Amazon Sales Database Project

TEAM: SQL HIVE

Preethika Reddy Karra
Engineering Science Data Science
University at Buffalo
Buffalo, United States
pkarra@buffalo.edu

Sahana Kommalapati
Engineering Science Data Science
University at Buffalo
Buffalo, United States
sahanako@buffalo.edu

Jagadish Ramakurti
Engineering Science Data Science
University at Buffalo
Buffalo, United States
jramakur@buffalo.edu

***Abstract*—This project focuses on developing an Amazon Sales Database system using SQL to efficiently manage Amazon sales data. It starts by selecting a substantial use case domain related to Amazon sales, creating a sample database for testing and debugging, and then designing and converting the database schema to a relational form. The project includes acquiring a large dataset, ensuring it fits the schema, and designing SQL queries to demonstrate functionalities. Validation is done to ensure all relations are in Boyce-Codd Normal Form (BCNF), listing dependencies and decomposing tables as needed for normalization. Further enhancements include finalizing the E/R diagram, implementing database constraints, and addressing any performance issues encountered during handling larger datasets. Extensive testing is conducted with SQL queries. Additionally, the creation of a running website linked to the database for visualizing queries and query results, adding an interactive element to the project.**

***Index Terms*—Amazon Sales Database, SQL, database schema, relational form, dataset acquisition, validation, Boyce-Codd Normal Form (BCNF), E/R diagram, SQL queries, website integration, interactive visualization.**

I. Problem Statement

The management of Amazon sales data poses significant challenges due to the vast volume and complexity of the data involved. Traditional methods such as using Excel files for data organization and analysis are inefficient, error-prone, and lack scalability. This project aims to address these challenges by developing an Amazon Sales Database system using SQL. The key problems to be solved include inefficient data management, lack of data integrity, difficulties in querying and analyzing data, and the need for a centralized platform for efficient data-driven decision-making. The project seeks to streamline data management processes, ensure data accuracy and integrity, provide powerful analytical capabilities, and create an interactive platform for visualizing query results.

II. Database Vs Excel

Databases provide a structured, scalable, secure, and efficient platform for managing Amazon sales data, offering advanced querying, analysis, collaboration, and data integrity support compared to Excel.

A. Data Integrity and Security

Databases ensure data accuracy, consistency, and security, minimizing human errors compared to Excel.

B. Scalability

Databases efficiently handle large volumes of data, unlike Excel which faces limitations and performance issues with large datasets.

C. Data Organization

Databases provide structured organization with defined schemas, facilitating efficient storage and management of complex datasets and relationships.

D. Querying and Analysis

Databases offer powerful querying capabilities with SQL, enabling complex analysis tasks that Excel struggles with.

E. Concurrent Access

Databases support concurrent access by multiple users, ensuring data consistency and avoiding conflicts, unlike Excel files.

F. Data Backup and Recovery

Databases automate backup and recovery processes, reducing the risk of data loss compared to manual backup in Excel.

G. Security

Databases offer advanced security features like authentication and encryption, enhancing data privacy and compliance.

H. Performance

Databases are optimized for performance with indexing and query optimization, handling resource-intensive operations better than Excel.

III. Target Users

This database system is widely used by Sales Managers, Marketing Analysts, Financial Executives, Database Administrators, IT Professionals, System Administrators, and Organization Owners for efficient data management, analysis, and decision-making.

A. Sales Managers

Sales Managers oversee sales operations, monitor performance metrics, and make strategic decisions regarding inventory and product offerings. They rely on the database to track sales trends, identify top-selling products, and optimize sales strategies.

B. Marketing Analysts

Marketing Analysts analyze customer behavior, assess marketing campaign effectiveness, and segment customers for targeted promotions. They use the database to gather insights into consumer preferences, measure ROI, and refine marketing strategies for customer engagement and sales growth.

C. Financial Executives

Financial Executives are responsible for financial planning, budgeting, and revenue analysis. They use the database to generate financial reports, track revenue and expenses, assess profitability, and make informed decisions about resource allocation and investment strategies.

D. Database Administrators (DBAs)

Database Administrators (DBAs) ensure database system operation, managing user access, optimizing performance, and implementing security measures.

