Stage 3: Database Implementation and Indexing

Bennett Wu [bwu40]

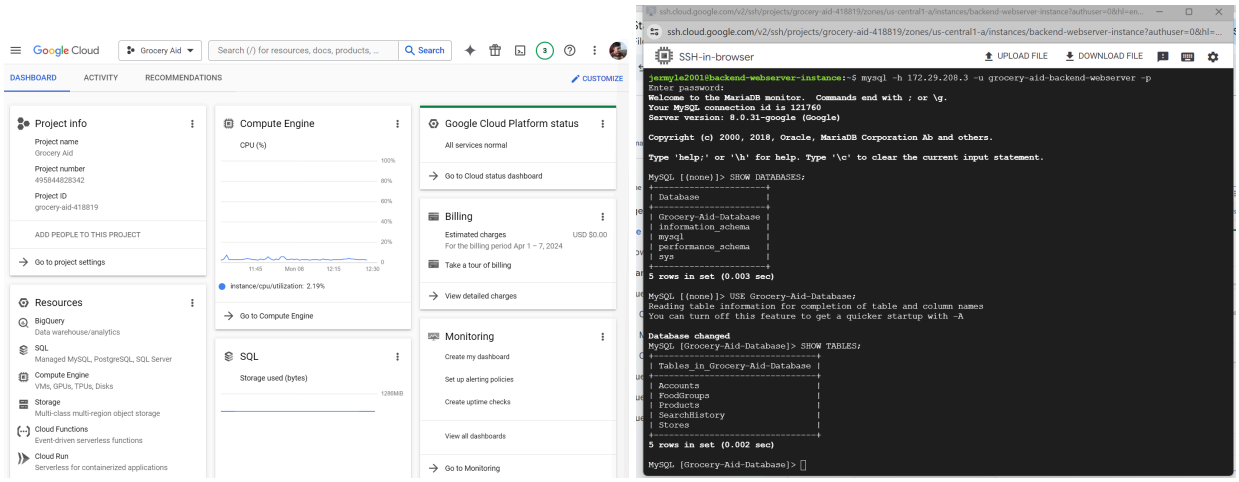
Jeremy Lee [jeremyl6]

Paul Jeong [paulj3]

Shreya Singh [shreya15]

Table Creation

The Database was implemented on the Google Cloud Platform, utilizing MySQL. A screenshot of the platform is included in the following:



The Data Definition Language (DDL) commands used to create each of the tables are as follows:

Accounts:

```
CREATE TABLE Accounts (
        user_id INT PRIMARY KEY,
        first_name VARCHAR( 255 ),
        last_name VARCHAR( 255 ),
        password_hash VARCHAR( 255 ),
        email_addr VARCHAR( 255 )
        );
```

SearchHistory:

```
CREATE TABLE SearchHistory (
        history_id INT PRIMARY KEY,
        user_id INT, FOREIGN KEY ( user_id ) REFERENCES Accounts( user_id ),
        search_string VARCHAR( 255 ),
        timestamp INT
        );
```

Products:

```
CREATE TABLE Products (
        product_id INT PRIMARY KEY AUTO_INCREMENT,
        store_id INT, FOREIGN KEY ( store_id ) REFERENCES Stores( store_id ),
        name VARCHAR( 255 ),
        price DOUBLE
        );
```

Stores:

```
CREATE TABLE Stores (
        store_id INT PRIMARY KEY
        store_name VARCHAR( 255 )
        );
```

FoodGroups:

```
CREATE TABLE FoodGroups (
        food_id INT PRIMARY KEY AUTO_INCREMENT,
        product_id INT, FOREIGN KEY ( product_id ) REFERENCES Products(
        product_id ),
        dairy BOOLEAN,
        vegetables BOOLEAN,
        fruits BOOLEAN,
        meats BOOLEAN,
        condiments BOOLEAN,
        spices BOOLEAN,
        grains BOOLEAN,
        other BOOLEAN
        );
```

Row Insertion

After table creation, datasets were retrieved and parsed from online sources and subsequently inserted into the database tables. Screenshots of the counts of each table are as follows:

```
MySQL [Grocery-Aid-Database]> SELECT COUNT( user_id ) AS NumRows FROM Accounts;
+---------+
| NumRows |
+---------+
|    3547 |
+---------+
1 row in set (0.022 sec)
```

```
MySQL [Grocery-Aid-Database]> SELECT COUNT( history_id ) AS NumRows FROM SearchHistory;
+---------+
| NumRows |
+---------+
|    3547 |
+---------+
1 row in set (0.020 sec)
```

```
MySQL [Grocery-Aid-Database]> SELECT COUNT( store_id ) AS NumRows FROM Stores;
+---------+
| NumRows |
+---------+
|       4 |
+---------+
1 row in set (0.004 sec)
```

```
MySQL [Grocery-Aid-Database]> SELECT COUNT( product_id ) AS NumRows FROM Products;
+---------+
| NumRows |
+---------+
|  574709 |
+---------+
1 row in set (0.791 sec)
```

```
MySQL [Grocery-Aid-Database]> SELECT COUNT( food_id ) AS NumRows FROM FoodGroups;
+---------+
| NumRows |
+---------+
|  574709 |
+---------+
1 row in set (0.618 sec)
```

Advanced Generated Queries

After inserting rows into the database, the team developed 4 advanced queries, each involving *at least two* of the following SQL concepts:

- Join Multiple Relations
- Set Operators
- Aggregation Via GROUP BY
- Subqueries that cannot be easily replaced by a Join

Query 1

Query Application

In the first query, we aimed to solve the following problem:

Find the number of produce items each store possesses.

With this query, users would be able to assess which stores offered the most produce items available, likely aiding the user in the decision on where to shop, whether that be prioritizing a large variety of items, or avoiding the plethora of items altogether.

MySQL Query

```
SELECT COUNT( Products.product_id ) AS NumProducts, Stores.store_name
FROM Products JOIN Stores ON Products.store_id = Stores.store_id WHERE
Products.product_id IN (SELECT product_id FROM FoodGroup WHERE fruits = 1
UNION SELECT product_id FROM FoodGroup WHERE vegetables = 1) GROUP
BY Stores.store_id;
```

This query utilizes a Subquery to improve readability and understanding, as well as avoid the usage of an unnecessary Join statement - the query isolates the appropriate products without having to join a large number of rows. Additionally, this query also utilizes the Set Operator UNION in order to expand upon the conditions of which the query can select rows and Aggregation via the GROUP BY statement.

Query Results

The non-indexed result includes less than 15 rows due to grouping via Store.

```
MySQL [Grocery-Aid-Database]> SELECT COUNT( Products.product_id ) AS NumProducts, Stores.store_name FROM Products JOIN Stores ON Products.sto
re_id = Stores.store_id WHERE Products.product_id IN (SELECT product_id FROM FoodGroups WHERE fruits = 1 UNION SELECT product_id FROM FoodGro
ups WHERE vegetables = 1) GROUP BY Stores.store_id LIMIT 15;
+-------------+------------+
| NumProducts | store_name |
+-------------+------------+
|          30 | Amazon     |
|          31 | Costco     |
|       19849 | Walmart    |
|          42 | WholeFoods |
+-------------+------------+
4 rows in set (10.347 sec)
```

Query 2

Query Application

In the first query, we aimed to solve the following problem:

Find the number of items and average price of each store available in the Database

With this query, users would be able to assess the average price of each store in order to determine where to shop to maximize their finances, as well as the variety of items each store may offer, allowing users to leverage whether they want to prioritize item variety or item cost.

