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”

order_id	order_dat	status	fulfilment	sales_char	ship_servic	sku	qty	currency	amount	promotion	b2b
405-80787	04-30-22	Cancelled	Merchant	Amazon.in	Standard	SET389-KR	0	INR	647.62	No Promo	FALSE
171-91981	04-30-22	Shipped - [	Merchant	Amazon.in	Standard	JNE3781-K	1	INR	406	Amazon Pl	FALSE
404-06876	04-30-22	Shipped	Amazon	Amazon.in	Expedited	JNE3371-K	1	INR	329	IN Core Fr	TRUE
403-96153	04-30-22	Cancelled	Merchant	Amazon.in	Standard	J0341-DR-	0	INR	753.33	No Promo	FALSE
407-10697	04-30-22	Shipped	Amazon	Amazon.in	Expedited	JNE3671-T	1	INR	574	No Promo	FALSE
404-14905	04-30-22	Shipped	Amazon	Amazon.in	Expedited	SET264-KR	1	INR	824	IN Core Fr	FALSE
408-57484	04-30-22	Shipped	Amazon	Amazon.in	Expedited	J0095-SET	1	INR	653	IN Core Fr	FALSE
406-78077	04-30-22	Shipped - [	Merchant	Amazon.in	Standard	JNE3405-K	1	INR	399	Amazon Pl	FALSE
407-54430	04-30-22	Cancelled	Amazon	Amazon.in	Expedited	SET200-KR	0	INR	648.5733	IN Core Fr	FALSE
402-43937	04-30-22	Shipped	Amazon	Amazon.in	Expedited	JNE3461-K	1	INR	363	No Promo	FALSE
407-56336	04-30-22	Shipped	Amazon	Amazon.in	Expedited	JNE3160-K	1	INR	685	No Promo	FALSE
171-46384	04-30-22	Shipped	Amazon	Amazon.in	Expedited	JNE3500-K	1	INR	364	No Promo	FALSE
405-55136	04-30-22	Shipped - [	Merchant	Amazon.in	Standard	JNE3405-K	1	INR	399	Amazon Pl	FALSE
408-79556	04-30-22	Shipped	Amazon	Amazon.in	Expedited	SET182-KR	1	INR	657	No Promo	FALSE
408-12983	04-30-22	Shipped - [	Merchant	Amazon.in	Standard	J0351-SET	1	INR	771	Amazon Pl	FALSE
403-49655	04-30-22	Shipped - [	Merchant	Amazon.in	Standard	PJNE3368-	1	INR	544	Amazon Pl	FALSE

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.

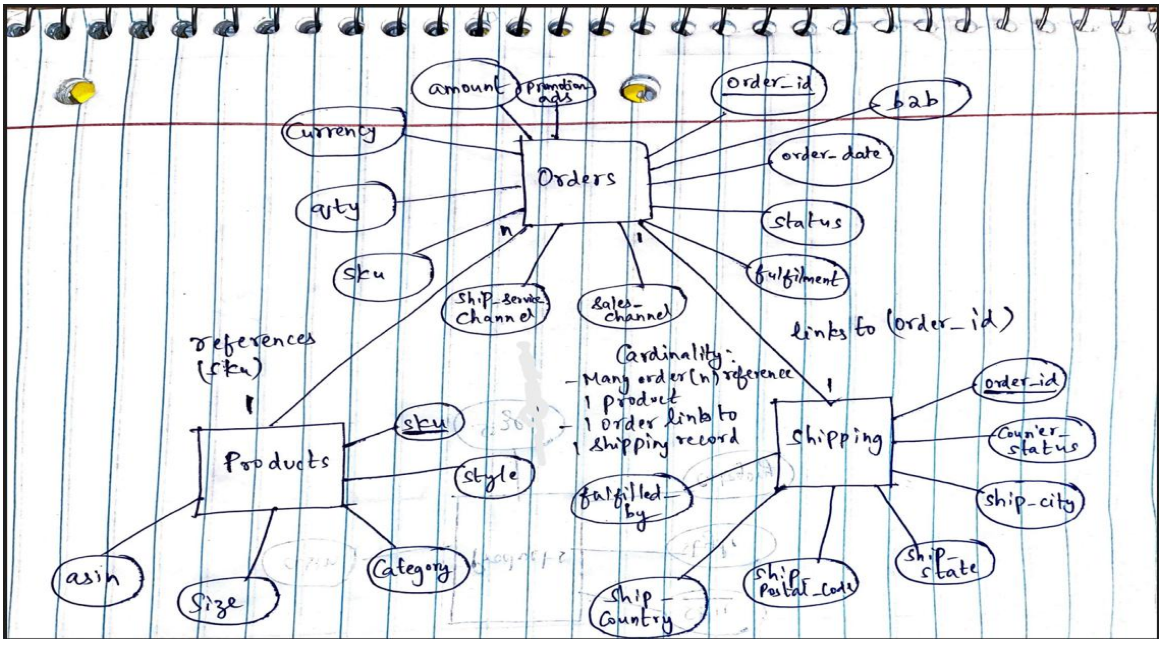
All the code for transforming and cleaning the dataset is included in "e_commerce_database.py"

Conceptual Design

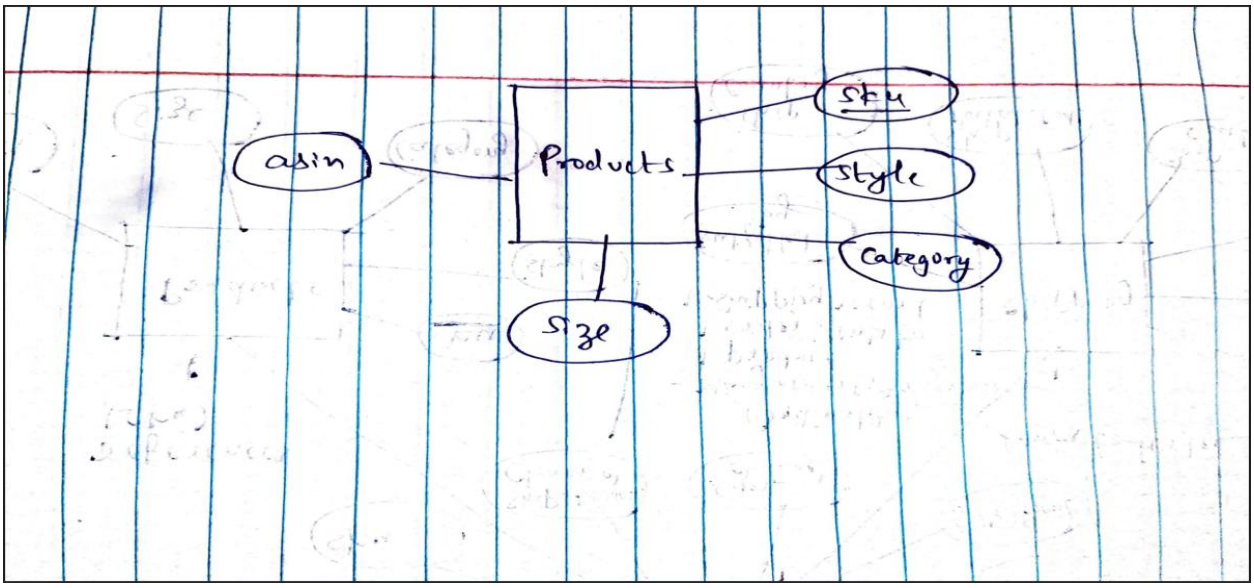
ER Diagram:

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

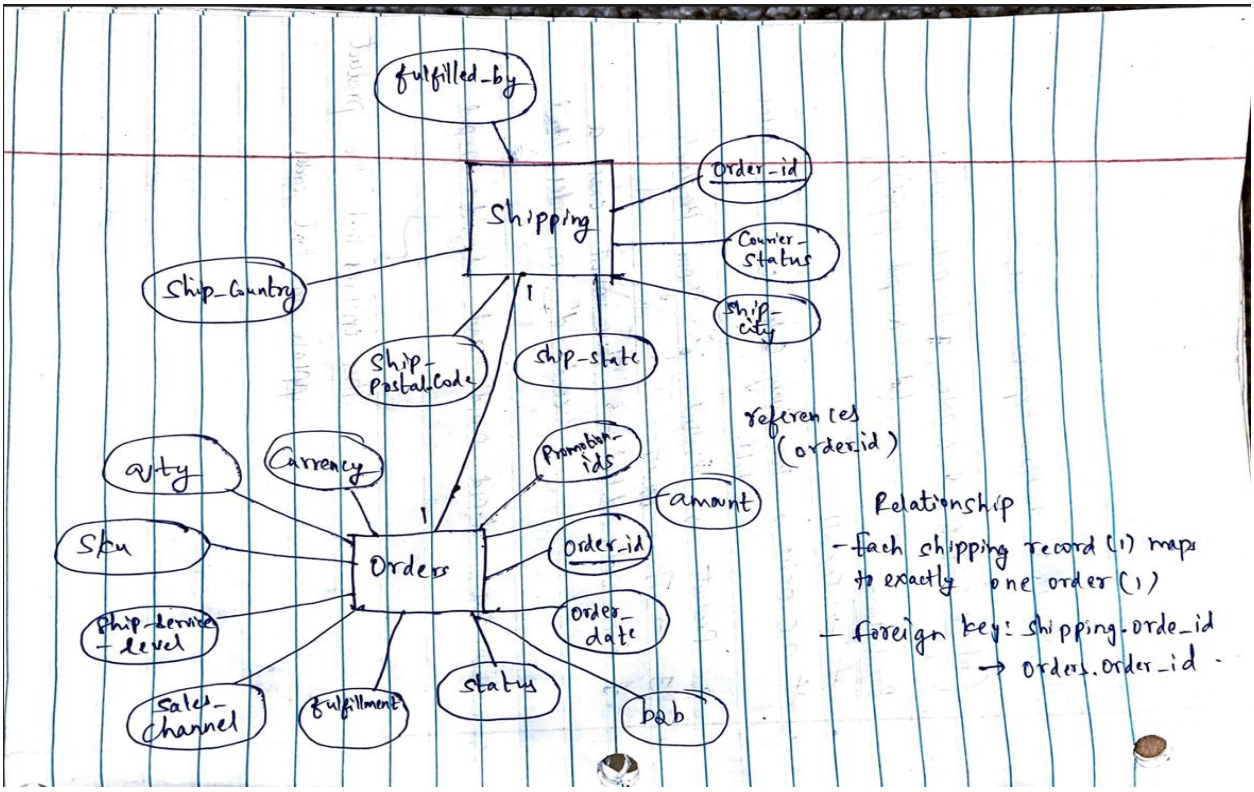
ER DIAGRAM OF THE TABLE "ORDERS":



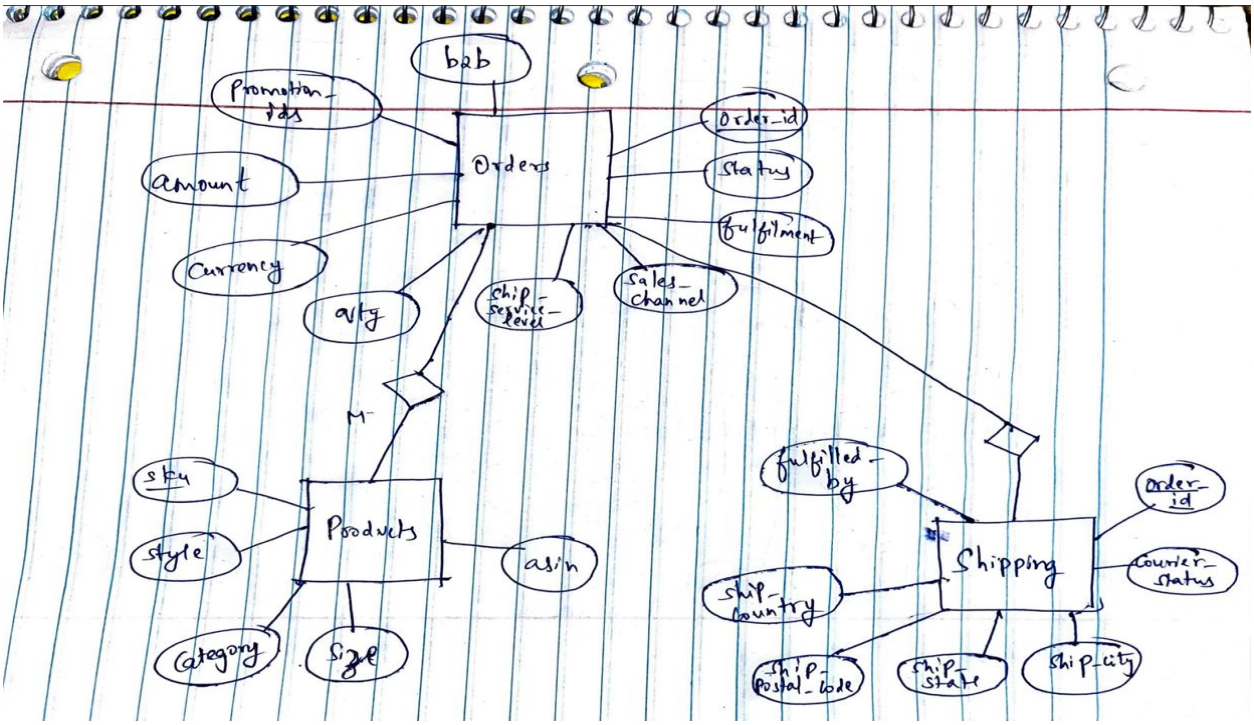
ER DIAGRAM OF THE TABLE “PRODUCTS”:



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:

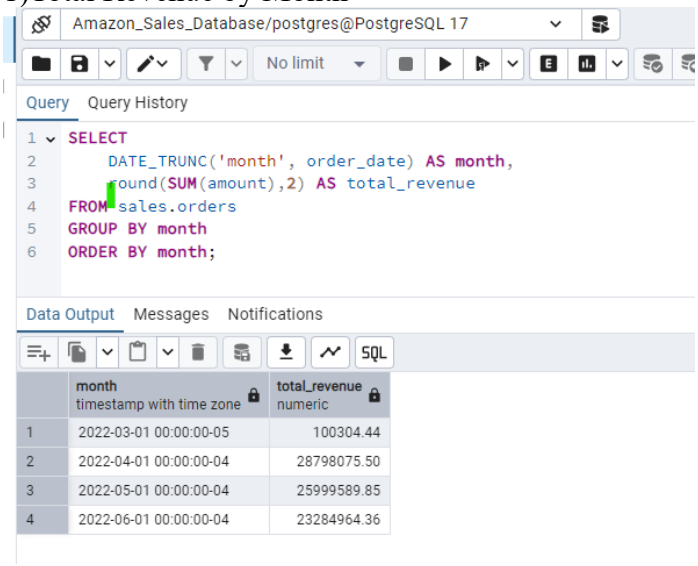
```
--command " "\\copy sales.orders (order_id, order_date, status, fulfilment, sales_channel, ship_service_level, sku, qty, currency, amount, promotion_ids, b2b) FROM 'C:/Users/mohin/Desktop/UMD/DB_SYS~1/PROJEC~1/PROJEC~1/orders.csv' DELIMITER ',' CSV HEADER QUOTE '\"' ESCAPE '"';""
--command " "\\copy 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 '"';""

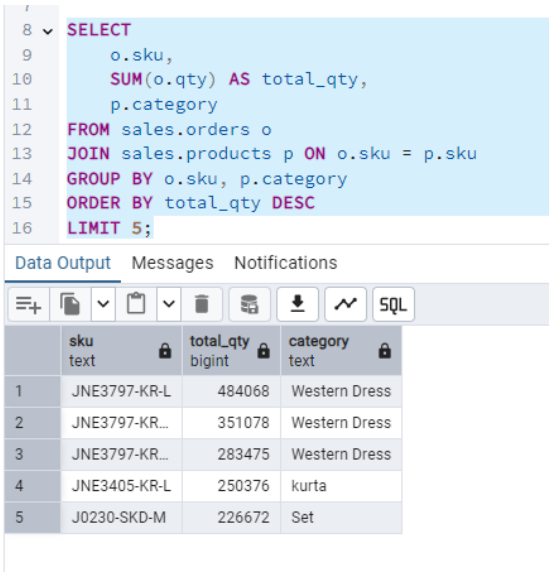
Analysis Done based on the Concepts taught in Class

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

```sql
SELECT COUNT(*) AS cancelled_but_shipped
FROM (
    SELECT o.order_id
    FROM sales.orders o
    JOIN sales.shipping s ON o.order_id = s.order_id
    WHERE o.status ILIKE 'cancelled%'
      AND s.courier_status ILIKE 'shipped%'
) AS cancelled_shipped_orders;
```

Data Output | Messages | Notifications

	cancelled_but_shipped bigint
1	6380

4)Orders That Were Cancelled But Shipped

```sql
SELECT o.order_id, o.status, s.courier_status
FROM sales.orders o
JOIN sales.shipping s ON o.order_id = s.order_id
WHERE o.status ILIKE 'cancelled%' AND s.courier_status ILIKE 'shipped%';

SELECT COUNT(*) AS cancelled_but_shipped
FROM (
    SELECT o.order_id
```

Data Output | Messages | Notifications

	order_id text	status text	courier_status text
1	405-8078784-5731545	Cancelled	Shipped
2	403-9615377-8133951	Cancelled	Shipped
3	404-6019946-2909948	Cancelled	Shipped
4	404-5933402-8801952	Cancelled	Shipped
5	404-6522553-9345930	Cancelled	Shipped
6	171-1224053-5752314	Cancelled	Shipped
7	407-4936046-5852304	Cancelled	Shipped
8	404-7452877-2137949	Cancelled	Shipped
9	402-2703790-2816329	Cancelled	Shipped

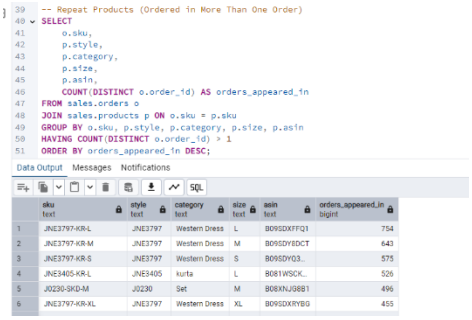
5)Average Order Value (AOV)

```sql
SELECT
    AVG(amount) AS avg_order_value
FROM sales.orders;
```

Data Output | Messages | Notifications

	avg_order_value numeric
1	649.6296980136933943

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



```sql
-- Repeat Products (Ordered in More Than One Order)
SELECT
    o.sku,
    p.style,
    p.category,
    p.size,
    p.asin,
    COUNT(DISTINCT o.order_id) AS orders_appeared_in
FROM sales.orders o
JOIN sales.products p ON o.sku = p.sku
GROUP BY o.sku, p.style, p.category, p.size, p.asin
HAVING COUNT(DISTINCT o.order_id) > 1
ORDER BY orders_appeared_in DESC;
```

Data Output | Messages | Notifications

	sku text	style text	category text	size text	asin text	orders_appeared_in bigint
1	JNE3797-KR-L	JNE3797	Western Dress	L	B09SDXFFQ1	754
2	JNE3797-KR-M	JNE3797	Western Dress	M	B09SDY8DCT	643
3	JNE3797-KR-S	JNE3797	Western Dress	S	B09SDYQ3...	575
4	JNE3405-KR-L	JNE3405	kurta	L	B081WSCK...	526
5	J0230-SKD-M	J0230	Set	M	B08XNJG8B1	496
6	JNE3797-KR-XL	JNE3797	Western Dress	XL	B09SDXRYBG	455

7)Sales by Category

```sql
-- Sales by Category

SELECT
    p.category,
    round(SUM(o.amount),3) AS total_sales
FROM
    sales.orders o
JOIN
    sales.products p ON o.sku = p.sku
GROUP BY
    p.category
ORDER BY
    total_sales DESC;
```

Data Output | Messages | Notifications

	category text	total_sales numeric
1	Set	3168745527.605
2	Western Dress	2184317501.175
3	kurta	1492347239.560
4	Top	261682659.403
5	Ethnic Dress	25903229.994
6	Blouse	8208759.345
7	Bottom	486863.121
8	Saree	465688.009
9	Dupatta	610.000

8)Sales by Region

```sql
--Sales by Region
SELECT
    s.ship_state,
    ROUND(SUM(o.amount),3) AS total_sales
FROM
    sales.shipping s
JOIN
    sales.orders o ON s.order_id = o.order_id
GROUP BY
    s.ship_state
ORDER BY
    total_sales DESC;
```

Data Output | Messages | Notifications

	ship_state text	total_sales numeric
1	MAHARASHTRA	13172033.301
2	KARNATAKA	10403440.028
3	UTTAR PRADESH	6869190.351
4	TELANGANA	6765267.955
5	TAMIL NADU	6374417.532
6	DELHI	4217121.443
7	KERALA	3819168.169
8	WEST BENGAL	3587018.385
9	ANDHRA PRADESH	3182730.867

9) Products with high return rates using window functions

