Inventory Management System

Team Name: **Schema Squad**
Hosted at: **https://imsdmql.streamlit.app**

Nikhil Gupta
Computer Science and Engineering
University at Buffalo
Buffalo, USA
ngupta22@buffalo.edu

Rahul Karkala Kudva
Computer Science and Engineering
University at Buffalo
Buffalo, USA
rahulkar@buffalo.edu

Indushree Byrareddy
Computer Science and Engineering
University at Buffalo
Buffalo, USA
indushre@buffalo.edu

***Abstract*—For businesses to meet customer demands and stay competitive, effective inventory management is essential. Conventional approaches that rely on traditional methods or Excel sheets frequently result in errors and inefficiency. This project suggests using an Inventory Management System (IMS) based on Relational Databases to address these issues. The goal of the IMS is to effectively manage inventory levels, reduce costs of operation, and simplify procedures like executing orders, managing warehouses, and tracking supplies by utilizing real-time data. By leveraging database technology, the IMS provides scalability, data consistency, and advanced query capabilities, enabling enterprises to maximize operational effectiveness, reduce stockouts, and guarantee inventory accuracy. The significance of this project is found in its ability to transform inventory management procedures throughout industries by providing companies with a centralized platform to efficiently manage inventory and promote long-term growth.**

I. PROBLEM STATEMENT

Managing inventory efficiently is crucial for businesses to maintain optimal operations and meet customer demands. However, traditional methods of inventory management using Excel files often lead to inefficiencies, errors, and lack of scalability. Our project aims to address these challenges by developing a comprehensive inventory management system using a relational database.

Why not Excel? Excel files have limitations in handling large volumes of data and concurrent users. A database provides scalability to accommodate growing data and user needs. Databases enforce data integrity constraints, ensuring accurate and consistent data storage and retrieval. Databases offer powerful query capabilities for complex data analysis and reporting compared to Excel's basic functionalities. Databases support concurrent access by multiple users while maintaining data consistency, which is challenging with Excel files.

Background and Objectives: The inefficiencies associated with manual inventory management can lead to stockouts, overstocking, and increased operational costs. Our objective is to develop a robust inventory management system that streamlines inventory tracking, order processing, and warehouse management. By automating these processes and providing real-time insights, our system aims to improve inventory accuracy, reduce stockouts, minimize holding costs, and enhance overall operational efficiency.

Contribution to the Problem Domain: Our project aims to provide an efficient inventory management system. Along with developing a centralized platform to manage inventory effectively, by leveraging database technology project focuses on designing the inventory database in a very efficient way to manage data effectively. Improved customer satisfaction, cost savings, and long-term company growth are all possible outcomes of this consolidation. Given the pivotal role of inventory management in optimizing supply chains and maintaining competitiveness, our contribution holds significant value.

II. TARGET USER

Warehouse managers in retail businesses track inventory levels, oversee the operations of multiple warehouses, and ensure timely order fulfillment using the inventory management system. To process customer orders, verify product availability, and give precise delivery estimates, sales representatives use the system. In order to maintain ideal inventory levels, supply chain managers use the database to forecast demand, analyze sales data, and work with suppliers. In addition to dealing with regular service duties and user account management, database administrators also make sure the database runs smoothly. The following are the intended users of the inventory management system database:

A. Warehouse Managers

Utilizing the database, warehouse managers will be able to effectively manage warehouse operations, track stock movements, and keep an eye on inventory levels. In addition to creating reports and optimizing warehouse layout based on inventory data, their responsibilities will include updating inventory records.

B. Sales and Customer Service Representatives

The database will be used by sales and customer support agents to process orders, verify product availability, and give clients accurate information about order status and delivery times.

C. Supply Chain Managers

Supply chain managers will use the database to forecast demand, analyze inventory trends, and make well-informed decisions about purchasing, production scheduling, and inventory replenishment tactics.

D. Administrators

Assuring the database's integrity, performance, and security will be the responsibility of database administrators. User access management, backup and recovery, and database optimization are among the responsibilities that they will have.

III. BCNF CHECK AND DECOMPOSITION

To ensure that all relations are in Boyce-Codd Normal Form (BCNF), we need to identify functional dependencies and eliminate any dependencies on non-superkey attributes. Below are the dependencies for each relation:

A. Location

- Dependencies:
  - $LocationID \rightarrow RegionName$, $CountryName, State$, $City, PostalCode$
- All attributes are functionally dependent on the $LocationID$ (primary key).
- No partial dependencies or transitive dependencies exist.