MySQL Query

> **SELECT** *Stores*.store_name, **AVG**(*Products*.price) **AS** AvgPrice, **COUNT**(*Products*.store_id) **FROM** *Products* **JOIN** *Stores* **ON** *Products*.store_id = *Stores*.store_id **JOIN** *FoodGroup* **ON** *Products*.product_id = *FoodGroup*.product_id **WHERE** other = 1 **GROUP BY** *Stores*.store_id;

This query utilizes a join on multiple relations to pull all the necessary identifying information, as well as Aggregation via the GROUP BY statement to complete its basic counting and mathematical operations.

Query Results

The non-indexed result includes less than 15 rows due to grouping via Store.

```
MySQL [Grocery-Aid-Database]> SELECT Stores.store_name, AVG( Products.price ) AS AvgPrice, COUNT( Products.store_id ) FROM Products JOIN Stor
es ON Products.store_id = Stores.store_id JOIN FoodGroups ON Products.product_id = FoodGroups.product_id WHERE other = 1 GROUP BY Stores.stor
e_id LIMIT 15;
+------------+--------------------+---------------------------+
| store_name | AvgPrice           | COUNT( Products.store_id ) |
+------------+--------------------+---------------------------+
| Walmart    |  5.105541222079056 |                    526429 |
| Costco     | 55.464345794394255 |                      1712 |
| WholeFoods |  8.407079081632583 |                      1568 |
| Amazon     | 21.571779725609616 |                      2624 |
+------------+--------------------+---------------------------+
4 rows in set (3.333 sec)
```

Query 3

Query Application

This query would allow users to find the cheapest price of the item they have searched for in each of the available stores. It also gives the price, the name of the product, and the store ID. In this example, we have used the item “eggs”, so this query will search across the Products table and find the minimum price of eggs in each of the available stores. Since there are no eggs available at the 4th store we only have three rows of data.

MySQL Query

```
SELECT DISTINCT Stores.store_name, Products.name, Products.price AS min_price
FROM Stores JOIN Products ON Stores.store_id = Products.store_id INNER JOIN
   ( SELECT store_id, MIN(price) AS price FROM Products WHERE name LIKE
    '%eggs%'GROUP BY store_id) AS min_prices
    ON Stores.store_id = min_prices.store_id AND Products.price = min_prices.price
WHERE Products.name LIKE '%eggs%'
ORDER BY Stores.store_name ASC, Products.price ASC;
```

This query utilizes a join on the Products and Stores table and joins them on the store_id. It also utilizes an Inner Join between the Products and min_prices table to only obtain the cheapest product for each search query (in this case eggs.) It also utilizes subqueries to create a table with the relevant information. It also uses GROUP BY to aggregate the results and store each shop’s information.

Query Results

Since there are only 4 stores, and one of them does not have the item searched for, we have 3 rows of results.

store_name	name	min_price
Amazon	The Ojai Cook Garlic Herb Lemonaise - Organic Mayonnaise Aioli Made With Cage-Free Eggs	12.95
Walmart	Great Value Large White Eggs	1.05
Whole Foods	Pasture Raised Large Grade A Eggs	7.99

Query 4

Query Application

This query allows the user to see the probability for finding a unique product at the available stores in the database. In this example we used “sriracha”. The result returns the number of products available at each store with sriracha in their name, and the probability of finding this product in the respective store.

MySQL Query

```
SELECT Stores.store_name, COUNT(Products.name) as count_product,
(COUNT(Products.name) /  (SELECT COUNT(*) FROM Products WHERE name LIKE
"%sriracha%") ) * 100 AS probability
FROM Products JOIN Stores on Products.store_id = Stores.store_id
WHERE Products.name LIKE "%sriracha%"
GROUP BY Stores.store_name
ORDER BY Stores.store_name ASC;
```

The query uses subqueries to count the number of the products with the query in their name, and find the total count of products to find the probability of finding that product in the respective store. It utilizes a JOIN operation to find the name of corresponding store names for each product. It also utilizes GROUP BY.

Query Results

Since there are only 4 stores, and one of them does not have the item searched for, we have 3 rows of results.

store_name	count_product	probability
Amazon	49	15.6051
Costco	1	0.3185
Walmart	264	84.0764

Indexing

Indexing Configurations

While evaluating how to improve the performance of the database and the advanced queries, the team determined several attributes of which indices could be added:

Index Name	Affected Columns
Store_products_idx (1)	Products.store_id (FK)
Products_foodgroup_idx (2)	FoodGroup.product_id (FK)
foodgroup_other_idx (3)	FoodGroup.other
store_name_idx (4)	Stores.store_name
first_name_idx (5)	Accounts.first_name
product_name_idx (6)	Products.name

The individual performance and effect of each column was then compared against the initial cost of each query. Table 1 shows the initial cost of each query, while Table 2 shows the performance change of each query with the corresponding index added into the database, shown below:

Non-Indexed Performance	
Query	**Total Cost**
1	244465.59
2	364347.73
3	3,766,283,889
4	392171.14

Table 1: Non-Indexed Performance

	Performance (Query #)				Performance Change (Query #) (INITAL - FINAL)			
Index Name	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**
store_products_idx	244,462.28	383,259.03	3,765,683,768.00	397,232.74	3.31	-18,911.30	600,121.00	-5,061.60
products_foodgr	240,472.	353,856.	3,643,70	389,874.	3,993.49	10,491.2	122,581,	2,296.34

oup_idx	10	49	1,961.63	80		4	927.37	
foodgroup_other_idx	239,426.58	723,686.98	31,160,767.70	390,166.42	5,039.01	-359,339.25	3,735,123,121.30	2,004.72
store_name_idx	379,487.76	379,487.76	31,356,827.67	386,116.00	-135,022.17	-15,140.03	3,734,927,061.33	6,055.14
first_name_idx	240,473.06	352,243.36	31,166,431.08	390,836.16	3,992.53	12,104.37	3,735,117,457.92	1,334.98
product_name_idx	240,473.06	347,529.92	31,158,286.82	389,683.52	3,992.53	16,817.81	3,735,125,602.18	2,487.62

Table 2: Effect of each Index on Performance of each Advanced Query

A link to the spreadsheet is also included here:
https://docs.google.com/spreadsheets/d/1BvPGekq7LShw3qMzfSV7L9_wHCi4UObQQhtqogh4l2I/edit?usp=sharing

With the results given by Table 2, the team formulated three possible configurations in hopes of improving performance:

Configuration #	Index Names
1	store_products_idx, store_name_idx
2	first_name_idx, product_name_idx
3	products_foodgroup_idx, product_name_idx

Performance Variations

The following section details the various results on each Query with each Configuration. Screenshots of these results are provided for reference, as well as a short analysis of each configuration's results.

