Part 1: Database Implementation

Proof of implementation on GCP:

```
mysql> ^C
juandiegosanz7@cloudshell:~ (cs411-project-440121)$ 0121)$ gcloimport csv group-122 gs://bizbite/cs411-business.csv     --database=bizbites     --table=Busine
ss     --columns=Address,Name,Phone;
Data from [gs://bizbite/cs411-business.csv] will be imported to [group-122].

Do you want to continue (Y/n)?  y

Importing data into Cloud SQL instance...done.
Imported data from [gs://bizbite/cs411-business.csv] into [https://sqladmin.googleapis.com/sql/v1beta4/projects/cs411-project-440121/instances/group-122].
juandiegosanz7@cloudshell:~ (cs411-project-440121)$ gcloud sql connect group-122 --user=root
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4945
Server version: 8.0.31-google (Google)

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use bizbites
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```

Proof of 100 rows in at least three different tables:

```
mysql> use bizbites
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select count(*) from Business;
+----------+
| count(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.02 sec)

mysql> select count(*) from food;
+----------+
| count(*) |
+----------+
|    13182 |
+----------+
1 row in set (0.01 sec)

mysql> select count(*) from Restaurant;
+----------+
| count(*) |
+----------+
|     7629 |
+----------+
1 row in set (0.01 sec)
```

- DDL Commands

Restaurant

```
mysql> CREATE TABLE Restaurant (
    ->     Id CHAR(36) PRIMARY KEY DEFAULT (UUID()),
    ->     Address VARCHAR(255),
    ->     Name VARCHAR(100),
    ->     Phone VARCHAR(30)
    -> );
Query OK, 0 rows affected (0.07 sec)
```

Food (Nutrients field in UML was expanded to singular nutrient counts because of good data availability)

```
mysql> CREATE TABLE food (
    ->     ID CHAR(36) PRIMARY KEY DEFAULT (UUID()),
    ->     restaurant VARCHAR(255),
    ->     food_category VARCHAR(255),
    ->     item_name VARCHAR(255),
    ->     item_description VARCHAR(255),
    ->     calories FLOAT,
    ->     total_fat FLOAT,
    ->     saturated_fat FLOAT,
    ->     trans_fat FLOAT,
    ->     cholesterol FLOAT,
    ->     sodium FLOAT,
    ->     carbohydrates FLOAT,
    ->     dietary_fiber FLOAT,
    ->     sugar FLOAT,
    ->     protein FLOAT,
    ->     potassium FLOAT,
    ->     price FLOAT
    -> );
Query OK, 0 rows affected (0.14 sec)
```

Business

```
mysql> CREATE TABLE Business (
    ->     Id CHAR(36) PRIMARY KEY DEFAULT (UUID()),
    ->     Address VARCHAR(255),
    ->     Name VARCHAR(100),
    ->     Phone VARCHAR(30)
    -> );
Query OK, 0 rows affected (0.19 sec)
```

Employee:

```
mysql> CREATE TABLE Employee (
    ->     Id CHAR(36) PRIMARY KEY DEFAULT (UUID()),
    ->     Name VARCHAR(100),
    ->     Preferences TEXT,
    ->     Dietary_Needs TEXT,
    ->     PhoneNumber VARCHAR(30),
    ->     Business_Id CHAR(36),
    ->     FOREIGN KEY (Business_Id) REFERENCES Business(Id) ON DELETE CASCADE
    -> );
Query OK, 0 rows affected (0.33 sec)
```

Product:

```
mysql> CREATE TABLE Product (
    ->     R_Id CHAR(36),
    ->     F_Id CHAR(36),
    ->     PRIMARY KEY (R_Id, F_Id),
    ->     FOREIGN KEY (R_Id) REFERENCES Restaurant(Id),
    ->     FOREIGN KEY (F_Id) REFERENCES Food(Id)
    -> );
Query OK, 0 rows affected (0.13 sec)
```

Deals:

```
mysql> DESCRIBE Deals;
+--------------------+--------------+------+-----+---------+-------+
| Field              | Type         | Null | Key | Default | Extra |
+--------------------+--------------+------+-----+---------+-------+
| R_Id               | char(36)     | NO   | PRI | NULL    |       |
| B_Id               | char(36)     | NO   | PRI | NULL    |       |
| Discount_Percentage | decimal(5,2) | YES  |     | NULL    |       |
+--------------------+--------------+------+-----+---------+-------+
3 rows in set (0.01 sec)
```

Advanced Queries:

1. Query to find high-value meal deals

```
SELECT
   r.Name as Restaurant_Name,
   AVG(d.Discount_Percentage) as Avg_Discount,
   COUNT(DISTINCT f.food_category) as Menu_Categories,
   AVG(f.price) as Avg_Price
FROM Restaurant r
JOIN Deals d ON r.Id = d.R_Id
JOIN Product p ON r.Id = p.R_Id
JOIN Food f ON p.F_Id = f.ID
WHERE d.Discount_Percentage > (
   SELECT AVG(Discount_Percentage) * .3
   FROM Deals
)
GROUP BY r.Id, r.Name
HAVING AVG(f.price) < (
   SELECT AVG(price) * 2
   FROM Food
)
ORDER BY Avg_Discount DESC;
```

```
mysql> SELECT
    ->     r.Name as Restaurant_Name,
    ->     AVG(d.Discount_Percentage) as Avg_Discount,
    ->     COUNT(DISTINCT f.food_category) as Menu_Categories,
    ->     AVG(f.price) as Avg_Price
    -> FROM Restaurant r
    -> JOIN Deals d ON r.Id = d.R_Id
    -> JOIN Product p ON r.Id = p.R_Id
    -> JOIN Food f ON p.F_Id = f.ID
    -> WHERE d.Discount_Percentage > (
    ->     SELECT AVG(Discount_Percentage) * .1
    ->     FROM Deals
    -> )
    -> GROUP BY r.Id, r.Name
    -> HAVING AVG(f.price) < (
    ->     SELECT AVG(price) * 1.2
    ->     FROM Food
    -> )
    -> ORDER BY Avg_Discount DESC;
+-----------------+--------------+-----------------+--------------------+
| Restaurant_Name | Avg_Discount | Menu_Categories | Avg_Price          |
+-----------------+--------------+-----------------+--------------------+
| Jack in the Box |     0.150000 |               1 | 3.9000000953674316 |
| McDonald's      |     0.150000 |               1 |               6.25 |
| Taco Bell       |     0.150000 |               1 |  7.199999809265137 |
| Baskin Robbins  |     0.100000 |               1 | 2.4700000286102295 |
+-----------------+--------------+-----------------+--------------------+
4 rows in set (0.01 sec)
```

2. Query to analyze food categories by nutritional value

```
SELECT
   food_category,
   AVG(calories) as avg_calories,
   AVG(protein) as avg_protein,
   COUNT(*) as item_count,
   AVG(price) as avg_price
FROM Food
WHERE calories > 0
GROUP BY food_category
HAVING AVG(calories) < (
   SELECT AVG(calories) * 1.5
   FROM Food
   WHERE calories > 0
)
AND COUNT(*) > (
   SELECT COUNT(*)/20
   FROM Food
)
ORDER BY avg_protein DESC;
```