B. Warehouse

- Dependencies:
  - $WarehouseID \rightarrow WarehouseAddress$, $WarehouseName, LocationID$
  - $LocationID \rightarrow RegionName$, $CountryName, State$, $City, PostalCode$
- The $WarehouseID$ uniquely determines $WarehouseAddress$ and $WarehouseName$.
- $LocationID$ determines $RegionName, CountryName, State, City$, $PostalCode$.
- No partial dependencies or transitive dependencies exist.

C. Employee

- Dependencies:
  - $EmployeeID \rightarrow EmployeeEmail$, $EmployeeName, EmployeePhone$, $EmployeeHireDate, EmployeeJobTitle$, $WarehouseID$
- The $EmployeeID$ uniquely determines all other attributes.
- No partial dependencies or transitive dependencies exist.

D. Category

- Dependencies:
  - $CategoryID \rightarrow CategoryName$
- The $CategoryID$ uniquely determines $CategoryName$.
- No partial dependencies or transitive dependencies exist.

E. ProductOrder

- Dependencies:
  - $OrderID \rightarrow OrderItemID$, $ProductID, OrderItemQuantity$, $PerUnitPrice, TotalItemQuantity$, $ProductName, ProductDescription$, $ProductStandardCost, Profit$, $ProductListPrice, CategoryID$
  - $ProductID \rightarrow ProductName$, $ProductDescription, ProductStandardCost$, $Profit, ProductListPrice, CategoryID$
  - $CategoryID \rightarrow CategoryName$
- The above functional dependencies indicate that **OrderID** and **ProductID** are candidate keys for the ProductOrder relation.

F. Customer

- Dependencies:
  - $CustomerID \rightarrow CustomerEmail$, $CustomerName, CustomerAddress$, $CustomerCreditLimit, CustomerPhone$
- The $CustomerID$ uniquely determines all other attributes.
- No partial dependencies or transitive dependencies exist.

G. OrderTable

- Dependencies:
  - $OrderID \rightarrow OrderDate$, $Status, CustomerID$
- The $OrderID$ uniquely determines $OrderDate$, $Status$, and $CustomerID$.
- No partial dependencies or transitive dependencies exist.

BCNF Check and Decomposition: The functional dependencies in the **ProductOrder** relation indicate that **OrderID** and **ProductID** are candidate keys. Therefore, there is a partial dependency in this table. We'll decompose the ProductOrder relation into **Product** and **OrderItem** relations such that each relation is in BCNF:

H. Product

- Dependencies:
  - $ProductID \rightarrow ProductName$, $ProductDescription, ProductStandardCost$, $Profit, ProductListPrice, CategoryID$
  - $CategoryID \rightarrow CategoryName$
- $ProductID$ uniquely determines all other attributes.
- $CategoryID$ determines $CategoryName$.
- No partial dependencies or transitive dependencies exist.

I. OrderItem

- Dependencies:
  - $OrderItemID \rightarrow OrderID$, $ProductID, OrderItemQuantity$, $PerUnitPrice, TotalItemQuantity$
- The $OrderItemID$ uniquely determines all other attributes.
- No partial dependencies or transitive dependencies exist.

Conclusion: All tables appear to be in BCNF as there are no non-trivial functional dependencies on any proper subset of the primary keys. Therefore, no further decomposition is necessary.

