SQL RDBMS Implementation for E-commerce (with Front End)

Nabeel Khan
School of Engineering and Applied Science – Data Science
Buffalo, USA
nkhan6@buffalo.edu

Imran Anwar Rahman Khan
School of Engineering and Applied Science – Data Science
Buffalo, USA
imrananw@buffalo.edu

Vijaya Rana
School of Engineering and Applied Science – Data Science
Buffalo, USA
vrana3 @buffalo.edu

ABSTRACT— ***In this paper, we discuss the Relational Database implementation for an e-commerce platform. An e-commerce platform needs to store many data points, of users, sellers, customers, orders, delivery times, etc. In this paper, we explain our approach to implementing an RDBMS system for application in an eCommerce system scenario.***

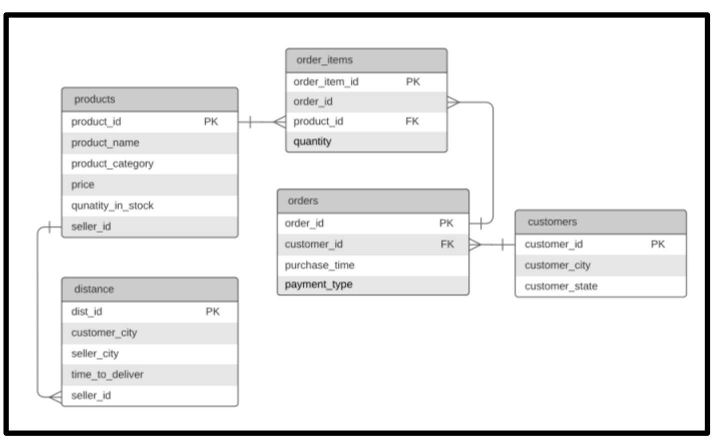
I. INTRODUCTION

Storing information in an RDBMS system is helpful for any business, especially e-commerce businesses. It can help them solve many business analytics related problems, such as:

- Top Selling Category of products (Daily, Weekly, Monthly, Yearly)
- Most purchased category by location (city, state)
- Most Popular/Frequent payment method for the users over time:
- Most Valuable payment method of the users over time
- Breakdown of payment type per category
- Breakdown of payment type by location

These figures can help in providing useful insights into the performance of the business and help with important decisions.

II. E/R DIAGRAM



III. TASKS FOR THE PROJECT

These are the important listed tasks for the project

- Extract Data from Kaggle and augmented it using Python.
- Data wrangling & cleaning with pandas.
- Validations to maintaining referential integrity
- Dashboard creation to implement run time functionalities
- Unit Testing with all scenarios for the two dashboards results

IV. STRUCTURE FOR THE DATABASE

In our database we have final tables with the below functionalities

Products - This table stores the following information:

- product_name
- product_category
- price
- quantity_in_stock.

Orders-This table stores the following information

- payment type
- order id
- customer_id
- purchase time

Customers -This table stores the following information:

- customer_id
- customer_city
- customer_state

Order_items--This table stores the following information:

- order_item_id
- order_id
- product_id
- quantity

Distance-This table store the following information

- distance_id
- customer_city
- seller_city
- time_to_deliver
- seller_id

We **assume** that we have a fixed number of repeated customers who will be placing repeated orders with the business.

V. FINAL LIST OF TABLES

To satisfy the BCNF conditions and to implement some new functionalities we modified the existing tables and below is the final 5 tables we are using

- products
- order_items
- distance
- orders
- customers

VI. JUSTIFICATION OF RELATIONS IN BCNF

1 - Customers Relation:

```
CREATE TABLE customers(customer_id INT PRIMARY
KEY,customer_city VARCHAR(20),customer_state
VARCHAR(3));
```

Data Output

	customer_id [PK] integer	customer_city character varying (20)	customer_state character varying (20)
1	1	Los Angeles	California
2	2	Los Angeles	California
3	3	Chicago	Illinois
4	4	Chicago	Illinois
5	5	Chicago	Illinois
6	6	Houston	Texas
7	7	Houston	Texas
8	8	Phoenix	Arizona
9	9	Phoenix	Arizona
10	10	Phoenix	Arizona

Valid Functional Dependencies :

customer_id → {customer_id, customer_city, customer_state}

Only one real FD exists with others being subsets or derivations from the above FD.

We can see that here that

- There exists no multi-valued attribute, hence the relation customers is in **1st Normal Form (1NF)**.
- The relation is in 1NF and there exists no partial dependency. There exists no proper subset of the candidate key customer_id, hence there can exist no partial dependency which indicates that the relation is in **2nd Normal Form (2NF).**
- The relation is in 2NF and there also exists no transitive functional dependency for Non Prime Attributes which means that the relation customers in in **3rd Normal Form (3NF).**
- Hence relation is in **3NF**
- If we closure of customer_id

  { customer_id}+ = {customer_city, customer_state}

  Therefore, the LHS is a super key as its closure includes all the attributes.