```sql
-- Products with high return rates using window functions
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 r
FROM product_returns
WHERE total_orders > 0
ORDER BY return_rate DESC
LIMIT 10;
```

Data Output | Messages | Notifications

	sku text	category text	returned_orders bigint	total_orders bigint	return_rate double precision	return_rank bigint
1	AN201-RED-XL	Bottom	0	4	0	1
2	AN201-RED-XXL	Bottom	0	1	0	1
3	AN202-ORANGE-M	Bottom	0	4	0	1
4	AN202-ORANGE-S	Bottom	0	16	0	1
5	AN202-ORANGE-XXL	Bottom	0	1	0	1
6	AN204-PURPLE-L	Bottom	0	25	0	1
7	AN204-PURPLE-M	Bottom	0	1	0	1
8	AN204-PURPLE-S	Bottom	0	1	0	1
9	AN204-PURPLE-XL	Bottom	0	1	0	1

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

```sql
--Customer Lifetime Value CLV
SELECT
    ship_postal_code,
    ship_country,
    COUNT(DISTINCT o.order_id) AS total_orders,
    SUM(o.amount) AS total_revenue,
    AVG(o.amount) AS avg_order_value,
    MIN(o.order_date) AS first_order_date,
    MAX(o.order_date) AS last_order_date,
    (MAX(o.order_date) - MIN(o.order_date)) AS customer_lifetime_days
FROM sales.orders o
JOIN sales.shipping s ON o.order_id = s.order_id
WHERE o.status ILIKE 'shipped%'
GROUP BY ship_postal_code, ship_country
HAVING COUNT(DISTINCT o.order_id) > 1
ORDER BY total_revenue DESC
LIMIT 10;
```

Data Output | Messages | Notifications

	ship_postal_code text	ship_country text	total_orders bigint	total_revenue numeric	avg_order_value numeric	first_order_date date	last_order_date date	customer_lifetime_days integer
1	201301	IN	819	546383	667.1343101343101343	2022-03-31	2022-06-28	89
2	122001	IN	577	401140	695.2166377816291161	2022-04-01	2022-06-28	88
3	560068	IN	504	336535	667.7281746031746032	2022-04-01	2022-06-28	88
4	560037	IN	506	323579	639.4841897233201581	2022-04-01	2022-06-28	88
5	560076	IN	468	301331	643.8696581196581197	2022-03-31	2022-06-29	90
6	560043	IN	409	265042	648.0244498777506112	2022-04-01	2022-06-28	88
7	560066	IN	394	260118	660.1979695431472081	2022-04-01	2022-06-29	89
8	560100	IN	402	259918	646.5621890547263682	2022-03-31	2022-06-28	89
9	401107	IN	397	256184	645.2997481108312343	2022-04-01	2022-06-28	88

11) Customer Behavior Analysis

```sql
--Customer Behavior Analysis
WITH customer_orders AS (
    SELECT
        s.ship_postal_code,
        s.ship_country,
        o.order_id,
        o.order_date,
        o.amount,
        ROW_NUMBER() OVER (
            PARTITION BY s.ship_postal_code, s.ship_country
            ORDER BY o.order_date
        ) AS order_rank,
        COUNT(*) OVER (
            PARTITION BY s.ship_postal_code, s.ship_country
        ) AS total_orders,
        SUM(o.amount) OVER (
            PARTITION BY s.ship_postal_code, s.ship_country
```

Data Output | Messages | Notifications

	ship_postal_code text	ship_country text	total_orders bigint	total_spent numeric	avg_order_value numeric	first_order_date date	last_order_date date	customer_lifetime_days integer	avg_days_between_orders numeric
1	201301	IN	819	546383	667.1343101343101343	2022-03-31	2022-06-28	89	0.10880195599022004890
2	122001	IN	577	401140	695.2166377816291161	2022-04-01	2022-06-28	88	0.15277777777777777778
3	560068	IN	504	336535	667.7281746031746032	2022-04-01	2022-06-28	88	0.17495029821073558648
4	560037	IN	506	323579	639.4841897233201581	2022-04-01	2022-06-28	88	0.17425742574257425743
5	560076	IN	468	301331	643.8696581196581197	2022-03-31	2022-06-29	90	0.19271948608137044968
6	560043	IN	409	265042	648.0244498777506112	2022-04-01	2022-06-28	88	0.21568627450980392157
7	560066	IN	394	260118	660.1979695431472081	2022-04-01	2022-06-29	89	0.22646310432569974555
8	560100	IN	402	259918	646.5621890547263682	2022-03-31	2022-06-28	89	0.22194513715710723192
9	401107	IN	397	256184	645.2997481108312343	2022-04-01	2022-06-28	88	0.22222222222222222222

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:

```
176  ------------------------------
177  DROP INDEX IF EXISTS orders_orderid_idx;
178  DROP INDEX IF EXISTS orders_orderdate_idx;
179  DROP INDEX IF EXISTS orders_sku_idx;
180  DROP INDEX IF EXISTS products_sku_idx;
181  DROP INDEX IF EXISTS products_category_idx;
182  DROP INDEX IF EXISTS shipping_orderid_idx;
183
184
185
```

Data Output | Messages | Notifications

```
NOTICE:  index "orders_orderid_idx" does not exist, skipping
NOTICE:  index "orders_orderdate_idx" does not exist, skipping
NOTICE:  index "orders_sku_idx" does not exist, skipping
NOTICE:  index "products_sku_idx" does not exist, skipping
NOTICE:  index "products_category_idx" does not exist, skipping
NOTICE:  index "shipping_orderid_idx" does not exist, skipping
DROP INDEX

Query returned successfully in 76 msec.
```

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;