IV. E/R DIAGRAM

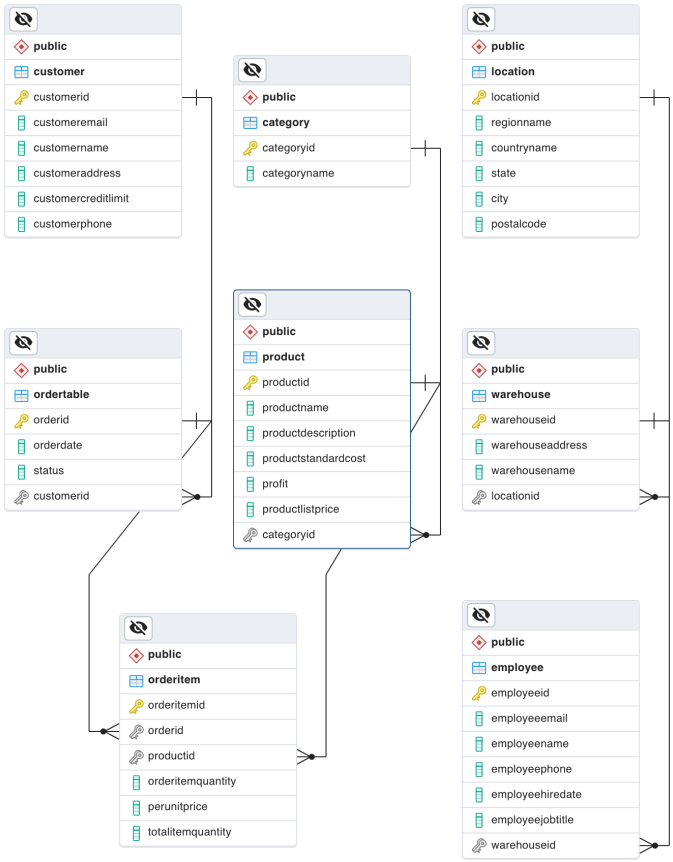


Fig. 1. E/R Diagram

V. DATABASE TABLE STRUCTURE

A. Location Table

- **LocationID** (Primary Key, INT): Unique identifier for each location.
- **RegionName** (VARCHAR(255)): Name of the region.
- **CountryName** (VARCHAR(255)): Name of the country.
- **State** (VARCHAR(255)): State name.
- **City** (VARCHAR(255)): City name.
- **PostalCode** (VARCHAR(255)): Postal code of the location.

B. Warehouse Table

- **WarehouseID** (Primary Key, INT): Unique identifier for each warehouse.
- **WarehouseAddress** (VARCHAR(255)): Address of the warehouse.
- **WarehouseName** (VARCHAR(255)): Name of the warehouse.
- **LocationID** (Foreign Key, INT): References the Location table's LocationID.
- *Actions on Foreign Key*: No action on delete.

C. Employee Table

- **EmployeeID** (Primary Key, INT): Unique identifier for each employee.
- **EmployeeEmail** (VARCHAR(255)): Email address of the employee.
- **EmployeeName** (VARCHAR(255)): Name of the employee.
- **EmployeePhone** (VARCHAR(255)): Phone number of the employee.
- **EmployeeHireDate** (DATE): Date when the employee was hired.
- **EmployeeJobTitle** (VARCHAR(255)): Job title of the employee.
- **WarehouseID** (Foreign Key, INT): References the Warehouse table's WarehouseID.
- *Actions on Foreign Key*: No action on delete.

D. Category Table

- **CategoryID** (Primary Key, INT): Unique identifier for each category.
- **CategoryName** (VARCHAR(255)): Name of the category.

E. Product Table

- **ProductID** (Primary Key, INT): Unique identifier for each product.
- **ProductName** (VARCHAR(255)): Name of the product.
- **ProductDescription** (TEXT): Description of the product.
- **ProductStandardCost** (DECIMAL(10,2)): Standard cost of the product.
- **Profit** (DECIMAL(10,2)): Profit margin of the product.
- **ProductListPrice** (DECIMAL(10,2)): List price of the product.
- **CategoryID** (Foreign Key, INT): References the Category table's CategoryID.
- *Actions on Foreign Key*: No action on delete.

F. Customer Table

- **CustomerID** (Primary Key, INT): Unique identifier for each customer.
- **CustomerEmail** (VARCHAR(255)): Email address of the customer.

- **CustomerName** (VARCHAR(255)): Name of the customer.
- **CustomerAddress** (TEXT): Address of the customer.
- **CustomerCreditLimit** (DECIMAL(10,2)): Credit limit of the customer.
- **CustomerPhone** (VARCHAR(255)): Phone number of the customer.

G. OrderTable Table

- **OrderID** (Primary Key, INT): Unique identifier for each order.
- **OrderDate** (DATE): Date when the order was placed.
- **Status** (VARCHAR(255)): Status of the order.
- **CustomerID** (Foreign Key, INT): References the Customer table's CustomerID.
- *Actions on Foreign Key*: No action on delete.

H. OrderItem Table

- **OrderItemID** (Primary Key, INT): Unique identifier for each order item.
- **OrderID** (Foreign Key, INT): References the OrderTable table's OrderID.
- **ProductID** (Foreign Key, INT): References the Product table's ProductID.
- **OrderItemQuantity** (INT): Quantity of the ordered item.
- **PerUnitPrice** (DECIMAL(10,2)): Price per unit of the ordered item.
- **TotalItemQuantity** (INT): Total quantity of the ordered item.
- *Actions on Foreign Keys*: No action on delete.

VI. Problems Handling Large Databases

Handling larger datasets can pose several challenges, including performance issues such as slower query execution times and increased resource consumption. In our scenario, with 10,000 rows in each relation, these challenges may become more evident. Here are some potential problems we may encounter and how we can address them:

A. Slow Query Performance

As the dataset grows, queries that involve joins or aggregations may become slower. This can be due to the need for full table scans to find matching rows.

B. Excessive Resource Consumption

With larger datasets, database server resources such as memory and CPU may be strained, leading to performance degradation or even server crashes. Optimizing queries and indexes can help mitigate this issue. Additionally, we may need to consider partitioning the tables or optimizing database server settings for better resource utilization.

C. Concurrency Issues

With increased data volume, concurrent access by multiple users or processes may lead to contention and performance issues. Implementing proper locking mechanisms and transaction management strategies can help ensure data integrity and mitigate concurrency-related problems.

