Project Report - Enterprise Resource Planning

Anup Atul Thakkar*
Computer Science and Engineering
University at Buffalo
Buffalo, United States
anupatul@buffalo.edu

Pushkaraj Joshi*
Computer Science and Engineering
University at Buffalo
Buffalo, United States
pjoshi6@buffalo.edu

Sagar Jitendra Thacker*
Computer Science and Engineering
University at Buffalo
Buffalo, United States
sagarjit@buffalo.edu

***Abstract*—In modern organizations, it's important to manage resources, both for organizational efficiency and auditing purposes. More often than not, organizations deal with a lot of inventory items, vendors, employees, customers, etc. If they don't have an efficient way to handle the data, then operations will get messy real fast. This is apart from the audit-related roadblocks they will face when it's time to report their financials at year-end. To solve this problem, organizations maintain something called Enterprise Resource Planning (ERP), which can go into hundreds, if not thousands of tables.**

***Index Terms*—sql, database, streamlit, python, postgresql**

I. Introduction

ERPs ideally involve complex databases that are carefully managed by a host of database operators or admins. We need to use database systems, because if we use Excel files to maintain the data, then the amount of time needed to make updates to the different related tables will be really complicated and would consume a lot of man-hours, which is expensive. If the number of tables/files is large, then the cost of maintaining the excel files will be prohibitive. This is excluding human error because human maintenance is not the perfect way to deal with large amounts of data. A simple "0" error would cause massive disruptions in the operations and would take a lot of time to correct, which would again mean deploying excess resources for mitigating such issues. A database management system would be a perfect system to maintain an ERP. By defining relations, we would be enforcing integrity and reducing the number of errors present in the data. This is apart from the efficiency gains in CRUD operations, which would lead to a drastic improvement in resource planning and a reduction in cost for mitigating issues.

II. Target User

- **The User**: Anyone who needs to manage the resources in their organization, such as orders, inventory, supplier information, etc. would be a user.
- **The Administer**: Anyone who needs to do control checks (such as supplier and purchaser is not the same person), or maintain the database characteristics (Integrity, tables, backups, etc) would be an administrator.
- **Real-life scenario description**: Our goal is to simulate a simple system for entering data into an ERP and allow users to do basic querying.

* Equal Contribution

III. Database

A. Schema

We have used postgresql as our database for this project. The schema contains 12 tables:

- customers
- categories
- categories_description
- supplier
- products
- territories
- employees
- employee_territories
- shippers
- postal_address_lookup
- orders
- order_details

The customers table contain all the details related to the customer such as name, title, postal code, country, etc. The categories tables contain category names of different category and categories description has multiple subcategories within each category. Supplier contains details company name, title, postal code, etc with respect to a supplier. Products contains name, quantity per unit, discontinued or not, which supplier sends the product and to which category it belongs to. Territories contains territory id, names. Employees contains first & last name, title, contact, etc. Employee territory tells us to which territory(ies) a employee is assigned to. Shipper has details of all the shipper used to ship the products and postal address lookup table maps postal code and country to a city. The most important table are orders and order details where order contains details of a single order and order details list the products, quantity, discount, etc for each product in that order.

B. Dataset

The dataset for this project was manually augmented using python script. The most rows in the tables are in the order and order_details table with approximately 10k and 25k rows respectively.

C. Dataset modification and departure from Milestone 1

In milestone 1 there were 10 tables and some tables were not in BCNF form. We decomposed categories into two tables i.e., categories and categories description because categories had a column which had multiple values. Also, We remove city attribute from the respective tables and created a new table called postal_address_lookup to bring the tables in BCNF form. We have also created one function and one trigger for employee & employee territory tables. Whenever a new employee is added to the database then they get mapped to a random territory. We have also added indexes to improve complex query performance.

D. Constraints in the Database

Various constraints were added to the tables in order to ensure the integrity of the databases. All the foreign keys, referencing the primary key from the parent table, are modified whenever the value in the parent table changes. For example if the value in the parent table changes, then the value "Cascades" in the child table through the following command after the foreign key.

```
ON UPDATE CASCADE
```

If the value in the parent table is deleted, then the value is set to "NULL" in the child table through the following command after the foreign key.

```
ON DELETE SET NULL
```

