

CSE 4/560 Data Models and Query Languages - Spring 24

Report

Team Name: SQLUnited - Team 31

1st Lokesh Konjeti
Computer Science
University at Buffalo
Buffalo, USA
UBID: lokeshko
lokeshko@buffalo.edu

2nd Phani Visweswara Sandeep
Chodavarapu
Computer Science
University at Buffalo
Buffalo, USA
UBID: phanivis
phanivis@buffalo.edu

3rd Likhit Sastry Juttada
ES Data Science
University at Buffalo
Buffalo, USA
UBID: likhitsu
likhitsu@buffalo.edu

Abstract: This report emphasizes the indispensability of databases for e-commerce businesses, highlighting their role in efficiently storing, managing, and analyzing vast quantities of sales data to derive valuable insights. It discusses the superiority of databases over spreadsheets in terms of data organization, security, and scalability, making them essential tools for improving customer experience, anticipating market demands, and maximizing profits in the e-commerce domain.

Keywords: E-commerce, database management, sales data analysis, data security, scalability.

I. PROBLEM STATEMENT

E-commerce businesses receive millions of orders in a day, storing the information regarding the orders, products and their transactions becomes necessary as it helps keep track of sales trends of various products, manage inventory, anticipate rapidly changing demands etc.

Thus a database which contains the information about customers, products, sales and their interactions would be a perfect tool for any company to derive valuable insights, improve customer experience, gain profits and provide better service

The need of a database is inevitable as it provides efficient mechanisms to store, retrieve and update huge amounts of data with minimal effort. For example, SQL Language is fairly easy to read and Dividing data into smaller tables to reduce data redundancy is an in built feature of the databases which helps to reduce the storage needed to store the data . Being able to derive such valuable information is the core strength of a database, which a spreadsheet fails to deliver.

Additionally, there might be sensitive data like credit card information which can be safeguarded using the strong safety constraints provided by the database software. The security features provided by a spreadsheet are not as robust as the one given by a database.

Finally, adding new data to a database is much easier than adding it to a spreadsheet. Thus having a database for sales data is advantageous when compared to a standard spreadsheet.

II. Target User

The database could be used internally within the e-commerce company to make strategic business decisions by using the insights derived from the database. There could be a team of database engineers who would run a set of complex queries to derive

valuable information and present it to the board of directors, who would then make business decisions.

The database would be centrally managed by a database administrator or a team of database engineers, who will be responsible to keep the database in an active condition. They will make sure to implement user privileges to restrict unauthorised access to the database. In certain scenarios they might also implement data pipelines to insert new data into the database.

III. Tasks to be accomplished

1. USE CASES

There are some critical questions to be addressed before proceeding with the database and here we will try to discuss the same.

Some critical questions are how would we use the data: Using the schema created for the data, when used by different end users, such as Store manager, customer, Application Manager. There are many ways the database can be queried. Store managers can query the database to view their sales analysis, inventory management. Customers can view their Orders and store inventory. Data analysts can use the tables (such as customer, order, product, store) to craft machine learning models to improve sales.

2. SAMPLE DATABASE

Creating a small subset of our database by hand to facilitate debugging, below is a screenshot of how sample data looks like

```
CREATE TABLE "Order" (
    OrderID INTEGER PRIMARY KEY,
    OrderDate DATE NOT NULL,
    CustomerID INTEGER NOT NULL,
    ProductID INTEGER NOT NULL,
    QuantityOrdered REAL NOT NULL,
    OrderAmount REAL NOT NULL,
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID),
    FOREIGN KEY (ProductID) REFERENCES Product(ProductID) );

CREATE TABLE Storeproduct
(
    StoreID integer,
    ProductID integer,
    CONSTRAINT storeproduct_productid_fkey FOREIGN KEY (productid)
        REFERENCES public.product (productid) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE RESTRICT,
    CONSTRAINT storeproduct_storeid_fkey FOREIGN KEY (storeid)
        REFERENCES public.store (storeid) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE RESTRICT;
);

-- INSERT statements for Customer table
INSERT INTO Customer (CustomerID, FirstName, LastName, DateOfBirth, Address) VALUES
(1, 'John', 'Doe', '1990-05-15', '123 Main St'),
(2, 'Jane', 'Smith', '1985-08-20', '456 Elm St'),
(3, 'Michael', 'Johnson', '1978-11-10', '789 Oak St'),
(4, 'Emily', 'Brown', '1995-03-25', '101 Pine St'),
(5, 'David', 'Wilson', '1982-09-30', '202 Maple St');

-- INSERT statements for Manager table
INSERT INTO Manager (ManagerID, FirstName, LastName, Mobile) VALUES
(1, 'Adam', 'Jones', 1234567890),
(2, 'Sarah', 'Clark', 2345678901),
(3, 'Ryan', 'Taylor', 3456789012),
```