- **Thus we can say relation in BCNF**

2 - Order_items Relation:

```
CREATE TABLE order_items
(
order_item_id UUID PRIMARY KEY DEFAULT
uuid_generate_v4(),
order_id INT,
product_id INT,
quantity INT,
FOREIGN KEY (product_id) REFERENCES
products(product_id)ON UPDATE CASCADE ON
DELETE SET NULL
);
```

Data Output

	order_item_id [PK] uuid	order_id integer	product_id integer	quantity integer
1	6b5f6dfe-1446-4f4a-82b6-1e1798945a56	180101381	165	4
2	6601ea6b-a55f-4c5f-97fb-be89766f8d98	180101381	6	10
3	5f8b0729-d0d7-43e6-bacf-6ef612707994	180101381	97	1
4	86356d58-11ed-4020-99aa-da73ab8cc4cc	180101381	90	7
5	5f456a93-1df6-4d21-ae44-a4c1cac2a1ba	180101381	180	2
6	f6241b04-8aa5-4af4-aecf-2eeea7ebdf64	180101381	142	9
7	ce6a7576-9290-4cde-ae70-d211de541171	180101381	121	2
8	1ce844b9-1569-402c-aaac-499a42e5f1c8	180101381	109	3
9	58454e6a-21dc-4fc7-a56c-eb7af7d1224b	180101381	252	2
10	ed95a9d5-e465-44ae-9e4b-ca27a9b8c406	180101381	66	3

Valid Functional Dependencies :

order_item_id → {order_item_id, **order_id**, **product_id**, **quantity**}

Only one real FD exists with others being subsets or derivations from the above FD.

We can see that here that

- There exists no multi-valued attribute, hence the relation order_items is in **1st Normal Form (1NF)**.
- The relation is in 1NF and there exists no partial dependency. There exists no proper subset of the candidate key order_items_id, hence there can exist no partial dependency which indicates that the relation is in **2nd Normal Form (2NF).**
- The relation is in 2NF and there also exists no transitive functional dependency for Non Prime Attributes which means that the relation order_items is in **3rd Normal Form. (3NF).**
- Hence relation is in **3NF**
- If we closure of order_item_id

  { order_item_id }+ = {order_item_id, order_id, product_id, quantity}

  Therefore, the LHS is a super key as its closure includes all the attributes.

- **Thus we can say relation in BCNF**

3 - Orders Relation:

```
CREATE TABLE orders ( order_id INT
PRIMARY KEY,customer_id INT,
purchase_time DATE, payment_type
VARCHAR(20),FOREIGN KEY (customer_id)
REFERENCES
customers(customer_id),FOREIGN KEY
(order_id) REFERENCES orders(order_id)ON
UPDATE CASCADE ON DELETE SET NULL);
```

Data Output

	order_id [PK] integer	customer_id integer	purchase_time date	payment_type character varying
1	180101381	381	2018-01-01	cash
2	180101231	231	2018-01-01	debit_card
3	180101332	332	2018-01-01	debit_card
4	180101249	249	2018-01-01	credit_card
5	180101175	175	2018-01-01	cash
6	180101239	239	2018-01-01	debit_card
7	180101132	132	2018-01-01	credit_card
8	180101138	138	2018-01-01	credit_card
9	180101098	98	2018-01-01	debit_card
10	180101421	421	2018-01-01	food_stamp

Valid Functional Dependencies :

order_id → {order_id, customer_id, purchase_time,payment_type}

Only one real FD exists with others being subsets or derivations from the above FD.

We can see that here that

- There exists no multi-valued attribute, hence the relation orders is in **1st Normal Form (1NF)**.
- The relation is in 1NF and there exists no partial dependency. There exists no proper subset of the candidate key order_id, hence there can exist no partial dependency which indicates that the relation is in **2nd Normal Form (2NF).**
- The relation is in 2NF and there also exists no transitive functional dependency for Non Prime Attributes which means that the relation orders is in **3rd Normal Form. (3NF).**
- Hence relation is in **3NF**
- If we closure of **order_id**

{ order_id }+ = {order_id, customer_id, purchase_time,payment_type}Therefore, the LHS is a super key as its closure includes all the attributes.