E. Attributes of the Tables

- Categories.
  - id: The Primary key.
  - name: Name of the type/category of the products.
- FDs in table: id $\rightarrow name$
- Supplier.
  - id: The Primary key.
  - company_name: Company name of the supplier.
  - contact_name: Contact persons name for the company.
  - contact_title: Role of the contact person.
  - postal_code: Postal Code of suppliers city.
  - country: Suppliers country.
  - contact: Phone number of the contact person.
- FDs in table: id$\rightarrow$ $company\_name, contact\_name, postal\_code, contact$
- Products.
  - id: The Primary key.
  - name: Name of the product.
  - supplier_id: Supplier ID of the supplier proving the product.
  - category_id: Category of the product.
  - quantity_per_unit: Number of pieces in 1 unit of the product.
  - unit_price: Price of Product per unit.
  - stock: Number of available product count.
  - discontinued: Boolean field specifying whether the product is still active or discontinued.
- FDs in table: id$\rightarrow$ $name, supplier\_id, category\_id, discontinued, quantity\_per\_unit, unit\_price, stock$
- Categories_Description.
  - category_id: The Primary key.
  - name: Description of the category.
- Order_Details.
  - order_id: The Primary key.
  - product_id: ID of the ordered product.
  - unit_price: Price of the unit quantity of the product.
  - quantity: Quantity of the product ordered.
  - discount: Discount applied on the order.
- FDs in table: order_id$\rightarrow$ $product\_id, unit\_price, quantity, discount$
- Customers.
  - id: The Primary key.
  - name: Name of the company of the customer.
  - contact_name: Point of contact from the customer company.
  - title: Designation of the point of contact.
  - postal_code: Postal code for the customer company.
  - country: Country where the customer company is located.
  - contact: Phone number of the point of contact from the customer company.
- FDs in table: id$\rightarrow$ $name, contact\_name, title, postal\_code, country, contact$
- Orders.
  - id: The Primary key
  - customer_id: ID of the customer making the order.
  - employee_id: ID of the employee facilitating the order.
  - order_date: Date on which the order was placed.
  - delivery_date: Date on which the order was delivered.
  - shipped_date: Date on which the order was shipped to the customer.
  - shipper_id: Refers to the id shipper who shipped the order.
  - weight: Weight of the order being shipped.
  - ship_name: Name of the ship being used to ship the product.
  - ship_postal_code: Postal code where the order is being shipped.
  - ship_country: Country where the order is being shipped.
- FDs in table: id$\rightarrow$ $customer\_id, employee\_id, order\_date, delivery\_date, shipped\_date, shipper\_id, weight, ship\_name, ship\_postal\_code, ship\_country$
- Shippers.
  - id: The Primary key
  - name: Name of the shipper.
  - contact: Contact number of the shipper.
- FDs in table: id$\rightarrow$ $name, contact$
- Postal Address Lookup.
  - postal_code: Postal Code unique with respect to the country.

  - country: Country which defines where the postal code belong.
  - city: City association based on the postal code and the country.
- FDs in table: postal_code, country$\rightarrow city$
- Employees.
  - id: The primary key.
  - first_name: The first name of the employee.
  - last_name: The last name of the employee.
  - title: The designation of the employee.
  - birthdate: The birth date of the employee.
  - hire_date: The date on which the employee was hired.
  - postal_code: The postal code of the city where the employee is located.
  - country: The country where the employee is located.
  - reports_to: The id of the employee which the employee reports to.
  - contact: The contact number of the employee.
- FDs: id$\rightarrow first_name, last_name, title, birthdate, hire_date, contact, reports_to, postal_code, country$
- Territories.
  - id: The territory id.
  - name: Name of the territory.
- FDs in table: id$\rightarrow name.$
- Employee Territories.
  - employee_id: Employee ID.
  - territory_id: The ID of the territory assigned to the employee.
- FDs in table: employee_id$\rightarrow territory_id.$

IV. Website

To demonstrate our final project, we have built a website. The details of both the front-end and back-end are mentioned below.

A. Setting up Python Environment and Database Connection:

To run the web app, all the dependencies needs to be installed. We have provided with an python script called 'requirements.py' which will take care of all the dependent components that are required for the app to work. We have used psycopg2 which is a Python-PostgreSQL Database Adapter. To set database environment variables such as DATABASE, USER, PASSWORD, and HOST, we have provided an 'config.py' file.

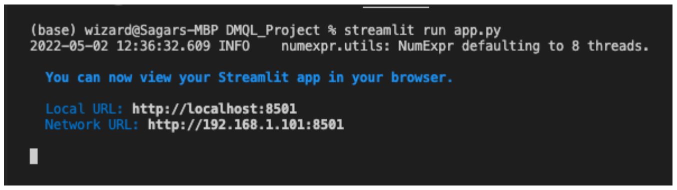
B. Project Directory Overview

The 'Application Code' directory contains 'app.py' and a sub-directory called 'pages' which contains different pages of our web app. We also have 'multipage.py', 'config.py' and 'sql_connect.py' in main directory which are used structuring and configuring our app.

C. Running the Code

The file 'app.py' is responsible to start the streamlit server by default on the local host and on port number 8501. It is in the main project directory and it can be run by the following command:

```
streamlit run app.py
```



```
(base) wizard@Sagars-MBP DMQL_Project % streamlit run app.py
2022-05-02 12:36:32.609 INFO    numexpr.utils: NumExpr defaulting to 8 threads.

  You can now view your Streamlit app in your browser.

  Local URL: http://localhost:8501
  Network URL: http://192.168.1.101:8501

```

Fig. 1. Streamlit run command

D. Back-end

We used a python framework called Streamlit to develop our website backend. Our main file is 'app.py' located in the Application Code directory which is responsible for starting the web app. To add the multi-page functionality we have a python script for each page which can be located under the pages directory. When you run the web app, the first page is the home page.

1) Home Page: This page is responsible for displaying each table and an option to perform insert, update, and delete for that respective table. User can navigate different tables using the drop down in the app navigation panel to view different tables. We have also implemented pagination feature for huge tables. To view each table and perform DML commands we have handled different SQL commands in the backend.

2) Handling Form issues in Streamlit: Everytime an action is performed on the Streamlit the app gets refreshed. Because of this all the data inside the input fields of form gets refreshed on submission. To resolve this and store the values of the input fields of the form Session State of the Streamlit were used. In session state one can create a variable and store the value in that variable. Even when the app is refreshed the value is retained in the session state variable and can be used after the app has refreshed. Hence using session states the values entered in the input fields of the form were retained and used for the respective operations.

