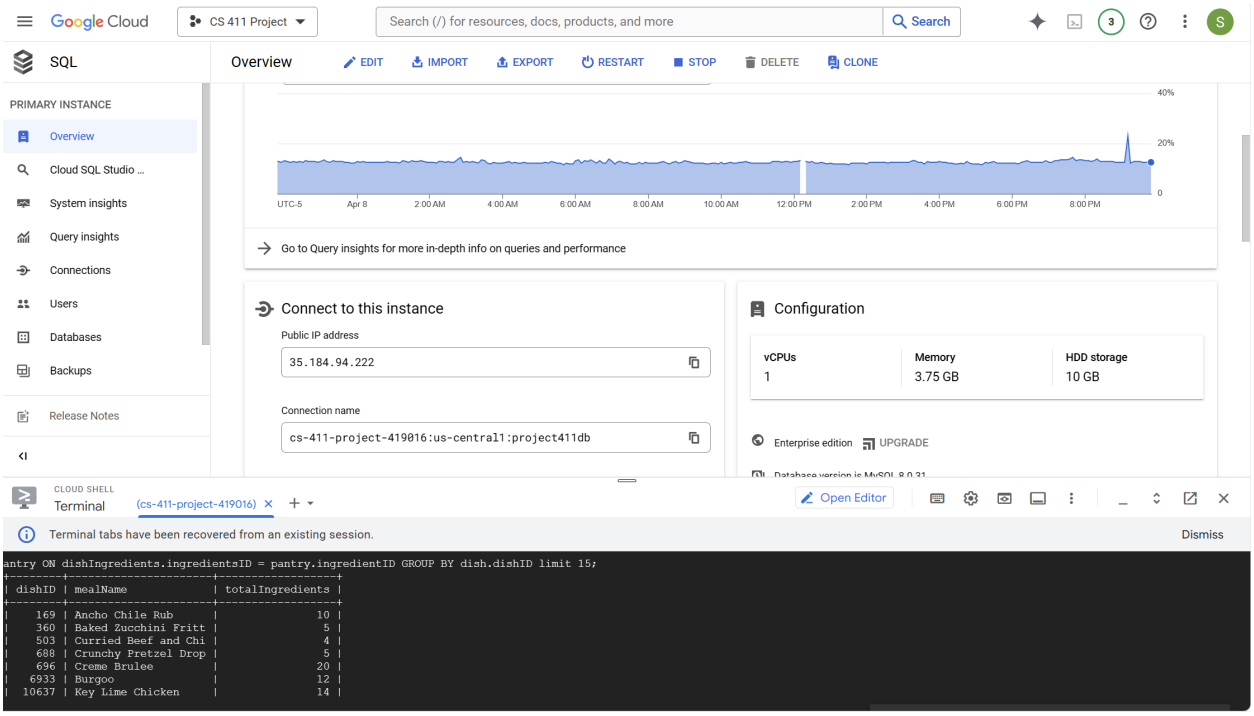
Stage 3: Database Implementation and Indexing (30%)

Samvit Dehdia, Divya Jain, Frank Lu, Kathryn Thompson

GCP Database Setup



DDL Commands to Create Tables:

```
CREATE TABLE users          (UID            INT NOT NULL AUT
O_INCREMENT,
                             username       VARCHAR(20),
                             password_hash  VARCHAR(20),
                             name           VARCHAR(50),
```

```
                                allergies      VARCHAR(100),
                                PRIMARY KEY    (UID));
```

```
CREATE TABLE userPantries    (pantryID       INT NOT NULL AUT
O_INCREMENT,
                              memberID       INT NOT NULL,
                              PRIMARY KEY    (pantryID),
                              FOREIGN KEY    (memberID)
REFERENCES users(UID));
```

```
CREATE TABLE pantry          (pantryID       INT NOT NULL,
                              ingredientID   INT NOT NULL,
                              quantity       FLOAT,
                              FOREIGN KEY    (pantryID)
REFERENCES userPantries,
                              FOREIGN KEY    (ingredientID)
REFERENCES ingredients(ingredientID));
```

```
CREATE TABLE mealPlan        (planID         INT NOT NULL,
                              dishID         INT NOT NULL,
                              notes          VARCHAR(255),
                              datePlanned    DATE,
                              timePlanned    TIME,
                              memberID       INT NOT NULL,
                              PRIMARY KEY    (planID)
                              FOREIGN KEY    (dishID)
REFERENCES dish,
                              FOREIGN KEY    (memberID)
REFERENCES users(UID),  ON DELETE CASCADE);
```

```
CREATE TABLE dishIngredients (dishID         INT NOT NULL,
                              ingredientsID  INT NOT NULL,
```

```
                                PRIMARY KEY    (dishID, ingredi
entsID)
                                FOREIGN KEY    (dishID)
REFERENCES dish(dishID),
                                FOREIGN KEY    (ingredientsID)
REFERENCES ingredients);
```

```
CREATE TABLE dish            (dishID         INT NOT NULL AUT
O_INCREMENT,
                                mealName       VARCHAR(20),
                                instructions   VARCHAR(255),
                                description    VARCHAR(255),
                                PRIMARY KEY    (dishID));
```

```
CREATE TABLE ingredients     (ingredientID   INT NOT NULL AUT
O_INCREMENT,
                                name           VARCHAR(20),
                                PRIMARY KEY    (ingredientID));
```