Fig. 1. Schema definition of a table

The screenshot shows the pgAdmin interface. In the top window, a query is being run: 'SELECT * FROM PRODUCT;'. The results window below shows a table with columns: productid [PK] integer, productname text, productcategory text, and productprice real. The data is as follows:

productid	productname	productcategory	productprice
1	Laptop	Electronics	999.99
2	Smartphone	Electronics	599.99
3	T-shirt	Apparel	19.99
4	Running Shoes	Footwear	79.99
5	Coffee Maker	Appliances	49.99

Fig. 2. Running a simple query on the defined schema.

The screenshot shows the pgAdmin interface. In the top window, a complex query is being run involving multiple tables: Order, Product, and Customer. The results window below shows a table with columns: orderid integer, productname text, firstname text, and address text. The data is as follows:

orderid	productname	firstname	address
1	Laptop	John	123 Main St
2	Smartphone	Jane	456 Elm St
3	T-shirt	Michael	789 Oak St
4	Running Shoes	Emily	101 Pine St
5	Coffee Maker	David	202 Maple St

Fig. 3. Running a simple query on the defined schema.

3. DECOMPOSING INTO SUB-TABLES

A. Based on Design Theory, we have defined the following Functional Dependencies in the base table.

- OrderID --> CustomerID, CustomerFirstName, CustomerLastName, CustomerAddress, CustomerDOB
- ProductID --> ProductCategory, ProductName, QuantityAvailable
- StoreID, ProductID --> QuantityAvailable, ManagerID
- StoreID --> StoreName, StoreAddress, ManagerID
- ManagerID --> FirstName, LastName, Mobile, StoreID
- OrderID, ProductID --> UnitPrice, QuantityOrdered, OrderAmount
- CustomerID --> CustomerFirstName, CustomerLastName, CustomerAddress, CustomerDOB
- ProductCategory --> ProductName
- StoreName --> StoreAddress

B. There are two super keys in this dataset, which can determine the tuples uniquely in the table.

- OrderID, ProductID, StoreID
- OrderID, ProductID, ManagerID

C. Boyce Codd Normal Form

BCNF can be used to decompose the table into smaller sub tables to define a schema such that we avoid data inconsistencies and redundancies.

A BCNF violation occurs when a non-trivial functional dependency exists where the determinant (the left side) is not a super key. Based on the Functional dependencies identified and the super keys, the base table is decomposed into the following schema of six tables.

Manager : ManagerID, FirstName, LastName, Mobile, StoreID

Order : OrderID, ProductID, UnitPrice, CustomerID, QuantityOrdered, OrderAmount, OrderDate

Customer : CustomerID, CustomerFirstName, CustomerLastName, CustomerAddress, CustomerDOB

Product : ProductID, ProductCategory, ProductName, UnitPrice

Store : StoreID, StoreName, StoreAddress, ManagerID

Inventory : StoreID, ProductID, QuantityAvailable

4. CREATING DATA USING FAKER

The dataset acquired from the Kaggle repository did not contain all the attributes required for our application, so we have generated certain attributes using python programs that utilise the faker library, below is a snippet of the implementation of faker library to generate random data.

```
In [22]: from faker import Faker
import csv

def generate_dummy_data(num_records):
    fake = Faker()
    data = []
    for _ in range(num_records):
        unit_price = fake.random_int(min=1000, max=9999)
        quantity_ordered = fake.random_int(min=2, max=11)
        sale_amount = unit_price*fake.random_int(min=2, max=11)
        #first_name = fake.first_name()
        #last_name = fake.last_name()
        #mobile_number = fake.phone_number()
        data.append([unit_price, quantity_ordered, sale_amount])
    return data

def write_to_csv(filename, data):
    with open(filename, mode='w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(['Unit Price', 'Quantity Ordered', 'Sale Amount'])
        writer.writerows(data)

num_records = 9800
filename = 'dummy_store_data3.csv'

dummy_data = generate_dummy_data(num_records)
write_to_csv(filename, dummy_data)

print(f'{num_records} records generated and saved to {filename}.')
```