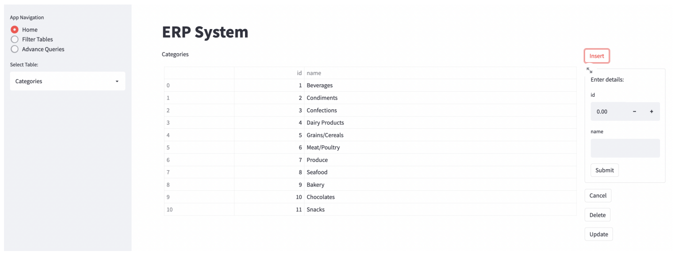
3) Filter Tables Page: This page provides a lot of flexibility to the user to slice through different tables based on different filters. In simple terms, user can play with SELECT statement in SQL using the UI components. Again, here the user has the freedom to choose table, choice of attribute to view and apply different algebraic, boolean, logical, and operators like BETWEEN, LIKE. Based on the filter values and attributes the appropriate SQL command is executed to fetch the results from the database.

4) Advanced Queries Page: In this page we have defined various use cases related to our ERP system. Each use cases captures various database concepts such as joins over multiple tables, order by, group by and having operations; along with various other filtering options based on where clause. User can choose values for each filters/fields and based on those values each time a SQL query is executed in the backend and the result is displayed in the form of a table.

E. Front-end

After starting the web app, we land upon the home page of the web app.

1) Home Page: In fig 2, on the left side which contains the navigational panel, home option is selected by default which indicates we are on the home page. Below which we can observe other radio buttons that let's us navigate to different pages in our web app. Below that there is a drop down that let's user to choose which table to show and that table is displayed in the middle. Right of the table we can observe three button - Insert, Update, and Delete; upon clicked opens a form in which values can be filled based on each table and the respective SQL command is executed on submit.



Select Table: Categories

Categories

	id	name
0	1	Beverages
1	2	Condiments
2	3	Confections
3	4	Dairy Products
4	5	Grains/Cereals
5	6	Meat/Poultry
6	7	Produce
7	8	Seafood
8	9	Bakery
9	10	Chocolates
10	11	Snacks

Enter details:
id: 0.00
name:

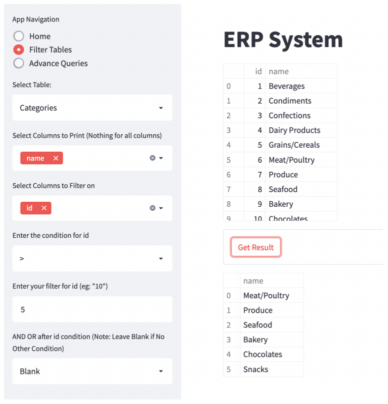
Fig. 2. Home Page

2) Filter Tables Page: In fig 3, we can observe an snippet of how the page works. On the left hand, Filter tables option is selected indicating we are on that page and below that user can select various options to choose from on how to slice the table. Based on the values user mentions, a SQL query gets executed in the backend when 'Get Result' button is clicked. The result of the query is displayed below the button. Table above the button is the original table displayed for the reference.

3) Advance Queries Page: In fig 4, we can observe that on the navigation panel Advance Queries option is selected and below that there is a drop down that contains a list of different use cases. Use cases are independent and cater different aspects of ERP systems. Also, each use case has different set of filters based on which the underlying query changes and every time the value changes the corresponding query with new values gets executed and the result is displayed.

V. Query Optimization

To build reliable and scalable system it is necessary to optimize the system performance.



Select Table: Categories

Select Columns to Print (Nothing for all columns): name

Select Columns to Filter on: id

Enter the condition for id: >

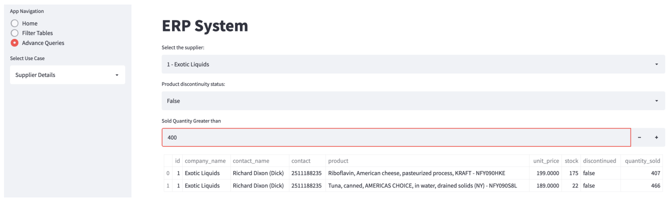
Enter your filter for id (eg: "10"): 5

AND OR after id condition (Note: Leave Blank if No Other Condition): Blank

	id	name
0	1	Beverages
1	2	Condiments
2	3	Confections
3	4	Dairy Products
4	5	Grains/Cereals
5	6	Meat/Poultry
6	7	Produce
7	8	Seafood
8	9	Bakery
9	10	Chocolates

	name
0	Meat/Poultry
1	Produce
2	Seafood
3	Bakery
4	Chocolates
5	Snacks

Fig. 3. Filter Tables Page



Select Use Case: Supplier Details

Select the supplier: 1 - Exotic Liquids

Product discontinuity status: False

Sold Quantity Greater than: 400

	id	company_name	contact_name	contact	product	unit_price	stock	discontinued	quantity_sold
0	1	Exotic Liquids	Richard Dixon (Dick)	2511188235	Riboflavin, American cheese, pasteurized process, KRAFT - NFY090HKE	199.0000	175	false	407
1	1	Exotic Liquids	Richard Dixon (Dick)	2511188235	Tuna, canned, AMERICAS CHOICE, in water, drained solids (NY) - NFY090S8L	189.0000	22	false	466