Tables:

```
mysql> select count(ingredientsID) from dishIngredients;
+----------------------+
| count(ingredientsID) |
+----------------------+
|                 9333 |
+----------------------+
1 row in set (0.00 sec)
```

```
mysql> select count(ingredientID) from ingredients;
+----------------------+
| count(ingredientID) |
+----------------------+
|                 1987 |
+----------------------+
1 row in set (0.00 sec)
```

```
mysql> select count(id) from dishes;
+-----------+
| count(id) |
+-----------+
|    494955 |
+-----------+
1 row in set (0.26 sec)
```

Advanced Queries:

Query 1:

The query below finds all the dishes that the user specified in the where statement of the subquery is not allergic to.

```
SELECT dishID, mealName
FROM dish d1 NATURAL JOIN dishIngredients
WHERE NOT EXISTS
    ( SELECT 1
        FROM users JOIN ingredients ON allergies =
            ingredientName JOIN dishIngredients
        ON ingredientsID = ingredientID
        WHERE di.dishID = d1.dishID and user.UID = 1)
```

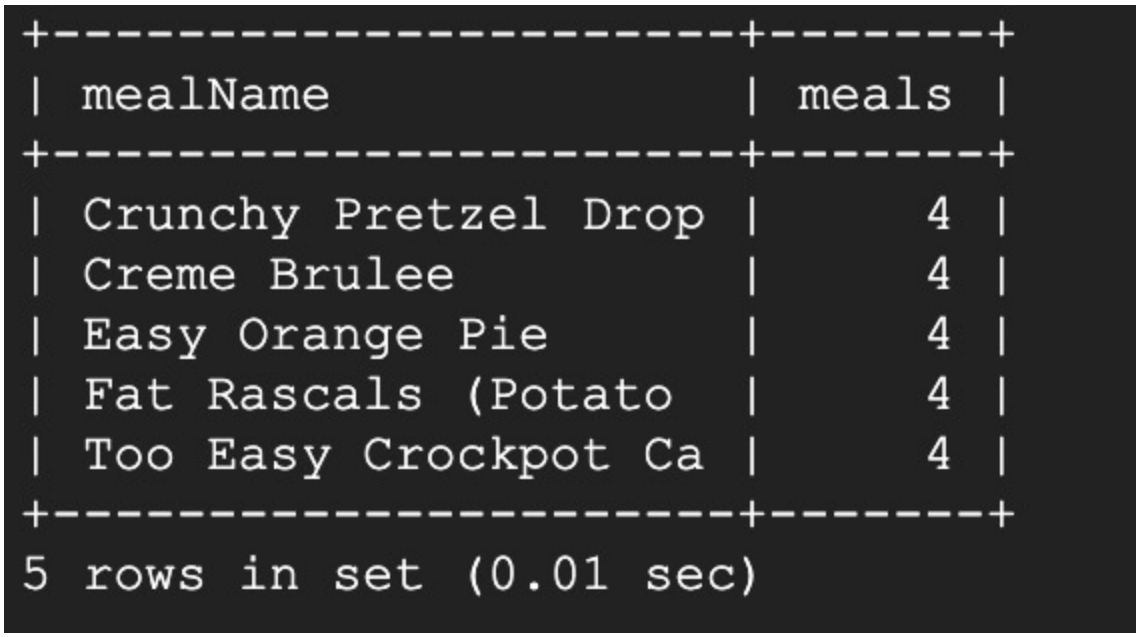
```
GROUP BY dishID, mealName;
LIMIT 15;
```

```
mysql> select dishID, mealName
    -> from dish d1 natural join dishIngredients
    -> where not exists
    ->   ( select 1
    ->     from users join ingredients on allergies = ingredientName join dishIngredients di on ingredientsID = ingredientID
    ->     where di.dishID = d1.dishID)
    -> group by dishID, mealName
    -> limit 15;
+--------+----------------------+
| dishID | mealName             |
+--------+----------------------+
|      0 | name                 |
|    169 | Ancho Chile Rub      |
|    360 | Baked Zucchini Fritt |
|    503 | Curried Beef and Chi |
|   6933 | Burgoo               |
|   7111 | Gingersnap Gravy     |
|  10637 | Key Lime Chicken     |
|  11967 | Italian Pot Roast    |
|  12978 | Easy Pickled Eggs    |
|  14813 | Pork Chop And New Po |
|  17088 | Five Hour Beef Stew  |
|  19485 | Italian-Style Spinac |
|  21132 | Baked Cranberry Oatm |
|  23259 | Dublin Pork Sparerib |
|  23809 | Easy Orange Pie      |
+--------+----------------------+
15 rows in set (0.17 sec)
```