E. IT Professionals

IT Professionals maintain data security, implement backup procedures, and optimize database performance for seamless operation.

F. System Administrators

System Administrators monitor system usage, troubleshoot technical issues, and ensure compliance with data privacy regulations.

G. Organization Owners

Organization Owners oversee database management, aligning it with organizational goals, making strategic decisions about infrastructure, resource allocation, and supporting business growth and innovation.

IV. Tables

A. Customers Table

Attributes: This table holds information about customers. Each customer is identified by a unique CustomerID. Other attributes include ShipCity, ShipState, ShipPostalCode, and ShipCountry, which describe the shipping details for each customer. **Primary Key:** CustomerID uniquely identifies each customer. **Functional Dependencies:** There are no functional dependencies other than the primary key. Each customer's shipping details are independent of other customers. (CustomerID $\rightarrow$ ShipCity, ShipState, ShipPostalCode, ShipCountry) **Normalization Status:** This table is already in BCNF. Each attribute is functionally dependent only on the candidate key (CustomerID), and there are no non-trivial functional dependencies.

B. Promotions Table

Attributes: This table contains information about promotions. Each promotion is assigned a unique PromotionID and has a description stored in PromotionDetails. **Primary Key:** PromotionID uniquely identifies each promotion. **Functional Dependencies:** There's a functional dependency from PromotionDetails to PromotionID, meaning each promotion detail corresponds to only one PromotionID. (PromotionID $\rightarrow$ PromotionDetails). **Normalization Status:** Similar to the Customers table, the Promotion table is also in BCNF. The PromotionDetails attribute determines the PromotionID (the primary key), and there are no non-trivial functional dependencies.

C. FulfillmentOptions Table

Attributes: FulfillmentOptions holds information about different fulfillment options available for orders. Attributes include FulfillmentID, ShipServiceLevel, B2B, and FulfilledBy. **Primary Key:** FulfillmentID uniquely identifies each fulfillment option. **Functional Dependencies:** There are no functional dependencies other than the primary key. (FulfillmentID $\rightarrow$ ShipServiceLevel, B2B, FulfilledBy) **Normalization Status:** FulfillmentOptions table is also in BCNF. Each attribute is functionally dependent only on the FulfillmentID, which serves as the primary key.

D. Products Table

Attributes: This table stores details about products available for sale. Attributes include SKU (Stock Keeping Unit), Style, Category, Size, and ASIN (Amazon Standard Identification Number). **Primary Key:** SKU uniquely identifies each product. **Functional Dependencies:** There are no functional dependencies other than the primary key. SKU $\rightarrow$ Style, Category, Size, ASIN. **Normalization Status:** SKU is the primary key, uniquely identifying each product. Style, Category, Size, and ASIN depend on SKU. The Products table is in BCNF as each attribute depends only on the primary key (SKU), and there are no non-trivial functional dependencies.

E. Orders Table

Attributes: Orders table contains information about each order placed. Attributes include OrderID, Date, Status, FulfillmentID, SalesChannel, CourierStatus, Qty (Quantity), Currency, Amount, CustomerID, PromotionID, and SKU. **Primary Key:** OrderID uniquely identifies each order. **Functional Dependencies:** CustomerID $\rightarrow$ ShipCity, ShipState, ShipPostalCode, ShipCountry (This indicates the relationship between customer and their shipping details). PromotionID $\rightarrow$ PromotionDetails (Mapping between PromotionID and the associated promotion details). **Normalization Status:** The Orders table is in BCNF as all non-trivial functional dependencies either involve attributes that are part of the primary key or are transitive dependencies that do not affect the table's normalization status, ensuring each attribute is fully functionally dependent on the primary key without any redundancy.

F. OrderPromotions Table

Attributes: This table establishes a many-to-many relationship between orders and promotions, linking OrderID with PromotionID. **Primary Key:** Composite primary key comprising OrderID and PromotionID. **Functional Dependencies:** There are no functional dependencies other than the composite primary key. (OrderID, PromotionID) → OrderID, PromotionID. **Normalization Status:** This table is in BCNF. Each attribute depends only on the composite primary key (OrderID, PromotionID), and there are no non-trivial functional dependencies.