```
mysql> SELECT
    ->     r.Name as Restaurant_Name,
    ->     AVG(d.Discount_Percentage) as Avg_Discount,
    ->     COUNT(DISTINCT f.food_category) as Menu_Categories,
    ->     AVG(f.price) as Avg_Price
    -> FROM Restaurant r
    -> JOIN Deals d ON r.Id = d.R_Id
    -> JOIN Product p ON r.Id = p.R_Id
    -> JOIN Food f ON p.F_Id = f.ID
    -> WHERE d.Discount_Percentage > (
    ->     SELECT AVG(Discount_Percentage) * .3
    ->     FROM Deals
    -> )
    -> GROUP BY r.Id, r.Name
    -> HAVING AVG(f.price) < (
    ->     SELECT AVG(price) * 2
    ->     FROM Food
    -> )
    -> ORDER BY Avg_Discount DESC;
+-----------------+--------------+-----------------+--------------------+
| Restaurant_Name | Avg_Discount | Menu_Categories | Avg_Price          |
+-----------------+--------------+-----------------+--------------------+
| Hardee's        |     0.300000 |               1 | 20.280000686645508 |
| McDonald's      |     0.250000 |               1 | 21.969999313354492 |
| McDonald's      |     0.250000 |               1 | 18.850000381469727 |
| Quiznos         |     0.250000 |               1 | 21.049999237060547 |
| Jack in the Box |     0.150000 |               1 | 3.9000000953674316 |
| McDonald's      |     0.150000 |               1 |               6.25 |
| Taco Bell       |     0.150000 |               1 |  7.199999809265137 |
| Baskin Robbins  |     0.100000 |               1 | 2.4700000286102295 |
+-----------------+--------------+-----------------+--------------------+
8 rows in set (0.01 sec)
```

3. Query to compare business deal advantages

```
SELECT
   b.Name,
   COUNT(d.R_Id) as restaurant_deals,
   AVG(d.Discount_Percentage) as avg_discount,
   (
      SELECT COUNT(DISTINCT R_Id)
      FROM Deals d2
      WHERE d2.B_Id = b.Id
      AND d2.Discount_Percentage > 15
   ) as premium_deals
FROM Business b
JOIN Deals d ON b.Id = d.B_Id
GROUP BY b.Id, b.Name
HAVING AVG(d.Discount_Percentage) > (
   SELECT AVG(Discount_Percentage)
   FROM Deals
```

)
ORDER BY avg_discount DESC;

```
mysql> SELECT
    ->     b.Name,
    ->     COUNT(d.R_Id) as restaurant_deals,
    ->     AVG(d.Discount_Percentage) as avg_discount,
    ->     (
    ->         SELECT COUNT(DISTINCT R_Id)
    ->         FROM Deals d2
    ->         WHERE d2.B_Id = b.Id
    ->         AND d2.Discount_Percentage > 0.15
    ->     ) as premium_deals
    -> FROM Business b
    -> JOIN Deals d ON b.Id = d.B_Id
    -> GROUP BY b.Id, b.Name
    -> HAVING AVG(d.Discount_Percentage) > (
    ->     SELECT AVG(Discount_Percentage)
    ->     FROM Deals
    -> )
    -> ORDER BY avg_discount DESC
    -> LIMIT 15;
+-----------------------------+------------------+--------------+---------------+
| Name                        | restaurant_deals | avg_discount | premium_deals |
+-----------------------------+------------------+--------------+---------------+
| Terrell PLC                 |                1 |     0.300000 |             1 |
| Briggs-Hall                 |                1 |     0.300000 |             1 |
| Collier-James               |                1 |     0.300000 |             1 |
| Walker and Sons             |                1 |     0.300000 |             1 |
| Brown, Powell and Aguilar   |                1 |     0.300000 |             1 |
| Guzman, Hernandez and Parker |                1 |     0.300000 |             1 |
| Henry, Baker and Garcia     |                1 |     0.300000 |             1 |
| Hamilton-Wilson             |                1 |     0.300000 |             1 |
| Brown-Clark                 |                1 |     0.300000 |             1 |
| Castillo, Sanders and Martin |                1 |     0.300000 |             1 |
| Carter-Vargas               |                1 |     0.300000 |             1 |
| Adams and Sons              |                1 |     0.300000 |             1 |
| Harrison-Kane               |                1 |     0.300000 |             1 |
| Nicholson, Young and Blair  |                1 |     0.300000 |             1 |
| Mejia and Sons              |                1 |     0.300000 |             1 |
+-----------------------------+------------------+--------------+---------------+
15 rows in set (0.01 sec)
```

4. Query to find restaurants with the best healthy options

```
SELECT
  food_category,
  COUNT(*) as Total_Items,
  ROUND(AVG(protein), 1) as Avg_Protein_Grams,
  ROUND(AVG(calories), 1) as Avg_Calories,
  ROUND(AVG(protein / NULLIF(calories, 0) * 100), 2) as Avg_Protein_Calorie_Ratio,
  ROUND(MIN(price), 2) as Min_Price,
  ROUND(MAX(price), 2) as Max_Price,
  ROUND(AVG(protein / NULLIF(price, 0)), 2) as Protein_Price_Value
FROM Food
WHERE protein > 0
```

```
AND price < (
   SELECT AVG(price) * 1.5 FROM Food
)
GROUP BY food_category
HAVING
   Avg_Protein_Grams >= 10
   AND COUNT(*) >= 2
ORDER BY
   Avg_Protein_Calorie_Ratio DESC;
```

```
mysql> SELECT
    ->     food_category,
    ->     COUNT(*) as Total_Items,
    ->     ROUND(AVG(protein), 1) as Avg_Protein_Grams,
    ->     ROUND(AVG(calories), 1) as Avg_Calories,
    ->     ROUND(AVG(protein / NULLIF(calories, 0) * 100), 2) as Avg_Protein_Calorie_Ratio,
    ->     ROUND(MIN(price), 2) as Min_Price,
    ->     ROUND(MAX(price), 2) as Max_Price,
    ->     ROUND(AVG(protein / NULLIF(price, 0)), 2) as Protein_Price_Value
    -> FROM Food
    -> WHERE protein > 0
    -> GROUP BY food_category
    -> HAVING
    ->     Avg_Protein_Grams >= 10
    ->     AND COUNT(*) >= 2
    -> ORDER BY
    ->     Avg_Protein_Calorie_Ratio DESC;
+--------------------+-------------+------------------+--------------+--------------------------+-----------+-----------+---------------------+
| food_category      | Total_Items | Avg_Protein_Grams | Avg_Calories | Avg_Protein_Calorie_Ratio | Min_Price | Max_Price | Protein_Price_Value |
+--------------------+-------------+------------------+--------------+--------------------------+-----------+-----------+---------------------+
| Entrees            |         737 |             37.1 |          698 |                      5.8 |      1.03 |     24.88 |                5.15 |
| Salads             |         280 |             22.8 |        417.5 |                     5.79 |      1.01 |     24.91 |                2.79 |
| Burgers            |         288 |             34.7 |        696.8 |                     4.87 |      1.05 |     24.98 |                4.61 |
| Sandwiches         |        1308 |             27.8 |        588.2 |                     4.84 |         1 |        25 |                3.72 |
| Soup               |         114 |             16.9 |        365.2 |                     4.77 |      1.06 |     24.94 |                2.43 |
| Appetizers & Sides |         765 |             25.4 |        487.8 |                     4.66 |      1.06 |     24.97 |                3.51 |
| Pizza              |         504 |             21.4 |        573.1 |                        3 |         1 |     24.76 |                3.37 |
+--------------------+-------------+------------------+--------------+--------------------------+-----------+-----------+---------------------+
7 rows in set (0.02 sec)
```

Part 2: Indexing

Advanced SQL Query 1 Indexing

Non indexed:

```
Database changed
mysql> EXPLAIN ANALYZE
    -> SELECT
    ->     r.Name as Restaurant_Name,
    ->     AVG(d.Discount_Percentage) as Avg_Discou
    ->     COUNT(DISTINCT f.food_category) as Menu_
    ->     AVG(f.price) as Avg_Price
    -> FROM Restaurant r
    -> JOIN Deals d ON r.Id = d.R_Id
    -> JOIN Product p ON r.Id = p.R_Id
    -> JOIN Food f ON p.F_Id = f.ID
    -> WHERE d.Discount_Percentage > (
    ->     SELECT AVG(Discount_Percentage) * .3
    ->     FROM Deals
    -> )
    -> GROUP BY r.Id, r.Name
    -> HAVING AVG(f.price) < (
    ->     SELECT AVG(price) * 2
    ->     FROM Food
    -> )
    -> ORDER BY Avg_Discount DESC;
```

```
| -> Sort: Avg_Discount DESC  (actual time=6.776..6.777 rows=8 loops=1)
    -> Filter: (avg(Food.price) < (select #3))  (actual time=6.713..6.749 rows=8 loops=1)
        -> Stream results  (actual time=0.539..0.572 rows=8 loops=1)
            -> Group aggregate: avg(Food.price), avg(Deals.Discount_Percentage), count(distinct Food.food_category), avg(Food.price)  (actual time=0.534..0.559 rows=8 loops=1)
                -> Sort: r.Id, r.`Name`  (actual time=0.527..0.530 rows=8 loops=1)
                    -> Stream results  (cost=40.57 rows=20) (actual time=0.243..0.507 rows=8 loops=1)
                        -> Nested loop inner join  (cost=40.57 rows=20) (actual time=0.240..0.500 rows=8 loops=1)
                            -> Nested loop inner join  (cost=33.68 rows=20) (actual time=0.230..0.457 rows=8 loops=1)
                                -> Nested loop inner join  (cost=26.80 rows=20) (actual time=0.220..0.421 rows=8 loops=1)
                                    -> Covering index scan on p using F_Id  (cost=6.15 rows=59) (actual time=0.040..0.051 rows=59 loops=1)
                                    -> Filter: (d.Discount_Percentage > (select #2))  (cost=0.25 rows=0.3) (actual time=0.006..0.006 rows=0 loops=59)
                                        -> Index lookup on d using PRIMARY (R_Id=p.R_Id)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=0 loops=59)
                                        -> Select #2 (subquery in condition; run only once)
                                            -> Aggregate: avg(Deals.Discount_Percentage)  (cost=67.60 rows=1) (actual time=0.136..0.136 rows=1 loops=1)
                                                -> Table scan on Deals  (cost=34.30 rows=333) (actual time=0.039..0.098 rows=333 loops=1)
                                -> Single-row index lookup on r using PRIMARY (Id=p.R_Id)  (cost=0.26 rows=1) (actual time=0.004..0.004 rows=1 loops=8)
                            -> Single-row index lookup on f using PRIMARY (ID=p.F_Id)  (cost=0.26 rows=1) (actual time=0.005..0.005 rows=1 loops=8)
    -> Select #3 (subquery in condition; run only once)
        -> Aggregate: avg(Food.price)  (cost=2653.65 rows=1) (actual time=6.157..6.158 rows=1 loops=1)
            -> Table scan on Food  (cost=1354.95 rows=12987) (actual time=0.021..5.143 rows=13182 loops=1)
|
```

Table scan on Food for Select #3: Cost 1354.95
Nested loop join (Outer): Cost 48.24
Nested loop join (Middle): Cost 41.36
Nested loop join (Inner): Cost 27.55
Subquery in WHERE on Deals: Cost 67.6

CREATE INDEX idx_food_price ON Food(price);

- Attempted to reduce cost for aggregations and HAVING filters
    - Table scan on Food for Select #3: Cost 1354.95
    - Nested loop join (Outer): Cost 48.24
    - Nested loop join (Middle): Cost 41.36
    - Nested loop join (Inner): Cost 27.55
    - Subquery in WHERE on Deals: Cost 67.6

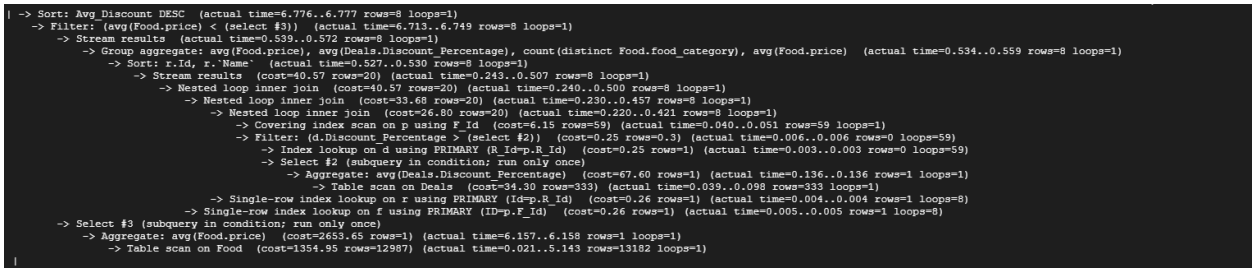
```
mysql> CREATE INDEX idx_food_price ON Food(price);
Query OK, 0 rows affected (0.32 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN ANALYZE
    -> SELECT
    ->     r.Name as Restaurant_Name,
    ->     AVG(d.Discount_Percentage) as Avg_Discount,
    ->     COUNT(DISTINCT f.food_category) as Menu_Categories,
    ->     AVG(f.price) as Avg_Price
    -> FROM Restaurant r
    -> JOIN Deals d ON r.Id = d.R_Id
    -> JOIN Product p ON r.Id = p.R_Id
    -> JOIN Food f ON p.F_Id = f.ID
    -> WHERE d.Discount_Percentage > (
    ->     SELECT AVG(Discount_Percentage) * .3
    ->     FROM Deals
    -> )
    -> GROUP BY r.Id, r.Name
    -> HAVING AVG(f.price) < (
    ->     SELECT AVG(price) * 2
    ->     FROM Food
    -> )
    -> ORDER BY Avg_Discount DESC;
```