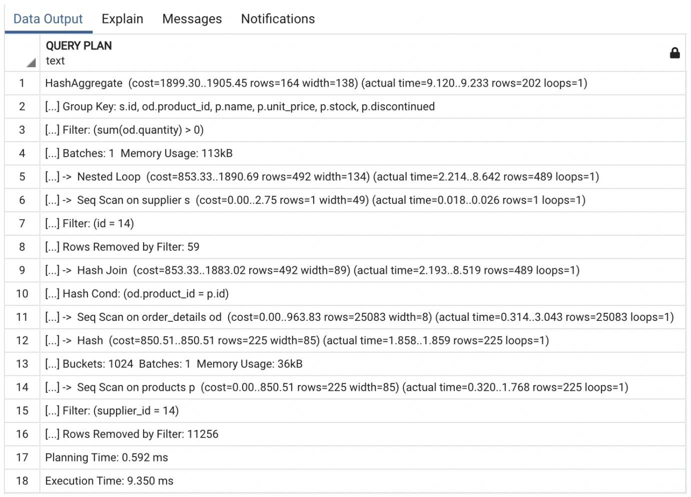
Fig. 4. Advance Queries Page

A. Problematic queries

1) Query 1:

```
SELECT
    s.id,
    s.company_name,
    s.contact_name,
    s.contact,
    p.name AS product,
    p.unit_price,
    p.stock,
    p.discontinued,
    SUM(od.quantity) AS quantity_sold
FROM
    supplier s,
    products p,
    order_details od
WHERE
    s.id = p.supplier_id
    AND p.id = od.product_id
    and s.id = '14'
GROUP BY
    od.product_id,
    s.id,
    s.company_name,
    s.contact_name,
    s.contact,
    p.name,
    p.unit_price,
    p.stock,
    p.discontinued
```

```
HAVING
    SUM(od.quantity) > 0
ORDER BY
    s.id;
```



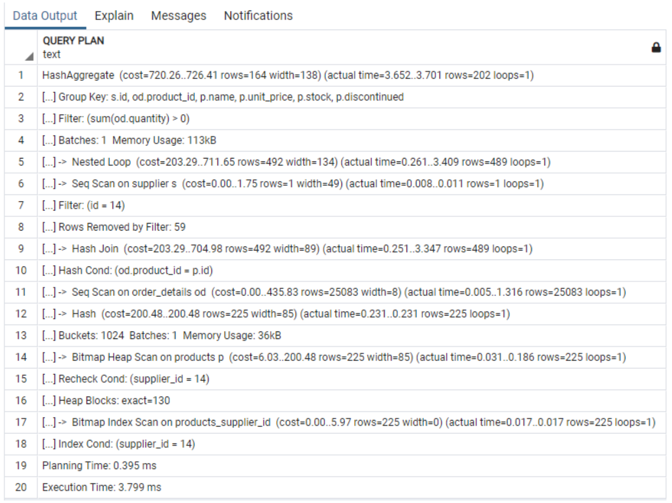
Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	HashAggregate (cost=1899.30..1905.45 rows=164 width=138) (actual time=9.120..9.233 rows=202 loops=1)
2	[...] Group Key: s.id, od.product_id, p.name, p.unit_price, p.stock, p.discontinued
3	[...] Filter: (sum(od.quantity) > 0)
4	[...] Batches: 1 Memory Usage: 113kB
5	[...] -> Nested Loop (cost=853.33..1890.69 rows=492 width=134) (actual time=2.214..8.642 rows=489 loops=1)
6	[...] -> Seq Scan on supplier s (cost=0.00..2.75 rows=1 width=49) (actual time=0.018..0.026 rows=1 loops=1)
7	[...] Filter: (id = 14)
8	[...] Rows Removed by Filter: 59
9	[...] -> Hash Join (cost=853.33..1883.02 rows=492 width=89) (actual time=2.193..8.519 rows=489 loops=1)
10	[...] Hash Cond: (od.product_id = p.id)
11	[...] -> Seq Scan on order_details od (cost=0.00..963.83 rows=25083 width=8) (actual time=0.314..3.043 rows=25083 loops=1)
12	[...] -> Hash (cost=850.51..850.51 rows=225 width=85) (actual time=1.858..1.859 rows=225 loops=1)
13	[...] Buckets: 1024 Batches: 1 Memory Usage: 36kB
14	[...] -> Seq Scan on products p (cost=0.00..850.51 rows=225 width=85) (actual time=0.320..1.768 rows=225 loops=1)
15	[...] Filter: (supplier_id = 14)
16	[...] Rows Removed by Filter: 11256
17	Planning Time: 0.592 ms
18	Execution Time: 9.350 ms

Fig. 5. Problematic Query 1

The fig 5 shows the analysis of the query. As we can see that there are multiple tables involved and multiple joins happening on the basis of the foreign keys in the table the execution time and cost of the query was higher.

We decided to use index on the product_id of order_details table and supplier_id in products table. Indexing of these columns on which the joins are happening led to faster retrieval of the results and in less cost. The result of the analysis of the same query after indexing can be observed in fig 6.



Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	HashAggregate (cost=720.26..726.41 rows=164 width=138) (actual time=3.652..3.701 rows=202 loops=1)
2	[...] Group Key: s.id, od.product_id, p.name, p.unit_price, p.stock, p.discontinued
3	[...] Filter: (sum(od.quantity) > 0)
4	[...] Batches: 1 Memory Usage: 113kB
5	[...] -> Nested Loop (cost=203.29..711.65 rows=492 width=134) (actual time=0.261..3.409 rows=489 loops=1)
6	[...] -> Seq Scan on supplier s (cost=0.00..1.75 rows=1 width=49) (actual time=0.008..0.011 rows=1 loops=1)
7	[...] Filter: (id = 14)
8	[...] Rows Removed by Filter: 59
9	[...] -> Hash Join (cost=203.29..704.98 rows=492 width=89) (actual time=0.251..3.347 rows=489 loops=1)
10	[...] Hash Cond: (od.product_id = p.id)
11	[...] -> Seq Scan on order_details od (cost=0.00..435.83 rows=25083 width=8) (actual time=0.005..1.316 rows=25083 loops=1)
12	[...] -> Hash (cost=200.48..200.48 rows=225 width=85) (actual time=0.231..0.231 rows=225 loops=1)
13	[...] Buckets: 1024 Batches: 1 Memory Usage: 36kB
14	[...] -> Bitmap Heap Scan on products p (cost=6.03..200.48 rows=225 width=85) (actual time=0.031..0.186 rows=225 loops=1)
15	[...] Recheck Cond: (supplier_id = 14)
16	[...] Heap Blocks: exact=130
17	[...] -> Bitmap Index Scan on products_supplier_id (cost=0.00..5.97 rows=225 width=0) (actual time=0.017..0.017 rows=225 loops=1)
18	[...] Index Cond: (supplier_id = 14)
19	Planning Time: 0.395 ms
20	Execution Time: 3.799 ms

Fig. 6. Optimized Query 1

In fig 6, we can see various improvements over fig 5. For example, the hash aggregate cost for the non-indexed query is significantly higher as compared to the indexed query. Another interesting statistic to look at is the nested loop cost, which is again significantly higher for the non-indexed query as compared to the indexed query. Most importantly, the execution time of the indexed query is far less as compared to the non-indexed query.

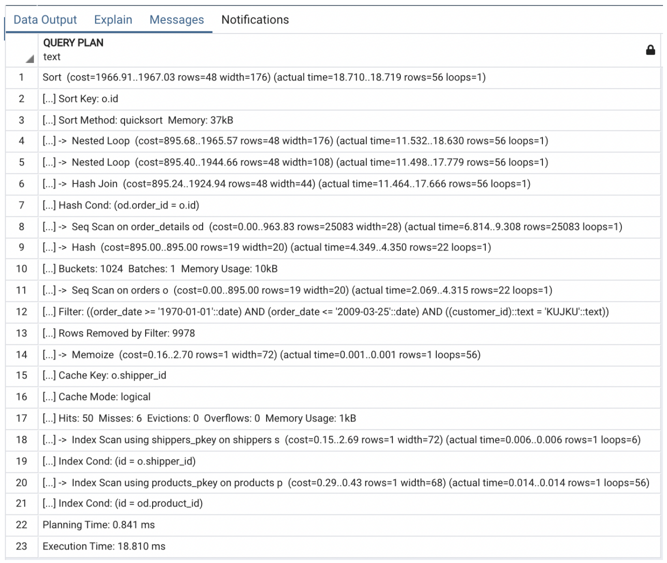
2) Query 2:

```
Select
    o.id as order_id,
    o.order_date,
    o.shipped_date,
    o.delivery_date,
    s.name as shippers_name,
    p.name as product_name,
    od.quantity,
    od.unit_price,
    od.discount,
    od.quantity * od.unit_price *
    (1 - od.discount) as total_price
from
    orders o,
    shippers s,
    order_details od,
    products p
where
    o.shipper_id = s.id
    and o.id = od.order_id
    and od.product_id = p.id
    and o.customer_id = 'KUJKU'
    AND o.order_date BETWEEN '1970-01-01'
    and '2009-03-25'
order by
    o.id;
```

The image 7 shows the analysis of another problematic query. The query is used to get the details of the customers and their corresponding orders and its details. As we can see that four different tables are joined to get the desired output. Also along with the joins, there are certain filters imposed like order dates between two dates and customer id equal to customer id of a certain customer. The retrieval of the results is costly and time consuming as we can see from the below analysis of the query

To improve the process of retrieving the results for the query, we decided to add indexes on the columns which were used for joining the four different tables. Indexes were applied on order_id, product_id of the order_details table and customer_id of the order table. This resulted in fast and cost efficient retrieval of the result. The analysis of the query after indexing is shown in the below image