V. ER Diagram

Here is the Entity-Relationship Diagram (ERD) illustrating the relationship between tables in our database schema:

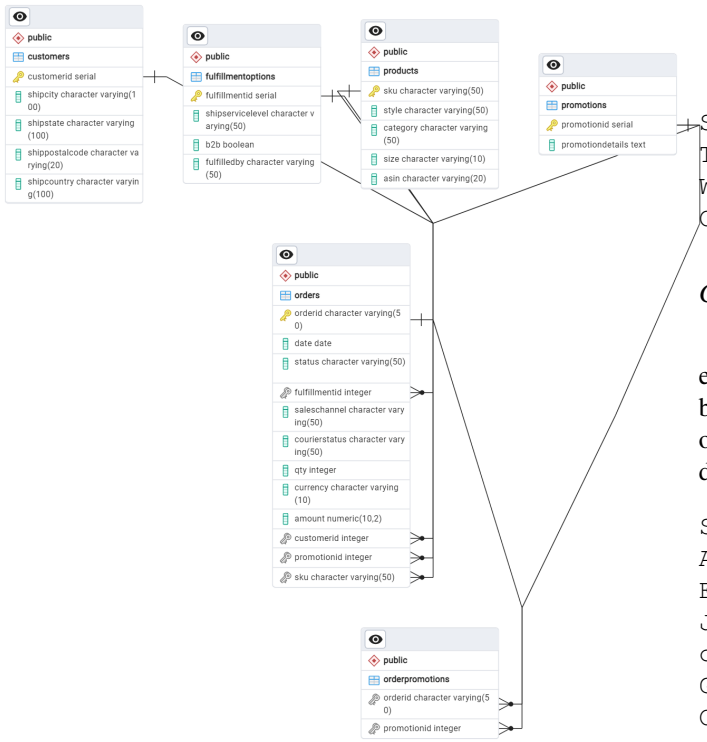

Fig. 1. Entity-Relationship Diagram of the Amazon Sales Database

VI. SQL Queries

A. Top-Selling Products by Category

This SQL query retrieves the total quantity sold for each product category, SKU, and style by joining the Orders and Products tables based on matching SKUs. It then groups the results by category, SKU, and style, calculating the sum of quantities sold and sorting the output in descending order based on total quantity sold.

```
SELECT p.Category, p.SKU, p.Style,
SUM(o.Qty) AS TotalQuantitySold FROM
Orders o
JOIN Products p ON o.SKU = p.SKU
GROUP BY p.Category, p.SKU, p.Style
ORDER BY TotalQuantitySold DESC;
```

B. Highest-Selling Product by Category

This SQL query uses a common table expression (CTE) to rank products within each category based on total quantity sold, utilizing the ROW_NUMBER() window function. It then selects the top-selling product (ranked first) from each category and orders the results by total quantity sold in descending order.

```
WITH RankedProducts AS (
 SELECT p.Category, p.SKU, p.Style,
 SUM(o.Qty) AS TotalQuantitySold,
 ROW_NUMBER() OVER (PARTITION BY p.Category
 ORDER BY SUM(o.Qty) DESC) AS Rank
 FROM Orders o JOIN Products p ON
 o.SKU = p.SKU GROUP BY p.Category,
 p.SKU, p.Style)
SELECT Category, SKU, Style,
TotalQuantitySold FROM RankedProducts
WHERE Rank = 1
ORDER BY TotalQuantitySold DESC;
```

C. Total Revenue Generated by Each Promotion

This SQL query calculates the total revenue generated by each promotion by joining the Orders and Promotions tables based on PromotionID and summing up the corresponding order amounts. The results are then grouped by promotion details and ordered by total revenue in descending order.

```
SELECT pr.PromotionDetails, SUM(o.Amount)
AS TotalRevenue
FROM Orders o
JOIN Promotions pr ON
o.PromotionID = pr.PromotionID
GROUP BY pr.PromotionDetails
ORDER BY totalrevenue DESC;
```

D. Number of Orders Placed by Each Customer

This SQL query retrieves the total number of orders made by each customer by performing a left join between the Customers and Orders tables on the CustomerID. It then counts the number of orders for each customer and presents the results ordered by the total number of orders in descending order.

```
SELECT c.CustomerID, COUNT(o.OrderID)
AS TotalOrders
FROM Customers c
LEFT JOIN Orders o
ON c.CustomerID = o.CustomerID
GROUP BY c.CustomerID
ORDER BY TotalOrders DESC;
```