```
| -> Sort: Avg_Discount DESC  (actual time=6.776..6.777 rows=8 loops=1)
    -> Filter: (avg(Food.price) < (select #3))  (actual time=6.713..6.749 rows=8 loops=1)
        -> Stream results  (actual time=0.539..0.572 rows=8 loops=1)
            -> Group aggregate: avg(Food.price), avg(Deals.Discount_Percentage), count(distinct Food.food_category), avg(Food.price)  (actual time=0.534..0.559 rows=8 loops=1)
                -> Sort: r.Id, r.`Name`  (actual time=0.527..0.530 rows=8 loops=1)
                    -> Stream results  (cost=40.57 rows=20) (actual time=0.243..0.507 rows=8 loops=1)
                        -> Nested loop inner join  (cost=40.57 rows=20) (actual time=0.240..0.500 rows=8 loops=1)
                            -> Nested loop inner join  (cost=33.68 rows=20) (actual time=0.230..0.457 rows=8 loops=1)
                                -> Nested loop inner join  (cost=26.80 rows=20) (actual time=0.220..0.421 rows=8 loops=1)
                                    -> Covering index scan on p using F_Id  (cost=6.15 rows=59) (actual time=0.040..0.051 rows=59 loops=1)
                                    -> Filter: (d.Discount_Percentage > (select #2))  (cost=0.25 rows=0.3) (actual time=0.006..0.006 rows=0 loops=59)
                                        -> Index lookup on d using PRIMARY (R_Id=p.R_Id)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=0 loops=59)
                                        -> Select #2 (subquery in condition; run only once)
                                            -> Aggregate: avg(Deals.Discount_Percentage)  (cost=67.60 rows=1) (actual time=0.136..0.136 rows=1 loops=1)
                                                -> Table scan on Deals  (cost=34.30 rows=333) (actual time=0.039..0.098 rows=333 loops=1)
                                -> Single-row index lookup on r using PRIMARY (Id=p.R_Id)  (cost=0.26 rows=1) (actual time=0.004..0.004 rows=1 loops=8)
                            -> Single-row index lookup on f using PRIMARY (ID=p.F_Id)  (cost=0.26 rows=1) (actual time=0.005..0.005 rows=1 loops=8)
    -> Select #3 (subquery in condition; run only once)
        -> Aggregate: avg(Food.price)  (cost=2653.65 rows=1) (actual time=6.157..6.158 rows=1 loops=1)
            -> Table scan on Food  (cost=1354.95 rows=12987) (actual time=0.021..5.143 rows=13182 loops=1)
|
```

CREATE INDEX idx_food_category ON Food(food_category);

- Attempted to reduce cost in COUNT(DISTINCT Food.food_category)
    - Table scan on Food for Select #3: Cost 1354.95
    - Nested loop join (Outer): Cost 48.24
    - Nested loop join (Middle): Cost 41.36
    - Nested loop join (Inner): Cost 27.55
    - Subquery in WHERE on Deals: Cost 67.6

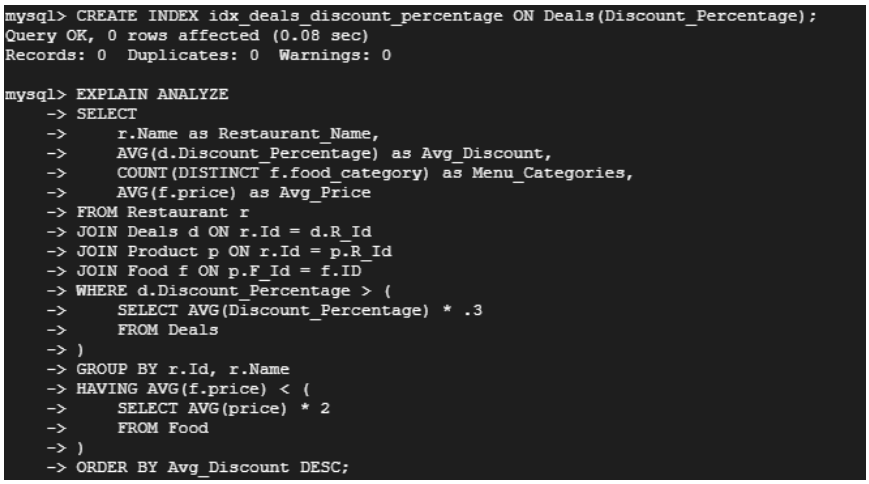
```
mysql> CREATE INDEX idx_food_category ON Food(food_category);
Query OK, 0 rows affected (0.38 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN ANALYZE
    -> SELECT
    ->     r.Name as Restaurant_Name,
    ->     AVG(d.Discount_Percentage) as Avg_Discount,
    ->     COUNT(DISTINCT f.food_category) as Menu_Categories,
    ->     AVG(f.price) as Avg_Price
    -> FROM Restaurant r
    -> JOIN Deals d ON r.Id = d.R_Id
    -> JOIN Product p ON r.Id = p.R_Id
    -> JOIN Food f ON p.F_Id = f.ID
    -> WHERE d.Discount_Percentage > (
    ->     SELECT AVG(Discount_Percentage) * .3
    ->     FROM Deals
    -> )
    -> GROUP BY r.Id, r.Name
    -> HAVING AVG(f.price) < (
    ->     SELECT AVG(price) * 2
    ->     FROM Food
    -> )
    -> ORDER BY Avg_Discount DESC;
```



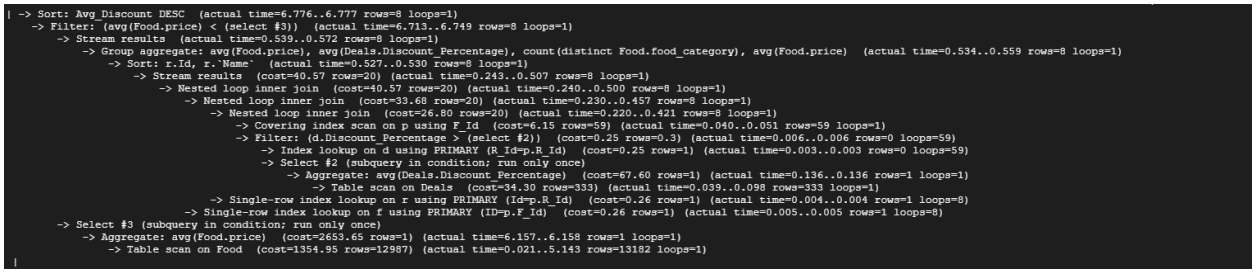
```
| -> Sort: Avg_Discount DESC  (actual time=6.776..6.777 rows=8 loops=1)
    -> Filter: (avg(Food.price) < (select #3))  (actual time=6.713..6.749 rows=8 loops=1)
        -> Stream results  (actual time=0.539..0.572 rows=8 loops=1)
            -> Group aggregate: avg(Food.price), avg(Deals.Discount_Percentage), count(distinct Food.food_category), avg(Food.price)  (actual time=0.534..0.559 rows=8 loops=1)
                -> Sort: r.Id, r.`Name`  (actual time=0.527..0.530 rows=8 loops=1)
                    -> Stream results  (cost=40.57 rows=20) (actual time=0.243..0.507 rows=8 loops=1)
                        -> Nested loop inner join  (cost=40.57 rows=20) (actual time=0.240..0.500 rows=8 loops=1)
                            -> Nested loop inner join  (cost=33.68 rows=20) (actual time=0.230..0.457 rows=8 loops=1)
                                -> Nested loop inner join  (cost=26.80 rows=20) (actual time=0.220..0.421 rows=8 loops=1)
                                    -> Covering index scan on p using F_Id  (cost=6.15 rows=59) (actual time=0.040..0.051 rows=59 loops=1)
                                    -> Filter: (d.Discount_Percentage > (select #2))  (cost=0.25 rows=0.3) (actual time=0.006..0.006 rows=0 loops=59)
                                        -> Index lookup on d using PRIMARY (R_Id=p.R_Id)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=0 loops=59)
                                        -> Select #2 (subquery in condition; run only once)
                                            -> Aggregate: avg(Deals.Discount_Percentage)  (cost=67.60 rows=1) (actual time=0.136..0.136 rows=1 loops=1)
                                                -> Table scan on Deals  (cost=34.30 rows=333) (actual time=0.039..0.098 rows=333 loops=1)
                                -> Single-row index lookup on r using PRIMARY (Id=p.R_Id)  (cost=0.26 rows=1) (actual time=0.004..0.004 rows=1 loops=8)
                            -> Single-row index lookup on f using PRIMARY (ID=p.F_Id)  (cost=0.26 rows=1) (actual time=0.005..0.005 rows=1 loops=8)
    -> Select #3 (subquery in condition; run only once)
        -> Aggregate: avg(Food.price)  (cost=2653.65 rows=1) (actual time=6.157..6.158 rows=1 loops=1)
            -> Table scan on Food  (cost=1354.95 rows=12987) (actual time=0.021..5.143 rows=13182 loops=1)
|
```

CREATE INDEX idx_deals_discount_percentage ON Deals(Discount_Percentage);

- Attempted to reduce cost to filter Deals.Discount_Percentage more efficiently in WHERE
    - Table scan on Food for Select #3: Cost 1354.95
    - Nested loop join (Outer): Cost 48.24
    - Nested loop join (Middle): Cost 41.36
    - Nested loop join (Inner): Cost 27.55
    - Subquery in WHERE on Deals: Cost 67.6