Query 2:

This query finds the most popular meals that are in the meal plans. It gets the name of these meals and outputs them according to the number of times it appears in the users' meal plans.

```
SELECT mealName, count(*) AS meals
FROM mealplan NATURAL JOIN dish
GROUP BY mealName
ORDER BY meals DESC;
```



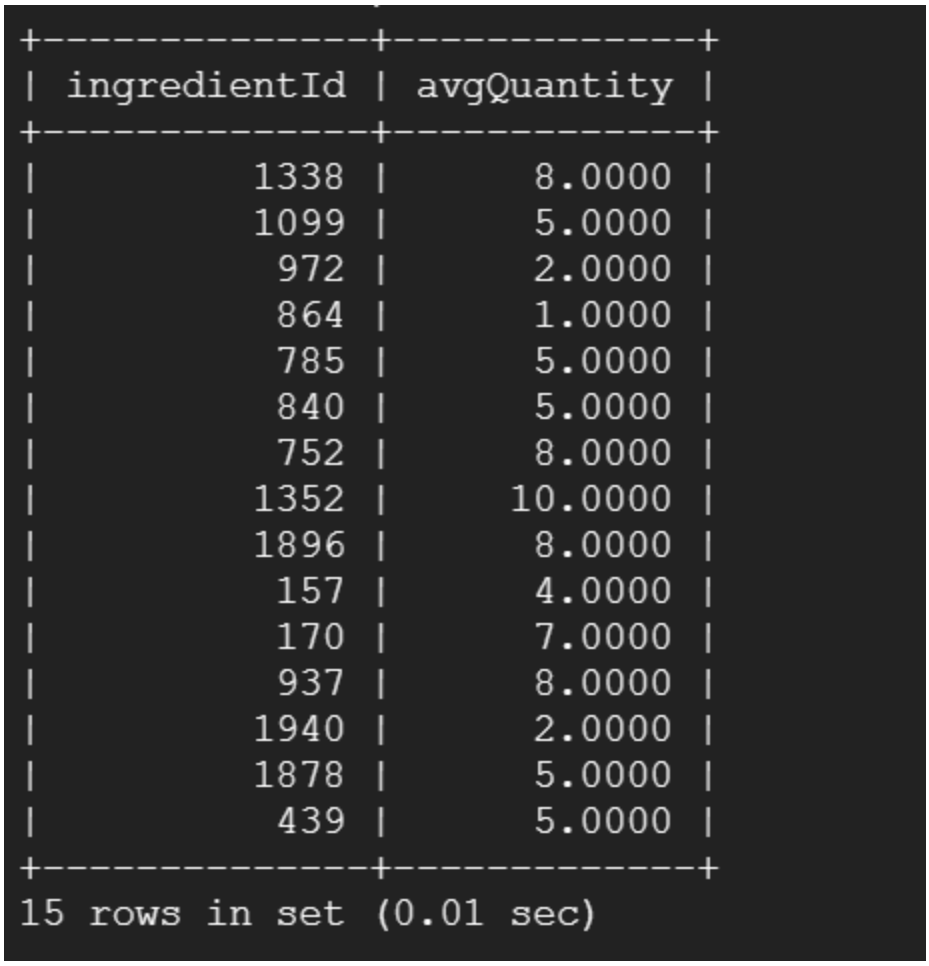
```
+-----------------------+--------+
| mealName              | meals  |
+-----------------------+--------+
| Crunchy Pretzel Drop  |      4 |
| Creme Brulee          |      4 |
| Easy Orange Pie       |      4 |
| Fat Rascals (Potato   |      4 |
| Too Easy Crockpot Ca  |      4 |
+-----------------------+--------+
5 rows in set (0.01 sec)
```

Query 3:

Find the average quantity of each ingredient in all user pantries

This query returns the average quantity of each ingredient present in users' pantries.