Fig. 4. Faker code to Generate data

5. ER Diagram

A. The final Database schema including the six sub tables is shown in Figure 4.

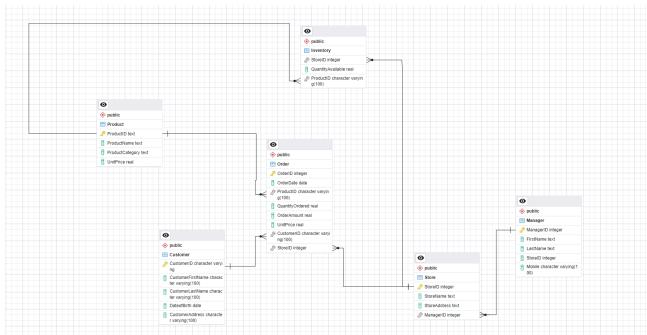


Fig. 5. Final ER Diagram

The base table can be decomposed into the same schema even while decomposing using the third normal form condition.

Relationship between Tables:

- Manager to Store: One-to-many relationship. One manager can manage multiple stores, but each store is managed by only one manager.
 - Store to Store Product: One-to-many relationship. One store can have many products, but the store-product pair is unique in the Store Product table.
 - Product to Order: One-to-many relationship. One product can be part of many orders, but each order is for one or more specific products.
 - Product to Store Product: One-to-many relationship. A product can be present in many stores, and the Store Product table tracks the inventory of each product in each store.
 - Customer to Order: One-to-many relationship. A customer can place many orders, but each order is placed by one specific customer.

B. RESTRICT COMMAND

To make the Database robust to errors, we prevent the deletion of primary keys referenced to foreign keys in other tables. The Figure below shows the implementation of the RESTRICT command to achieve the aforementioned result.

```
CREATE TABLE Storeproduct
(
    StoreID integer,
    ProductID integer,
    CONSTRAINT storeproduct_productid_fkey FOREIGN KEY (productid)
        REFERENCES public.product (productid) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE RESTRICT,
    CONSTRAINT storeproduct_storeid_fkey FOREIGN KEY (storeid)
        REFERENCES public.store (storeid) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE RESTRICT;
);
```

Fig. 6. Restrict Command on Primary Keys

6. PROBLEMS WHILE HANDLING LARGE DATASETS

Certain queries took longer than the rest to execute and return the results. This is a major issue in most of the databases in the real world. The long execution times are generally due to complex query structures and inefficient searching algorithms like Sequential search.

We came across some queries that took longer to execute. And we solved those issues through indexing concepts

Indexes in SQL databases help improve query performance by allowing the database system to locate and retrieve specific rows more efficiently. They serve as a data structure that organizes the data in a way that enables quick lookups based on the values of one or more columns.

Indexes can also speed up sorting and grouping operations by allowing the database to access the data in the desired order without the need for sorting the entire dataset.

7. TESTING DATABASE WITH QUERIES

The Schema is defined using create.sql command and the data is bulk loaded from csv file to the database by load.sql dataset.

Below are a few commands that we run to test the functionality of the database.

Insert Query:

The screenshot shows the pgAdmin 4 interface with a query editor and a results table.

Query Editor:

```
1 -- Insert a new store and assign a manager
2 INSERT INTO "Store" ("StoreID", "StoreName", "StoreAddress", "ManagerID")
3 VALUES (1, 'Main Store', '123 Main Street', 1010);
4
5 -- display the result
6 SELECT * FROM "Store"
7 WHERE "StoreID" = 1;
8
9
```

Data Output:

	StoreID	StoreName	StoreAddress	ManagerID
1	1	Main Store	123 Main Street	1010

Fig. 7. Insert Query 1

The screenshot shows the pgAdmin 4 interface with a query editor and a results table.

Query Editor:

```
1 -- Insert a new product into the products table
2 INSERT INTO "Product" ("ProductID", "ProductName", "ProductCategory", "UnitPrice")
3 VALUES ('OFF-BI-10003681', 'Apple airpods pro max 3', 'Technology', 20);
4
5 -- Display the inserted result
6 SELECT * FROM "Product"
7 WHERE "ProductID" = 'OFF-BI-10003681'
```

Results Table:

	ProductID	ProductName	ProductCategory	UnitPrice
1	OFF-BI-10003681	Apple airpods pro max 3	Technology	20

Fig. 8. Insert Query 2

Delete Commands:

```

1
2 -- Delete Store with ID 1
3 DELETE FROM "Store"
4 WHERE "StoreID" = 1
    
```

Query returned successfully in 49 msec.