```
mysql> CREATE INDEX idx_deals_discount_percentage ON Deals(Discount_Percentage);
Query OK, 0 rows affected (0.08 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN ANALYZE
    -> SELECT
    ->     r.Name as Restaurant_Name,
    ->     AVG(d.Discount_Percentage) as Avg_Discount,
    ->     COUNT(DISTINCT f.food_category) as Menu_Categories,
    ->     AVG(f.price) as Avg_Price
    -> FROM Restaurant r
    -> JOIN Deals d ON r.Id = d.R_Id
    -> JOIN Product p ON r.Id = p.R_Id
    -> JOIN Food f ON p.F_Id = f.ID
    -> WHERE d.Discount_Percentage > (
    ->     SELECT AVG(Discount_Percentage) * .3
    ->     FROM Deals
    -> )
    -> GROUP BY r.Id, r.Name
    -> HAVING AVG(f.price) < (
    ->     SELECT AVG(price) * 2
    ->     FROM Food
    -> )
    -> ORDER BY Avg_Discount DESC;
```



```
| -> Sort: Avg_Discount DESC  (actual time=6.776..6.777 rows=8 loops=1)
    -> Filter: (avg(Food.price) < (select #3))  (actual time=6.713..6.749 rows=8 loops=1)
        -> Stream results  (actual time=0.539..0.572 rows=8 loops=1)
            -> Group aggregate: avg(Food.price), avg(Deals.Discount_Percentage), count(distinct Food.food_category), avg(Food.price)  (actual time=0.534..0.559 rows=8 loops=1)
                -> Sort: r.Id, r.`Name`  (actual time=0.527..0.530 rows=8 loops=1)
                    -> Stream results  (cost=40.57 rows=20) (actual time=0.243..0.507 rows=8 loops=1)
                        -> Nested loop inner join  (cost=40.57 rows=20) (actual time=0.240..0.500 rows=8 loops=1)
                            -> Nested loop inner join  (cost=33.68 rows=20) (actual time=0.230..0.457 rows=8 loops=1)
                                -> Nested loop inner join  (cost=26.80 rows=20) (actual time=0.220..0.421 rows=8 loops=1)
                                    -> Covering index scan on p using F_Id  (cost=6.15 rows=59) (actual time=0.040..0.051 rows=59 loops=1)
                                    -> Filter: (d.Discount_Percentage > (select #2))  (cost=0.25 rows=0.3) (actual time=0.006..0.006 rows=0 loops=59)
                                        -> Index lookup on d using PRIMARY (R_Id=p.R_Id)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=0 loops=59)
                                        -> Select #2 (subquery in condition; run only once)
                                            -> Aggregate: avg(Deals.Discount_Percentage)  (cost=67.60 rows=1) (actual time=0.136..0.136 rows=1 loops=1)
                                                -> Table scan on Deals  (cost=34.30 rows=333) (actual time=0.039..0.098 rows=333 loops=1)
                                -> Single-row index lookup on r using PRIMARY (Id=p.R_Id)  (cost=0.26 rows=1) (actual time=0.004..0.004 rows=1 loops=8)