- **Thus we can say relation in BCNF**

4 - Products Relation:

```
CREATE TABLE products(product_id SERIAL
PRIMARY KEY,product_name
VARCHAR(32),product_category
VARCHAR(32),price REAL,quantity_in_stock
INT, seller_id INT );
```

Data Output

	product_id [PK] integer	product_name character varying (32)	product_category character varying (32)	price real	quantity_in_stock integer	seller_id integer
1	1	Asparagus	Fresh vegetables	21	121	1
2	2	Broccoli	Fresh vegetables	13	113	2
3	3	Carrots	Fresh vegetables	17	128	2
4	4	Cauliflower	Fresh vegetables	19	81	2
5	5	Celery	Fresh vegetables	22	125	1
6	6	Corn	Fresh vegetables	21	82	2
7	7	Cucumbers	Fresh vegetables	21	136	2
8	8	Lettuce	Fresh vegetables	22	97	2
9	9	Greens	Fresh vegetables	15	84	2
10	10	Mushrooms	Fresh vegetables	21	149	2

Valid Functional Dependencies :

product_id →

{product_id, product_name, product_category, price, quantity_in_stock, seller_id}

Only one real FD exists with others being subsets or derivations from the above FD.

We can see that here that

- There exists no multi-valued attribute, hence the relation Orders is in **1st Normal Form (1NF)**.
- The relation is in 1NF and there exists no partial dependency. There exists no proper subset of the candidate key product_id, hence there can exist no partial dependency which indicates that the relation is in **2nd Normal Form (2NF).**
- The relation is in 2NF and there also exists no transitive functional dependency for Non Prime Attributes which means that the relation products is in **3rd Normal Form. (3NF).**
- Hence relation is in **3NF**
- If we closure of customer_id

  { product_id }+ = { product_id, product_name, product_category, price quantity_in_stock, seller_id}

  Therefore, the LHS is a super key as its closure includes all the attributes.

- **Thus we can say relation in BCNF**

5 - Distances Relation:

```
CREATE TABLE distances
(dist_id INT PRIMARY KEY, customer_city
VARCHAR(20),  seller_city  VARCHAR(20),
time_to_deliver INT, seller_id INT );
```

Data Output

	dist_id [PK] integer	customer_city character varying (20)	seller_city character varying (20)	time_to_deliver integer	seller_id integer
1	1	Los Angeles	Albany	79	1
2	2	Los Angeles	Amsterdam	79	2
3	3	Los Angeles	Auburn	79	3
4	4	Los Angeles	Batavia	79	4
5	5	Los Angeles	Beacon	79	5
6	6	Los Angeles	Binghamton	79	6
7	7	Los Angeles	Buffalo	79	7
8	8	Los Angeles	Canandaigua	79	8
9	9	Los Angeles	Cohoes	79	9
10	10	Los Angeles	Corning	79	10

Valid Functional Dependencies :

dist_id → {customer_city, seller_city, time_to_deliver, seller_id}

Only one real FD exists with others being subsets or derivations from the above FD.

We can see that here that there exists no multi-valued attribute, hence the relation Distances is in **1st Normal Form (1NF)**.

- The relation is in 1NF and there exists no partial dependency. There exists no proper subset of the candidate key transaction_id, hence there can exist no partial dependency which indicates that the relation is in **2nd Normal Form (2NF).**
- The relation is in 2NF and there also exists no transitive functional dependency for Non Prime Attributes which means that the relation transactions is in **3rd Normal Form. (3NF).**
- Hence relation is in **3NF**

VII. DATASET CHALLENGES FACED & RESOLUTION

- ***Challenge 1*** - Our dataset contains 100,000 orders, and due to scale we found it challenging to insert data in our database, due to erroneous entries.

- ***Solution.*** We used Python for data prepping, missing value handling, erroneous value handling, and datatype correction. Additionally we had to do brainstorming to figure out which attributes we needed to filter out.

- ***Challenge 2*** - Generating the unique and consistent order_id but different order_item_id at run time from the dashboard was challenging.

- ***Solution*** - We assumed order_id to be a combination of date and customer id, hence all order_items from a customer on a single day will automatically fall into a single unique order_id. This allowed us to keep our frontend simple by only keeping a single order item.

VIII. DATABASE QUERIES WITH EXECUTION RESULTS

Note: Few variables in these queries were selected from the frontend

1 - Top N most frequent customers:

```
SELECT
orders.customer_id, sum(order_items.quantity *
products.price) as total_value FROM orders
INNER JOIN order_items ON orders.order_id =
order_items.order_id
INNER JOIN products
ON order_items.product_id=products.product_id
```

Data Output

	customer_id integer	total_value double precision
1	184	15495
2	87	7351
3	273	20117
4	394	33495
5	51	7604
6	272	12021
7	70	8329
8	190	15364
9	350	31587
10	278	19436
11	424	26916
12	406	28810
13	176	15225