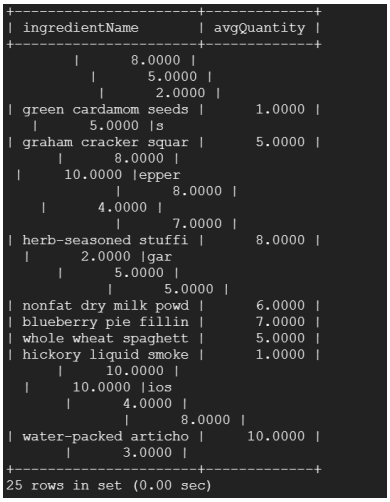
```
SELECT ingredients.ingredientId,
    AVG(pantry.quantity) as avgQuantity
FROM pantry
JOIN ingredients ON pantry.ingredientID =
    ingredients.ingredientID
GROUP BY ingredients.ingredientID
LIMIT 15;
```



ingredientId	avgQuantity
1338	8.0000
1099	5.0000
972	2.0000
864	1.0000
785	5.0000
840	5.0000
752	8.0000
1352	10.0000
1896	8.0000
157	4.0000
170	7.0000
937	8.0000
1940	2.0000
1878	5.0000
439	5.0000

15 rows in set (0.01 sec)

We are returning the ingredientId because there is a formatting issue with the ingredientName, but the values for the ingredient names are returned correctly.



ingredientName	avgQuantity
green cardamom seeds	1.0000
graham cracker squar	5.0000
herb-seasoned stuffi	8.0000
nonfat dry milk powd	6.0000
blueberry pie fillin	7.0000
whole wheat spaghett	5.0000
hickory liquid smoke	1.0000
water-packed articho	10.0000

25 rows in set (0.00 sec)

Query 4:

Based on the allergies listed by the user, this query will return the meals that they cannot make as they will be allergic to them.

```
SELECT name, mealName
FROM users JOIN ingredients ON
    users.allergies = ingredients.ingredientName
JOIN dishIngredients ON ingredients.ingredientID
    = dishIngredients.ingredientsID
JOIN dish ON dishIngredients.dishID = dish.dishID
GROUP BY name, mealname
ORDER BY name
LIMIT 15;
```

name	mealName
Alexander Zuniga	"the Deck's&quo
Alexander Zuniga	Abc Bread
Alexander Zuniga	Asparagus Omelette
Alexander Zuniga	Banana Oatmeal Cooki
Alexander Zuniga	Banana Split Cake
Alexander Zuniga	Beef & Cheddar S
Alexander Zuniga	Bourbon Bread Puddin
Alexander Zuniga	Bourbon Street Fritt
Alexander Zuniga	Choc-Cherry Muffins
Alexander Zuniga	Chocolate Griddle Ca
Alexander Zuniga	Cinnamon Chocolate C
Alexander Zuniga	Classic Cheesecake
Alexander Zuniga	Coconut-Cream Bread
Alexander Zuniga	Creme Brulee
Alexander Zuniga	Ct's Baked Egg Patti

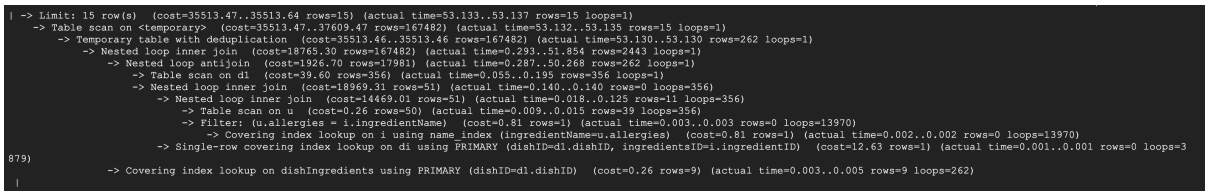
Indexing

Indexing for Query 1:

With no index:

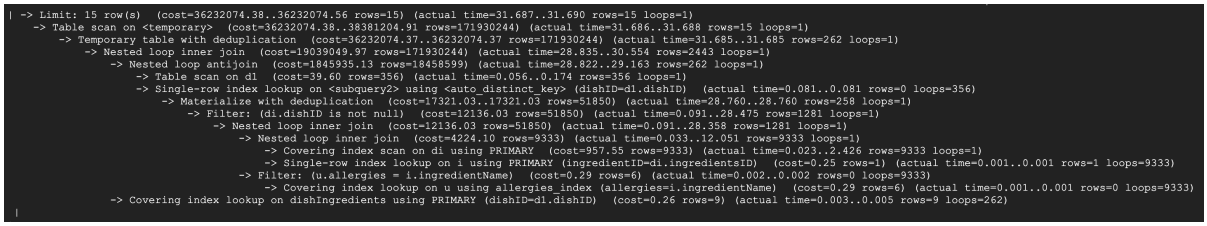
```
| -> Limit: 15 row(s)  (cost=325655.37..325655.54 rows=15) (actual time=232.926..232.930 rows=15 loops=1)
    -> Table scan on <temporary>  (cost=325655.37..344961.39 rows=1544284) (actual time=232.925..232.928 rows=15 loops=1)
        -> Temporary table with deduplication  (cost=325655.36..325655.36 rows=1544284) (actual time=232.922..232.922 rows=262 loops=1)
            -> Nested loop inner join  (cost=171226.99 rows=1544284) (actual time=0.207..231.317 rows=2443 loops=1)
                -> Nested loop antijoin  (cost=16708.18 rows=165796) (actual time=0.204..229.755 rows=262 loops=1)
                    -> Table scan on d1  (cost=39.60 rows=356) (actual time=0.051..0.233 rows=356 loops=1)
                    -> Nested loop inner join  (cost=5444.51 rows=466) (actual time=0.644..0.644 rows=0 loops=356)
                        -> Nested loop inner join  (cost=4568.96 rows=466) (actual time=0.014..0.215 rows=367 loops=356)
                            -> Table scan on u  (cost=0.26 rows=50) (actual time=0.010..0.017 rows=39 loops=356)
                            -> Covering index lookup on di using PRIMARY (dishID=d1.dishID)  (cost=0.25 rows=9) (actual time=0.003..0.004 rows=9 loops=13970)
                        -> Filter: (u.allergies = i.ingredientName)  (cost=2.33 rows=1) (actual time=0.001..0.001 rows=0 loops=130654)
                            -> Single-row index lookup on i using PRIMARY (ingredientID=di.ingredientsID)  (cost=2.33 rows=1) (actual time=0.001..0.001 rows=1 loops=130654)
                -> Covering index lookup on dishIngredients using PRIMARY (dishID=d1.dishID)  (cost=0.26 rows=9) (actual time=0.003..0.005 rows=9 loops=262)
 |
```

With index on ingredients(ingredientName):



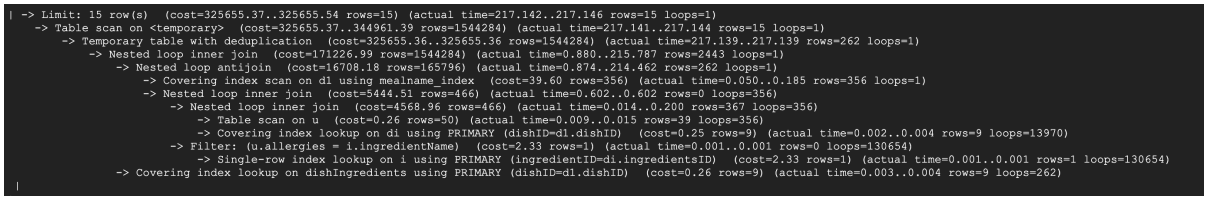
Our hypothesis with adding this index was that having an index on ingredientName as a part of the ingredients table would allow for faster table look ups when joining ingredients table with other tables such as users. This would result in a faster retrieval of rows that are relevant. This hypothesis proved to be true as we were able to significantly lower the cost, improving performance.

With index on users(allergies):



We had a similar hypothesis for placing an index on allergies, however, we were proved to be wrong as the cost increased, decreasing performance. One such reason could be the limited uniqueness in our allergies column which could meaning that the number of rows scanned during execution is not significantly reduced which would cause increased costs.

With index on dish(mealname):



We were hoping for an optimized query by placing an index on mealname, but the cost did not change. This could mean that the dish table may not heavily rely on filtering or sorting of meal names meaning there would not be a noticeable improvement on costs.