                            -> Single-row index lookup on f using PRIMARY (ID=p.F_Id)  (cost=0.26 rows=1) (actual time=0.005..0.005 rows=1 loops=8)
    -> Select #3 (subquery in condition; run only once)
        -> Aggregate: avg(Food.price)  (cost=2653.65 rows=1) (actual time=6.157..6.158 rows=1 loops=1)
            -> Table scan on Food  (cost=1354.95 rows=12987) (actual time=0.021..5.143 rows=13182 loops=1)
|
```

These indexes did not reduce costs due to low selectivity of indexed cols. This can be due to the amount of discount, or food_category not being helpful because there is not much variance. The indexes are meant to reduce the number of rows scanned, when referencing a large portion of the table.

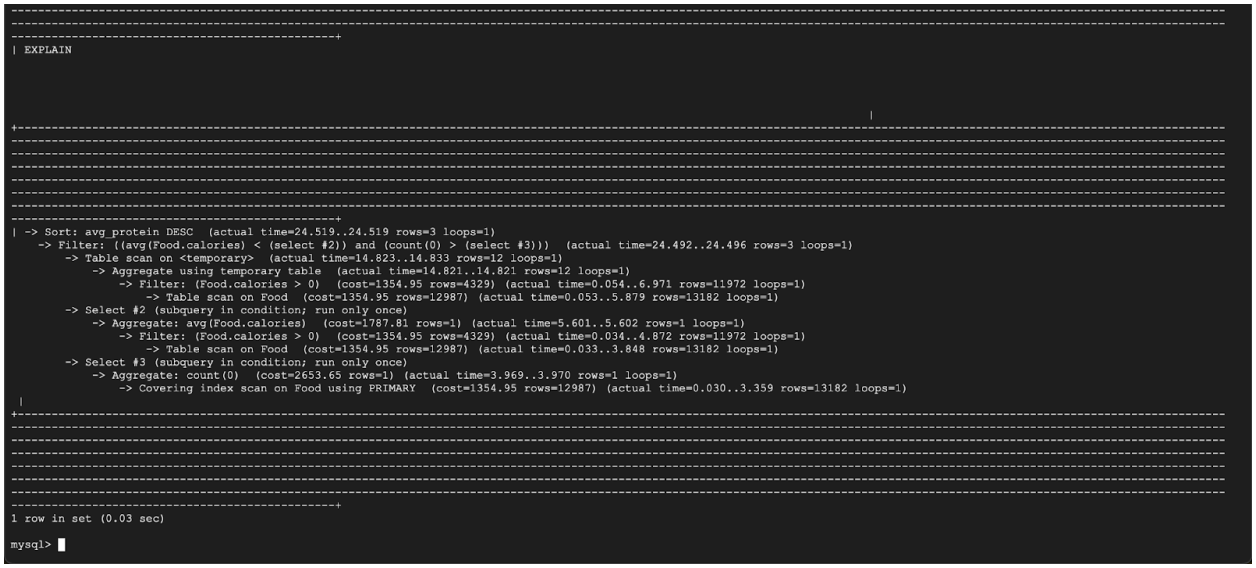
These indexes did not reduce cost to the simplicity of the query. Complex aggregations and groupings such as GROUP BY, COUNT(DISTINCT), and HAVING require the database to process a large amount of data to process calculations. Indexes on columns used in aggregations don't typically reduce the workload significantly because the optimizer still has to compute the aggregate for all relevant rows.

Another reason for these indexes to not work could be due to redundant access paths already existed through primary or foreign keys, making the new indexes unnecessary.

Advanced SQL Query 2 Indexing

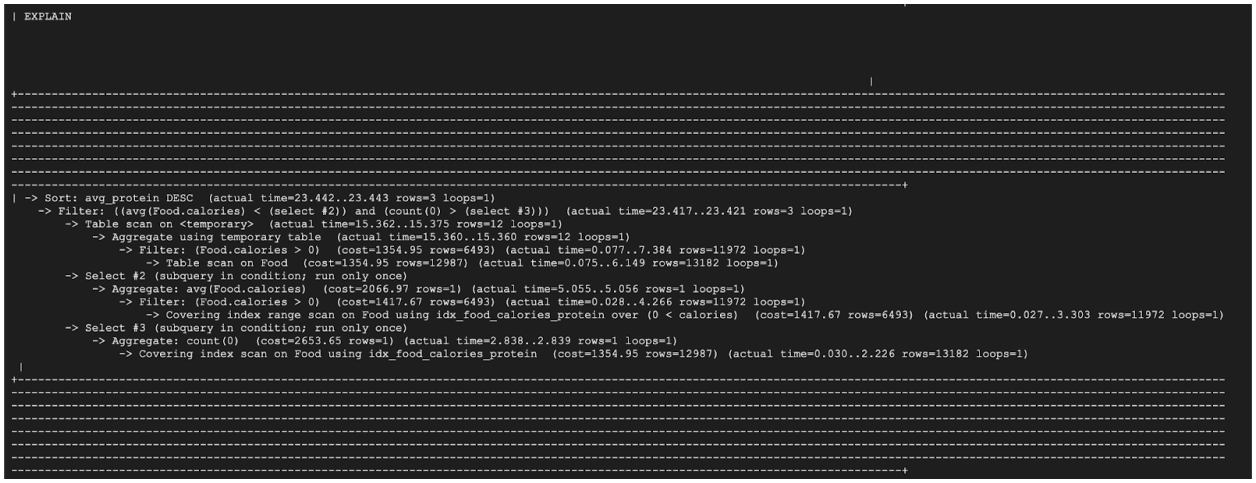
```
SELECT
   food_category,
   AVG(calories) as avg_calories,
   AVG(protein) as avg_protein,
   COUNT(*) as item_count,
   AVG(price) as avg_price
FROM Food
WHERE calories > 0
GROUP BY food_category
HAVING AVG(calories) < (
   SELECT AVG(calories) * 1.5
   FROM Food
   WHERE calories > 0
)
AND COUNT(*) > (
   SELECT COUNT(*)/20
   FROM Food
)
ORDER BY avg_protein DESC;
```

Default Index
Cost = 1354.95



Index 1

mysql> CREATE INDEX idx_food_calories_protein ON Food (calories, protein);
Query OK, 0 rows affected (0.35 sec)
Records: 0 Duplicates: 0 Warnings: 0

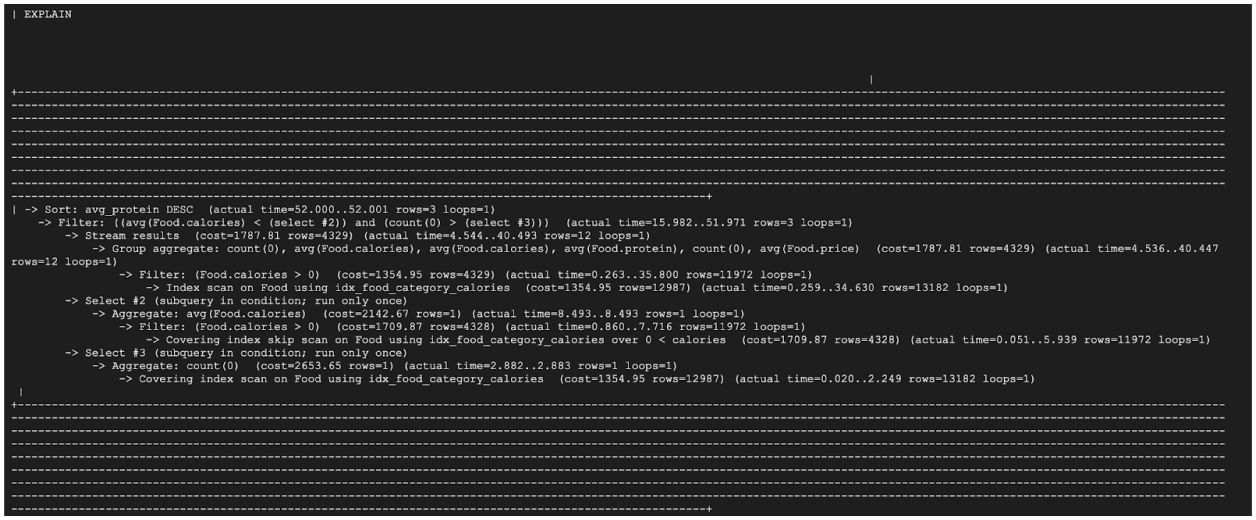


Result: Cost remained at 1354.95

I believe this index did not improve performance because the query filters based on calories and protein. However, this index wasn't applied to the food_category. Grouping and sorting was applied to food_category, so indexing on protein wasn't really necessary. After all, the query filters based on a condition (AVG(calories) < some value) that isn't directly covered by the index.

Index 2

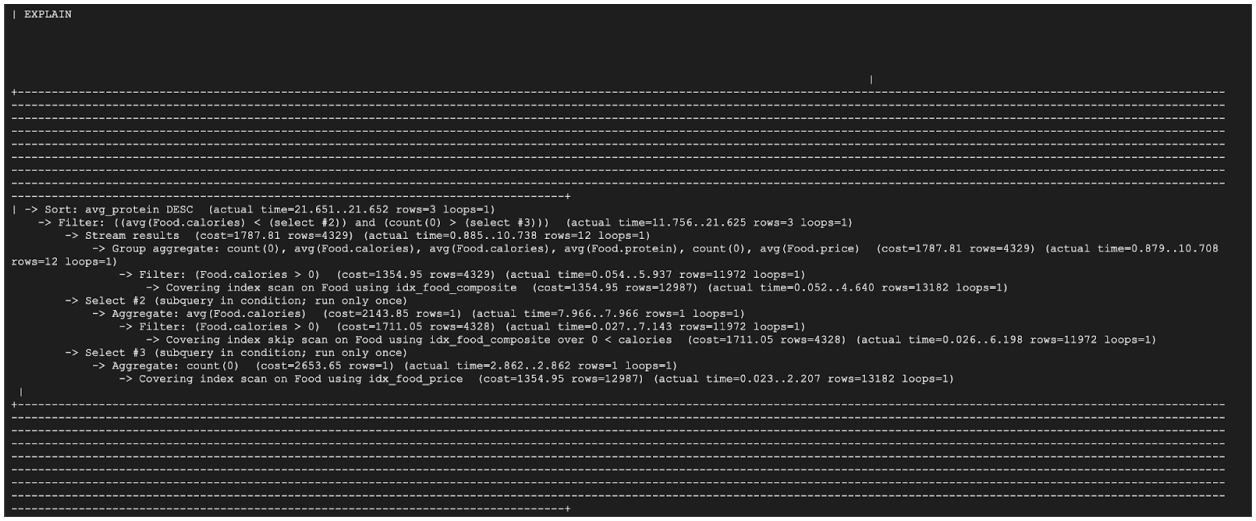
mysql> CREATE INDEX idx_food_category_calories ON Food (food_category, calories);
Query OK, 0 rows affected (0.43 sec)