2 - Top N most selling product_categories (in past M days):

```
SELECT
products.product_category,
sum(order_items.quantity * products.price) as
total_value
FROM order_items
INNER JOIN products ON order_items.product_id =
products.product_id
INNER JOIN orders ON orders.order_id =
order_items.order_id
WHERE orders.purchase_time > current_date - {m}
GROUP BY products.product_category ORDER BY
total_value DESC LIMIT {n}
```

Data Output

	product_category character varying (32)	total_value double precision
1	Snacks	644749
2	Medicine	576771
3	Baked goods	563685
4	Frozen	534302
5	Canned foods	531980
6	Personal care	497885
7	Cheese	447335
8	Condiments / Sauces	446663
9	Fresh vegetables	368390
10	Fresh fruits	343822

3 - Mapping of customer IDs with the products that they have bought:

```
SELECT customers.customer_id,
products.product_name FROM products
INNER JOIN order_items  ON
products.product_id = order_items.order_id
INNER JOIN orders  ON order_items.order_id =
orders.order_id
INNER JOIN  customers ON orders.customer_id =
customers.customer_id
```

Data Output

	customer_id integer	product_name character varying (32)
1	381	Lunchmeat
2	381	Corn
3	381	Tuna
4	381	Swiss
5	381	Yeast
6	381	Dried fruit
7	381	Pizza
8	381	Hummus
9	381	Cigarettes
10	381	Lime juice
11	381	Pasta sauce
12	231	Magazine
13	231	Cigarettes
14	231	Ketchup
15	231	Non-stick spray
16	332	Potatoes
17	332	Breakfasts
18	332	Lime juice
19	332	Arsenic
20	332	Bouillon cubes

4 - Inserting data into orders table:

```
INSERT into orders (order_id, customer_id,
purchase_time, payment_type) VALUES ("""  +
vals_1 + ") ON CONFLICT DO NOTHING
```

* These values are taken from the front-end panel

(from the data which the user enters)

5 - Inserting data into order_items table:

```
INSERT into order_items (order_id ,product_id,
quantity)  VALUES(""" + vals_2 + ")”
```

* These values are taken from the front-end panel

(from the data which the user enters)

6 - Calculating the delivery time between seller and customer cities:

```
SELECT time_to_deliver, seller_city
FROM distances
WHERE seller_id = {} AND customer_city = {}
```

7 - Getting Various Products options

(to show on front-end - after user selects the category):

SELECT DISTINCT product_id, product_name, price, seller_id from products WHERE product_category = '{}'

IX. QUERY EXECUTION ANALYSIS AND WAYS TO IMPROVE THEM

Problematic Query #1:

Query to get the product_name and product_category for the products sold to customers in states (New York)

```
SELECT product_name, product_category
FROM products
WHERE product_id IN (SELECT product_id
FROM order_items
WHERE order_id IN(SELECT order_id FROM orders
WHERE customer_id IN(SELECT customer_id
FROM customers
WHERE customer_state  IN ('New York'))))
```

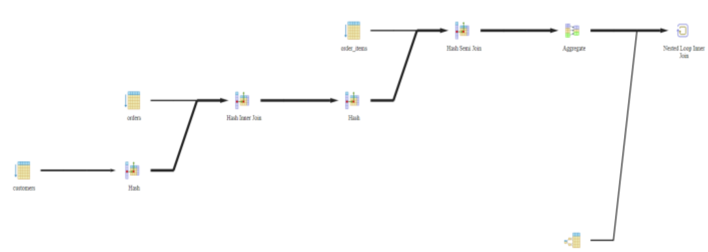
Run Time = 189 ms

OUTPUT

Data Output

	product_name character varying (32)	product_category character varying (32)
1	Hand soap	Personal care
2	Fries	Frozen
3	Provolone	Cheese
4	Kiwis	Fresh fruits
5	Pancake	Various groceries
6	Facial cleanser	Personal care
7	Bread crumbs	Baking
8	Flour	Baking
9	Baked beans	Canned foods
10	Yeast	Baking
11	Mop head	Cleaning products
12	Jam	Condiments / Sauces
13	Pencils	Office supplies
14	Bacon	Meat
15	Moisturizing lotion	Personal care
16	Pie! Pie! Pie!	Baked goods