```
186  -- Products with high return rates using window functions
187  EXPLAIN ANALYZE
188  WITH product_returns AS (
189    SELECT
190      p.sku,
191      p.category,
```

Data Output | Messages | Notifications

SQL

	QUERY PLAN text
1	Limit (cost=120481.62..120481.65 rows=10 width=60) (actual time=5947.034..5951.256 rows=10 loops=1)
2	-> Sort (cost=120481.62..120491.65 rows=4012 width=60) (actual time=5947.032..5951.253 rows=10 loops=1)
3	Sort Key: (((((product_returns.returned_orders)::double precision / (NULLIF(product_returns.total_orders, 0))::double pre...
4	Sort Method: top-N heapsort Memory: 26kB
5	-> WindowAgg (cost=120234.46..120394.93 rows=4012 width=60) (actual time=5938.923..5948.849 rows=7074 loops...
6	-> Sort (cost=120234.45..120244.48 rows=4012 width=44) (actual time=5938.909..5943.579 rows=7074 loops=1)
7	Sort Key: (((product_returns.returned_orders)::double precision / (NULLIF(product_returns.total_orders, 0))::doubl...
8	Sort Method: quicksort Memory: 634kB
9	-> Subquery Scan on product_returns (cost=118299.39..119994.33 rows=4012 width=44) (actual time=5922.09...
10	-> Finalize GroupAggregate (cost=118299.39..119954.21 rows=4012 width=36) (actual time=5922.091..5939...
11	Group Key: p.sku, p.category
12	Filter: (count(o.order_id) > 0)

Total rows: 31 | Query complete 00:00:06.014

execution plan and timing

	QUERY PLAN text
1	Limit (cost=120481.62..120481.65 rows=10 width=60) (actual time=5947.034..5951.256 rows=10 loops=1)
2	-> Sort (cost=120481.62..120491.65 rows=4012 width=60) (actual time=5947.032..5951.253 rows=10 loops=1)
3	Sort Key: (((((product_returns.returned_orders)::double precision / (NULLIF(product_returns.total_orders, 0))::double precision)) * '100'::double precision)) DESC
4	Sort Method: top-N heapsort Memory: 26kB
5	-> WindowAgg (cost=120234.46..120394.93 rows=4012 width=60) (actual time=5938.923..5948.849 rows=7074 loops=1)
6	-> Sort (cost=120234.45..120244.48 rows=4012 width=44) (actual time=5938.909..5943.579 rows=7074 loops=1)
7	Sort Key: (((product_returns.returned_orders)::double precision / (NULLIF(product_returns.total_orders, 0))::double precision)) DESC
8	Sort Method: quicksort Memory: 634kB
9	-> Subquery Scan on product_returns (cost=118299.39..119994.33 rows=4012 width=44) (actual time=5922.094..5940.954 rows=7074 loops=1)
10	-> Finalize GroupAggregate (cost=118299.39..119954.21 rows=4012 width=36) (actual time=5922.091..5939.687 rows=7074 loops=1)
11	Group Key: p.sku, p.category
12	Filter: (count(o.order_id) > 0)
13	-> Gather Merge (cost=118299.39..119683.42 rows=12035 width=36) (actual time=5922.080..5933.640 rows=12576 loops=1)
14	Workers Planned: 1
15	Workers Launched: 1
16	-> Sort (cost=117299.38..117329.47 rows=12035 width=36) (actual time=5874.238..5875.058 rows=6288 loops=2)
17	Sort Key: p.sku, p.category
18	Sort Method: quicksort Memory: 538kB
19	Worker 0: Sort Method: quicksort Memory: 535kB
20	-> Partial HashAggregate (cost=116363.36..116483.71 rows=12035 width=36) (actual time=5816.946..5820.033 rows=6288 loops=2)
21	Group Key: p.sku, p.category
22	Batches: 1 Memory Usage: 1169kB
23	Worker 0: Batches: 1 Memory Usage: 1169kB
24	-> Parallel Hash Join (cost=8500.28..56928.95 rows=4754753 width=53) (actual time=56.786..2673.041 rows=5335380 loops=2)
25	Hash Cond: (p.sku = o.sku)
26	-> Parallel Seq Scan on products p (cost=0.00..1803.94 rows=70794 width=20) (actual time=0.038..28.004 rows=60175 loops=2)
27	-> Parallel Hash (cost=7873.46..7873.46 rows=50146 width=47) (actual time=56.273..56.274 rows=60175 loops=2)
28	Buckets: 131072 Batches: 1 Memory Usage: 10912kB
29	-> Parallel Seq Scan on orders o (cost=0.00..7873.46 rows=50146 width=47) (actual time=0.017..24.413 rows=60175 loops=2)
30	Planning Time: 2.013 ms
31	Execution Time: 5952.270 ms

Total rows: 31 Query complete 00:00:06.014

Scenario 2: With Statistics but Without Indexes

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

```
211  ANALYZE VERBOSE sales.orders;
212  ANALYZE VERBOSE sales.products;
213  ANALYZE VERBOSE sales.shipping;
```

Data Output | Messages | Notifications

```
INFO:  analyzing "sales.orders"
INFO:  "orders": scanned 7372 of 7372 pages, containing 120350 live rows and 0 dead rows; 30000 rows in sample, 120350 estimated total rows
ANALYZE

Query returned successfully in 1 secs 401 msec.
```

```
211  ANALYZE VERBOSE sales.orders;
212  ANALYZE VERBOSE sales.products;
213  ANALYZE VERBOSE sales.shipping;
```

Data Output | Messages | Notifications

```
INFO:  analyzing "sales.products"
INFO:  "products": scanned 1096 of 1096 pages, containing 120350 live rows and 0 dead rows; 30000 rows in sample, 120350 estimated total rows
ANALYZE

Query returned successfully in 597 msec.
```

```
213  ANALYZE VERBOSE sales.shipping;
```

Data Output | Messages | Notifications

```
INFO:  analyzing "sales.shipping"
INFO:  "shipping": scanned 1512 of 1512 pages, containing 120350 live rows and 0 dead rows; 30000 rows in sample, 120350 estimated total rows
ANALYZE

Query returned successfully in 863 msec.
```

✓ Running the same query again with EXPLAIN ANALYZE.

```
216  -- Products with high return rates using window functions
217  EXPLAIN ANALYZE
218  WITH product_returns AS (
219    SELECT
220      p.sku,
221      p.category,
222      SUM(CASE WHEN o.status = 'Returned' THEN 1 ELSE 0 END) AS returned_orders,
223      COUNT(o.order_id) AS total_orders
224    FROM sales.orders o
225    JOIN sales.products p ON o.sku = p.sku
226    GROUP BY p.sku, p.category
227  )
228
229  SELECT
230    sku,
231    category,
232    returned_orders,
233    total_orders,
234    (returned_orders::FLOAT / NULLIF(total_orders, 0)) * 100 AS return_rate,
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Limit (cost=104265.56..104265.58 rows=10 width=60) (actual time=4144.863..4147.673 rows=10 loops=1)
2	-> Sort (cost=104265.56..104275.59 rows=4012 width=60) (actual time=4144.861..4147.669 rows=10 loops=1)
3	Sort Key: ((((product_returns.returned_orders)::double precision / (NULLIF(product_returns.total_orders, 0))::double precisi...
4	Sort Method: top-N heapsort Memory: 26kB
5	-> WindowAgg (cost=104018.39..104178.86 rows=4012 width=60) (actual time=4137.277..4145.034 rows=7074 loops=1)
6	-> Sort (cost=104018.38..104028.41 rows=4012 width=44) (actual time=4137.251..4140.487 rows=7074 loops=1)
7	Sort Key: (((product_returns.returned_orders)::double precision / (NULLIF(product_returns.total_orders, 0))::double pr...
8	Sort Method: quicksort Memory: 634kB
9	-> Subquery Scan on product_returns (cost=103587.70..103778.26 rows=4012 width=44) (actual time=4131.196..4...
10	-> Finalize HashAggregate (cost=103587.70..103738.14 rows=4012 width=36) (actual time=4130.720..4136.071...
11	Group Key: p.sku, p.category
12	Filter: (count(o.order_id) > 0)
13	Batches: 1 Memory Usage: 1169kB
14	-> Gather (cost=100819.65..103347.00 rows=24070 width=36) (actual time=4106.683..4116.225 rows=1720...