Fig. 9. Delete Query 1

```

1
2 -- Delete from Product with ProductID = OFF-BI-10003681
3 DELETE FROM "Product"
4 WHERE "ProductID" = 'OFF-BI-10003681'
    
```

Query returned successfully in 61 msec.

Fig. 10. Delete Query 2

Update Commands:

```

1
2 -- Update date of birth of customer from 1905-03-19 to 1905-03-20
3 UPDATE "Customer"
4 SET "DateOfBirth" = DATE('1905-03-20')
5 WHERE "CustomerID" = 'AA-10315'
6
7 -- Display the updated result
8 SELECT * FROM "Customer"
9 WHERE "CustomerID" = 'AA-10315';
10
    
```

CustomerID	CustomerFirstName	CustomerLastName	DateOfBirth	CustomerAddress
AA-10315	Alex	Avila	1905-03-20	San Francisco, California, 94122

Fig. 11. Update Query 1

```

1
2 -- Update unit price of a specific product from 2472 to 2400
3 UPDATE "Product"
4 SET "UnitPrice" = 2400
5 WHERE "ProductID" = 'FUR-BO-10000112'
6
7 -- Display the result
8 SELECT * FROM "Product"
9 WHERE "ProductID" = 'FUR-BO-10000112';
10
    
```

ProductID	ProductName	ProductCategory	UnitPrice
FUR-BO-10000112	Bush Birmingham Collection Bookcase, Dark Cherry	Furniture	2400

Fig. 12. Update Query 2

Select Commands:

```

1
2 -- Get orders with a total amount greater than $109000.00 (using WHERE and ORDER BY)
3 SELECT * FROM "Order"
4 WHERE "OrderAmount" > 109000
5 ORDER BY "OrderAmount" DESC;
    
```

OrderID	OrderDate	ProductID	QuantityOrdered	OrderAmount	UnitPrice	CustomerID	StoreID
1	2024-03-04	OFF-AR-10004602	6	109989	9999	M2-17740	8622
2	2024-01-24	OFF-PA-10000418	5	109923	9993	CC-12145	6643
3	2024-12-01	FUR-GH-10004063	4	109868	9988	DB-12910	7220
4	2024-03-04	OFF-AP-10000275	4	109835	9985	RM-19675	5360
5	2024-04-29	OFF-AP-10001634	9	109791	9981	DK-13375	3365
6	2024-02-19	OFF-FU-10000206	11	109747	9977	LA-16780	6389
7	2024-04-29	OFF-ST-10000321	4	109659	9969	AB-10015	5558
8	2024-03-15	TEC-PH-10003589	10	109593	9963	SB-20290	1606
9	2024-01-15	FUR-B0-10004409	5	109527	9957	OT-18730	5417
10	2024-09-02	OFF-BI-10004738	8	109459	9949	ML-17395	3596
11	2024-04-17	OFF-B-10000666	2	109417	9947	RP-19390	9244
12	2024-09-09	FUR-TA-10001539	4	109241	9931	PR-18880	4150
13	2024-05-04	TEC-AC-10001314	8	109219	9929	TS-21205	6773
14	2024-02-27	OFF-PA-10002195	10	109043	9913	AZ-10750	9281

Fig. 13. Select Query 1

```

1
2 -- Query the product with highest price( using sub-query)
3 SELECT * FROM "Product"
4 WHERE "UnitPrice" = (SELECT MAX("UnitPrice") FROM "Product");
5
6
    
```

ProductID	ProductName	ProductCategory	UnitPrice
TEC-MA-10001972	Okidata C331dn Printer	Technology	9999

Fig. 14. Select Query 2

```

1
2 -- Get the order details of a specific customer (using JOIN)
3 SELECT o."OrderID", o."OrderDate", str."StoreID", pr."ProductName", o."OrderAmount"
4 FROM "Order" AS o
5 JOIN "Store" AS str ON o."StoreID" = str."StoreID"
6 JOIN "Product" AS pr ON pr."ProductID" = o."ProductID"
7 WHERE o."CustomerID" = 'ME-17320';
    
```