Non-Indexed Results

Query 1

```
MySQL [Grocery-Aid-Database]> SELECT COUNT( Products.product_id ) AS NumProducts, Stores.store_name FROM Products JOIN Stores ON Products.sto
re_id = Stores.store_id WHERE Products.product_id IN (SELECT product_id FROM FoodGroups WHERE fruits = 1 UNION SELECT product_id FROM FoodGro
ups WHERE vegetables = 1) GROUP BY Stores.store_id LIMIT 15;
+-------------+------------+
| NumProducts | store_name |
+-------------+------------+
|          30 | Amazon     |
|          31 | Costco     |
|       19849 | Walmart    |
|          42 | WholeFoods |
+-------------+------------+
4 rows in set (10.347 sec)
```

```
| -> Group aggregate: count(Products.product_id)  (cost=155989.68 rows=776141) (actual time=10578.841..10578.841 rows=0 loops=1)
    -> Nested loop inner join  (cost=78375.55 rows=776141) (actual time=10578.839..10578.839 rows=0 loops=1)
        -> Index scan on Stores using PRIMARY  (cost=1.40 rows=4) (actual time=1.161..2.663 rows=4 loops=1)
        -> Filter: <in_optimizer>(Products.product_id,<exists>(select #2))  (cost=5040.89 rows=194035) (actual time=2644.042..2644.042 rows=0
 loops=4)
            -> Covering index lookup on Products using Products_ibfk_1 (store_id=Stores.store_id)  (cost=5040.89 rows=194035) (actual time=0.
726..115.607 rows=143677 loops=4)
            -> Select #2 (subquery in condition; dependent)
                -> Limit: 1 row(s)  (cost=4.54..4.54 rows=0.2) (actual time=0.017..0.017 rows=0 loops=574709)
                    -> Table scan on <union temporary>  (cost=4.54..4.54 rows=0.2) (actual time=0.017..0.017 rows=0 loops=574709)
                        -> Union materialize with deduplication  (cost=2.04..2.04 rows=0.2) (actual time=0.017..0.017 rows=0 loops=574709)
                            -> Limit table size: 1 unique row(s)
                                -> Limit: 1 row(s)  (cost=1.01 rows=0.1) (actual time=0.008..0.008 rows=0 loops=574709)
                                    -> Filter: (FoodGroup.fruits = 1)  (cost=1.01 rows=0.1) (actual time=0.008..0.008 rows=0 loops=574709)
                                        -> Index lookup on FoodGroup using product_id (product_id=<cache>(Products.product_id))  (cost=1.01 r
ows=1) (actual time=0.007..0.008 rows=1 loops=574709)
                            -> Limit table size: 1 unique row(s)
                                -> Limit: 1 row(s)  (cost=1.01 rows=0.1) (actual time=0.008..0.008 rows=0 loops=574709)
                                    -> Filter: (FoodGroup.vegetables = 1)  (cost=1.01 rows=0.1) (actual time=0.007..0.007 rows=0 loops=574709
)
                                        -> Index lookup on FoodGroup using product_id (product_id=<cache>(Products.product_id))  (cost=1.01 r
ows=1) (actual time=0.007..0.007 rows=1 loops=574709)
 |
```

Query 2

```
MySQL [Grocery-Aid-Database]> SELECT Stores.store_name, AVG( Products.price ) AS AvgPrice, COUNT( Products.store_id ) FROM Products JOIN Stor
es ON Products.store_id = Stores.store_id JOIN FoodGroups ON Products.product_id = FoodGroups.product_id WHERE other = 1 GROUP BY Stores.stor
e_id LIMIT 15;
+------------+--------------------+---------------------------+
| store_name | AvgPrice           | COUNT( Products.store_id ) |
+------------+--------------------+---------------------------+
| Walmart    |  5.105541222079056 |                    526429 |
| Costco     | 55.464345794394255 |                      1712 |
| WholeFoods |  8.407079081632583 |                      1568 |
| Amazon     | 21.571779725609616 |                      2624 |
+------------+--------------------+---------------------------+
4 rows in set (3.333 sec)
```

```
-+
| -> Table scan on <temporary>  (actual time=4072.881..4072.882 rows=4 loops=1)
    -> Aggregate using temporary table  (actual time=4072.877..4072.877 rows=4 loops=1)
        -> Inner hash join (Stores.store_id = Products.store_id)  (cost=134961.73 rows=57376) (actual time=3671.410..3812.577 rows=574709 loo
ps=1)
            -> Table scan on Stores  (cost=0.00 rows=4) (actual time=1.752..1.767 rows=4 loops=1)
            -> Hash
                -> Nested loop inner join  (cost=112005.51 rows=57376) (actual time=5.748..3502.287 rows=574709 loops=1)
                    -> Filter: (FoodGroup.other = 1)  (cost=58690.24 rows=57376) (actual time=3.123..453.561 rows=574709 loops=1)
                        -> Table scan on FoodGroup  (cost=58690.24 rows=573760) (actual time=3.117..398.379 rows=574709 loops=1)
                    -> Single-row index lookup on Products using PRIMARY (product_id=FoodGroup.product_id)  (cost=0.83 rows=1) (actual time=0
.005..0.005 rows=1 loops=574709)
 |
```