Total rows: 29 Query complete 00:00:04.246

Data Output | Messages | Notifications

	QUERY PLAN text
1	Limit (cost=104265.56..104265.58 rows=10 width=60) (actual time=4144.863..4147.673 rows=10 loops=1)
2	-> Sort (cost=104265.56..104275.59 rows=4012 width=60) (actual time=4144.861..4147.669 rows=10 loops=1)
3	Sort Key: ((((product_returns.returned_orders)::double precision / (NULLIF(product_returns.total_orders, 0))::double precisi...
4	Sort Method: top-N heapsort Memory: 26kB
5	-> WindowAgg (cost=104018.39..104178.86 rows=4012 width=60) (actual time=4137.277..4145.034 rows=7074 loops=1)
6	-> Sort (cost=104018.38..104028.41 rows=4012 width=44) (actual time=4137.251..4140.487 rows=7074 loops=1)
7	Sort Key: (((product_returns.returned_orders)::double precision / (NULLIF(product_returns.total_orders, 0))::double pr...
8	Sort Method: quicksort Memory: 634kB
9	-> Subquery Scan on product_returns (cost=103587.70..103778.26 rows=4012 width=44) (actual time=4131.196..4...
10	-> Finalize HashAggregate (cost=103587.70..103738.14 rows=4012 width=36) (actual time=4130.720..4136.071...
11	Group Key: p.sku, p.category
12	Filter: (count(o.order_id) > 0)
13	Batches: 1 Memory Usage: 1169kB
14	-> Gather (cost=100819.65..103347.00 rows=24070 width=36) (actual time=4106.683..4116.225 rows=1720...
15	Workers Planned: 2
16	Workers Launched: 2
17	-> Partial HashAggregate (cost=99819.65..99940.00 rows=12035 width=36) (actual time=4025.077..4027...
18	Group Key: p.sku, p.category
19	Batches: 1 Memory Usage: 1169kB
20	Worker 0: Batches: 1 Memory Usage: 1169kB
21	Worker 1: Batches: 1 Memory Usage: 1169kB
22	-> Parallel Hash Join (cost=2688.87..57623.55 rows=3375688 width=53) (actual time=27.269..1700.5...
23	Hash Cond: (o.sku = p.sku)
24	-> Parallel Seq Scan on orders o (cost=0.00..7873.46 rows=50146 width=47) (actual time=0.032..3...
25	-> Parallel Hash (cost=1803.94..1803.94 rows=70794 width=20) (actual time=26.949..26.950 rows...
26	Buckets: 131072 Batches: 1 Memory Usage: 7680kB
27	-> Parallel Seq Scan on products p (cost=0.00..1803.94 rows=70794 width=20) (actual time=0....
28	Planning Time: 11.610 ms
29	Execution Time: 4151.664 ms

Total rows: 29 Query complete 00:00:04.246

✓ 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

```
241  -- Indexes on orders table
242  CREATE INDEX orders_orderid_idx ON sales.orders(order_date);
243  CREATE INDEX orders_orderdate_idx ON sales.orders(order_date);
244  -- Indexes on products table
245  CREATE INDEX products_sku_idx ON sales.products(sku);
246  CREATE INDEX products_category_idx ON sales.products(category);
247  -- Indexes on shipping table
248  CREATE INDEX shipping_orderid_idx ON sales.shipping(order_id);
249
250
251
252
```

Data Output | Messages | Notifications

```
CREATE INDEX

Query returned successfully in 892 msec.
```

2) Ran ANALYZE again to update stat with indexes

```
252  ANALYZE VERBOSE sales.orders;
253  ANALYZE VERBOSE sales.products;
254  ANALYZE VERBOSE sales.shipping;
255
```

Data Output | Messages | Notifications

```
INFO:  analyzing "sales.orders"
INFO:  "orders": scanned 7372 of 7372 pages, containing 120350 live rows and 0 dead rows; 30000 rows in sample, 120350 estimated total rows
ANALYZE

Query returned successfully in 631 msec.
```

```
253  ANALYZE VERBOSE sales.products;
254  ANALYZE VERBOSE sales.shipping;
255
```

Data Output | Messages | Notifications

```
INFO:  analyzing "sales.products"
INFO:  "products": scanned 1096 of 1096 pages, containing 120350 live rows and 0 dead rows; 30000 rows in sample, 120350 estimated total rows
ANALYZE

Query returned successfully in 554 msec.
```

```
254  ANALYZE VERBOSE sales.shipping;
255
```

Data Output | Messages | Notifications

```
INFO:  analyzing "sales.shipping"
INFO:  "shipping": scanned 1512 of 1512 pages, containing 120350 live rows and 0 dead rows; 30000 rows in sample, 120350 estimated total rows
ANALYZE

Query returned successfully in 736 msec.
```