D. Storage Requirements

Larger datasets consume more storage space, which may necessitate scaling up our database infrastructure or implementing data archiving and purging strategies to manage storage costs effectively.

E. Backup and Recovery

Backing up and restoring large databases can be time-consuming and resource-intensive. It's essential to have robust backup and recovery procedures in place, including regular backups, incremental backups, and disaster recovery planning.

VII. Proposed Solutions

In our case, we address the Slow Query Performance problem, by creating indexes on columns frequently used in WHERE clauses, JOIN conditions, or ORDER BY clauses. For example, indexing the CustomerID column in the OrderTable and OrderItem tables can speed up queries that involve filtering or joining orders and order items by the customer.

```
CREATE INDEX idx_OrderTable_CustomerID
ON OrderTable (CustomerID);

CREATE INDEX idx_OrderItem_OrderID
ON OrderItem (OrderID);

CREATE INDEX idx_OrderItem_ProductID
ON OrderItem (ProductID);
```

VIII. Testing Dataset with Different SQL Queries

Here are SQL queries for various operations and select queries with different types of statements:

A. Insert Queries

Insert a new location

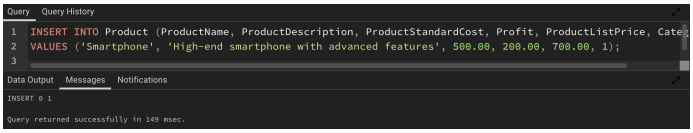
```
INSERT INTO Location (RegionName,
    ↪ CountryName, State, City,
    ↪ PostalCode)
VALUES ('North America', 'USA', '
    ↪ California', 'Los Angeles', '90001'
    ↪ );
```

Execution Result

Insert a new product

```
INSERT INTO Product (ProductName,
    ↪ ProductDescription,
    ↪ ProductStandardCost, Profit,
    ↪ ProductListPrice, CategoryID)
VALUES ('Smartphone', 'High-end␣
    ↪ smartphone␣with␣advanced␣features',
    ↪  500.00, 200.00, 700.00, 1);
```

Execution Result

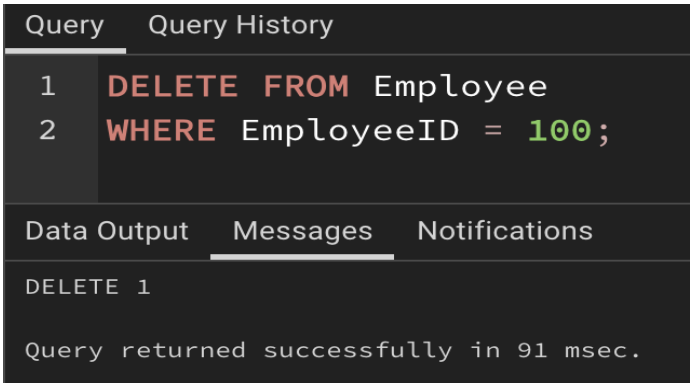


B. Delete Queries

Delete an employee

```
DELETE FROM Employee
WHERE EmployeeID = 100;
```

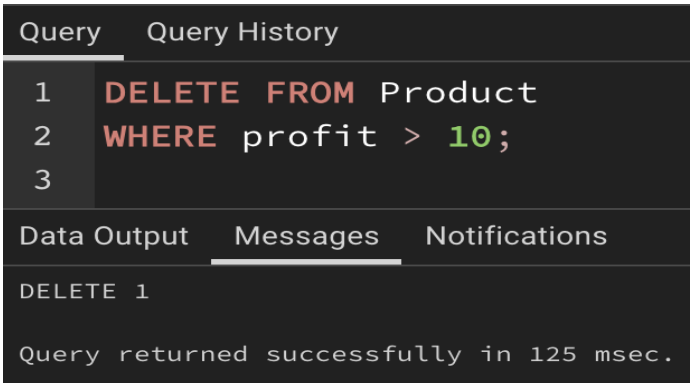
Execution Result



Delete product whose profit is greater than 10

```
DELETE FROM Product
WHERE profit > 10;
```

Execution Result

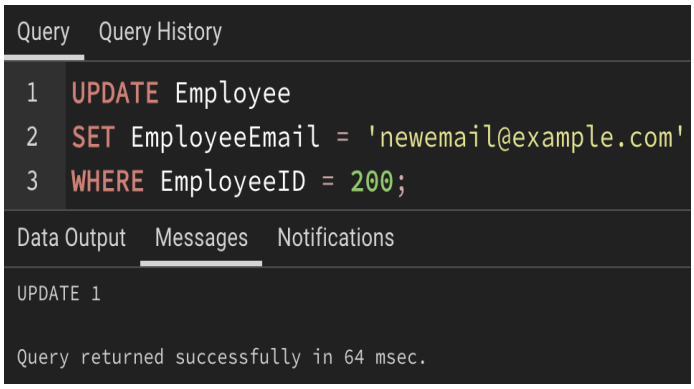


C. Update Queries

Update employee's email

```
UPDATE Employee
SET EmployeeEmail = 'newemail@example.com
    ↪ '
WHERE EmployeeID = 200;
```