Query 3

```
MySQL [Grocery-Aid-Database]> SELECT DISTINCT Stores.store_name, Products.name, Products.price AS min_price
    -> FROM Stores JOIN Products ON Stores.store_id = Products.store_id INNER JOIN
    ->     ( SELECT store_id, MIN(price) AS price FROM Products WHERE name LIKE
    ->      '%eggs%'GROUP BY store_id) AS min_prices
    ->      ON Stores.store_id = min_prices.store_id AND Products.price = min_prices.price
    -> WHERE Products.name LIKE '%eggs%'
    -> ORDER BY Stores.store_name ASC, Products.price ASC;
+-------------+-----------------------------------------------------------------------------------------------+-----------+
| store_name  | name                                                                                          | min_price |
+-------------+-----------------------------------------------------------------------------------------------+-----------+
| Amazon      | The Ojai Cook Garlic Herb Lemonaise - Organic Mayonnaise Aioli Made With Cage-Free Eggs |     12.95 |
| Walmart     | Great Value Large White Eggs                                                                  |      1.05 |
| Whole Foods | Pasture Raised Large Grade A Eggs                                                             |      7.99 |
+-------------+-----------------------------------------------------------------------------------------------+-----------+
3 rows in set (5.296 sec)
```

```
-----------------------------------------+
| -> Sort: Stores.store_name, Products.price  (actual time=6400.943..6400.944 rows=3 loops=1)
    -> Table scan on <temporary>  (cost=1255026171.05..1307306983.44 rows=4182464793) (actual time=6400.920..6400.921 rows=3 loops=1)
        -> Temporary table with deduplication  (cost=1255026171.04..1255026171.04 rows=4182464793) (actual time=6400.915..6400.915 rows=3 loops=1)
            -> Nested loop inner join  (cost=836779691.79 rows=4182464793) (actual time=4749.949..6400.841 rows=4 loops=1)
                -> Nested loop inner join  (cost=418468540.56 rows=4182464793) (actual time=4748.280..6398.475 rows=4 loops=1)
                    -> Filter: ((Products.`name` like '%eggs%') and (Products.price is not null))  (cost=60378.86 rows=64672) (actual time=13.194..1661.448 rows=1138 loops=1)
                        -> Table scan on Products  (cost=60378.86 rows=582106) (actual time=2.995..1261.754 rows=574709 loops=1)
                    -> Covering index lookup on min_prices using <auto_key0> (store_id=Products.store_id, price=Products.price)  (actual time=4.162..4.162 rows=0 loops=1138)
                        -> Materialize  (cost=73313.26..73313.26 rows=64672) (actual time=4735.074..4735.074 rows=3 loops=1)
                            -> Group aggregate: min(Products.price)  (cost=66846.06 rows=64672) (actual time=56.120..4735.004 rows=3 loops=1)
                                -> Filter: (Products.`name` like '%eggs%')  (cost=60378.86 rows=64672) (actual time=12.516..4733.944 rows=1138 loops=1)
                                    -> Index scan on Products using Products_ibfk_1  (cost=60378.86 rows=582106) (actual time=2.115..4327.603 rows=574709 loops=1)
                -> Single-row index lookup on Stores using PRIMARY (store_id=Products.store_id)  (cost=0.10 rows=1) (actual time=0.590..0.590 rows=1 loops=4)
 |
```

Query 4

```
MySQL [Grocery-Aid-Database]> SELECT Stores.store_name, COUNT(Products.name) as count_product, (COUNT(Products.name) /  (SELECT COUNT(*) FROM
 Products WHERE name LIKE "%sriracha%") ) * 100 AS probability
    -> FROM Products JOIN Stores on Products.store_id = Stores.store_id
    -> WHERE Products.name LIKE "%sriracha%"
    -> GROUP BY Stores.store_name
    -> ORDER BY Stores.store_name ASC;
+------------+---------------+-------------+
| store_name | count_product | probability |
+------------+---------------+-------------+
| Amazon     |            49 |     15.6051 |
| Costco     |             1 |      0.3185 |
| Walmart    |           264 |     84.0764 |
+------------+---------------+-------------+
3 rows in set (1.571 sec)
```

```
| -> Sort: Stores.store_name  (actual time=943.152..943.155 rows=3 loops=1)
    -> Table scan on <temporary>  (actual time=943.110..943.111 rows=3 loops=1)
        -> Aggregate using temporary table  (actual time=943.105..943.105 rows=3 loops=1)
            -> Nested loop inner join  (cost=1.14 rows=1) (actual time=1.304..941.655 rows=314 loops=1)
                -> Filter: (Products.`name` like '%sriracha%')  (cost=0.79 rows=1) (actual time=0.649..940.455 rows=314 loops=1)
                    -> Table scan on Products  (cost=0.79 rows=1) (actual time=0.615..553.517 rows=574709 loops=1)
                -> Single-row index lookup on Stores using PRIMARY (store_id=Products.store_id)  (cost=0.35 rows=1) (actual time=0.001..0.001 rows=1 loops=314)
-> Select #2 (subquery in projection; run only once)
    -> Aggregate: count(0)  (cost=0.89 rows=1) (actual time=942.896..942.896 rows=1 loops=1)
        -> Filter: (Products.`name` like '%sriracha%')  (cost=0.79 rows=1) (actual time=1.289..942.601 rows=314 loops=1)
            -> Table scan on Products  (cost=0.79 rows=1) (actual time=1.139..579.074 rows=574709 loops=1)
 |
```

Total Cost without any Indices

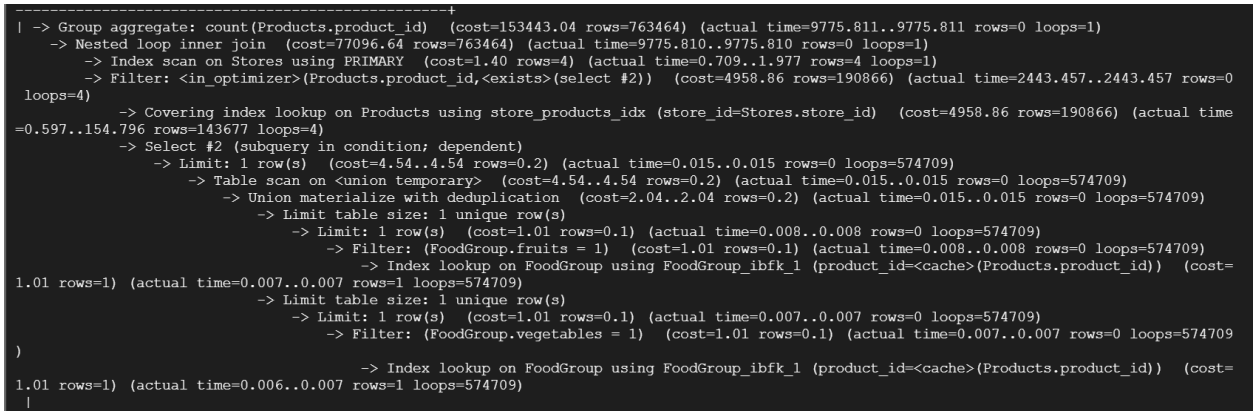
Query	Total Cost
1	244465.59
2	364347.73
3	3,766,283,889
4	392171.14

These results represent the baseline performance of our database - shown in the data, Query 3 has an abnormally high cost compared to the other queries.

Configurations with Indices

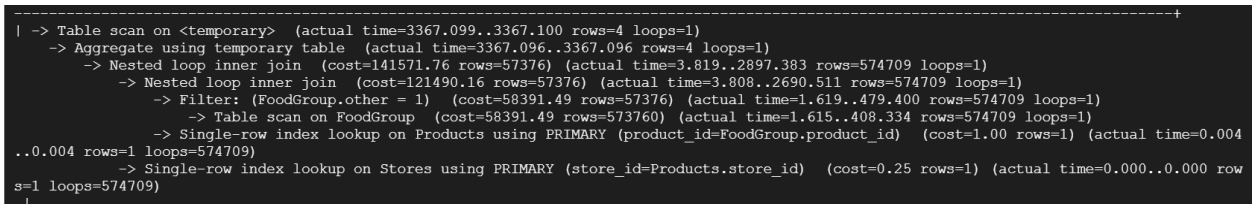
Config 1: store_products_idx, store_name_idx