In conclusion, the index we would use for query one is ingredientName on ingredients relation.

Indexing for Query 2:

With no index:

```
| -> Limit: 15 row(s)  (actual time=0.992..0.995 rows=15 loops=1)
    -> Table scan on <temporary>  (actual time=0.990..0.992 rows=15 loops=1)
        -> Aggregate using temporary table  (actual time=0.989..0.989 rows=300 loops=1)
            -> Nested loop inner join  (cost=135.25 rows=300) (actual time=0.050..0.741 rows=300 loops=1)
                -> Table scan on pantry  (cost=30.25 rows=300) (actual time=0.033..0.208 rows=300 loops=1)
                -> Single-row index lookup on ingredients using PRIMARY (ingredientID=pantry.ingredientID)  (cost=0.25 rows=1) (actual time=0.001..0.002 rows=1 loops=300)
 |
```

With index on ingredients(ingredientName):

```
| -> Limit: 15 row(s)  (actual time=3.747..3.750 rows=15 loops=1)
    -> Table scan on <temporary>  (actual time=3.746..3.748 rows=15 loops=1)
        -> Aggregate using temporary table  (actual time=3.743..3.743 rows=300 loops=1)
            -> Nested loop inner join  (cost=895.90 rows=1987) (actual time=0.057..3.513 rows=300 loops=1)
                -> Covering index scan on ingredients using ingredients_index  (cost=200.45 rows=1987) (actual time=0.031..0.592 rows=1987 loops=1)
                -> Index lookup on pantry using ingredientID (ingredientID=ingredients.ingredientID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0 loops=1987)
 |
```

The cost of adding this index increased leading to decreased query optimization. Potential reasons of this could include unnecessary indexing which could lead to decreased improvement.

With index on quantity_index on pantry(quantity):

```
| -> Limit: 15 row(s)  (actual time=0.940..0.943 rows=15 loops=1)
    -> Table scan on <temporary>  (actual time=0.938..0.941 rows=15 loops=1)
        -> Aggregate using temporary table  (actual time=0.937..0.937 rows=300 loops=1)
            -> Nested loop inner join  (cost=135.25 rows=300) (actual time=0.040..0.692 rows=300 loops=1)
                -> Table scan on pantry  (cost=30.25 rows=300) (actual time=0.026..0.201 rows=300 loops=1)
                -> Single-row index lookup on ingredients using PRIMARY (ingredientID=pantry.ingredientID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=300)
 |
```

The cost of adding an index on quantity in pantry stayed the same which could potentially be because the pantry table does not heavily rely on filtering or sorting of quantity so there would not be a noticeable improvement on costs.

With index on pantry(ingredientID):

```
| -> Limit: 15 row(s)  (actual time=4.587..4.594 rows=15 loops=1)
    -> Table scan on <temporary>  (actual time=4.585..4.590 rows=15 loops=1)
        -> Aggregate using temporary table  (actual time=4.583..4.583 rows=300 loops=1)
            -> Nested loop inner join  (cost=135.25 rows=300) (actual time=0.053..4.307 rows=300 loops=1)
                -> Table scan on pantry  (cost=30.25 rows=300) (actual time=0.026..0.249 rows=300 loops=1)
                -> Single-row index lookup on ingredients using iiid_index (ingredientID=pantry.ingredientID)  (cost=0.25 rows=1) (actual time=0.013..0.013 rows=1 loops=300)
 |
```

The cost of adding an index on ingredientID in pantry stayed the same which could potentially be because the pantry table does not heavily rely on filtering or sorting of ingredientIDS so there would not be a noticeable improvement on costs.

In conclusion, we would be best off not indexing any columns for this query as we don’t see a change or improvement in costs.

Indexing for Query 3:

With no index:

```
| -> Sort: meals DESC  (actual time=0.117..0.118 rows=5 loops=1)
    -> Table scan on <temporary>  (actual time=0.105..0.106 rows=5 loops=1)
        -> Aggregate using temporary table  (actual time=0.103..0.103 rows=5 loops=1)
            -> Nested loop inner join  (cost=9.25 rows=20) (actual time=0.046..0.068 rows=20 loops=1)
                -> Covering index scan on mealplan using dishID  (cost=2.25 rows=20) (actual time=0.030..0.034 rows=20 loops=1)
                -> Single-row index lookup on dish using PRIMARY (dishID=mealplan.dishID)  (cost=0.26 rows=1) (actual time=0.001..0.001 rows=1 loops=20)
 |
```