EXECUTION PLAN



QUERY PLAN

	QUERY PLAN text
1	Nested Loop (cost=2256.71..2266.80 rows=263 width=21) (actual time=22.299..22.704 rows=263 loops=1)
2	[...] -> HashAggregate (cost=2256.55..2259.18 rows=263 width=4) (actual time=22.281..22.309 rows=263 loops=1
3	[...] Group Key: order_items.product_id
4	[...] Batches: 1 Memory Usage: 45kB
5	[...] -> Hash Semi Join (cost=369.28..2205.75 rows=20322 width=4) (actual time=3.000..17.966 rows=31791 loops
6	[...] Hash Cond: (order_items.order_id = orders.order_id)
7	[...] -> Seq Scan on order_items (cost=0.00..1398.85 rows=80585 width=8) (actual time=0.012..4.516 rows=80585
8	[...] -> Hash (cost=318.41..318.41 rows=4070 width=4) (actual time=2.971..2.973 rows=4206 loops=1)
9	[...] Buckets: 8192 (originally 4096) Batches: 1 (originally 1) Memory Usage: 212kB
10	[...] -> Hash Join (cost=11.26..318.41 rows=4070 width=4) (actual time=0.103..2.474 rows=4206 loops=1)
11	[...] Hash Cond: (orders.customer_id = customers.customer_id)
12	[...] -> Seq Scan on orders (cost=0.00..264.39 rows=16139 width=8) (actual time=0.008..0.870 rows=16139 loops=
13	[...] -> Hash (cost=9.80..9.80 rows=117 width=4) (actual time=0.073..0.074 rows=117 loops=1)
14	[...] Buckets: 1024 Batches: 1 Memory Usage: 13kB
15	[...] -> Seq Scan on customers (cost=0.00..9.80 rows=117 width=4) (actual time=0.044..0.058 rows=117 loops=1)
16	[...] Filter: ((customer_state)::text = 'New York'::text)
17	[...] Rows Removed by Filter: 347
18	[...] -> Memoize (cost=0.16..0.25 rows=1 width=25) (actual time=0.001..0.001 rows=1 loops=263)
19	[...] Cache Key: order_items.product_id
20	[...] Cache Mode: logical

ANALYSIS

Explain

Graphical Analysis Statistics

#	Node	Rows Plan
1.	→ Nested Loop Inner Join (cost=2256.71..2266.8 rows=263 width=21)	263
2.	→ Aggregate (cost=2256.55..2259.18 rows=263 width=4)	263
3.	→ Hash Semi Join (cost=369.28..2205.75 rows=20322 width=4) Hash Cond: (order_items.order_id = orders.order_id)	20322
4.	→ Seq Scan on order_items as order_items (cost=0..1398.85 rows=80585 width=8)	80585
5.	→ Hash (cost=318.41..318.41 rows=4070 width=4)	4070
6.	→ Hash Inner Join (cost=11.26..318.41 rows=4070 width=4) Hash Cond: (orders.customer_id = customers.customer_id)	4070
7.	→ Seq Scan on orders as orders (cost=0..264.39 rows=16139 width=8)	16139
8.	→ Hash (cost=9.8..9.8 rows=117 width=4)	117
9.	→ Seq Scan on customers as customers (cost=0..9.8 rows=117 width=4) Filter: ((customer_state)::text = 'New York'::text)	117
10.	→ Memoize (cost=0.16..0.25 rows=1 width=25)	1
11.	→ Index Scan using products_pkey on products as products (cost=0.15..0.24 rows=1 width=25) Index Cond: (product_id = order_items.product_id)	1

STATISTICS

Explain

Graphical Analysis Statistics

Statistics per Node Type

Node type	Count
Aggregate	1
Hash	2
Hash Inner Join	1
Hash Semi Join	1
Index Scan	1
Memoize	1
Nested Loop Inner Join	1
Seq Scan	3

Statistics per Relation

Relation name	Scan count
Node type	Count
customers	1
Seq Scan	1
order_items	1
Seq Scan	1
orders	1
Seq Scan	1
products	1
Index Scan	1

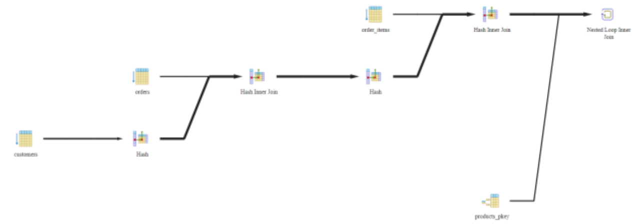
Note: We achieved the cost of 2256.71 ,which we will try to improve

Improvement for Problematic Query #1:

```
CREATE INDEX ind ON products(product_id)
SELECT product_name,product_category FROM products
INNER JOIN order_items ON products.product_id = order_items.product_id
INNER JOIN orders ON orders.order_id = order_items.order_id INNER JOIN customers ON customers.customer_id = orders.customer_id
WHERE customers.customer_state IN ('New York')
```

Run Time = 70ms

EXECUTION PLAN



STATISTICS

Explain

Graphical Analysis Statistics

Statistics per Node Type

Node type	Count
Hash	3
Hash Inner Join	3
Seq Scan	4

Statistics per Relation

Relation name	Scan count
Node type	Count
customers	1
Seq Scan	1
order_items	1
Seq Scan	1
orders	1
Seq Scan	1
products	1
Seq Scan	1

The nested query is seen to take a lot of cost. This can be improved. We made the nested query into a natural join query in addition to adding an index for the product_id column in the products relation.