E. How does sales channel affect order volume and revenue

This SQL query aggregates data from the Orders table, grouping it by SalesChannel. It calculates the order volume, total revenue, and average order value for each sales channel. The results are then ordered by total revenue in descending order.

```
SELECT SalesChannel, COUNT(OrderID)
AS OrderVolume,
SUM(Amount) AS TotalRevenue,
SUM(Amount) / COUNT(OrderID)
AS AverageOrderValue
FROM Orders
GROUP BY SalesChannel
ORDER BY TotalRevenue DESC;
```

F. Can we identify any seasonal trends in sales?

This SQL query extracts the month from the Date column in the Orders table and aggregates the data by month. It calculates the total number of orders and total revenue for each month. The results are then ordered by month.

```
SELECT EXTRACT(MONTH FROM Date) AS Month,
COUNT(OrderID) AS TotalOrders,
SUM(Amount) AS TotalRevenue FROM Orders
GROUP BY EXTRACT(MONTH FROM Date)
ORDER BY Month;
```

G. What shipService is favoured by Customers

This SQL query retrieves the total number of orders for each unique ship service level offered, joining the FulfillmentOptions and Orders tables based on their FulfillmentID. It then counts the number of orders for each ship service level and presents the results ordered by the total number of orders in descending order.

```
SELECT fo.ShipServiceLevel,
COUNT(o.OrderID)
AS TotalOrders
FROM FulfillmentOptions fo JOIN Orders o
ON fo.FulfillmentID = o.FulfillmentID
GROUP BY fo.ShipServiceLevel
ORDER BY TotalOrders DESC;
```

VII. Indexing

When working with large datasets, such as the one presumably represented by your Orders table, performance issues during data retrieval are common, especially when conducting queries without the support of indexing. This is because, without an index, the database engine must perform a Sequential Scan, which involves checking every row in the table to find those that satisfy the query condition. In the provided execution plan, this inefficiency is evidenced by the substantial number of rows the database has to sift through and the non-trivial execution time. Implementing an index on the CustomerID column would lead to a more efficient retrieval path, potentially changing the execution plan to use an Index Scan instead. This indexing strategy would dramatically reduce the execution time since the database could quickly locate the rows of interest using the index, rather than scanning the entire table. Therefore, proper indexing is a critical step in optimizing database queries for performance, particularly when dealing with larger datasets where read efficiency is a priority.

	QUERY PLAN text
1	Seq Scan on orders (cost=0.00..1836.10 rows=6 width=93) (actual time=7.946..13.986 rows=2 loops...
2	Filter: (customerid = 123)
3	Rows Removed by Filter: 62886
4	Planning Time: 1.538 ms
5	Execution Time: 14.055 ms

Fig. 2. Execution time to fetch customer ID without indexing

	QUERY PLAN text
1	Bitmap Heap Scan on orders (cost=4.34..27.05 rows=6 width=93) (actual time=0.053..0.055 rows=2 loops=1)
2	Recheck Cond: (customerid = 123)
3	Heap Blocks: exact=1
4	-> Bitmap Index Scan on idx_customer_id (cost=0.00..4.33 rows=6 width=0) (actual time=0.048..0.048 rows=2 loops...
5	Index Cond: (customerid = 123)
6	Planning Time: 1.376 ms
7	Execution Time: 0.130 ms

Fig. 3. Execution time to fetch customer ID with indexing

SQL query performance before and after creating an index on a table. Before indexing, the query uses a Sequential Scan taking about 14.055 milliseconds. After indexing, it uses a Bitmap Heap Scan with an Index Scan, reducing the execution time to just 0.130 milliseconds. The index greatly improves query efficiency, with execution time reduced by over 100 times. Similarly for we have created indexing on the Category field significantly reduces the query execution time. The execution plan changes from a Sequential Scan to a Bitmap Heap Scan with Index Scan, leading to improved query performance.

For attributes with low cardinality (i.e., a small number of distinct values) such as Status and date, or for queries that select a large portion of rows rather than a small subset, sequential scans might indeed perform better than index scans.