OrderID	OrderDate	StoreID	ProductName	OrderAmount
1	2024-02-13	4739	Ibico Plastic Spiral Binding Combs	36619
2	2024-01-14	9803	Icebear OfficeWorks 42" Round Table	15392
3	2024-04-15	3018	Hoover Commercial Soft Guard Upright Vacuum And Disposable Filtration Bags	73634
4	2024-03-31	1808	C-Line Magnetic Cubicle Keepers, Clear Polypropylene	35289
5	2024-03-29	8741	Computer Printout Paper with Letter-Trim Perforations	96470
6	2024-08-04	6111	Global Leather Highback Executive Chair with Pneumatic Height Adjustment, Bla...	41500
7	2024-07-04	2062	Jabra BIZ 2300 Duo QD Corded Headset	53416
8	2024-01-27	3708	Anker Astro 15000mAh USB Portable Charger	4956
9	2024-04-27	8919	Wirebound Message Books, Four 3/4" x 5" Forms per Page, 600 Sets per Book	16644
10	2024-01-13	5668	GBC White Gloss Covers, Plain Front	28645
11	2024-03-18	1745	Advantus 10 Drawer Portable Organizer, Chrome Metal Frame, Smoke Drawers	11544
12	2024-02-29	8294	Global Deluxe High Back Manager's Chair	45246
13	2024-03-24	5079	Ibico Hi-Tech Manual Binding System	25308
14	2024-02-27	9454	Rogers Handheld Barrel Pencil Sharpener	19288

Fig. 15. Select Query 3

```

1
2 -- Find the total value of each product across all the stores
3 SELECT pro."ProductName", SUM(pro."UnitPrice" * inv."QuantityAvailable") AS "Total Product Value"
4 FROM "Product" AS pro
5 JOIN "Inventory" AS inv ON pro."ProductID" = inv."ProductID"
6 GROUP BY pro."ProductID"
7 ORDER BY "Total Product Value";
    
```

ProductName	Total Product Value
Cisco Desktop Collaboration Experience DX650 IP Video Phone	26467
Memorex Mini Travel Drive 4 GB USB 2.0 Flash Drive	27405
Xerox Blank Computer Paper	29722
Cisco SPA525G 5 Line IP Phone	32103
Brother MFC-9340CDW LED All-In-One Printer, Copier Scanner	32200
Zebra GK420t Direct Thermal/Thermal Transfer Printer	36157
Xerox 215	37444
Rediform 5.0:5 Phone Message Books	38324
Computer Printout Paper with Letter-Trim Fine Perforations	38722
4000 Highlighters	44688
Xerox 1938	47980
Xiaomi Mi3	49346
Sauder Facets Collection Locker/File Cabinet, Sky Alder Finish	50922
Hunt BOSTON Model 1606 High-Volume Electric Pencil Sharpener, Beige	52130

Fig. 16. Select Query 4

8. PROBLEMATIC QUERIES

1. Query 1

We have identified three queries which take comparatively more run time to retrieve the data. Upon inspection, we see that the cost of these queries is too high. Hence to optimize data retrieval from the database we create index on the most used column of the database to reduce the cost of these queries.

The first query is trying to retrieve the data for all the orders made by the customer with first name John. The cost for this query is mentioned in the figure below.

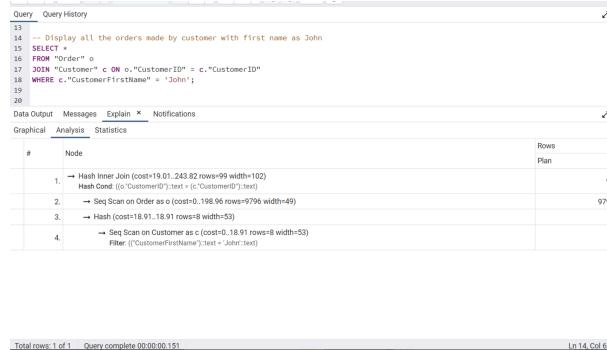


Fig. 17. Problematic Query 1

Upon Creating an Index on the database on the column CustomerFirstName the cost reduces which also results in reduced time to retrieve the data from the table. The figure below shows the corresponding values.