Records: 0 Duplicates: 0 Warnings: 0



Result: Cost remained at 1354.95

Index 3

mysql> CREATE INDEX idx_food_composite ON Food (food_category, calories, protein, price);
Query OK, 0 rows affected (0.41 sec)
Records: 0 Duplicates: 0 Warnings: 0

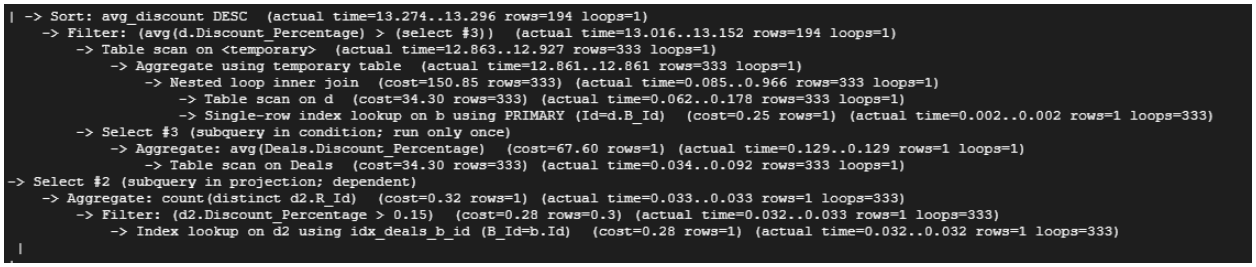


Result: Cost remained at 1354.95

While cost remained the same, the addition of the index has significantly improved the query's efficiency. The most notable improvement is the reduction in full table scans on the food table. The new execution plan now leverages index scans more effectively, which is generally much faster, especially for large datasets. This optimization is particularly evident in the subqueries, where index skip scans and covering index scans are employed to reduce the amount of data that needs to be processed. Although some specific operations, like the group aggregate and select #2 subquery, have seen an increase in cost and execution time, the overall performance gain from reduced table scans outweighs these minor increases.

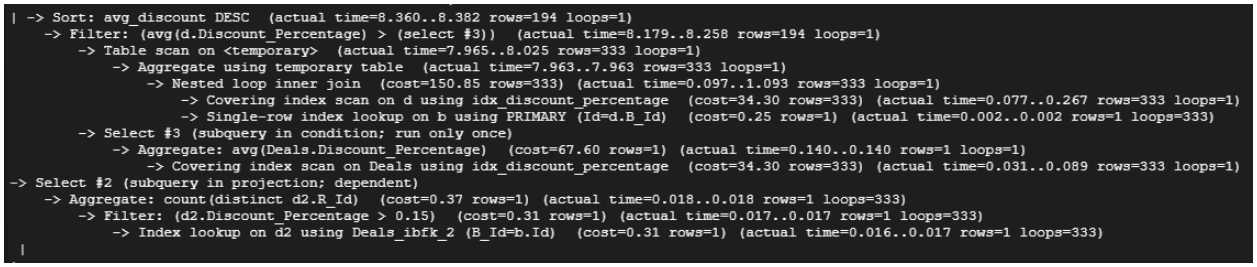
Advanced SQL Query 3 Indexing

1. CREATE INDEX idx_deals_b_id ON Deals(B_Id);



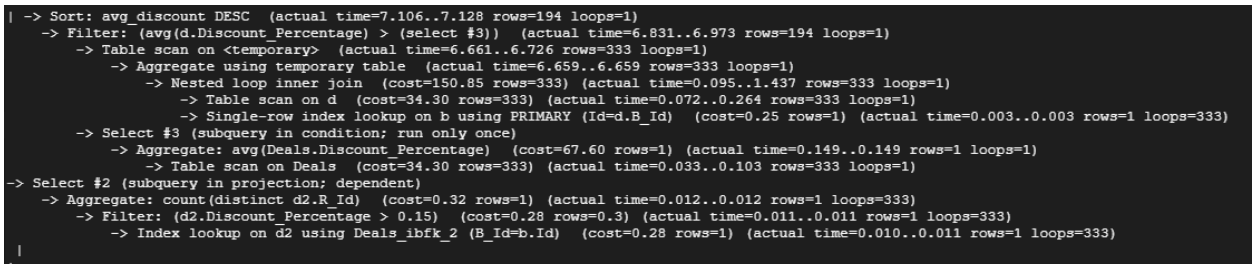
```
| -> Sort: avg_discount DESC  (actual time=13.274..13.296 rows=194 loops=1)
    -> Filter: (avg(d.Discount_Percentage) > (select #3))  (actual time=13.016..13.152 rows=194 loops=1)
        -> Table scan on <temporary>  (actual time=12.863..12.927 rows=333 loops=1)
            -> Aggregate using temporary table  (actual time=12.861..12.861 rows=333 loops=1)
                -> Nested loop inner join  (cost=150.85 rows=333) (actual time=0.085..0.966 rows=333 loops=1)
                    -> Table scan on d  (cost=34.30 rows=333) (actual time=0.062..0.178 rows=333 loops=1)
                    -> Single-row index lookup on b using PRIMARY (Id=d.B_Id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=333)
        -> Select #3 (subquery in condition; run only once)
            -> Aggregate: avg(Deals.Discount_Percentage)  (cost=67.60 rows=1) (actual time=0.129..0.129 rows=1 loops=1)
                -> Table scan on Deals  (cost=34.30 rows=333) (actual time=0.034..0.092 rows=333 loops=1)
-> Select #2 (subquery in projection; dependent)
    -> Aggregate: count(distinct d2.R_Id)  (cost=0.32 rows=1) (actual time=0.033..0.033 rows=1 loops=333)
        -> Filter: (d2.Discount_Percentage > 0.15)  (cost=0.28 rows=0.3) (actual time=0.032..0.033 rows=1 loops=333)
            -> Index lookup on d2 using idx_deals_b_id (B_Id=b.Id)  (cost=0.28 rows=1) (actual time=0.032..0.032 rows=1 loops=333)
 |
```

2. CREATE INDEX idx_discount_percentage ON Deals(Discount_Percentage);