Query 1



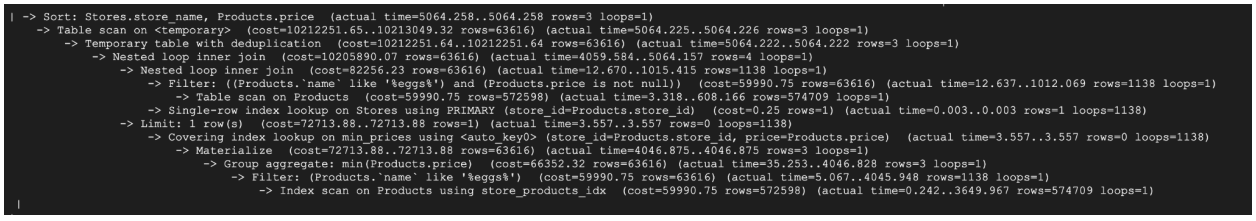
```
-> Group aggregate: count(Products.product_id)  (cost=153443.04 rows=763464) (actual time=9775.811..9775.811 rows=0 loops=1)
    -> Nested loop inner join  (cost=77096.64 rows=763464) (actual time=9775.810..9775.810 rows=0 loops=1)
        -> Index scan on Stores using PRIMARY  (cost=1.40 rows=4) (actual time=0.709..1.977 rows=4 loops=1)
        -> Filter: <in_optimizer>(Products.product_id,<exists>(select #2))  (cost=4958.86 rows=190866) (actual time=2443.457..2443.457 rows=0
 loops=4)
            -> Covering index lookup on Products using store_products_idx (store_id=Stores.store_id)  (cost=4958.86 rows=190866) (actual time
=0.597..154.796 rows=143677 loops=4)
            -> Select #2 (subquery in condition; dependent)
                -> Limit: 1 row(s)  (cost=4.54..4.54 rows=0.2) (actual time=0.015..0.015 rows=0 loops=574709)
                    -> Table scan on <union temporary>  (cost=4.54..4.54 rows=0.2) (actual time=0.015..0.015 rows=0 loops=574709)
                        -> Union materialize with deduplication  (cost=2.04..2.04 rows=0.2) (actual time=0.015..0.015 rows=0 loops=574709)
                            -> Limit table size: 1 unique row(s)
                                -> Limit: 1 row(s)  (cost=1.01 rows=0.1) (actual time=0.008..0.008 rows=0 loops=574709)
                                    -> Filter: (FoodGroup.fruits = 1)  (cost=1.01 rows=0.1) (actual time=0.008..0.008 rows=0 loops=574709)
                                        -> Index lookup on FoodGroup using FoodGroup_ibfk_1 (product_id=<cache>(Products.product_id))  (cost=
1.01 rows=1) (actual time=0.007..0.007 rows=1 loops=574709)
                            -> Limit table size: 1 unique row(s)
                                -> Limit: 1 row(s)  (cost=1.01 rows=0.1) (actual time=0.007..0.007 rows=0 loops=574709)
                                    -> Filter: (FoodGroup.vegetables = 1)  (cost=1.01 rows=0.1) (actual time=0.007..0.007 rows=0 loops=574709
)
                                        -> Index lookup on FoodGroup using FoodGroup_ibfk_1 (product_id=<cache>(Products.product_id))  (cost=
1.01 rows=1) (actual time=0.006..0.007 rows=1 loops=574709)
```

Query 2



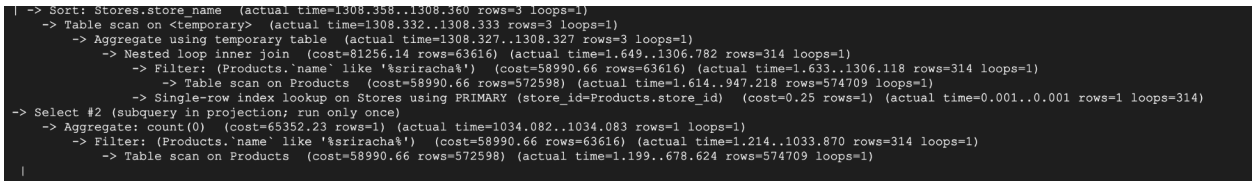
```
-> Table scan on <temporary>  (actual time=3367.099..3367.100 rows=4 loops=1)
    -> Aggregate using temporary table  (actual time=3367.096..3367.096 rows=4 loops=1)
        -> Nested loop inner join  (cost=141571.76 rows=57376) (actual time=3.819..2897.383 rows=574709 loops=1)
            -> Nested loop inner join  (cost=121490.16 rows=57376) (actual time=3.808..2690.511 rows=574709 loops=1)
                -> Filter: (FoodGroup.other = 1)  (cost=58391.49 rows=57376) (actual time=1.619..479.400 rows=574709 loops=1)
                    -> Table scan on FoodGroup  (cost=58391.49 rows=573760) (actual time=1.615..408.334 rows=574709 loops=1)
                -> Single-row index lookup on Products using PRIMARY (product_id=FoodGroup.product_id)  (cost=1.00 rows=1) (actual time=0.004
..0.004 rows=1 loops=574709)
            -> Single-row index lookup on Stores using PRIMARY (store_id=Products.store_id)  (cost=0.25 rows=1) (actual time=0.000..0.000 row
s=1 loops=574709)
```

Query 3



```
-> Sort: Stores.store_name, Products.price  (actual time=5064.258..5064.258 rows=3 loops=1)
    -> Table scan on <temporary>  (cost=10212251.65..10213049.32 rows=63616) (actual time=5064.225..5064.226 rows=3 loops=1)
        -> Temporary table with deduplication  (cost=10212251.64..10212251.64 rows=63616) (actual time=5064.222..5064.222 rows=3 loops=1)
            -> Nested loop inner join  (cost=10205890.07 rows=63616) (actual time=4059.584..5064.157 rows=4 loops=1)
                -> Nested loop inner join  (cost=82256.23 rows=63616) (actual time=12.670..1015.415 rows=1138 loops=1)
                    -> Filter: ((Products.`name` like '%eggs%') and (Products.price is not null))  (cost=59990.75 rows=63616) (actual time=12.637..1012.069 rows=1138 loops=1)
                        -> Table scan on Products  (cost=59990.75 rows=572598) (actual time=3.318..608.166 rows=574709 loops=1)
                    -> Single-row index lookup on Stores using PRIMARY (store_id=Products.store_id)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=1138)
                -> Limit: 1 row(s)  (cost=72713.88..72713.88 rows=1) (actual time=3.557..3.557 rows=0 loops=1138)
                    -> Covering index lookup on min_prices using <auto_key0> (store_id=Products.store_id, price=Products.price)  (actual time=3.557..3.557 rows=0 loops=1138)
                        -> Materialize  (cost=72713.88..72713.88 rows=63616) (actual time=4046.875..4046.875 rows=3 loops=1)
                            -> Group aggregate: min(Products.price)  (cost=66352.32 rows=63616) (actual time=35.253..4046.828 rows=3 loops=1)
                                -> Filter: (Products.`name` like '%eggs%')  (cost=59990.75 rows=63616) (actual time=5.067..4045.948 rows=1138 loops=1)
                                    -> Index scan on Products using store_products_idx  (cost=59990.75 rows=572598) (actual time=0.242..3649.967 rows=574709 loops=1)
```