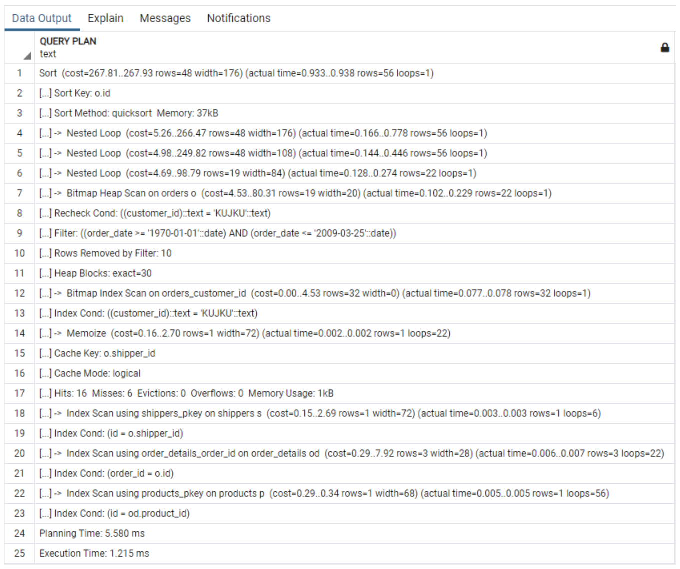
In fig 8, we can see various improvements over fig 7. For example, the sort cost for the non-indexed query is significantly higher as compared to the indexed query. Another interesting statistic to look at is the nested loop cost, which is again



```
Data Output   Explain   Messages   Notifications

     QUERY PLAN
     text
1    Sort  (cost=1966.91..1967.03 rows=48 width=176) (actual time=18.710..18.719 rows=56 loops=1)
2    [...] Sort Key: o.id
3    [...] Sort Method: quicksort  Memory: 37kB
4    [...] ->  Nested Loop  (cost=895.68..1965.57 rows=48 width=176) (actual time=11.532..18.630 rows=56 loops=1)
5    [...] ->  Nested Loop  (cost=895.40..1944.66 rows=48 width=108) (actual time=11.498..17.779 rows=56 loops=1)
6    [...] ->  Hash Join  (cost=895.24..1924.94 rows=48 width=44) (actual time=11.464..17.666 rows=56 loops=1)
7    [...] Hash Cond: (od.order_id = o.id)
8    [...] ->  Seq Scan on order_details od  (cost=0.00..963.83 rows=25083 width=28) (actual time=6.814..9.308 rows=25083 loops=1)
9    [...] ->  Hash  (cost=895.00..895.00 rows=19 width=20) (actual time=4.349..4.350 rows=22 loops=1)
10   [...] Buckets: 1024  Batches: 1  Memory Usage: 10kB
11   [...] ->  Seq Scan on orders o  (cost=0.00..895.00 rows=19 width=20) (actual time=2.069..4.315 rows=22 loops=1)
12   [...] Filter: ((order_date >= '1970-01-01'::date) AND (order_date <= '2009-03-25'::date) AND ((customer_id)::text = 'KUJKU'::text))
13   [...] Rows Removed by Filter: 9978
14   [...] ->  Memoize  (cost=0.16..2.70 rows=1 width=72) (actual time=0.001..0.001 rows=1 loops=56)
15   [...] Cache Key: o.shipper_id
16   [...] Cache Mode: logical
17   [...] Hits: 50  Misses: 6  Evictions: 0  Overflows: 0  Memory Usage: 1kB
18   [...] ->  Index Scan using shippers_pkey on shippers s  (cost=0.15..2.69 rows=1 width=72) (actual time=0.006..0.006 rows=1 loops=6)
19   [...] Index Cond: (id = o.shipper_id)
20   [...] ->  Index Scan using products_pkey on products p  (cost=0.29..0.43 rows=1 width=68) (actual time=0.014..0.014 rows=1 loops=56)
21   [...] Index Cond: (id = od.product_id)
22   Planning Time: 0.841 ms
23   Execution Time: 18.810 ms
```

Fig. 7. Problematic Query 2



```
Data Output   Explain   Messages   Notifications

     QUERY PLAN
     text
1    Sort  (cost=267.81..267.93 rows=48 width=176) (actual time=0.933..0.938 rows=56 loops=1)
2    [...] Sort Key: o.id
3    [...] Sort Method: quicksort  Memory: 37kB
4    [...] ->  Nested Loop  (cost=5.26..266.47 rows=48 width=176) (actual time=0.166..0.778 rows=56 loops=1)
5    [...] ->  Nested Loop  (cost=4.98..249.82 rows=48 width=108) (actual time=0.144..0.446 rows=56 loops=1)
6    [...] ->  Nested Loop  (cost=4.69..98.79 rows=19 width=84) (actual time=0.128..0.274 rows=22 loops=1)
7    [...] ->  Bitmap Heap Scan on orders o  (cost=4.53..80.31 rows=19 width=20) (actual time=0.102..0.229 rows=22 loops=1)
8    [...] Recheck Cond: ((customer_id)::text = 'KUJKU'::text)
9    [...] Filter: ((order_date >= '1970-01-01'::date) AND (order_date <= '2009-03-25'::date))
10   [...] Rows Removed by Filter: 10
11   [...] Heap Blocks: exact=30
12   [...] ->  Bitmap Index Scan on orders_customer_id  (cost=0.00..4.53 rows=32 width=0) (actual time=0.077..0.078 rows=32 loops=1)
13   [...] Index Cond: ((customer_id)::text = 'KUJKU'::text)
14   [...] ->  Memoize  (cost=0.16..2.70 rows=1 width=72) (actual time=0.002..0.002 rows=1 loops=22)
15   [...] Cache Key: o.shipper_id
16   [...] Cache Mode: logical
17   [...] Hits: 16  Misses: 6  Evictions: 0  Overflows: 0  Memory Usage: 1kB
18   [...] ->  Index Scan using shippers_pkey on shippers s  (cost=0.15..2.69 rows=1 width=72) (actual time=0.003..0.003 rows=1 loops=6)
19   [...] Index Cond: (id = o.shipper_id)
20   [...] ->  Index Scan using order_details_order_id on order_details od  (cost=0.29..7.92 rows=3 width=28) (actual time=0.006..0.007 rows=3 loops=22)
21   [...] Index Cond: (order_id = o.id)
22   [...] ->  Index Scan using products_pkey on products p  (cost=0.29..0.34 rows=1 width=68) (actual time=0.005..0.005 rows=1 loops=56)
23   [...] Index Cond: (id = od.product_id)
24   Planning Time: 5.580 ms
25   Execution Time: 1.215 ms
```

