

Stage 3: Database Implementation and Indexing (30%)

Samvit Dehdia, Divya Jain, Frank Lu, Kathryn Thompson

GCP Database Setup

The screenshot displays the Google Cloud SQL console for a MySQL instance. The instance is named 'cs-411-project-419016:us-central1:project411db'. The configuration shows 1 vCPU, 3.75 GB memory, and 10 GB HDD storage. The instance is in the 'Enterprise edition' and has an 'UPGRADE' button. The 'Connect to this instance' section shows the public IP address '35.184.94.222' and the connection name 'cs-411-project-419016:us-central1:project411db'. The 'Overview' section shows a graph of CPU usage over time, with a peak of 40%.

```
entry ON dishIngredients.ingredientsID = pantry.ingredientsID GROUP BY dish.dishID limit 15;
+-----+-----+-----+
| dishID | mealName | totalIngredients |
+-----+-----+-----+
| 169 | Ancho Chile Rub | 10 |
| 360 | Baked Zucchini Fritt | 5 |
| 503 | Curried Beef and Chi | 4 |
| 688 | Crunchy Pretzel Drop | 5 |
| 696 | Creme Brulee | 20 |
| 6933 | Burgoo | 12 |
| 10637 | Key Lime Chicken | 14 |
+-----+-----+-----+
```

DDL Commands to Create Tables:

| | | |
|--------------------|---------------|------------------|
| CREATE TABLE users | (UID | INT NOT NULL AUT |
| 0_INCREMENT, | username | VARCHAR(20), |
| | password_hash | VARCHAR(20), |
| | name | VARCHAR(50), |

| | | |
|--|--|--|
| | allergies PRIMARY KEY | VARCHAR(100), (UID)); |
| CREATE TABLE userPantries O_INCREMENT, | (pantryID memberID PRIMARY KEY FOREIGN KEY | INT NOT NULL AUT INT NOT NULL, (pantryID), (memberID) |
| REFERENCES users(UID)); | | |
| CREATE TABLE pantry | (pantryID ingredientID quantity FOREIGN KEY | INT NOT NULL, INT NOT NULL, FLOAT, (pantryID) |
| REFERENCES userPantries, | FOREIGN KEY | (ingredientID) |
| REFERENCES ingredients(ingredientID)); | | |
| CREATE TABLE mealPlan | (planID dishID notes datePlanned timePlanned memberID PRIMARY KEY FOREIGN KEY | INT NOT NULL, INT NOT NULL, VARCHAR(255), DATE, TIME, INT NOT NULL, (planID) (dishID) |
| REFERENCES dish, | FOREIGN KEY | (memberID) |
| REFERENCES users(UID), ON DELETE CASCADE); | | |
| CREATE TABLE dishIngredients | (dishID ingredientsID | INT NOT NULL, INT NOT NULL, |

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

| | | |
|-------------------|--------------|------------------|
| 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 |
|-----------------------|-------------|
| | 8.0000 |
| | 5.0000 |
| | 2.0000 |
| green cardamom seeds | 1.0000 |
| | 5.0000 |
| graham cracker squar | 5.0000 |
| | 8.0000 |
| | 10.0000 |
| | 8.0000 |
| | 4.0000 |
| | 7.0000 |
| herb-seasoned stuffi | 8.0000 |
| | 2.0000 |
| | 5.0000 |
| | 5.0000 |
| nonfat dry milk powd | 6.0000 |
| blueberry pie fillin | 7.0000 |
| whole wheat spaghetti | 5.0000 |
| hickory liquid smoke | 1.0000 |
| | 10.0000 |
| | 10.0000 |
| | 4.0000 |
| | 8.0000 |
| water-packed articho | 10.0000 |
| | 3.0000 |

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'squo |
| 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 anti join (cost=16708.18 rows=165796) (actual time=0.204..229.755 rows=262 loops=1)
|         -> Table scan on di (cost=39.60 rows=356) (actual time=0.051..0.233 rows=356 loops=1)
|         -> Nested loop inner join (cost=8444.51 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=di.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):

```

1 -> Limit: 15 row(s) (cost=35513.47..35513.64 rows=15) (actual time=53.133..53.137 rows=15 loops=1)
   -> Table scan on <temporary> (cost=35513.47..37609.47 rows=167482) (actual time=53.132..53.135 rows=15 loops=1)
     -> Temporary table with deduplication (cost=35513.46..35513.46 rows=167482) (actual time=53.130..53.130 rows=262 loops=1)
       -> Nested loop inner join (cost=18745.30 rows=167482) (actual time=0.293..51.854 rows=2443 loops=1)
         -> Nested loop antijoin (cost=1926.70 rows=17981) (actual time=0.287..50.268 rows=262 loops=1)
           -> Table scan on dl (cost=39.60 rows=356) (actual time=0.055..0.195 rows=356 loops=1)
             -> Nested loop inner join (cost=18969.31 rows=51) (actual time=0.140..0.140 rows=0 loops=356)
               -> Nested loop inner join (cost=14409.01 rows=51) (actual time=0.018..0.125 rows=11 loops=356)
                 -> Table scan on u (cost=0.26 rows=50) (actual time=0.009..0.015 rows=39 loops=356)
                   -> Filter: (u.allergies = i.ingredientName) (cost=0.81 rows=1) (actual time=0.003..0.003 rows=0 loops=13970)
                     -> Covering index lookup on i using name_index (ingredientName=u.allergies) (cost=0.81 rows=1) (actual time=0.002..0.002 rows=0 loops=13970)
                       -> Single-row covering index lookup on di using PRIMARY (dishID=dl.dishID, ingredientsID=i.ingredientID) (cost=12.63 rows=1) (actual time=0.001..0.001 rows=0 loops=3879)
                         -> Covering index lookup on dishIngredients using PRIMARY (dishID=dl.dishID) (cost=0.26 rows=9) (actual time=0.003..0.005 rows=9 loops=262)
1