Note:We achieved the cost of 378.22 ,which is highly improved

Problematic Query #2:

```
SELECT DISTINCT distances.seller_id, distances.seller_city FROM distances
INNER JOIN products ON distances.seller_id = products.seller_id
INNER JOIN order_items ON products.product_id =order_items.product_id
INNER JOIN orders ON order_items.order_id = orders.order_id
INNER JOIN customers ON customers.customer_id = orders.customer_id
WHERE customers.customer_state = 'NY'
SELECT DISTINCT seller_id,seller_city
```

Run Time = 965ms

	seller_id integer	seller_city character varying (20)
1	34	Newburgh
2	37	Norwich
3	14	Fulton
4	42	Oswego
5	36	North Tonawanda
6	11	Cortland
7	13	Elmira
8	5	Beacon
9	35	Niagara Falls
10	22	Jamestown

ANALYSIS

Explain

Graphical Analysis Statistics

#	Node	Rows Plan
1.	→ Aggregate (cost=80093.62..80108.13 rows=1451 width=13)	1451
2.	→ Hash Inner Join (cost=817.65..56319.31 rows=4754862 width=13) Hash Cond: (products.seller_id = distances.seller_id)	4754862
3.	→ Hash Inner Join (cost=378.22..2336.68 rows=20320 width=4) Hash Cond: (order_items.product_id = products.product_id)	20320
4.	→ Hash Inner Join (cost=369.28..2273.55 rows=20320 width=4) Hash Cond: (order_items.order_id = orders.order_id)	20320
5.	→ Seq Scan on order_items as order_items (cost=0..1398.85 rows=80585 width=8)	80585
6.	→ Hash (cost=318.41..318.41 rows=4070 width=4)	4070
7.	→ Hash Inner Join (cost=11.26..318.41 rows=4070 width=4) Hash Cond: (orders.customer_id = customers.customer_id)	4070
8.	→ Seq Scan on orders as orders (cost=0..264.39 rows=16139 width=8)	16139
9.	→ Hash (cost=9.8..9.8 rows=117 width=4)	117
10.	→ Seq Scan on customers as customers (cost=0..9.8 rows=117 wid... Filter: ((customer_state)::text = 'New York'::text)	117
11.	→ Hash (cost=5.64..5.64 rows=264 width=8)	264
12.	→ Seq Scan on products as products (cost=0..5.64 rows=264 width=8)	264

STATISTICS

Explain

Graphical Analysis Statistics

Statistics per Node Type

Node type	Count
Aggregate	4
Hash	3
Hash Inner Join	2
Hash Semi Join	1
Index Scan	1
Memoize	1
Nested Loop Inner Join	1
Seq Scan	4

Statistics per Relation

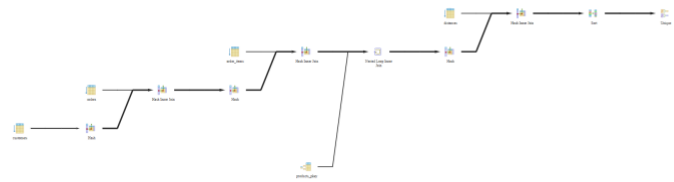
Relation name / Node type	Scan count / Count
customers	1
Seq Scan	1
distances	1
Seq Scan	1
order_items	1
Seq Scan	1
orders	1
Seq Scan	1
products	1
Index Scan	1

Note:We achieved the cost of 2725.68 ..2740 ,which we will try to improve

Improvement for Problematic Query #2:

```
FROM distances WHERE seller_id IN
(SELECT seller_id FROM products
WHERE product_id IN
(SELECT product_id FROM order_items
WHERE order_id IN(SELECT order_id FROM ORDERS
WHERE customer_id IN
(SELECT DISTINCT customer_id FROM customers
WHERE customer_state = 'NY'))))
```