Query 4



```
-> Sort: Stores.store_name  (actual time=1308.358..1308.360 rows=3 loops=1)
    -> Table scan on <temporary>  (actual time=1308.332..1308.333 rows=3 loops=1)
        -> Aggregate using temporary table  (actual time=1308.327..1308.327 rows=3 loops=1)
            -> Nested loop inner join  (cost=81256.14 rows=63616) (actual time=1.649..1306.782 rows=314 loops=1)
                -> Filter: (Products.`name` like '%sriracha%')  (cost=58990.66 rows=63616) (actual time=1.633..1306.118 rows=314 loops=1)
                    -> Table scan on Products  (cost=58990.66 rows=572598) (actual time=1.614..947.218 rows=574709 loops=1)
                -> Single-row index lookup on Stores using PRIMARY (store_id=Products.store_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=314)
-> Select #2 (subquery in projection; run only once)
    -> Aggregate: count(0)  (cost=65352.23 rows=1) (actual time=1034.082..1034.083 rows=1 loops=1)
        -> Filter: (Products.`name` like '%sriracha%')  (cost=58990.66 rows=63616) (actual time=1.214..1033.870 rows=314 loops=1)
            -> Table scan on Products  (cost=58990.66 rows=572598) (actual time=1.199..678.624 rows=574709 loops=1)
```

	Performance (Query #)				Performance Change (Query #) (INITAL - FINAL)			
Index Name	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**
Config 1 (1, 4)	240475.9	379846.1	31,164,3	382571.2	3989.61	-15498.4	3,735,119	9599.88

	8	5	92.92	6		2	,496	

In configuration 1, the database gains a performance improvement for query 2, but suffers from performance degradations for the other queries. However, the performance degradations for queries 1 and 4 are relatively reasonable, whereas the performance degradation for query 3 is very large.

Config 2: first_name_idx, product_name_idx

Query 1

```
-----------------------------------------------------+
| -> Group aggregate: count(Products.product_id)  (cost=155256.26 rows=772492) (actual time=10763.737..10763.737 rows=0 loops=1)
    -> Nested loop inner join  (cost=78007.06 rows=772492) (actual time=10763.735..10763.735 rows=0 loops=1)
        -> Index scan on Stores using PRIMARY  (cost=1.40 rows=4) (actual time=1.674..2.727 rows=4 loops=1)
        -> Filter: <in_optimizer>(Products.product_id,<exists>(select #2))  (cost=5017.19 rows=193123) (actual time=2690.250..2690.250 rows=0
 loops=4)
            -> Covering index lookup on Products using Products_ibfk_1 (store_id=Stores.store_id)  (cost=5017.19 rows=193123) (actual time=0.
444..102.898 rows=143677 loops=4)
            -> Select #2 (subquery in condition; dependent)
                -> Limit: 1 row(s)  (cost=4.54..4.54 rows=0.2) (actual time=0.017..0.017 rows=0 loops=574709)
                    -> Table scan on <union temporary>  (cost=4.54..4.54 rows=0.2) (actual time=0.017..0.017 rows=0 loops=574709)
                        -> Union materialize with deduplication  (cost=2.04..2.04 rows=0.2) (actual time=0.017..0.017 rows=0 loops=574709)
                            -> Limit table size: 1 unique row(s)
                                -> Limit: 1 row(s)  (cost=1.01 rows=0.1) (actual time=0.009..0.009 rows=0 loops=574709)
                                    -> Filter: (FoodGroup.fruits = 1)  (cost=1.01 rows=0.1) (actual time=0.008..0.008 rows=0 loops=574709)
                                        -> Index lookup on FoodGroup using FoodGroup_ibfk_1 (product_id=<cache>(Products.product_id))  (cost=
1.01 rows=1) (actual time=0.008..0.008 rows=1 loops=574709)
                            -> Limit table size: 1 unique row(s)
                                -> Limit: 1 row(s)  (cost=1.01 rows=0.1) (actual time=0.008..0.008 rows=0 loops=574709)
                                    -> Filter: (FoodGroup.vegetables = 1)  (cost=1.01 rows=0.1) (actual time=0.008..0.008 rows=0 loops=574709
)
                                        -> Index lookup on FoodGroup using FoodGroup_ibfk_1 (product_id=<cache>(Products.product_id))  (cost=
1.01 rows=1) (actual time=0.007..0.008 rows=1 loops=574709)
 |
```

Query 2

```
-+
| -> Table scan on <temporary>  (actual time=5095.042..5095.043 rows=4 loops=1)
    -> Aggregate using temporary table  (actual time=5095.037..5095.037 rows=4 loops=1)
        -> Inner hash join (Stores.store_id = Products.store_id)  (cost=131197.54 rows=57376) (actual time=4684.491..4832.093 rows=574709 loo
ps=1)
            -> Table scan on Stores  (cost=0.00 rows=4) (actual time=1.424..1.436 rows=4 loops=1)
            -> Hash
                -> Nested loop inner join  (cost=108241.33 rows=57376) (actual time=3.890..4533.919 rows=574709 loops=1)
                    -> Filter: (FoodGroup.other = 1)  (cost=58615.37 rows=57376) (actual time=2.617..679.601 rows=574709 loops=1)
                        -> Table scan on FoodGroup  (cost=58615.37 rows=573760) (actual time=2.612..621.337 rows=574709 loops=1)
                    -> Single-row index lookup on Products using PRIMARY (product_id=FoodGroup.product_id)  (cost=0.76 rows=1) (actual time=0
.006..0.006 rows=1 loops=574709)
 |
```

Query 3

```
| -> Sort: Stores.store_name, Products.price  (actual time=6491.679..6491.679 rows=3 loops=1)
    -> Table scan on <temporary>  (cost=10453425.66..10454232.73 rows=64368) (actual time=6491.659..6491.660 rows=3 loops=1)
        -> Temporary table with deduplication  (cost=10453425.65..10453425.65 rows=64368) (actual time=6491.655..6491.655 rows=3 loops=1)
            -> Nested loop inner join  (cost=10446988.86 rows=64368) (actual time=4599.159..6491.599 rows=4 loops=1)
                -> Nested loop inner join  (cost=82630.89 rows=64368) (actual time=12.014..1902.313 rows=1138 loops=1)
                    -> Filter: ((Products.`name` like '%eggs%') and (Products.price is not null))  (cost=60102.12 rows=64368) (actual time=11.991..1898.972 rows=1138 loops=1)
                        -> Table scan on Products  (cost=60102.12 rows=579369) (actual time=3.105..1375.851 rows=574709 loops=1)
                    -> Single-row index lookup on Stores using PRIMARY (store_id=Products.store_id)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=1138)
                -> Limit: 1 row(s)  (cost=72975.70..72975.70 rows=1) (actual time=4.032..4.032 rows=0 loops=1138)
                    -> Covering index lookup on min_prices using <auto_key0> (store_id=Products.store_id, price=Products.price)  (actual time=4.032..4.032 rows=0 loops=1138)
                        -> Materialize  (cost=72975.70..72975.70 rows=64368) (actual time=4587.130..4587.130 rows=3 loops=1)
                            -> Group aggregate: min(Products.price)  (cost=66538.91 rows=64368) (actual time=51.365..4587.083 rows=3 loops=1)
                                -> Filter: (Products.`name` like '%eggs%')  (cost=60102.12 rows=64368) (actual time=11.492..4585.603 rows=1138 loops=1)
                                    -> Index scan on Products using Products_ibfk_1  (cost=60102.12 rows=579369) (actual time=1.616..4185.289 rows=574709 loops=1)
 |
```