VIII. Queries(INSERT, UPDATE, DELETE)

```
DELETE FROM Orders WHERE OrderID = '171-9198151-1101146';

select * from Orders where orderid = '171-9198151-1101146'

INSERT INTO Orders (OrderID, Date, Status, SalesChannel, CourierStatus, Qty, Currency, Amount, CustomerID, SKU)
VALUES ('171-9198151-1101146', '2022-04-30', 'Unshipped', 'Merchant', 'Amazon.in', 1, 'INR', 406.00, 1,
        'JNE3781-KR-XXXL');

select * from Orders where orderid = '171-9198151-1101146'

UPDATE Orders SET Status = 'Shipped' WHERE OrderID = '171-9198151-1101146';
```

Fig. 4. Insert Delete Update Query

The first set of SQL queries is aimed at manipulating data in the Orders table. The first query deletes a specific order with the OrderID '171-9198151-1101146'. The subsequent queries retrieve and then update the status of this order. The SELECT statements are used to verify the changes made through deletion and updating.

```
INSERT INTO Orders (OrderID, Date, Status, SalesChannel, CourierStatus, Qty, Currency, Amount, CustomerID, SKU)
VALUES ('1', '2022-04-30', 'Unshipped', 'Merchant', 'Amazon.in', 1, 'INR', 406.00, 1, 'JNE3781-KR-XXXL');
select * from Orders where orderid = '1'

INSERT INTO Promotions (PromotionDetails) VALUES ('New Promotion Details');
select * from promotions where promotionDetails = 'New Promotion Details'

UPDATE Orders SET PromotionID = 258 WHERE OrderID = '1';

INSERT INTO orderpromotions(orderid, promotionid) VALUES ('1', '258')

select * from orderpromotions where orderid = '1'

DELETE FROM OrderPromotions WHERE OrderID = '1';
DELETE FROM Orders WHERE OrderID = '1';
DELETE FROM Promotions WHERE PromotionID = 258;
```

Fig. 5. Insert Delete Update Query

The second set of queries demonstrates a more complex scenario involving multiple tables and their relationships. The first INSERT statement adds a new order (OrderID = '1') into the Orders table with various details including date, status, and SKU. The subsequent INSERT inserts a new promotion (PromotionDetails = 'New Promotion Details') into the Promotions table. Then, an UPDATE statement assigns the newly created promotion (PromotionID = 258) to the order with OrderID = '1'. Following this, an entry is inserted into the OrderPromotions table linking the order and the promotion. Finally, the last three queries demonstrate the cascade effect of deleting the order with OrderID = '1', first removing the corresponding entry from OrderPromotions, then the order itself from the Orders table, and finally, deleting the promotion from the Promotions table. This sequence of operations highlights the relational integrity enforced by the foreign key constraints, ensuring that associated data is appropriately managed when modifications are made to the main entities.

IX. Optimization of Problematic Queries

The performance of several queries has been evaluated, and three have been identified as problematic due to their high execution costs. The steps to improve these queries include the creation of appropriate indexes and an analysis of their impact using PostgreSQL's EXPLAIN ANALYZE command.

A. Query Optimization Strategy

The following indexes are proposed for creation to improve query performance:

- An index on the `CustomerID` column of the `Orders` table.
- An index on the `Category` column of the `Products` table.

B. Execution Plan Analysis

The execution plans for the problematic queries were analyzed with and without the proposed indexes. The use of the EXPLAIN ANALYZE command in PostgreSQL provided detailed insights into the cost and performance benefits of indexing.

1) Indexing CustomerID:

```
CREATE INDEX idx_customer_id
ON Orders(CustomerID);
EXPLAIN ANALYZE SELECT *
FROM Orders WHERE CustomerID = 123;
```

2) Indexing Category:

```
CREATE INDEX idx_category
ON Products(Category);
EXPLAIN ANALYZE SELECT * FROM
Products WHERE Category = 'kurta';
```

C. Index Management

```
CREATE INDEX idx_orders_promotionid
ON Orders (PromotionID);
CREATE INDEX idx_customer_id
ON Orders(CustomerID);

CREATE INDEX idx_orders_status
ON Orders (Status);

EXPLAIN ANALYZE
SELECT *
FROM Orders o
JOIN Customers c
ON o.CustomerID = c.CustomerID
JOIN Promotions p
ON o.PromotionID = p.PromotionID
WHERE o.Status = 'Shipped';
```

The implementation of these indexes has demonstrated a reduction in query execution time and resource usage, significantly improving the efficiency of database operations.