RUN TIME: 65 ms



ANALYSIS

Explain

Graphical Analysis Statistics

#	Node	Rows Plan
1.	→ Aggregate (cost=2725.68..2740.19 rows=1451 width=13)	1451
2.	→ Hash Inner Join (cost=2271.04..2676.54 rows=9828 width=13) Hash Cond: (distances.seller_id = products.seller_id)	9828
3.	→ Seq Scan on distances as distances (cost=0..258.08 rows=14508 width=13)	14508
4.	→ Hash (cost=2270.51..2270.51 rows=42 width=4)	42
5.	→ Aggregate (cost=2270.09..2270.51 rows=42 width=4)	42
6.	→ Nested Loop Inner Join (cost=2259.34..2269.44 rows=263 width=4)	263
7.	→ Aggregate (cost=2259.19..2261.82 rows=263 width=4)	263
8.	→ Hash Semi Join (cost=371.91..2208.38 rows=20322 width=4) Hash Cond: (order_items.order_id = orders.order_id)	20322
9.	→ Seq Scan on order_items as order_items (cost=0..1398.85 rows=80585 wi...	80585
10.	→ Hash (cost=321.04..321.04 rows=4070 width=4)	4070
11.	→ Hash Inner Join (cost=13.9..321.04 rows=4070 width=4) Hash Cond: (orders.customer_id = customers.customer_id)	4070
12.	→ Seq Scan on orders as orders (cost=0..264.39 rows=16139 widt...	16139

STATISTICS

Explain

Graphical Analysis Statistics

Statistics per Node Type

Node type	Count
Aggregate	1
Hash	4
Hash Inner Join	4
Seq Scan	5

Statistics per Relation

Relation name / Node type	Scan count / Count
customers	1
Seq Scan	1
distances	1
Seq Scan	1
order_items	1
Seq Scan	1
orders	1
Seq Scan	1
products	1
Seq Scan	1

SOLUTION APPROACH

The nested query is seen to take a lot of cost. This can be improved. This can be improved.We made the second nested query into a natural join query in addition to adding an index for the num purchases column in the places relation.

Problematic Query #3:

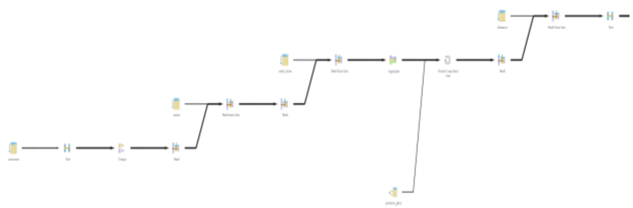
```
SELECT customers.customer_id,
products.product_name FROM products
INNER JOIN order_items  ON products.product_id
= order_items.order_id  INNER JOIN orders  ON
order_items.order_id = orders.order_id
INNER JOIN  customers ON orders.customer_id =
customers.customer_id
```

Run Time = 103 ms

OUTPUT

Data Output

	customer_id integer	product_name character varying (32)
1	381	Lunchmeat
2	381	Corn
3	381	Tuna
4	381	Swiss
5	381	Yeast
6	381	Dried fruit
7	381	Pizza
8	381	Hummus
9	381	Cigarettes
10	381	Lime juice



ANALYSIS

Explain

Graphical | Analysis | Statistics

#	Node	Rows Plan
1.	→ Hash Inner Join (cost=489.51..2528.35 rows=80585 width... **Hash Cond**: (orders.customer_id = customers.customer_id)	80585
2.	→ Hash Inner Join (cost=475.07..2300.44 rows=80585 ... **Hash Cond**: (order_items.order_id = orders.order_id)	80585
3.	→ Hash Inner Join (cost=8.94..1622.72 rows=8058... **Hash Cond**: (order_items.product_id = products.produ ct_id)	80585
4.	→ Seq Scan on order_items as order_items (c...	80585
5.	→ Hash (cost=5.64..5.64 rows=264 width=13)	264
6.	→ Seq Scan on products as products (co...	264
7.	→ Hash (cost=264.39..264.39 rows=16139 width=8)	16139
8.	→ Seq Scan on orders as orders (cost=0..264....	16139
9.	→ Hash (cost=8.64..8.64 rows=464 width=4)	464
10.	→ Seq Scan on customers as customers (cost=0..8...	464

STATISTICS

Explain

Graphical | Analysis | Statistics

Statistics per Node Type

Node type	Count
Hash	3
Hash Inner Join	3
Seq Scan	4

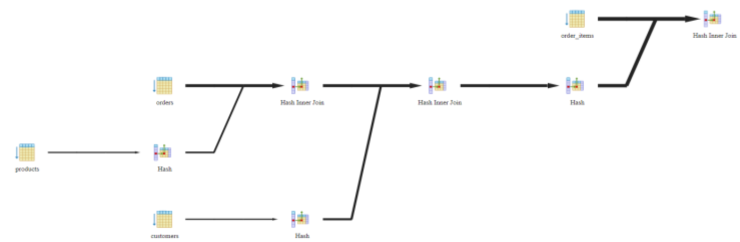
Statistics per Relation

Relation name	Scan count
Node type	**Count**
customers	1
Seq Scan	1
order_items	1
Seq Scan	1
orders	1
Seq Scan	1
products	1
Seq Scan	1