With index on mealplan(dishID):

```
| -> Sort: meals DESC  (actual time=0.144..0.145 rows=5 loops=1)
    -> Table scan on <temporary>  (actual time=0.127..0.129 rows=5 loops=1)
        -> Aggregate using temporary table  (actual time=0.126..0.126 rows=5 loops=1)
            -> Nested loop inner join  (cost=9.25 rows=20) (actual time=0.061..0.083 rows=20 loops=1)
                -> Covering index scan on mealplan using dish_index  (cost=2.25 rows=20) (actual time=0.042..0.047 rows=20 loops=1)
                -> Single-row index lookup on dish using PRIMARY (dishID=mealplan.dishID)  (cost=0.26 rows=1) (actual time=0.001..0.002 rows=1 loops=20)
 |
```

Since in this query, we are joining the meal plan table and the dish table on the dishID attribute, we thought indexing this attribute would lead to lower cost executing the query. However, although we are now running an index scan with the new index, the cost is exactly the same. This is likely due to the fact that we are not directly accessing this attribute to compare.

With index on dish(mealname):

```
| -> Sort: meals DESC  (actual time=0.442..0.443 rows=5 loops=1)
    -> Table scan on <temporary>  (actual time=0.418..0.419 rows=5 loops=1)
        -> Aggregate using temporary table  (actual time=0.416..0.416 rows=5 loops=1)
            -> Inner hash join (dish.dishID = mealplan.dishID)  (cost=718.25 rows=20) (actual time=0.145..0.351 rows=20 loops=1)
                -> Covering index scan on dish using mealName_index  (cost=0.21 rows=356) (actual time=0.025..0.196 rows=356 loops=1)
                -> Hash
                    -> Table scan on mealplan  (cost=2.25 rows=20) (actual time=0.081..0.086 rows=20 loops=1)
 |
```

Since we are selecting and grouping by the meal name, we thought that adding index on the meal name would help decrease the cost. However, cost is increased significantly. An inner hash join is used instead of a nested loop inner join, which might somehow be the reason that the cost increases.

With index on dish(mealname(5)):

```
| -> Sort: meals DESC  (actual time=0.283..0.284 rows=5 loops=1)
    -> Table scan on <temporary>  (actual time=0.271..0.272 rows=5 loops=1)
        -> Aggregate using temporary table  (actual time=0.270..0.270 rows=5 loops=1)
            -> Inner hash join (dish.dishID = mealplan.dishID)  (cost=718.25 rows=20) (actual time=0.081..0.233 rows=20 loops=1)
                -> Table scan on dish  (cost=0.21 rows=356) (actual time=0.043..0.165 rows=356 loops=1)
                -> Hash
                    -> Table scan on mealplan  (cost=2.25 rows=20) (actual time=0.019..0.023 rows=20 loops=1)
 |
```

Since most of the meals can be distinguished by the first few letters, we thought that adding an index on the first 5 letters of the meal name might decrease the cost since we don't need to read too many letters when comparing. However, like the previous index, cost is increased significantly and inner hash join is used instead of nested loop inner join.

In conclusion, based on the costs of running the query with these 4 indices, I think the most optimal one should be the one with the default indexing.

Indexing for Query 4:

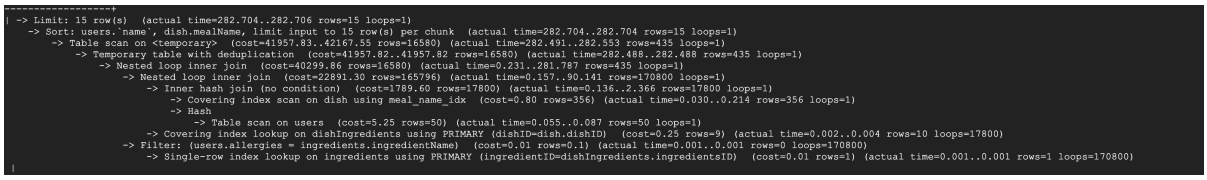
With no indexing:

```
---------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=280.564..280.566 rows=15 loops=1)
    -> Sort: users.`name`, dish.mealName, limit input to 15 row(s) per chunk  (actual time=280.563..280.564 rows=15 loops=1)
        -> Table scan on <temporary>  (cost=41957.83..42167.55 rows=16580) (actual time=280.334..280.401 rows=435 loops=1)
            -> Temporary table with deduplication  (cost=41957.82..41957.82 rows=16580) (actual time=280.330..280.330 rows=435 loops=1)
                -> Nested loop inner join  (cost=40299.86 rows=16580) (actual time=3.363..279.564 rows=435 loops=1)
                    -> Nested loop inner join  (cost=22891.30 rows=165796) (actual time=0.252..90.413 rows=170800 loops=1)
                        -> Inner hash join (no condition)  (cost=1789.60 rows=17800) (actual time=0.231..2.574 rows=17800 loops=1)
                            -> Table scan on dish  (cost=0.80 rows=356) (actual time=0.096..0.367 rows=356 loops=1)
                            -> Hash
                                -> Table scan on users  (cost=5.25 rows=50) (actual time=0.091..0.107 rows=50 loops=1)
                        -> Covering index lookup on dishIngredients using PRIMARY (dishID=dish.dishID)  (cost=0.25 rows=9) (actual time=0.002..0.004 rows=10 loops=17800)
                    -> Filter: (users.allergies = ingredients.ingredientName)  (cost=0.01 rows=0.1) (actual time=0.001..0.001 rows=0 loops=170800)
                        -> Single-row index lookup on ingredients using PRIMARY (ingredientID=dishIngredients.ingredientsID)  (cost=0.01 rows=1) (actual time=0.001..0.001 rows=1 loops=170800)
 |
```

Initial cost of query before indexing : 41957.83

```
mysql> CREATE INDEX meal_name_idx ON dish(mealname);
Query OK, 0 rows affected (0.06 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Creating index called meal_name_idx on Meal Name as a part of Dish Table.



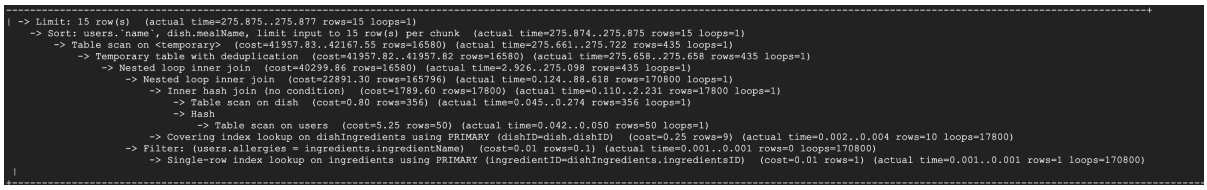
We initially believed that adding this index could potentially improve query performance by producing quicker lookups and data retrieval, to filter and sort through specific meal names. However, as seen by the stagnant change in cost, there was no significant improvement and the cost stayed the same.

```
mysql> DROP INDEX meal_name_idx ON dish;
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Drop meal_name_idx to test out other indices.

```
mysql> CREATE INDEX name_idx ON users(name);
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Creating index called name_idx on name as a part of users.



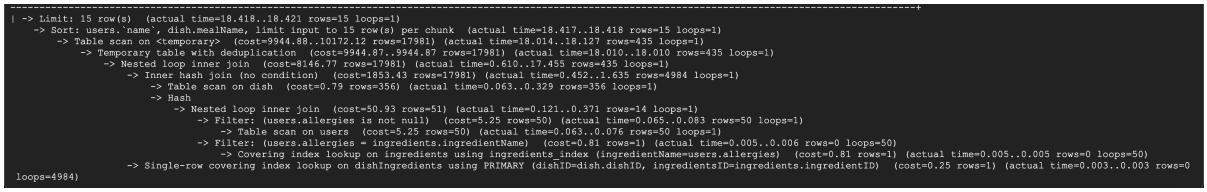
Similarly to our initial hypothesis with meal name index, we assumed that adding an index to name on users table would enhance query performance by allowing us to filter based on user names and improve data retrieval. However, the cost for this index was also stagnant and did not significantly change as the cost stayed the same.

```
mysql> DROP INDEX name_idx ON users;
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Drop name_idx to test out other indices.

Creating index called ingredientID_idx on ingredientID as a part of ingredients.

```
mysql> CREATE INDEX ingredientID_idx ON ingredients(ingredientID);
Query OK, 0 rows affected (0.07 sec)
Records: 0  Duplicates: 0  Warnings: 0
```



For this query we hypothesized that indexing a foreign key can make for efficient data retrieval and reduce the time for table scans when joining them with other tables. Our hypothesis proved to be true as the cost after adding this index decreases leading to significant improvement.

In conclusion for query four, we would use the index on ingredientID as a part of the ingredients relation.