```
| -> Sort: avg_discount DESC  (actual time=8.360..8.382 rows=194 loops=1)
    -> Filter: (avg(d.Discount_Percentage) > (select #3))  (actual time=8.179..8.258 rows=194 loops=1)
        -> Table scan on <temporary>  (actual time=7.965..8.025 rows=333 loops=1)
            -> Aggregate using temporary table  (actual time=7.963..7.963 rows=333 loops=1)
                -> Nested loop inner join  (cost=150.85 rows=333) (actual time=0.097..1.093 rows=333 loops=1)
                    -> Covering index scan on d using idx_discount_percentage  (cost=34.30 rows=333) (actual time=0.077..0.267 rows=333 loops=1)
                    -> Single-row index lookup on b using PRIMARY (Id=d.B_Id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=333)
        -> Select #3 (subquery in condition; run only once)
            -> Aggregate: avg(Deals.Discount_Percentage)  (cost=67.60 rows=1) (actual time=0.140..0.140 rows=1 loops=1)
                -> Covering index scan on Deals using idx_discount_percentage  (cost=34.30 rows=333) (actual time=0.031..0.089 rows=333 loops=1)
-> Select #2 (subquery in projection; dependent)
    -> Aggregate: count(distinct d2.R_Id)  (cost=0.37 rows=1) (actual time=0.018..0.018 rows=1 loops=333)
        -> Filter: (d2.Discount_Percentage > 0.15)  (cost=0.31 rows=1) (actual time=0.017..0.017 rows=1 loops=333)
            -> Index lookup on d2 using Deals_ibfk_2 (B_Id=b.Id)  (cost=0.31 rows=1) (actual time=0.016..0.017 rows=1 loops=333)
 |
```

3. CREATE INDEX idx_business_name ON Business(Name);



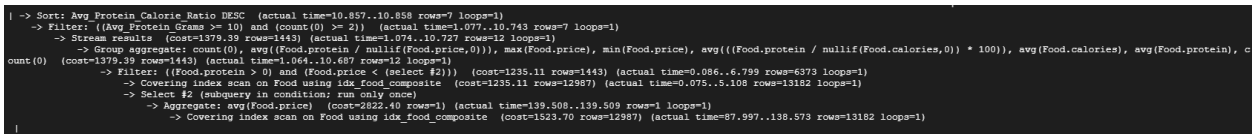
```
| -> Sort: avg_discount DESC  (actual time=7.106..7.128 rows=194 loops=1)
    -> Filter: (avg(d.Discount_Percentage) > (select #3))  (actual time=6.831..6.973 rows=194 loops=1)
        -> Table scan on <temporary>  (actual time=6.661..6.726 rows=333 loops=1)
            -> Aggregate using temporary table  (actual time=6.659..6.659 rows=333 loops=1)
                -> Nested loop inner join  (cost=150.85 rows=333) (actual time=0.095..1.437 rows=333 loops=1)
                    -> Table scan on d  (cost=34.30 rows=333) (actual time=0.072..0.264 rows=333 loops=1)
                    -> Single-row index lookup on b using PRIMARY (Id=d.B_Id)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=333)
        -> Select #3 (subquery in condition; run only once)
            -> Aggregate: avg(Deals.Discount_Percentage)  (cost=67.60 rows=1) (actual time=0.149..0.149 rows=1 loops=1)
                -> Table scan on Deals  (cost=34.30 rows=333) (actual time=0.033..0.103 rows=333 loops=1)
-> Select #2 (subquery in projection; dependent)
    -> Aggregate: count(distinct d2.R_Id)  (cost=0.32 rows=1) (actual time=0.012..0.012 rows=1 loops=333)
        -> Filter: (d2.Discount_Percentage > 0.15)  (cost=0.28 rows=0.3) (actual time=0.011..0.011 rows=1 loops=333)
            -> Index lookup on d2 using Deals_ibfk_2 (B_Id=b.Id)  (cost=0.28 rows=1) (actual time=0.010..0.011 rows=1 loops=333)
 |
```

JUSTIFICATION FOR QUERY #3:

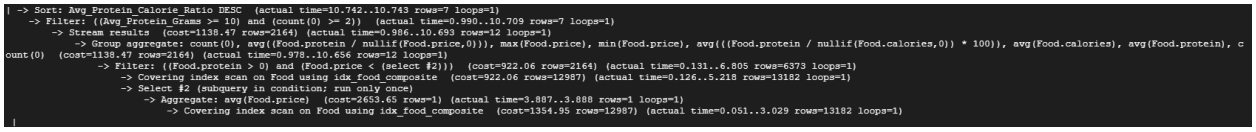
The cost for the first and third indexes (idx_deals_b_id and idx_business_name) didn't change because these indexes directly support the main join and grouping operations in the query, making data retrieval straightforward without adding extra processing. However, the second index on Discount_Percentage increased the query cost because it involves filtering on a specific discount threshold (> 0.15). This type of filtering requires the database to check more rows, which adds extra work. The additional steps needed to use the Discount_Percentage index for this filter caused the higher cost in the second query.

Advanced SQL Query 4 Indexing:

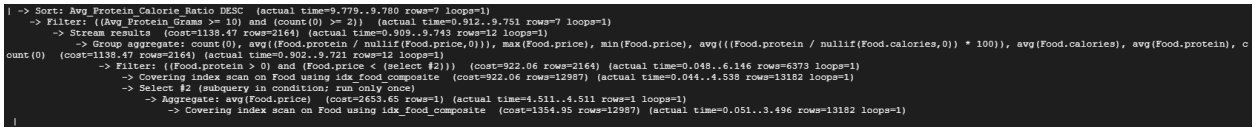
No Index:



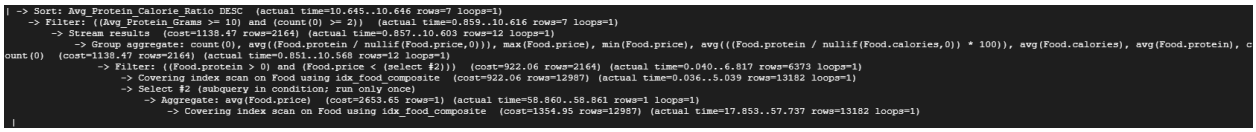
CREATE INDEX idx_food_protein ON Food(protein);



CREATE INDEX idx_food_protein_category ON Food(protein, food_category);



CREATE INDEX idx_food_covering ON Food(protein, food_category, calories, price);



All three tested indexes showed no improvement in the query cost (remaining at 1354.95), which is logical given the query's analytical nature. This query performs multiple aggregations (COUNT, AVG, MIN, MAX), calculates ratios, and requires grouping by food_category with a HAVING clause, meaning it needs to process most of the table's rows and columns regardless of indexing. When a query needs to access a large portion of the table and perform calculations across multiple columns, the query optimizer correctly determines that a full table scan is more efficient than using indexes, as the overhead of reading from indexes and then performing random access to get additional columns would be more expensive than simply reading the table sequentially. This is why even our covering index, which included all necessary columns, didn't improve performance.