3) Running query with EXPLAIN ANALYZE again.

```sql
-- Products with high return rates using window functions
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
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Limit (cost=74326.52..74326.54 rows=10 width=60) (actual time=3649.156..3653.528 rows=10 loops=1)
2	-> Sort (cost=74326.52..74336.55 rows=4012 width=60) (actual time=3649.154..3653.524 rows=10 loops=1)
3	Sort Key: (((((product_returns.returned_orders)::double precision / (NULLIF(product_returns.total_orders, 0))::double precision)) * '10...
4	Sort Method: top-N heapsort Memory: 26kB
5	-> WindowAgg (cost=74079.36..74239.82 rows=4012 width=60) (actual time=3639.698..3650.728 rows=7074 loops=1)
6	-> Sort (cost=74079.34..74089.37 rows=4012 width=44) (actual time=3639.677..3644.606 rows=7074 loops=1)
7	Sort Key: (((product_returns.returned_orders)::double precision / (NULLIF(product_returns.total_orders, 0))::double precision))...
8	Sort Method: quicksort Memory: 634kB
9	-> Subquery Scan on product_returns (cost=73648.66..73839.22 rows=4012 width=44) (actual time=3633.357..3641.479 row...
10	-> Finalize HashAggregate (cost=73648.66..73799.10 rows=4012 width=36) (actual time=3633.353..3640.208 rows=707...
11	Group Key: p.sku, p.category
12	Filter: (count(o.order_id) > 0)

Total rows: 33 Query complete 00:00:03.746

	QUERY PLAN text
1	Limit (cost=74326.52..74326.54 rows=10 width=60) (actual time=3649.156..3653.528 rows=10 loops=1)
2	-> Sort (cost=74326.52..74336.55 rows=4012 width=60) (actual time=3649.154..3653.524 rows=10 loops=1)
3	Sort Key: (((((product_returns.returned_orders)::double precision / (NULLIF(product_returns.total_orders, 0))::double precision)) * '10...
4	Sort Method: top-N heapsort Memory: 26kB
5	-> WindowAgg (cost=74079.36..74239.82 rows=4012 width=60) (actual time=3639.698..3650.728 rows=7074 loops=1)
6	-> Sort (cost=74079.34..74089.37 rows=4012 width=44) (actual time=3639.677..3644.606 rows=7074 loops=1)
7	Sort Key: (((product_returns.returned_orders)::double precision / (NULLIF(product_returns.total_orders, 0))::double precision))...
8	Sort Method: quicksort Memory: 634kB
9	-> Subquery Scan on product_returns (cost=73648.66..73839.22 rows=4012 width=44) (actual time=3633.357..3641.479 row...
10	-> Finalize HashAggregate (cost=73648.66..73799.10 rows=4012 width=36) (actual time=3633.353..3640.208 rows=707...
11	Group Key: p.sku, p.category
12	Filter: (count(o.order_id) > 0)
13	Batches: 1 Memory Usage: 1169kB
14	-> Gather (cost=70880.61..73407.96 rows=24070 width=36) (actual time=3603.363..3614.645 rows=17187 loops=1)
15	Workers Planned: 2
16	Workers Launched: 2
17	-> Partial HashAggregate (cost=69880.61..70000.96 rows=12035 width=36) (actual time=3539.571..3541.624 row...
18	Group Key: p.sku, p.category
19	Batches: 1 Memory Usage: 1169kB
20	Worker 0: Batches: 1 Memory Usage: 1169kB
21	Worker 1: Batches: 1 Memory Usage: 1169kB
22	-> Nested Loop (cost=0.30..26265.26 rows=3489228 width=53) (actual time=0.167..1355.816 rows=3556920 l...
23	-> Parallel Seq Scan on orders o (cost=0.00..7873.46 rows=50146 width=47) (actual time=0.013..22.027 ro...
24	-> Memoize (cost=0.30..0.73 rows=22 width=20) (actual time=0.004..0.014 rows=89 loops=120350)
25	Cache Key: o.sku
26	Cache Mode: logical
27	Hits: 36095 Misses: 5784 Evictions: 0 Overflows: 0 Memory Usage: 6461kB
28	Worker 0: Hits: 33327 Misses: 5696 Evictions: 0 Overflows: 0 Memory Usage: 6444kB
29	Worker 1: Hits: 33741 Misses: 5707 Evictions: 0 Overflows: 0 Memory Usage: 6449kB
30	-> Index Scan using products_sku_idx on products p (cost=0.29..0.72 rows=22 width=20) (actual time=0...
31	Index Cond: (sku = o.sku)

Total rows: 33 Query complete 00:00:03.746

We used the following commands to collect the table statistics:

ANALYZE VERBOSE sales.orders;
ANALYZE VERBOSE sales.products;
ANALYZE VERBOSE sales.shipping;

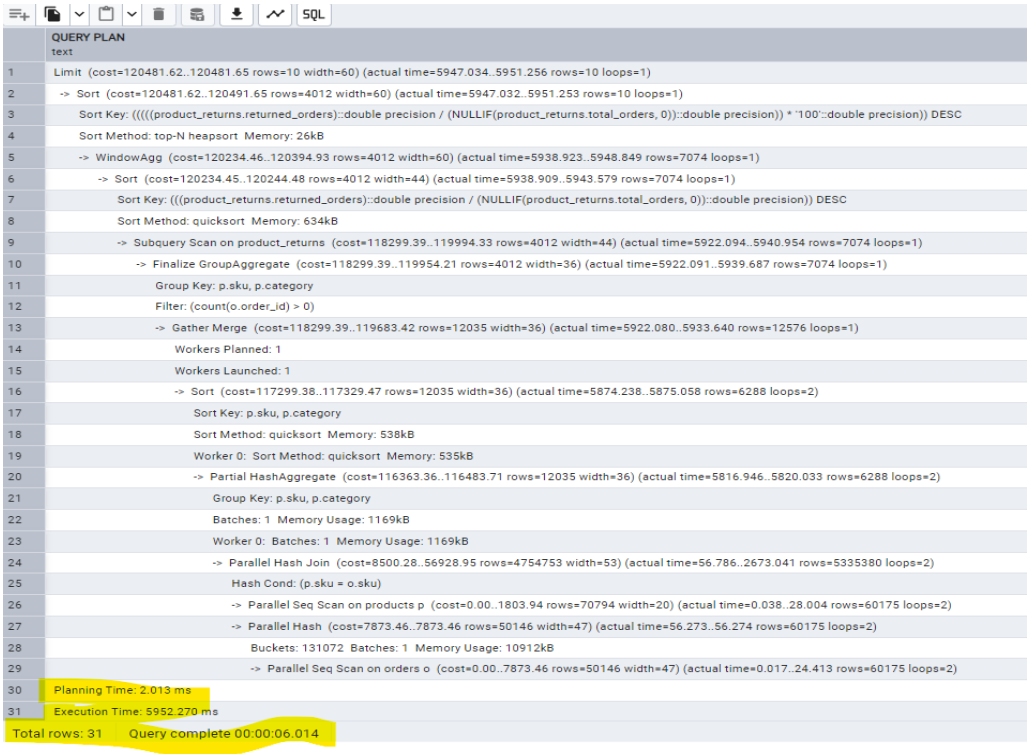
We used the following indexing scheme:

```
-- 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 text
1	Limit (cost=120481.62..120481.65 rows=10 width=60) (actual time=5947.034..5951.256 rows=10 loops=1)
2	-> Sort (cost=120481.62..120491.65 rows=4012 width=60) (actual time=5947.032..5951.253 rows=10 loops=1)
3	Sort Key: (((((product_returns.returned_orders)::double precision / (NULLIF(product_returns.total_orders, 0))::double precision)) * '100'::double precision)) DESC
4	Sort Method: top-N heapsort Memory: 26kB
5	-> WindowAgg (cost=120234.46..120394.93 rows=4012 width=60) (actual time=5938.923..5948.849 rows=7074 loops=1)
6	-> Sort (cost=120234.45..120244.48 rows=4012 width=44) (actual time=5938.909..5943.579 rows=7074 loops=1)
7	Sort Key: (((product_returns.returned_orders)::double precision / (NULLIF(product_returns.total_orders, 0))::double precision)) DESC
8	Sort Method: quicksort Memory: 634kB
9	-> Subquery Scan on product_returns (cost=118299.39..119994.33 rows=4012 width=44) (actual time=5922.094..5940.954 rows=7074 loops=1)
10	-> Finalize GroupAggregate (cost=118299.39..119954.21 rows=4012 width=36) (actual time=5922.091..5939.687 rows=7074 loops=1)
11	Group Key: p.sku, p.category
12	Filter: (count(o.order_id) > 0)
13	-> Gather Merge (cost=118299.39..119683.42 rows=12035 width=36) (actual time=5922.080..5933.640 rows=12576 loops=1)
14	Workers Planned: 1
15	Workers Launched: 1
16	-> Sort (cost=117299.38..117329.47 rows=12035 width=36) (actual time=5874.238..5875.058 rows=6288 loops=2)
17	Sort Key: p.sku, p.category
18	Sort Method: quicksort Memory: 538kB
19	Worker 0: Sort Method: quicksort Memory: 535kB
20	-> Partial HashAggregate (cost=116363.36..116483.71 rows=12035 width=36) (actual time=5816.946..5820.033 rows=6288 loops=2)
21	Group Key: p.sku, p.category
22	Batches: 1 Memory Usage: 1169kB
23	Worker 0: Batches: 1 Memory Usage: 1169kB
24	-> Parallel Hash Join (cost=8500.28..56928.95 rows=4754753 width=53) (actual time=56.786..2673.041 rows=5335380 loops=2)
25	Hash Cond: (p.sku = o.sku)
26	-> Parallel Seq Scan on products p (cost=0.00..1803.94 rows=70794 width=20) (actual time=0.038..28.004 rows=60175 loops=2)
27	-> Parallel Hash (cost=7873.46..7873.46 rows=50146 width=47) (actual time=56.273..56.274 rows=60175 loops=2)
28	Buckets: 131072 Batches: 1 Memory Usage: 10912kB
29	-> Parallel Seq Scan on orders o (cost=0.00..7873.46 rows=50146 width=47) (actual time=0.017..24.413 rows=60175 loops=2)
30	Planning Time: 2.013 ms
31	Execution Time: 5952.270 ms