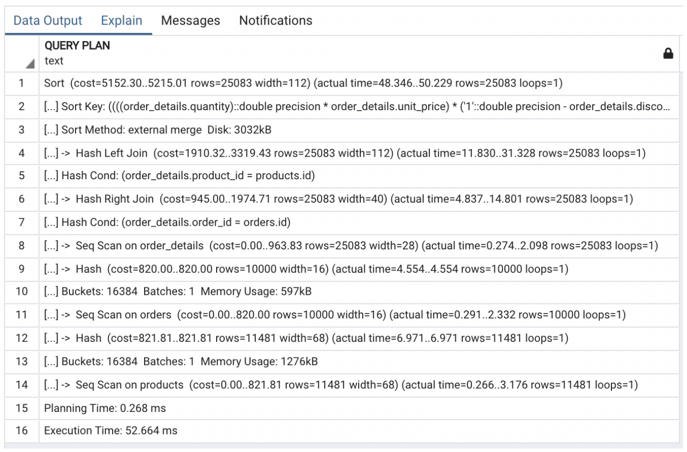
Fig. 8. Optimized Query 2

significantly higher for the non-indexed query as compared to the indexed query. Most importantly, the execution time of the indexed query is far less as compared to the non-indexed query.

3) Query 3:

```
select
    orders.id,
    orders.order_date,
    orders.shipped_date,
    orders.delivery_date,
    orders.shipped_date -
    orders.delivery_date as delay,
    order_details.quantity,
    order_details.unit_price,
    order_details.discount,
    order_details.quantity *
    order_details.unit_price *
    (1 - order_details.discount)
    as total_price,
    products.name as product_name
from
    orders
    left join order_details
    on orders.id = order_details.order_id
    left join products
    on order_details.product_id=products.id
order by
    total_price desc
```

The image 9 shows the analysis of the query. This query is used to calculate the delay between the shipped date and the delivery date of all the orders. As a large amount of data will be present in order details and order tables joining these two table is a costly operation. This can be seen from the analysis of the query.



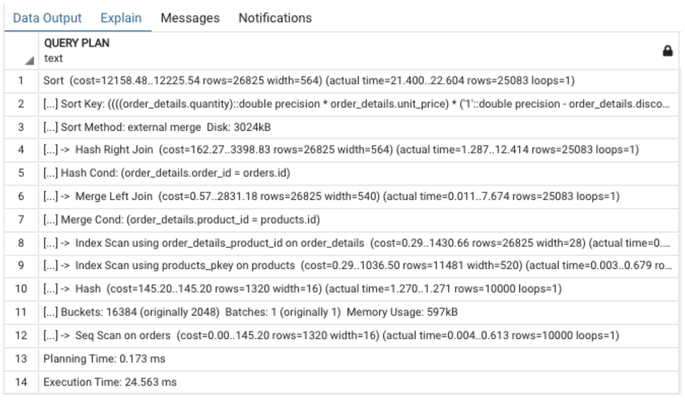
```
Data Output   Explain   Messages   Notifications

     QUERY PLAN
     text
1    Sort  (cost=5152.30..5215.01 rows=25083 width=112) (actual time=48.346..50.229 rows=25083 loops=1)
2    [...] Sort Key: ((((order_details.quantity)::double precision * order_details.unit_price) * ('1'::double precision - order_details.disco...
3    [...] Sort Method: external merge  Disk: 3032kB
4    [...] ->  Hash Left Join  (cost=1910.32..3319.43 rows=25083 width=112) (actual time=11.830..31.328 rows=25083 loops=1)
5    [...] Hash Cond: (order_details.product_id = products.id)
6    [...] ->  Hash Right Join  (cost=945.00..1974.71 rows=25083 width=40) (actual time=4.837..14.801 rows=25083 loops=1)
7    [...] Hash Cond: (order_details.order_id = orders.id)
8    [...] ->  Seq Scan on order_details  (cost=0.00..963.83 rows=25083 width=28) (actual time=0.274..2.098 rows=25083 loops=1)
9    [...] ->  Hash  (cost=820.00..820.00 rows=10000 width=16) (actual time=4.554..4.554 rows=10000 loops=1)
10   [...] Buckets: 16384  Batches: 1  Memory Usage: 597kB
11   [...] ->  Seq Scan on orders  (cost=0.00..820.00 rows=10000 width=16) (actual time=0.291..2.332 rows=10000 loops=1)
12   [...] ->  Hash  (cost=821.81..821.81 rows=11481 width=68) (actual time=6.971..6.971 rows=11481 loops=1)
13   [...] Buckets: 16384  Batches: 1  Memory Usage: 1276kB
14   [...] ->  Seq Scan on products  (cost=0.00..821.81 rows=11481 width=68) (actual time=0.266..3.176 rows=11481 loops=1)
15   Planning Time: 0.268 ms
16   Execution Time: 52.664 ms
```

Fig. 9. Problematic Query 3

To improve the efficiency of joining data of two different tables without any filters on it, indexes have been used. Indexing is applied on product_id and order_id of the order_details table. The analysis of the query after indexing can be seen in 10.