```

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):

```

1 -> Limit: 15 row(s) (cost=36232074.38..36232074.56 rows=15) (actual time=31.687..31.690 rows=15 loops=1)
   -> Table scan on <temporary> (cost=36232074.38..38381204.91 rows=171930244) (actual time=31.686..31.688 rows=15 loops=1)
     -> Temporary table with deduplication (cost=36232074.37..36232074.37 rows=171930244) (actual time=31.685..31.685 rows=262 loops=1)
       -> Nested loop inner join (cost=19039049.97 rows=171930244) (actual time=28.835..30.554 rows=2443 loops=1)
         -> Nested loop antijoin (cost=1845935.13 rows=18458599) (actual time=28.822..29.163 rows=262 loops=1)
           -> Table scan on dl (cost=39.60 rows=356) (actual time=0.056..0.174 rows=356 loops=1)
             -> Single-row index lookup on <subquery> using <auto distinct key> (dishID=dl.dishID) (actual time=0.081..0.081 rows=0 loops=356)
               -> Materialize with deduplication (cost=17321.03..17321.03 rows=51850) (actual time=28.760..28.760 rows=258 loops=1)
                 -> Filter: (dl.dishID is not null) (cost=12136.03 rows=51850) (actual time=0.091..28.475 rows=1281 loops=1)
                   -> Nested loop inner join (cost=12136.03 rows=51850) (actual time=0.091..28.358 rows=1281 loops=1)
                     -> Nested loop inner join (cost=4224.10 rows=9333) (actual time=0.033..12.051 rows=9333 loops=1)
                       -> Single-row index lookup on di using PRIMARY (ingredientID=dl.ingredientsID) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=9333)
                         -> Filter: (u.allergies = i.ingredientName) (cost=0.29 rows=6) (actual time=0.002..0.002 rows=0 loops=9333)
                           -> Covering index lookup on u using allergies_index (allergies=i.ingredientName) (cost=0.29 rows=6) (actual time=0.001..0.001 rows=0 loops=9333)
                             -> Covering index lookup on dishIngredients using PRIMARY (dishID=dl.dishID) (cost=0.26 rows=9) (actual time=0.003..0.005 rows=9 loops=262)
1

```

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):

```

1 -> Limit: 15 row(s) (cost=325655.37..325655.54 rows=15) (actual time=217.142..217.146 rows=15 loops=1)
   -> Table scan on <temporary> (cost=325655.37..344961.39 rows=1544284) (actual time=217.141..217.144 rows=15 loops=1)
     -> Temporary table with deduplication (cost=325655.36..325655.36 rows=1544284) (actual time=217.139..217.139 rows=262 loops=1)
       -> Nested loop inner join (cost=171226.99 rows=1544284) (actual time=0.880..215.787 rows=2443 loops=1)
         -> Nested loop antijoin (cost=16708.18 rows=165796) (actual time=0.874..214.462 rows=262 loops=1)
           -> Covering index scan on dl using mealname_index (cost=39.60 rows=356) (actual time=0.050..0.185 rows=356 loops=1)
             -> Nested loop inner join (cost=3444.51 rows=466) (actual time=0.602..0.602 rows=0 loops=356)
               -> Nested loop inner join (cost=4568.96 rows=466) (actual time=0.014..0.200 rows=367 loops=356)
                 -> Table scan on u (cost=0.26 rows=50) (actual time=0.009..0.015 rows=39 loops=356)
                   -> Covering index lookup on di using PRIMARY (dishID=dl.dishID) (cost=0.25 rows=9) (actual time=0.002..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=dl.ingredientsID) (cost=2.33 rows=1) (actual time=0.001..0.001 rows=1 loops=130654)
                         -> Covering index lookup on dishIngredients using PRIMARY (dishID=dl.dishID) (cost=0.26 rows=9) (actual time=0.003..0.004 rows=9 loops=262)
1