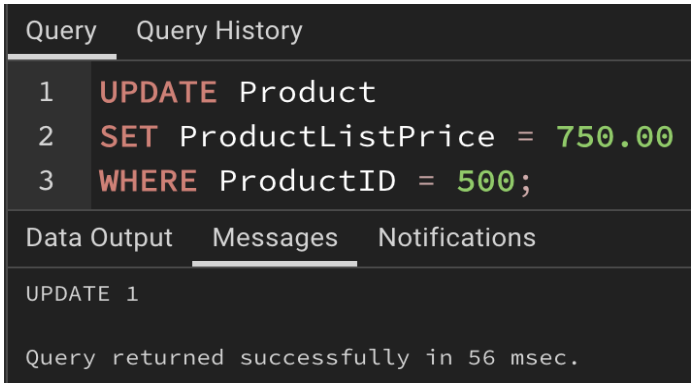
Execution Result



Update product's list price

```
UPDATE Product
SET ProductListPrice = 750.00
WHERE ProductID = 500;
```

Execution Result



D. Select Queries

Select all products with their categories:

```
SELECT p.ProductName, c.CategoryName
FROM Product p
INNER JOIN Category c ON p.CategoryID = c
    ↪ .CategoryID;
```

Select total quantity and revenue for each order:

```
SELECT OrderID, SUM(OrderItemQuantity) AS
    ↪  TotalQuantity, SUM(PerUnitPrice *
    ↪ OrderItemQuantity) AS TotalRevenue
FROM OrderItem
GROUP BY OrderID;
```

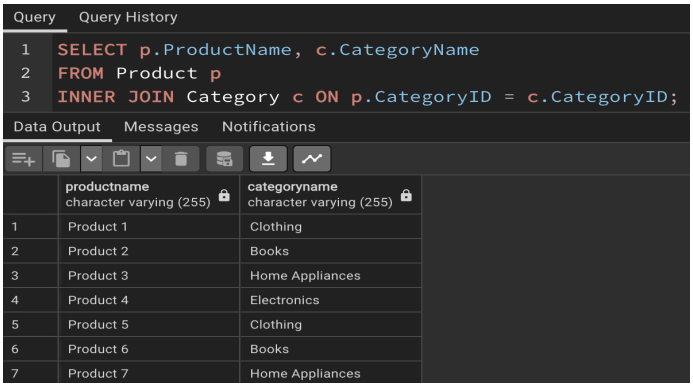
Select employees hired after a certain date:

```
SELECT *
FROM Employee
WHERE EmployeeHireDate > '2023-01-01';
```

Select customers with their total orders:

```
SELECT c.CustomerName, COUNT(o.OrderID)
    ↪ AS TotalOrders
FROM Customer c
LEFT JOIN OrderTable o ON c.CustomerID =
    ↪ o.CustomerID
GROUP BY c.CustomerName;
```

Execution Results

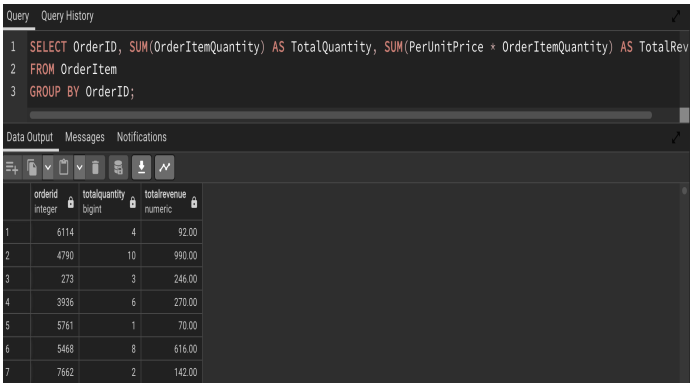


Query Query History

```
1  SELECT p.ProductName, c.CategoryName
2  FROM Product p
3  INNER JOIN Category c ON p.CategoryID = c.CategoryID;
```

Data Output Messages Notifications

	productname character varying (255)	categoryname character varying (255)
1	Product 1	Clothing
2	Product 2	Books
3	Product 3	Home Appliances
4	Product 4	Electronics
5	Product 5	Clothing
6	Product 6	Books
7	Product 7	Home Appliances