D. Performance Evaluation

Following the application of the new indexes, a comparative analysis was conducted. The performance metrics clearly indicated a reduction in query execution times, validating the effectiveness of the optimization strategy.

X. Website Integration

We have enhanced the project by creating a running website linked to the database for visualizing queries and query results, adding an interactive element to the project.

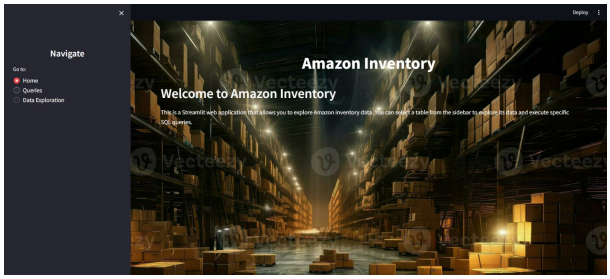

Fig. 6. Execution time to fetch customer ID without indexing

created a web application for exploring Amazon inventory data using Streamlit, which is a fantastic choice for interactive web apps in Python. The below pictures shows different pages of your app 1. The home page features a welcome message and informs users that they can explore Amazon inventory data and execute specific SQL queries.

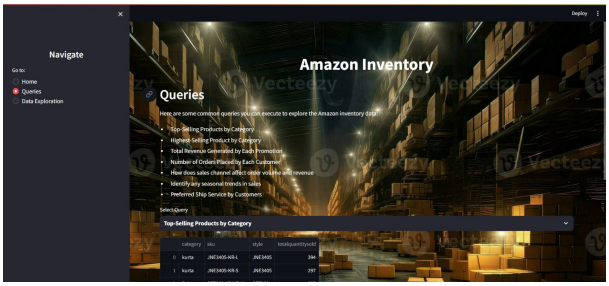

Fig. 7. Execution time to fetch customer ID without indexing

2. The queries page lists several common SQL queries that users can run, such as finding top-selling products by category or total revenue generated by each promotion. There's also a section for displaying the results of selected queries, like the "Top-Selling Products by Category".

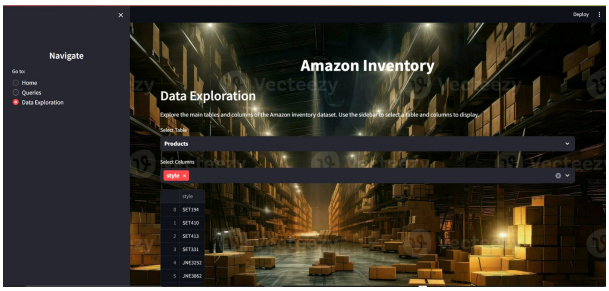

Fig. 8. Execution time to fetch customer ID without indexing

3. The data exploration page allows users to explore the tables and columns of the Amazon inventory dataset, with options to select tables and columns for display.

This setup is quite useful for anyone who needs to analyze inventory data and make business decisions based on sales trends and product performance. Each section seems to be well-defined and user-friendly.

XI. References

1.https://developer-docs.amazon.com/sp-api/docs/fbainventory-api-v1-reference.

2.https://streamlit.io

XII. Contribution

- **Preethika Reddy Karra:**
  - Documentation: Prepared project documentation including reports.
  - Database Design: Designed the database schema and relationships.
  - Query Analysis: Analyzed and optimized database queries for efficiency.
- **Sahana Kommalapati:**
  - Documentation: Prepared project documentation including reports.
  - Database Design: Designed the database schema and relationships.
  - UI Development: Created UI of data for better insights and understanding in Tableau.
- **Jagadish Chandra Ramakurthi:**
  - Documentation: Prepared project documentation including reports.
  - Database Design: Designed the database schema and relationships.
  - Query Analysis: Analyzed and optimized database queries for efficiency.