```

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..1.513 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 liid_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=40299.85 rows=16580) (actual time=3.363..279.564 rows=435 loops=1)
                    -> Nested loop inner join (cost=22891.50 rows=165796) (actual time=0.256..30.413 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.ingredientID) (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.

```

-> limit: 15 row(s) (actual time=282.704..282.706 rows=15 loops=1)
  -> Sort: users.name, dish.mealName, limit input to 15 row(s) per chunk (actual time=282.704..282.704 rows=15 loops=1)
    -> Table scan on <temporary> (cost=41957.83..42167.55 rows=16580) (actual time=282.491..282.553 rows=435 loops=1)
      -> Temporary table with deduplication (cost=41957.82..41957.82 rows=16580) (actual time=282.488..282.488 rows=435 loops=1)
        -> Nested loop inner join (cost=40299.86 rows=16580) (actual time=0.231..281.787 rows=435 loops=1)
          -> Nested loop inner join (cost=22891.30 rows=165796) (actual time=0.157..90.141 rows=170800 loops=1)
            -> Inner hash join (no condition) (cost=1789.60 rows=17800) (actual time=0.136..2.366 rows=17800 loops=1)
              -> Covering index scan on dish using meal_name_idx (cost=0.80 rows=356) (actual time=0.030..0.214 rows=356 loops=1)
                -> Hash
              -> Table scan on users (cost=5.25 rows=50) (actual time=0.055..0.087 rows=50 loops=1)
            -> Covering index lookup on dishIngredients using PRIMARY (dishID=dish.dishID) (cost=0.25 rows=9) (actual time=0.802..0.804 rows=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)

```

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.

```
| -> Limit: 15 row(s) (actual time=275.875..275.877 rows=15 loops=1)
  -> Sort: users.name, dish.mealName, limit input to 15 row(s) per chunk (actual time=275.874..275.875 rows=15 loops=1)
    -> Table scan on <temporary> (cost=41957.83..42167.55 rows=16580) (actual time=275.661..275.722 rows=435 loops=1)
      -> Temporary table with deduplication (cost=41957.82..41957.82 rows=16580) (actual time=275.658..275.658 rows=435 loops=1)
        -> Nested loop inner join (cost=10229.86 rows=16580) (actual time=2.526..275.098 rows=435 loops=1)
          -> Nested loop inner join (cost=22891.30 rows=165796) (actual time=0.124..88.618 rows=170800 loops=1)
            -> Inner hash join (no condition) (cost=1789.60 rows=17800) (actual time=0.110..2.231 rows=17800 loops=1)
              -> Table scan on dish (cost=0.80 rows=356) (actual time=0.045..0.274 rows=356 loops=1)
                -> Hash
              -> Table scan on users (cost=5.25 rows=50) (actual time=0.042..0.050 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)
      |
```

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
```

```
| -> Limit: 15 row(s) (actual time=18.418..18.421 rows=15 loops=1)
  -> Sort: users.name, dish.mealName, limit input to 15 row(s) per chunk (actual time=18.417..18.418 rows=15 loops=1)
    -> Table scan on <temporary> (cost=9944.88..10172.12 rows=17981) (actual time=18.014..18.127 rows=435 loops=1)
      -> Temporary table with deduplication (cost=9944.87..9944.87 rows=17981) (actual time=18.010..18.010 rows=435 loops=1)
        -> Nested loop inner join (cost=2146.77 rows=17981) (actual time=0.610..17.455 rows=435 loops=1)
          -> Inner hash join (no condition) (cost=1853.43 rows=17981) (actual time=0.452..1.635 rows=4984 loops=1)
            -> Table scan on dish (cost=0.79 rows=356) (actual time=0.063..0.329 rows=356 loops=1)
              -> Hash
            -> Nested loop inner join (cost=50.93 rows=51) (actual time=0.121..0.371 rows=14 loops=1)
              -> Filter: (users.allergies is not null) (cost=5.25 rows=50) (actual time=0.065..0.083 rows=50 loops=1)
                -> Table scan on users (cost=5.25 rows=50) (actual time=0.063..0.076 rows=50 loops=1)
              -> Filter: (users.allergies = ingredients.ingredientName) (cost=0.81 rows=1) (actual time=0.005..0.006 rows=0 loops=50)
                -> Covering index lookup on ingredients using ingredients_index (ingredientName=users.allergies) (cost=0.81 rows=1) (actual time=0.005..0.005 rows=0 loops=50)
            -> Single-row covering index lookup on dishIngredients using PRIMARY (dishID=dish.dishID, ingredientsID=ingredients.ingredientID) (cost=0.25 rows=1) (actual time=0.003..0.003 rows=0 loops=4984)
```

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.