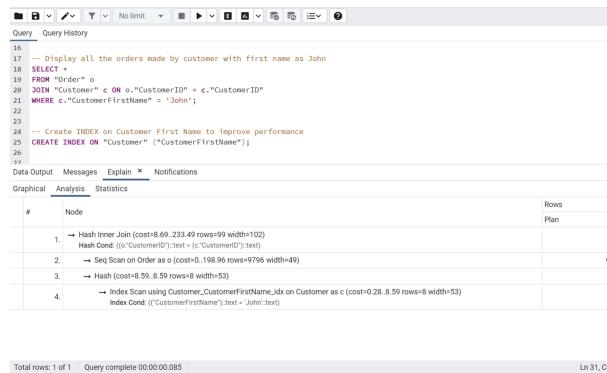


Fig. 18. Problematic Query 1, Solution

2. Query 2

Query two aggregates the data and try to retrieve the total number of orders placed by each customer. The respective cost and run time are mentioned in the figure below.

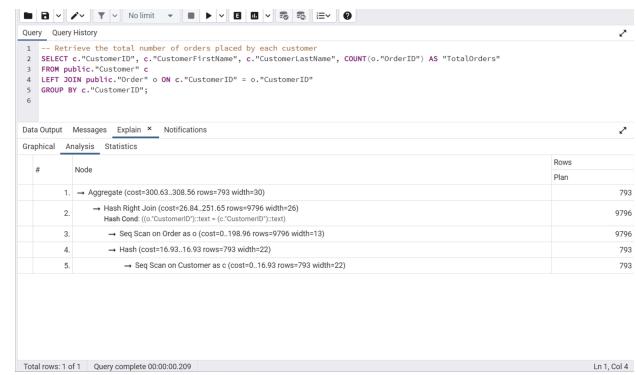


Fig. 19. Problematic Query 2

With respect to this query as CustomerID is a primary key, we can rewrite the query to eliminate the join condition. This operation could significantly improve the performance of the query. The figure below shows the cost of run time of the new query.



Fig. 20. Problematic Query 2, Solution

3. Query 3

The following query try to retrieve the data for the total sales amount for each store. This is again an aggregate operation which takes significantly more time to process and as the data in the database increase it is going to increase the run time of this query substantially. The cost and the run time for this query are on the figure below.

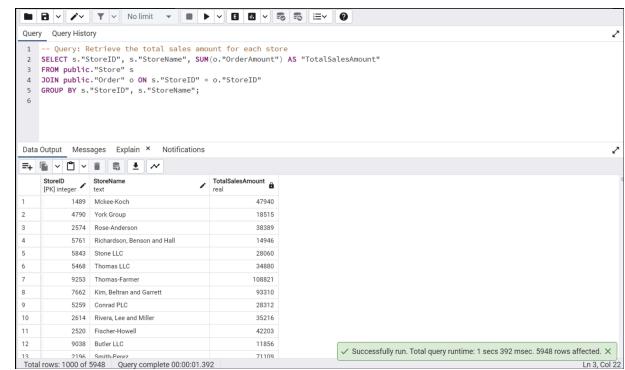


Fig. 21. Problematic Query 3

Hence, to solve this problem we create an index on the StoreID column in Order table so that the cost of the query decreased while the joining the tables. The optimized query now shows significant change in run time and cost is optimized. The figure below show shows the values respective values for this query.

```

Query History
1 -- The query involves multiple joins and grouping operations, which can be inefficient without proper indexing.
2 -- Create an index on the StoreID column in the "Order" table to speed up the join operation
3 CREATE INDEX ON public."Order" ("StoreID");
4
5 -- Query: Retrieve the total sales amount for each store
6 SELECT s."StoreID", s."StoreName", SUM(o."OrderAmount") AS "TotalSalesAmount"
7 FROM "Store" s
8 JOIN "Order" o ON s."StoreID" = o."StoreID"
9 GROUP BY s."StoreID", s."StoreName";
10

```

Data Output Messages Explain Notifications
Graphical Analysis Statistics

#	Node	Rows	Plan
1.	Aggregate (cost=482.51, rows=1 width=25)	5948	
2.	Hash Inner Join (cost=208.83, rows=9796 width=25)	9796	Hash Cond: (o."StoreID" = s."StoreID")
3.	Seq Scan on Order as o (cost=0, 198 rows=9796 width=8)	9796	
4.	Hash (cost=134.48, 134.48 rows=5948 width=21)	5948	
5.	Seq Scan on Store as s (cost=0, 134.48 rows=5948 width=21)	5948	

Total rows: 1 of 1 Query complete 00:00:204 Ln 8, Col 44

Fig. 22. Problematic Query 3, Solution

9. BONUS TASK: Deploying

In this task we chose to deploy our model on the Streamlit Community Cloud.