Improvement for Problematic Query #3:

```
SELECT customers.customer_id,
products.product_name FROM products
INNER JOIN order_items  ON products.product_id
= order_items.product_id  INNER JOIN orders
ON order_items.order_id = orders.order_id
INNER JOIN  customers ON orders.customer_id =
customers.customer_id
```

Run Time = 48 ms



ANALYSIS

Explain

Graphical | Analysis | Statistics

#	Node	Rows Plan
1.	→ Hash Inner Join (cost=334.15..2050.51 rows=1532 width=... **Hash Cond**: (order_items.order_id = products.product_id)	1532
2.	→ Seq Scan on order_items as order_items (cost=0..13...	80585
3.	→ Hash (cost=330.85..330.85 rows=264 width=21)	264
4.	→ Hash Inner Join (cost=23.38..330.85 rows=264 ... **Hash Cond**: (orders.customer_id = customers.custom er_id)	264
5.	→ Hash Inner Join (cost=8.94..315.71 rows=2... **Hash Cond**: (orders.order_id = products.product_ id)	264
6.	→ Seq Scan on orders as orders (cost=0.....	16139
7.	→ Hash (cost=5.64..5.64 rows=264 width...	264
8.	→ Seq Scan on products as product...	264
9.	→ Hash (cost=8.64..8.64 rows=464 width=4)	464
10.	→ Seq Scan on customers as customers ...	464

STATISTICS

Explain

Graphical | Analysis | Statistics

Statistics per Node Type

Node type	Count
Hash	3
Hash Inner Join	3
Seq Scan	4

Statistics per Relation

Relation name	Scan count
Node type	**Count**
customers	1
Seq Scan	1
order_items	1
Seq Scan	1
orders	1
Seq Scan	1
products	1
Seq Scan	1

SOLUTION APPROACH

The nested query is seen to take a lot of cost. This can be improved. The filter places.numwebpurchases > 15 also takes a considerable amount of time. This can be improved.

We made the second nested query into a natural join query in addition to adding an index for the numwebpurchases column in the places relation.

X. FRONT END

We have implemented two dashboard functionality for end to end process to give more realistic impact

1- Customer portal

This helps the customer this help customer to place the order.We have implemented additional 3 functionality (highlighted yellow) as below

- Functionality to calculate the delivery time
- Functionality to calculate the total price
- Functionality to add more items to cart and accordingly new price will be updated

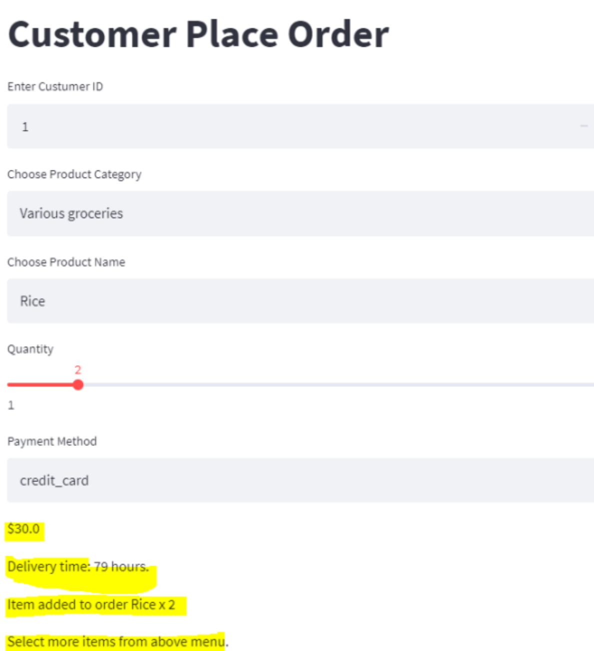


Fig 1.1

2: Summary dashboard

This helps to give the overall picture of the customer and purchase and revenue.We implement below functionalities

- On the basis of the number of previous days selected,we can choose the number of records we want to see.

  Now using this we can see the top n customers,products_categories
- Additionally we are calculating the total purchase of top

  customers,and total purchase within each top product category
- Functionality to view the above statistics as bar graph
- Functionality to view the monthly revenue generated on the basis of number of orders placed and there price.

Fig 1.2

WEBSITE URL REFERENCES

- Hosted Static version of the frontend https://lanbeee.github.io/DMQL
- Olist dataset

https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce

- Olist company profile
  https://pitchbook.com/profiles/company/102473-65#signals
- Brazil - Market Challenges. Retrieved July 01, 2021
  https://www.trade.gov/country-commercial-guides/brazil-market-challenges

M .TEAM MEMBERS CONTRIBUTION

We all contributed equally overall. Nabeel mostly handled frontend, Imran handled problematic queries, and Vijaya handled database and frontend interaction.