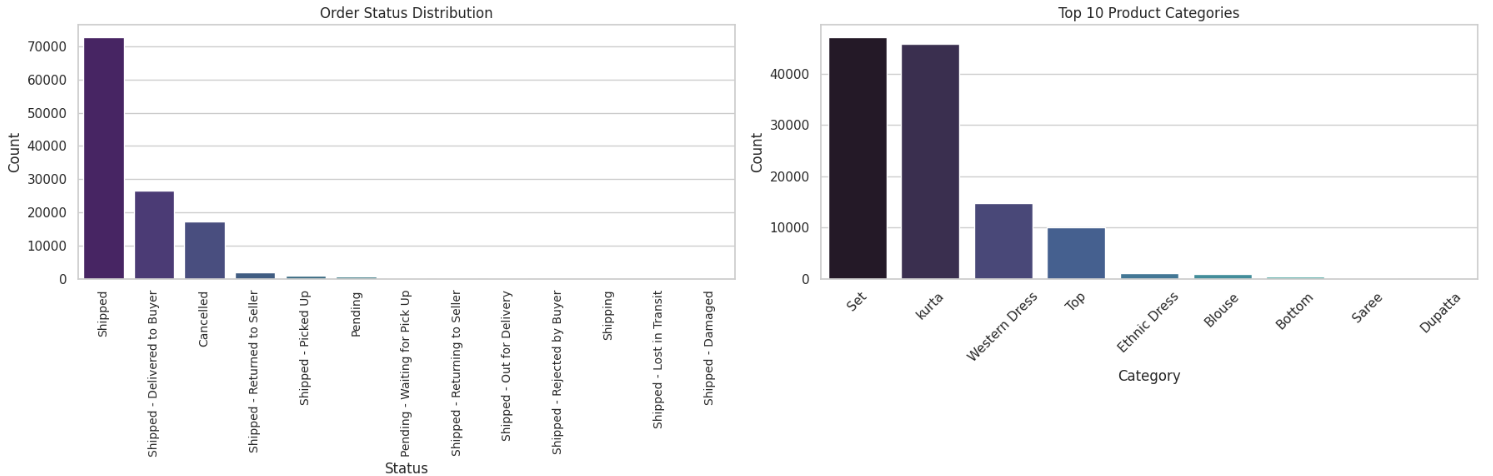
Join Type: Nested Loop Join



**QUERY PLAN IN TABULAR FORM:**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Without Indexes/Statistics** | **With Indexes & Statistics (ANALYZE)** |
| **Execution Time** | **5952.270 ms** | **3746 ms** |
| **Planning Time** | 2.013 ms | 0.961ms |
| **Scan Type - Orders** | Parallel Seq Scan on orders | Parallel Seq Scan on orders |
| **Scan Type - Products** | Parallel Seq Scan on products | Index Scan on products (products\_sku\_idx) |
| **Index Used** | ❌ None | ✅ products\_sku\_idx |
| **Join Type** | Parallel Hash Join | Nested Loop Join with Memoize |
| **Hashing / Caching** | Parallel Hash for join | Memoize (logical) with high cache hits (60K+ hits total) |
| **Sort Method** | Top-N Heapsort, Quicksort | Same: Top-N Heapsort, Quicksort |

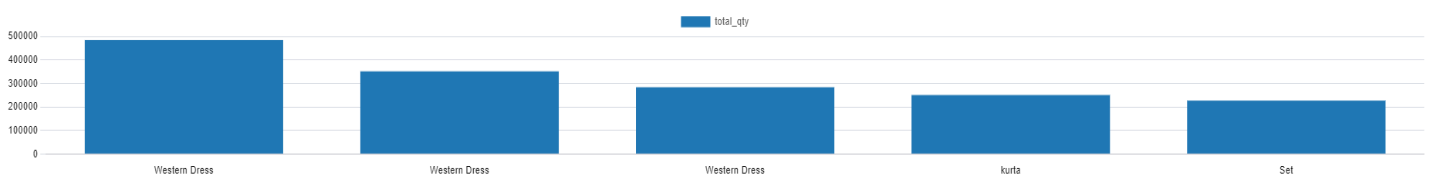
**Below is a plot to summarize our findings…**



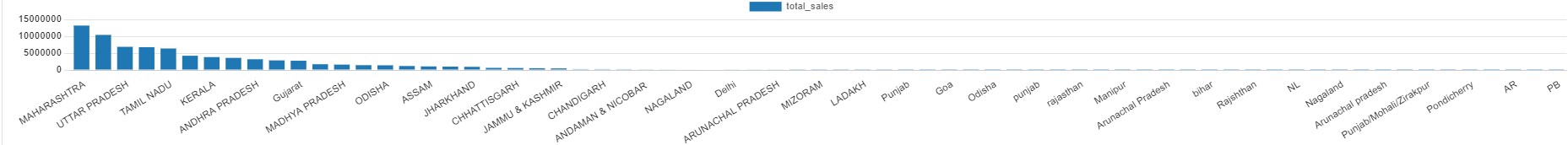
**Total\_revenue\_chart**

****

**Top 5 Best-Selling Products by Quantity**

****

**Sales by Region**

****

**Instructions for reproducing the experiments**

1. Run the python file using Google colab, three clean dataset files will be created out of raw dataset from this python file.
2. Run the DDL\_STATEMENTS.sql file using pgadmin
3. Run the DML\_STATEMENTS.sql file using pgadmin
4. Run the E-COMMERCE\_DATABASE.sql file using pgadmin – this has all the analysis queries, performace queries.

**Final Conclusion**

This project successfully demonstrates the end-to-end process of building, optimizing, and analyzing a mini e-commerce database modelled after Amazon’s sales and fulfillment structure. Starting from a raw dataset of over 128,976 records, we performed extensive data cleaning and transformation, ultimately arriving at a robust and structured dataset of 120,350 high-quality records.

The project involved designing a normalized relational schema grounded in ER modeling, implementing that schema in PostgreSQL, and populating it with realistic sales data. We created three interrelated tables—orders, products, and shipping—under the sales schema, with clear foreign key constraints to maintain data integrity.

A key aspect of the project was performance evaluation. We analyzed the execution time of complex SQL queries under three scenarios: without indexing or statistics, with statistics only, and with both statistics and indexes. Our benchmarking showed a significant performance improvement—over **37% faster execution time**—after applying indexes and collecting statistics using PostgreSQL's ANALYZE command. The optimizer’s use of Memoize and index scans in the fully optimized scenario demonstrated how informed query planning can reduce computational overhead.

We also explored a range of **analytical use cases including**:

* Sales trends by category and region
* Product return rates using window functions
* Customer lifetime value and buying behavior using CTEs
* Best-selling products and revenue forecasting

These insights highlight the power of SQL in deriving business intelligence from structured data.

In conclusion, this project provided hands-on experience in designing scalable database systems, writing analytical queries, applying performance optimization techniques, and using SQL as a tool for both data engineering and business analytics. The methodologies and findings here not only reinforce classroom concepts but also prepare us for real-world database and analytics challenges in e-commerce platforms.