Streamlit is a free and open-source framework to rapidly build and share beautiful machine learning and data science web apps. It is a Python-based library specifically designed for machine learning engineers. Streamlit enables the integration of AI and databases into applications effortlessly, giving you an edge in creating solutions that resonate with customers. Streamlit has inbuilt modules to create native UI components or also to use React.js components as well.

Using the endpoints of the database we connected the database to the Streamlit app and hosted the application on the Streamlit Community Cloud.

The application is deployed and is available at the following link:

<https://dmqlproject.streamlit.app/>

The database is hosted on AWS RDS, and its endpoints are integrated into a Python Streamlit app, leveraging the psycopg2 library for database interactions in Python.

The Application has 3 main pages.

1. Schema Definition Page
2. Table Preview Page
3. Query Page

Schema Definition Page

On this page the schema definition for all the tables is available, different tables can be selected using the navigation menu on the left.

Column Name	Data Type
OrderID	integer
OrderDate	date
ProductID	character varying
QuantityOrdered	real
OrderAmount	real
UnitPrice	real
CustomerID	character varying
StoreID	integer

Fig. 23. Application Deployment

Table Preview Page

Based on the selection of Table name and the Limit, this page will show the entire table up to *Limit number of rows.

OrderID	OrderDate	ProductID	QuantityOrdered	OrderAmount	UnitPrice	CustomerID	StoreID
1 1	2024-05-01	1000178	9.0	36832.0	8733.0	CG-12550	9605
2 2	2024-05-02	1000245	7.0	19922.0	2846.0	CG-12550	7871
3 3	2024-05-03	1000240	4.0	17148.0	4374.0	DV-13045	4228
4 4	2024-05-04	1000257	9.0	36470.0	3947.0	SJ-20355	4231
5 5	2024-05-05	1000276	2.0	73070.0	7307.0	SJ-20355	5741

Fig. 24. Application Deployment

Query Page

The user can write SQL Queries on this page to retrieve the tables associated with the respective queries.

Run Custom Query

Enter your custom SQL query below and click the 'Run Query' button.

Run Query

Fig. 25. Application Deployment

Note: Only “SELECT” queries can be executed here and all other queries will be discarded

10. TRIGGERS:

To ensure that there are no inconsistencies in the database, we have implemented a trigger on the inventory table. Once an order is placed and the data is inserted into the database a trigger is activated which will update the Quantity of the Product.

```
-- FUNCTION: public.update_inventory_quantity()
-- DROP FUNCTION IF EXISTS public.update_inventory_quantity();

CREATE OR REPLACE FUNCTION public.update_inventory_quantity()
RETURNS trigger
LANGUAGE 'plpgsql'
COST 100
VOLATILE NOT LEAKPROOF
AS $BODY$
BEGIN
    -- Decrease the quantity available for the product in the Inventory table
    UPDATE "Inventory"
    SET "QuantityAvailable" = "QuantityAvailable" - NEW."QuantityOrdered"
    WHERE "ProductID" = NEW."ProductID" AND "StoreID" = NEW."StoreID";

    RETURN NEW;
END;
$BODY$;

ALTER FUNCTION public.update_inventory_quantity()
OWNER TO postgres;

CREATE TRIGGER insert_trigger
AFTER INSERT ON orders
FOR EACH ROW
EXECUTE FUNCTION update_another_table();
```

13. TEAM CONTRIBUTION:

Name	Contribution
Phani Visweswara Sandeep Chodavarapu	33.3%
Likhit Sastry Juttada	33.3%
Lokesh Konjeti	33.3%

Fig. 24. Trigger

11. CHANGES SINCE Milestone 1

There are no major changes since Milestone 1, the schema remains similar but we added an additional column UnitPrice which is the Unit Price for each product.

12. CONCLUSION

In summary, the profound impact that databases have on the e-commerce industry cannot be emphasized enough. The intelligent deployment of strong database systems sticks out as a crucial success factor as companies continue to negotiate the complexity of enormous data landscapes. Databases offer better organization, improved security features, and unmatched scalability compared to spreadsheets, which struggle with large data volumes and complex data relationships. These characteristics simplify operations and give companies the analytical tools they need to predict market trends, customize customer experiences, and increase profitability. Thus, acquiring cutting-edge database technology is not merely a choice, but a basic requirement for any e-commerce business hoping to prosper in a cutthroat market.