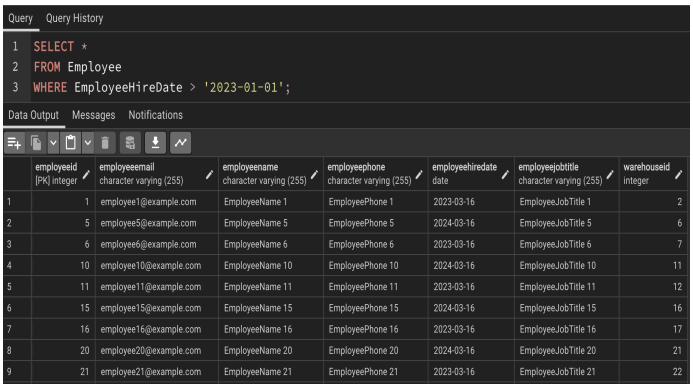


Query Query History

```
1  SELECT OrderID, SUM(OrderItemQuantity) AS TotalQuantity, SUM(PerUnitPrice * OrderItemQuantity) AS TotalRev
2  FROM OrderItem
3  GROUP BY OrderID;
```

Data Output Messages Notifications

	orderid integer	totalquantity bigint	totalrevenue numeric
1	6114	4	92.00
2	4790	10	990.00
3	273	3	246.00
4	3936	6	270.00
5	5761	1	70.00
6	5468	8	616.00
7	7662	2	142.00

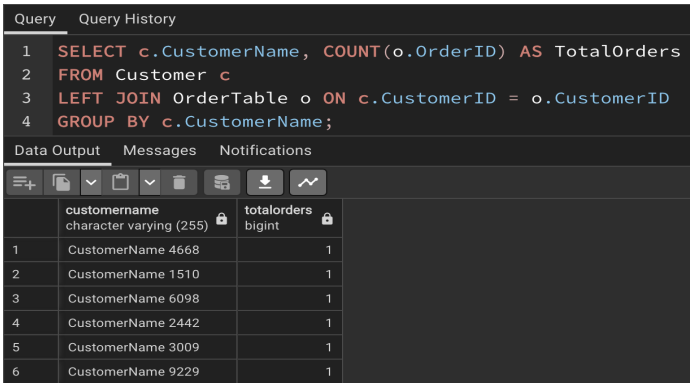


Query Query History

```
1  SELECT *
2  FROM Employee
3  WHERE EmployeeHireDate > '2023-01-01';
```

Data Output Messages Notifications

	employeeid [PK] integer	employeeemail character varying (255)	employeename character varying (255)	employeephone character varying (255)	employeehiredate date	employeejobtitle character varying (255)	warehouseid integer
1	1	employee1@example.com	EmployeeName 1	EmployeePhone 1	2023-03-16	EmployeeJobTitle 1	2
2	5	employee5@example.com	EmployeeName 5	EmployeePhone 5	2024-03-16	EmployeeJobTitle 5	6
3	6	employee6@example.com	EmployeeName 6	EmployeePhone 6	2023-03-16	EmployeeJobTitle 6	7
4	10	employee10@example.com	EmployeeName 10	EmployeePhone 10	2024-03-16	EmployeeJobTitle 10	11
5	11	employee11@example.com	EmployeeName 11	EmployeePhone 11	2023-03-16	EmployeeJobTitle 11	12
6	15	employee15@example.com	EmployeeName 15	EmployeePhone 15	2024-03-16	EmployeeJobTitle 15	16
7	16	employee16@example.com	EmployeeName 16	EmployeePhone 16	2023-03-16	EmployeeJobTitle 16	17
8	20	employee20@example.com	EmployeeName 20	EmployeePhone 20	2024-03-16	EmployeeJobTitle 20	21
9	21	employee21@example.com	EmployeeName 21	EmployeePhone 21	2023-03-16	EmployeeJobTitle 21	22



Query Query History

```
1  SELECT c.CustomerName, COUNT(o.OrderID) AS TotalOrders
2  FROM Customer c
3  LEFT JOIN OrderTable o ON c.CustomerID = o.CustomerID
4  GROUP BY c.CustomerName;
```

Data Output Messages Notifications

	customername character varying (255)	totalorders bigint
1	CustomerName 4668	1
2	CustomerName 1510	1
3	CustomerName 6098	1
4	CustomerName 2442	1
5	CustomerName 3009	1
6	CustomerName 9229	1

IX. Query Execution Analysis

To identify problematic queries and improve their performance, we can analyze their execution plans using the EXPLAIN command in PostgreSQL. Here are three example queries along with their execution plans and potential improvements:

A. Identify Problematic Queries

First, let's assume we have some suspicions or evidence like slow performance that the following queries are problematic:

Query A - Retrieving all products in a specific category

```
SELECT *
FROM Product
WHERE CategoryID = 3;
```

Query B - Joining order items to products to calculate total sales per product

```
SELECT p.ProductName, SUM(oi.
    ↪ OrderItemQuantity * oi.PerUnitPrice
    ↪ ) AS TotalSales
FROM Product p JOIN OrderItem oi ON p.
    ↪ ProductID = oi.ProductID
GROUP BY p.ProductName;
```

Query C - Fetching all orders for customers in a specific city

```
SELECT o.*
FROM OrderTable o
JOIN Customer c ON o.CustomerID = c.
    ↪ CustomerID
WHERE c.CustomerAddress LIKE '%
    ↪ CustomerAddress␣5%';
```

B. Use EXPLAIN to Analyze Queries

We use the EXPLAIN command to analyze these queries:

Query A - Retrieving all products in a specific category

```
EXPLAIN SELECT *
FROM Product
WHERE CategoryID = 3;
```

Result:

```
Seq Scan on product  (cost=0.00..239.00
    ↪ rows=2500 width=58)
  Filter: (categoryid = 3)
```

The EXPLAIN output for the query SELECT * FROM Product WHERE CategoryID = 3; indicates that PostgreSQL is performing a sequential scan on the entire Product table to find records where CategoryID equals 3. This means that it reads every row in the table and applies the filter to check if CategoryID equals 3. This approach can be inefficient, especially if the Product table contains a large number of rows and only a small subset of them have CategoryID equal to 3. Here's the breakdown of the query plan and suggested optimizations:

- **Seq Scan on product:** PostgreSQL scans each row in the Product table.
- **Cost:** The estimated startup cost is 0.00, and the total cost goes up to 239.00. The total cost is relatively high, indicating the inefficiency of scanning the entire table.
- **Rows:** PostgreSQL estimates that it will find 2500 rows that meet the filter condition, which is a potentially large number depending on the total number of rows in the table.
- **Width:** Average row size is estimated to be 58 bytes.

Query B - Joining order items to products to calculate total sales per product

```
EXPLAIN SELECT p.ProductName, SUM(oi.
    ↪ OrderItemQuantity * oi.PerUnitPrice
    ↪ ) AS TotalSales
FROM Product p JOIN OrderItem oi ON p.
    ↪ ProductID = oi.ProductID
GROUP BY p.ProductName;
```

Result:

```
HashAggregate  (cost=639.26..764.26 rows
    ↪ =10000 width=44)
  Group Key: p.productname
  ->  Hash Join  (cost=339.00..539.26
      ↪ rows=10000 width=21)
        Hash Cond: (oi.productid = p.
            ↪ productid)
        ->  Seq Scan on orderitem oi  (
            ↪ cost=0.00..174.00 rows
            ↪ =10000 width=13)
        ->  Hash  (cost=214.00..214.00
            ↪ rows=10000 width=16)
              ->  Seq Scan on product p
                  ↪ (cost=0.00..214.00
                  ↪ rows=10000 width=16)
```

The EXPLAIN output for our query shows the PostgreSQL query planner's strategy for executing a join between the Product and OrderItem tables, followed by an aggregation operation. Here's the breakdown of the query plan and suggested optimizations:

- **Hash Join:** The query planner decides to use a hash join to combine the Product and OrderItem tables. The Hash Join operation builds a hash table on the smaller table (Product) in memory and then scans the larger table (OrderItem) to find matching rows.
- **Sequential Scans:** Both tables (Product and OrderItem) are scanned sequentially (Seq Scan), indicating that there are no useful indexes for this join condition on productID.
- **Hash Aggregate:** After joining the data, a hash-based aggregation is performed to compute the sum of OrderItemQuantity * PerUnitPrice grouped by ProductName.
- **Cost and Rows Estimates:** The total cost of executing the query is estimated from 339.00 to 539.26, with the aggregation adding additional cost up to 764.26.

Query C - Fetching all orders for customers in a specific city

```
EXPLAIN SELECT o.*
FROM OrderTable o
JOIN Customer c ON o.CustomerID = c.
    ↪ CustomerID
WHERE c.CustomerAddress LIKE '%
    ↪ CustomerAddress␣5%';
```

Result:

```
Hash Join  (cost=291.89..482.15 rows=1111
    ↪  width=21)
  Hash Cond: (o.customerid = c.customerid
      ↪ )
  ->  Seq Scan on ordertable o  (cost
      ↪ =0.00..164.00 rows=10000 width
      ↪ =21)
  ->  Hash  (cost=278.00..278.00 rows
      ↪ =1111 width=4)
        ->  Seq Scan on customer c  (cost
            ↪ =0.00..278.00 rows=1111
            ↪ width=4)
              Filter: (customeraddress ~~
                  ↪  '%CustomerAddress␣5%
                  ↪ '::text)
```