Query 4

```
| -> Sort: Stores.store_name  (actual time=1313.685..1313.688 rows=3 loops=1)
    -> Table scan on <temporary>  (actual time=1313.667..1313.667 rows=3 loops=1)
        -> Aggregate using temporary table  (actual time=1313.665..1313.665 rows=3 loops=1)
            -> Inner hash join (Stores.store_id = Products.store_id)  (cost=86474.55 rows=64368) (actual time=1313.413..1313.453 rows=314 loops=1)
                -> Table scan on Stores  (cost=0.00 rows=4) (actual time=1.335..1.339 rows=4 loops=1)
                -> Hash
                    -> Filter: (Products.`name` like '%sriracha%')  (cost=60474.21 rows=64368) (actual time=2.620..1311.554 rows=314 loops=1)
                        -> Table scan on Products  (cost=60474.21 rows=579369) (actual time=2.597..948.451 rows=574709 loops=1)
-> Select #2 (subquery in projection; run only once)
    -> Aggregate: count(0)  (cost=66911.00 rows=1) (actual time=1623.085..1623.085 rows=1 loops=1)
        -> Filter: (Products.`name` like '%sriracha%')  (cost=60474.21 rows=64368) (actual time=58.032..1622.945 rows=314 loops=1)
            -> Covering index scan on Products using product_name_idx  (cost=60474.21 rows=579369) (actual time=1.587..1270.852 rows=574709 loops=1)
 |
```

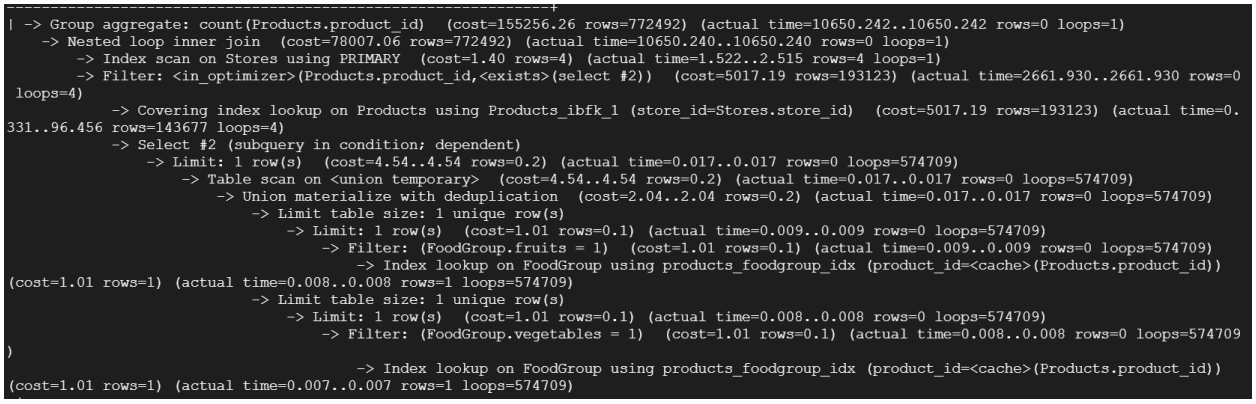
	Performance (Query #)	Performance Change (Query #) (INITAL

					- FINAL)			
Index Name	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**
Config 2 (5, 6)	243316.28	356670.37	31,889,370.10	395282.39	1149.31	7677.36	3,734,394,519	-3111.25

In configuration 2, the database gains a performance improvement for query 4, but suffers performance degradations in the other queries. However, the performance degradations for queries 1 and 2 are relatively reasonable, whereas the performance degradation for query 3 is very large.

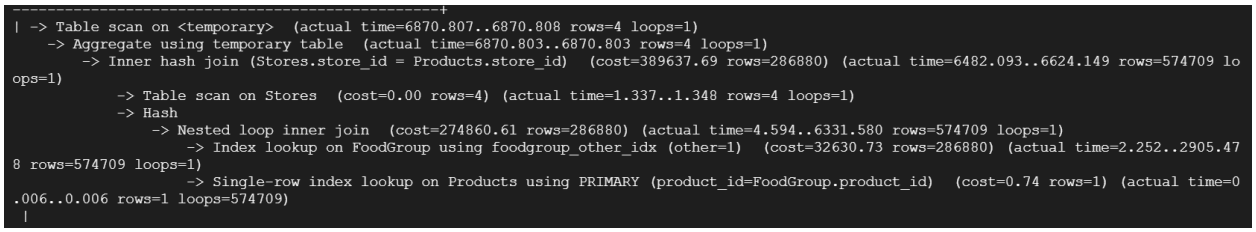
Config 3: products_foodgroup_idx, product_name_idx

Query 1



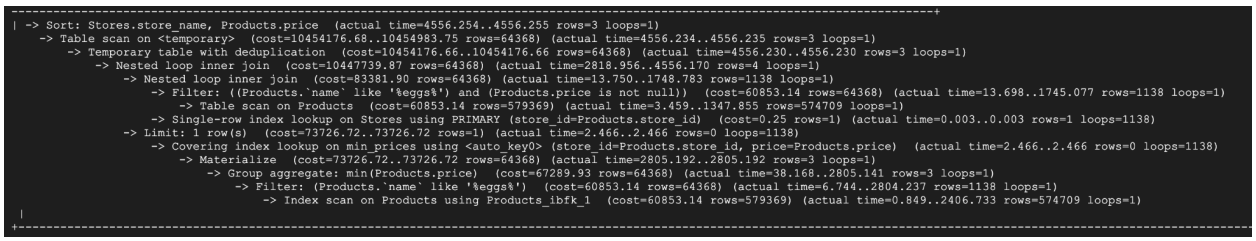
```
-> Group aggregate: count(Products.product_id)  (cost=155256.26 rows=772492) (actual time=10650.242..10650.242 rows=0 loops=1)
    -> Nested loop inner join  (cost=78007.06 rows=772492) (actual time=10650.240..10650.240 rows=0 loops=1)
        -> Index scan on Stores using PRIMARY  (cost=1.40 rows=4) (actual time=1.522..2.515 rows=4 loops=1)
        -> Filter: <in_optimizer>(Products.product_id,<exists>(select #2))  (cost=5017.19 rows=193123) (actual time=2661.930..2661.930 rows=0
 loops=4)
            -> Covering index lookup on Products using Products_ibfk_1 (store_id=Stores.store_id)  (cost=5017.19 rows=193123) (actual time=0.
331..96.456 rows=143677 loops=4)
            -> Select #2 (subquery in condition; dependent)
                -> Limit: 1 row(s)  (cost=4.54..4.54 rows=0.2) (actual time=0.017..0.017 rows=0 loops=574709)
                    -> Table scan on <union temporary>  (cost=4.54..4.54 rows=0.2) (actual time=0.017..0.017 rows=0 loops=574709)
                        -> Union materialize with deduplication  (cost=2.04..2.04 rows=0.2) (actual time=0.017..0.017 rows=0 loops=574709)
                            -> Limit table size: 1 unique row(s)
                                -> Limit: 1 row(s)  (cost=1.01 rows=0.1) (actual time=0.009..0.009 rows=0 loops=574709)
                                    -> Filter: (FoodGroup.fruits = 1)  (cost=1.01 rows=0.1) (actual time=0.009..0.009 rows=0 loops=574709)
                                        -> Index lookup on FoodGroup using products_foodgroup_idx (product_id=<cache>(Products.product_id))
(cost=1.01 rows=1) (actual time=0.008..0.008 rows=1 loops=574709)
                            -> Limit table size: 1 unique row(s)
                                -> Limit: 1 row(s)  (cost=1.01 rows=0.1) (actual time=0.008..0.008 rows=0 loops=574709)
                                    -> Filter: (FoodGroup.vegetables = 1)  (cost=1.01 rows=0.1) (actual time=0.008..0.008 rows=0 loops=574709
)
                                        -> Index lookup on FoodGroup using products_foodgroup_idx (product_id=<cache>(Products.product_id))
(cost=1.01 rows=1) (actual time=0.007..0.007 rows=1 loops=574709)
```