Total rows: 31 Query complete 00:00:06.014

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

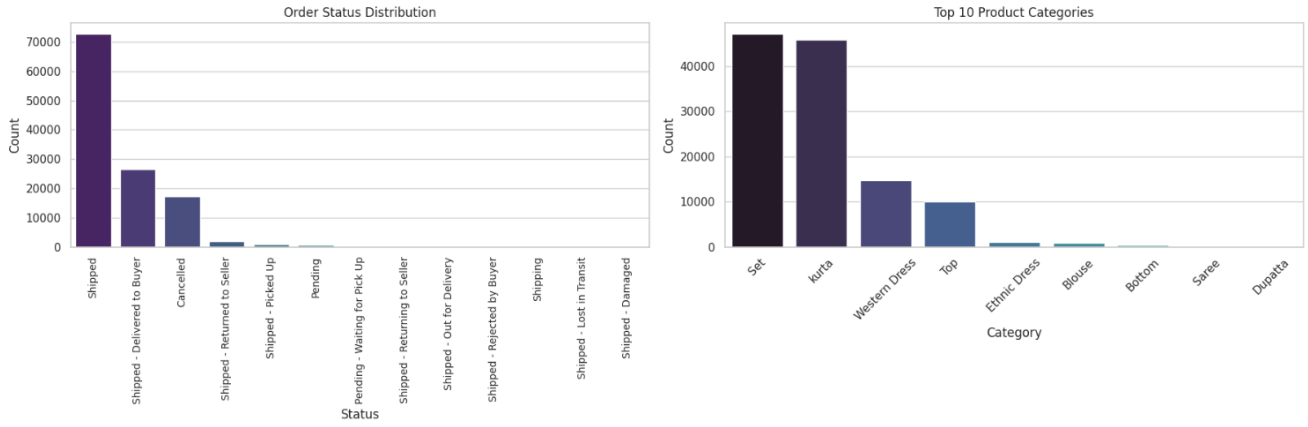
	QUERY PLAN text
1	Limit (cost=74326.52..74326.54 rows=10 width=60) (actual time=3649.156..3653.528 rows=10 loops=1)
2	-> Sort (cost=74326.52..74336.55 rows=4012 width=60) (actual time=3649.154..3653.524 rows=10 loops=1)
3	Sort Key: (((((product_returns.returned_orders)::double precision / (NULLIF(product_returns.total_orders, 0))::double precision)) * '10...
4	Sort Method: top-N heapsort Memory: 26kB
5	-> WindowAgg (cost=74079.36..74239.82 rows=4012 width=60) (actual time=3639.698..3650.728 rows=7074 loops=1)
6	-> Sort (cost=74079.34..74089.37 rows=4012 width=44) (actual time=3639.677..3644.606 rows=7074 loops=1)
7	Sort Key: (((product_returns.returned_orders)::double precision / (NULLIF(product_returns.total_orders, 0))::double precision))...
8	Sort Method: quicksort Memory: 634kB
9	-> Subquery Scan on product_returns (cost=73648.66..73839.22 rows=4012 width=44) (actual time=3633.357..3641.479 row...
10	-> Finalize HashAggregate (cost=73648.66..73799.10 rows=4012 width=36) (actual time=3633.353..3640.208 rows=707...
11	Group Key: p.sku, p.category
12	Filter: (count(o.order_id) > 0)
13	Batches: 1 Memory Usage: 1169kB
14	-> Gather (cost=70880.61..73407.96 rows=24070 width=36) (actual time=3603.363..3614.645 rows=17187 loops=1)
15	Workers Planned: 2
16	Workers Launched: 2
17	-> Partial HashAggregate (cost=69880.61..70000.96 rows=12035 width=36) (actual time=3539.571..3541.624 row...
18	Group Key: p.sku, p.category
19	Batches: 1 Memory Usage: 1169kB
20	Worker 0: Batches: 1 Memory Usage: 1169kB
21	Worker 1: Batches: 1 Memory Usage: 1169kB
22	-> Nested Loop (cost=0.30..26265.26 rows=3489228 width=53) (actual time=0.167..1355.816 rows=3556920 l...
23	-> Parallel Seq Scan on orders o (cost=0.00..7873.46 rows=50146 width=47) (actual time=0.013..22.027 ro...
24	-> Memoize (cost=0.30..0.73 rows=22 width=20) (actual time=0.004..0.014 rows=89 loops=120350)
25	Cache Key: o.sku
26	Cache Mode: logical
27	Hits: 36095 Misses: 5784 Evictions: 0 Overflows: 0 Memory Usage: 6461kB
28	Worker 0: Hits: 33327 Misses: 5696 Evictions: 0 Overflows: 0 Memory Usage: 6444kB
29	Worker 1: Hits: 33741 Misses: 5707 Evictions: 0 Overflows: 0 Memory Usage: 6449kB
30	-> Index Scan using products_sku_idx on products p (cost=0.29..0.72 rows=22 width=20) (actual time=0...
31	Index Cond: (sku = o.sku)

Total rows: 33 Query complete 00:00:03.746

QUERY PLAN IN TABULAR FORM:

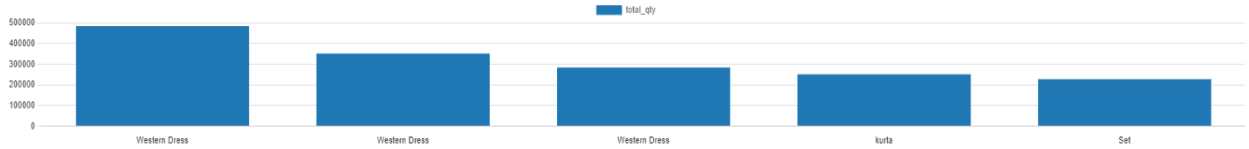
Aspect	Without Indexes/Statistics	With Indexes & Statistics (ANALYZE)
Execution Time	**5952.270 ms**	**3746 ms**
Planning Time	2.013 ms	0.961ms
Scan Type - Orders	Parallel Seq Scan on orders	Parallel Seq Scan on orders
Scan Type - Products	Parallel Seq Scan on products	Index Scan on products (products_sku_idx)
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.