The EXPLAIN output for the query involving a join between OrderTable and Customer tables filtered by CustomerAddress using a LIKE operator. Here's the breakdown of the query plan and suggested optimizations:

- **Hash Join:** Merges OrderTable and Customer tables based on customerid using a hash table.
- **Sequential Scans:** Both OrderTable and Customer tables scanned sequentially due to absence of suitable indexes.
- **Hash Conditions:** Hash condition applied to match customerid between the tables.
- **Cost and Rows Estimates:** Total cost of execution estimated between 291.89 and 482.15 units, expecting 1111 rows; customer table scanned for 1111 rows to filter CustomerAddress 5.

C. Suggest Improvements

Query A - Retrieving all products in a specific category Indexing: Adding an index on the CategoryID column will significantly improve the performance of this query. An index allows PostgreSQL to quickly locate rows with a specific CategoryID, avoiding a full table scan.

```
CREATE INDEX idx_product_categoryid ON
    ↪ Product(CategoryID);
```

Query B - Joining order items to products to calculate total sales per product Create Indexes on Join Columns: Since both tables are using a sequential scan for the join, adding indexes on the productID columns in both Product and OrderItem tables will help PostgreSQL use more efficient index scans instead of sequential scans.

```
CREATE INDEX idx_product_productid ON
    ↪ Product(ProductID);
CREATE INDEX idx_orderitem_productid ON
    ↪ OrderItem(ProductID);
```

Query C - Fetching all orders for customers in a specific city Indexing: Ensuring that there are indexes on CustomerID for both tables (OrderTable and Customer), if not already present. This will help the hash join operation to be more efficient.

```
CREATE INDEX IF NOT EXISTS
    ↪ idx_customer_customerid ON Customer
    ↪ (CustomerID);
CREATE INDEX IF NOT EXISTS
    ↪ idx_ordertable_customerid ON
    ↪ OrderTable(CustomerID);
```

Functional Index for LIKE Queries: Given that the filter uses LIKE '%CustomerAddress 5%', which is a suffix search, consider creating a functional index on the reversed CustomerAddress if this type of query is common. This would allow the index to be used for suffix searches.

```
CREATE INDEX idx_customer_reverse_address
    ↪  ON Customer (REVERSE(
    ↪ CustomerAddress));
```

Then, modify the query to use the index:

```
SELECT o.*
FROM OrderTable o
JOIN Customer c ON o.CustomerID = c.
    ↪ CustomerID
WHERE REVERSE(c.CustomerAddress) LIKE
    ↪ REVERSE('%CustomerAddress␣5%');
```

D. Compare Changes after Improvements

Query A - Retrieving all products in a specific category

```
EXPLAIN SELECT *
FROM Product
WHERE CategoryID = 3;
```

Result:

```
Bitmap Heap Scan on product  (cost
    ↪ =31.66..176.91 rows=2500 width=58)
  Recheck Cond: (categoryid = 3)
  ->  Bitmap Index Scan on
      ↪ idx_product_categoryid  (cost
      ↪ =0.00..31.04 rows=2500 width=0)
        Index Cond: (categoryid = 3)
```

Conclusion: After optimization, the cost of executing the query significantly decreased from approximately 239.00 to 176.91 units. Additionally, the estimated number of rows to be processed remained the same at 2500, but the execution plan changed to utilize an index scan instead of a sequential scan, resulting in a more efficient retrieval of rows matching the filter condition.

For **Query B - Joining order items to products to calculate total sales per product** and **Query C - Fetching all orders for customers in a specific city**, we have already performed the necessary indexing operations as part of VI. PROBLEMS HANDLING LARGE DATABASES and VII. PROPOSED SOLUTIONS; therefore giving us optimal cost. Here are the optimizations:

```
CREATE INDEX idx_OrderTable_CustomerID
ON OrderTable (CustomerID);

CREATE INDEX idx_OrderItem_OrderID
ON OrderItem (OrderID);

CREATE INDEX idx_OrderItem_ProductID
ON OrderItem (ProductID);
```

REFERENCES

[1] Kaggle. (n.d.). *Inventory management dataset*. Retrieved from https://www.kaggle.com/datasets/hetulparmar/inventory-management-dataset

[2] PostgreSQL. (n.d.). *PostgreSQL documentation*. Retrieved from https://www.postgresql.org/docs/current/index.html

[3] pgAdmin. (n.d.). *pgAdmin 4 documentation*. Retrieved from https://www.pgadmin.org/docs/pgadmin4/8.3/index.html

[4] pgAdmin. (n.d.). *ERD tool documentation*. Retrieved from https://www.pgadmin.org/docs/pgadmin4/latest/erd_tool.html