In fig 10, we can see various improvements over fig 9. For example, the sort cost for the non-indexed query is significantly higher as compared to the indexed query. Another interesting statistic to look at is the hash left join cost, which is again significantly higher for the non-indexed query as compared to the indexed query. Most importantly, the execution time of the indexed query is far less as compared to the non-indexed query.



```
Data Output   Explain   Messages   Notifications

   QUERY PLAN
   text
1  Sort  (cost=12158.48..12225.54 rows=26825 width=564) (actual time=21.400..22.604 rows=25083 loops=1)
2  [...] Sort Key: ((((order_details.quantity)::double precision * order_details.unit_price) * ('1'::double precision - order_details.disco...
3  [...] Sort Method: external merge  Disk: 3024kB
4  [...] ->  Hash Right Join  (cost=162.27..3398.83 rows=26825 width=564) (actual time=1.287..12.414 rows=25083 loops=1)
5  [...] Hash Cond: (order_details.order_id = orders.id)
6  [...] ->  Merge Left Join  (cost=0.57..2831.18 rows=26825 width=540) (actual time=0.011..7.674 rows=25083 loops=1)
7  [...] Merge Cond: (order_details.product_id = products.id)
8  [...] ->  Index Scan using order_details_product_id on order_details  (cost=0.29..1430.66 rows=26825 width=28) (actual time=0....
9  [...] ->  Index Scan using products_pkey on products  (cost=0.29..1036.50 rows=11481 width=520) (actual time=0.003..0.679 ro...
10 [...] ->  Hash  (cost=145.20..145.20 rows=1320 width=16) (actual time=1.270..1.271 rows=10000 loops=1)
11 [...] Buckets: 16384 (originally 2048)  Batches: 1 (originally 1)  Memory Usage: 597kB
12 [...] ->  Seq Scan on orders  (cost=0.00..145.20 rows=1320 width=16) (actual time=0.004..0.613 rows=10000 loops=1)
13 Planning Time: 0.173 ms
14 Execution Time: 24.563 ms
```

Fig. 10. Optimized Query 3

VI. Conclusion

For implementing a Enterprise Resource Planning solution, we had to implement a host of database techniques in order for it be practical and robust. From designing the ER diagram, converting the tables into BCNF form, implementing a website which a user can use to interact with the database to perform all CRUD operations, and observe statistics of different use cases through visualizations, we have created a system which can be operated by anyone looking to manage a small to medium organization.

VII. ER Diagram

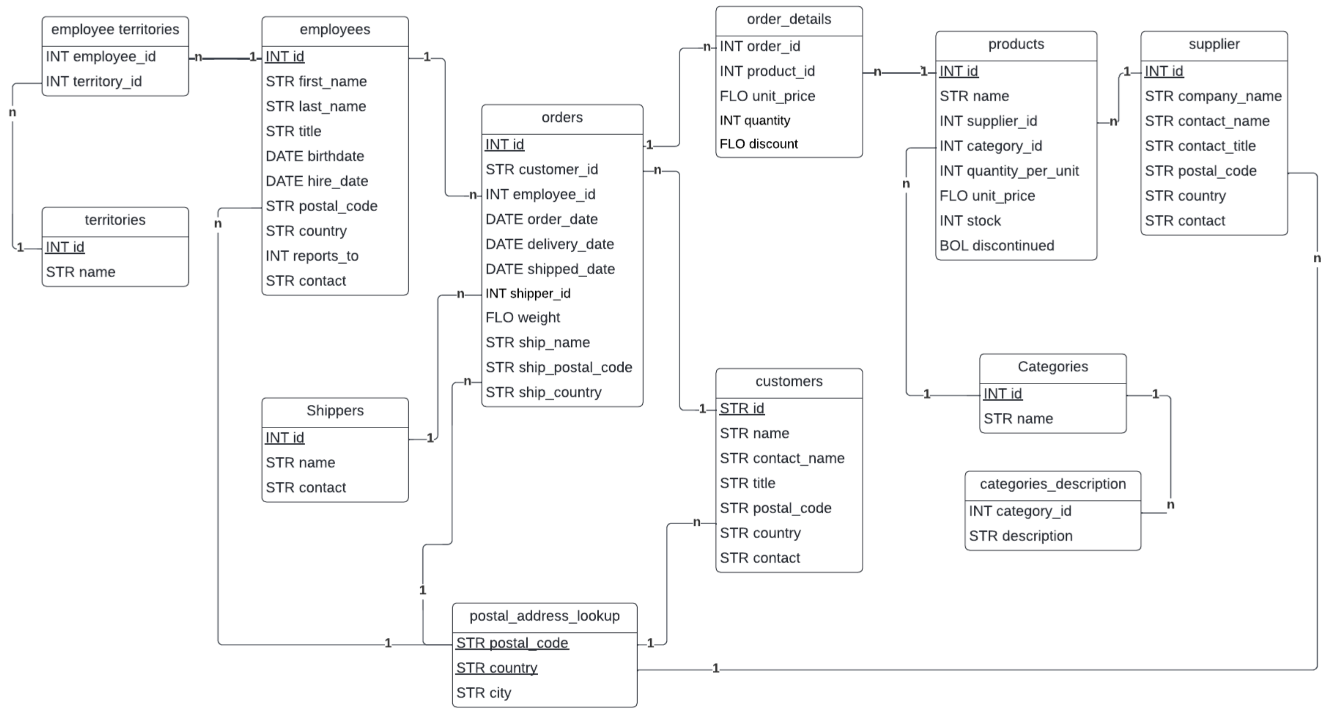


Fig. 11. ER Diagram