Query 2



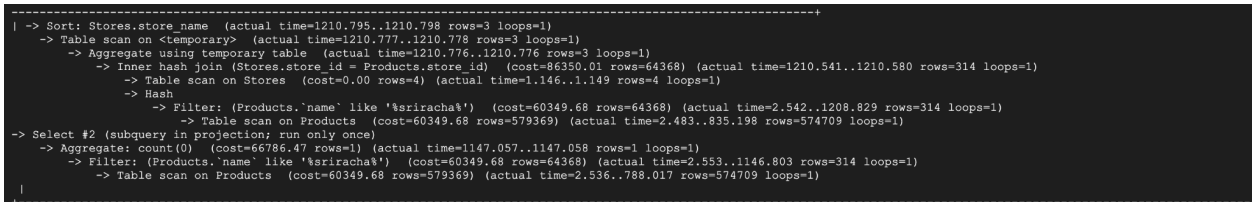
```
-> Table scan on <temporary>  (actual time=6870.807..6870.808 rows=4 loops=1)
    -> Aggregate using temporary table  (actual time=6870.803..6870.803 rows=4 loops=1)
        -> Inner hash join (Stores.store_id = Products.store_id)  (cost=389637.69 rows=286880) (actual time=6482.093..6624.149 rows=574709 lo
ops=1)
            -> Table scan on Stores  (cost=0.00 rows=4) (actual time=1.337..1.348 rows=4 loops=1)
            -> Hash
                -> Nested loop inner join  (cost=274860.61 rows=286880) (actual time=4.594..6331.580 rows=574709 loops=1)
                    -> Index lookup on FoodGroup using foodgroup_other_idx (other=1)  (cost=32630.73 rows=286880) (actual time=2.252..2905.47
8 rows=574709 loops=1)
                    -> Single-row index lookup on Products using PRIMARY (product_id=FoodGroup.product_id)  (cost=0.74 rows=1) (actual time=0
.006..0.006 rows=1 loops=574709)
```

Query 3



```
-> Sort: Stores.store_name, Products.price  (actual time=4556.254..4556.255 rows=3 loops=1)
    -> Table scan on <temporary>  (cost=10454176.68..10454983.75 rows=64368) (actual time=4556.234..4556.235 rows=3 loops=1)
        -> Temporary table with deduplication  (cost=10454176.66..10454176.66 rows=64368) (actual time=4556.230..4556.230 rows=3 loops=1)
            -> Nested loop inner join  (cost=10447739.87 rows=64368) (actual time=2818.956..4556.170 rows=4 loops=1)
                -> Nested loop inner join  (cost=83381.90 rows=64368) (actual time=13.750..1748.783 rows=1138 loops=1)
                    -> Filter: ((Products.`name` like '%eggs%') and (Products.price is not null))  (cost=60853.14 rows=64368) (actual time=13.698..1745.077 rows=1138 loops=1)
                        -> Table scan on Products  (cost=60853.14 rows=579369) (actual time=3.459..1347.855 rows=574709 loops=1)
                    -> Single-row index lookup on Stores using PRIMARY (store_id=Products.store_id)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=1138)
                -> Limit: 1 row(s)  (cost=73726.72..73726.72 rows=1) (actual time=2.466..2.466 rows=0 loops=1138)
                    -> Covering index lookup on min_prices using <auto_key0> (store_id=Products.store_id, price=Products.price)  (actual time=2.466..2.466 rows=0 loops=1138)
                        -> Materialize  (cost=73726.72..73726.72 rows=64368) (actual time=2805.192..2805.192 rows=3 loops=1)
                            -> Group aggregate: min(Products.price)  (cost=67289.93 rows=64368) (actual time=38.168..2805.141 rows=3 loops=1)
                                -> Filter: (Products.`name` like '%eggs%')  (cost=60853.14 rows=64368) (actual time=6.744..2804.237 rows=1138 loops=1)
                                    -> Index scan on Products using Products_ibfk_1  (cost=60853.14 rows=579369) (actual time=0.849..2406.733 rows=574709 loops=1)
```

Query 4



	Performance (Query #)				Performance Change (Query #) (INITAL - FINAL)			
Index Name	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**
Config 3 (2, 3)	243316.28	697129.77	3189763 1.29	394535.2	1149.31	-332782.04	3,734,386,258	-2364.06

In configuration 3, the database gains a performance improvement for queries 2 and 4, but suffers performance degradations in the other queries. However, the performance degradation for query 1 is relatively reasonable, whereas the performance degradation for query 3 is very large.

Overall, each configuration has its advantages and disadvantages which should be taken into consideration when choosing the final database design. One thing to note is that despite any changes, query 3 consistently had a performance degradation regardless of the indexing, which may be due to the query's innate design.

Final Design

We first tested out three different configurations and from our cost analysis we came to the conclusion that the best configuration is the third one (products_foodgroup_idx, product_name_idx), as this query reduces the cost for query 2 and query 4. By observing the cost analysis for query 3, we realized there is not much change being to this particular query and it already has a pretty high performance degradation.

Adding indices to the food_group table, which aims to classify each product into a particular food category can speed up the process of searching because this table contains 574709 records of data. Classifying each column of this table (like using the "other" column) can speed up the time it takes to look through this data because once we know we are only looking for data from one category we can reduce the cost of searching significantly. We believe that this is what is reducing the time for query 2, which looks through the "other" category column.