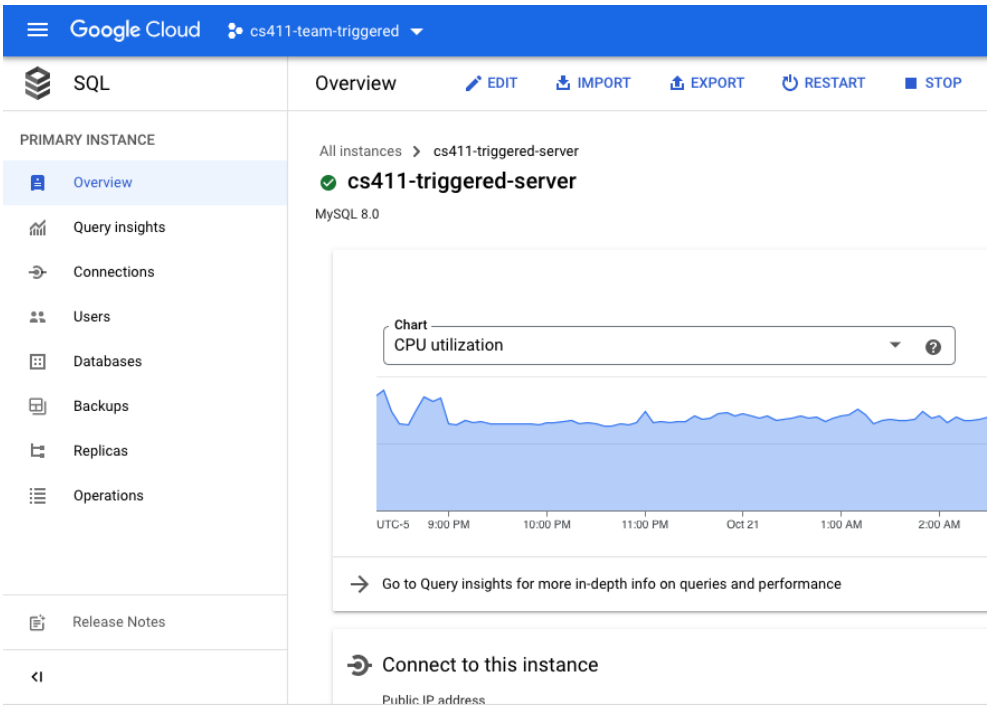
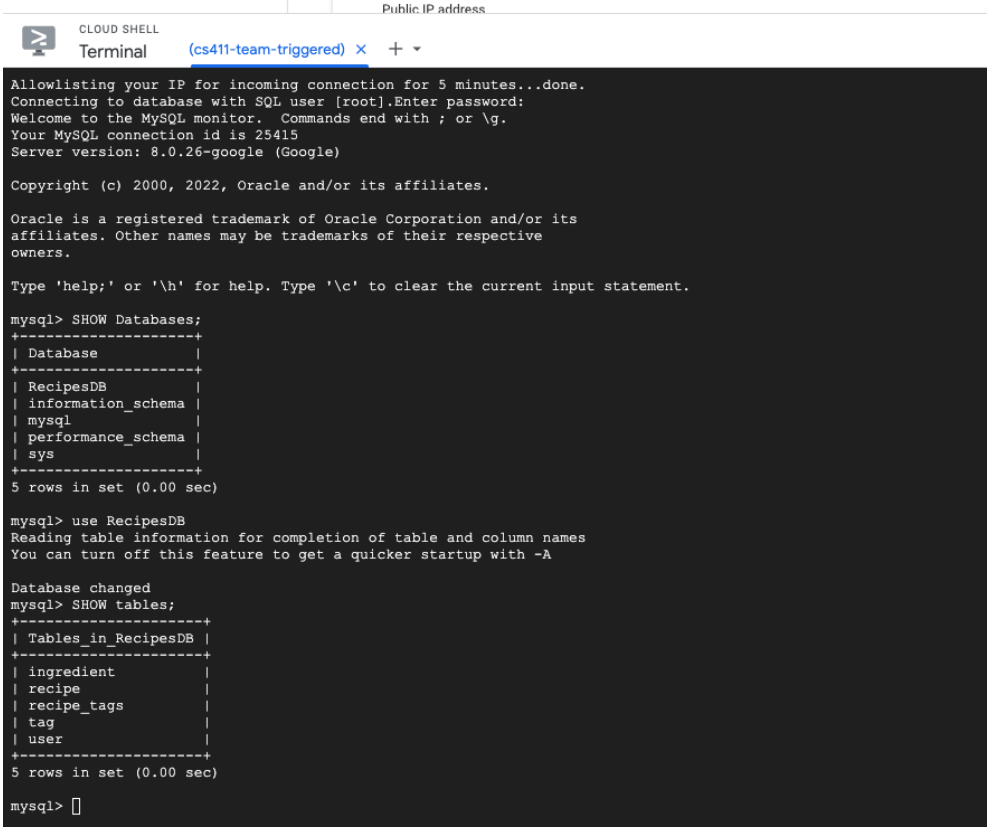

Database Implementation: Connect to GCP, DDL commands to create tables in the database, COUNT queries to ensure that at least three tables have at least 1000 entries in them

GCP Connection:





DDL Commands:

Ingredients

```
CREATE TABLE `RecipesDB`.`ingredient` (
  `name` VARCHAR(255) NOT NULL,
  `recipe_id` INT NOT NULL,
  PRIMARY KEY (`name`, `recipe_id`),
  INDEX `recipe_i_id_idx` (`recipe_id` ASC) VISIBLE,
  CONSTRAINT `recipe_i_id`
    FOREIGN KEY (`recipe_id`)
    REFERENCES `RecipesDB`.`recipe` (`recipe_id`)
    ON DELETE CASCADE
    ON UPDATE CASCADE);
```

Recipe

```
CREATE TABLE `RecipesDB`.`recipe` (
  `recipe_id` INT NOT NULL,
  `title` VARCHAR(255) NOT NULL,
  `cooking_method` VARCHAR(255) NOT NULL,
  `cuisine` VARCHAR(255) NULL,
  `image_path` VARCHAR(255) NULL,
  `prep_time` VARCHAR(255) NULL,
  `serving_time` VARCHAR(255) NOT NULL,
  PRIMARY KEY (`recipe_id`));
```

Recipe_Tags

```
CREATE TABLE `RecipesDB`.`recipe_tags` (
  `recipe_tag_id` INT NOT NULL,
  `tag_name` VARCHAR(255) NOT NULL,
  PRIMARY KEY (`recipe_tag_id`, `tag_name`),
  INDEX `tag_name_idx` (`tag_name` ASC) VISIBLE,
  CONSTRAINT `recipe_tag_id`
    FOREIGN KEY (`recipe_tag_id`)
    REFERENCES `RecipesDB`.`recipe` (`recipe_id`)
    ON DELETE CASCADE
    ON UPDATE CASCADE,
  CONSTRAINT `tag_name`
    FOREIGN KEY (`tag_name`)
    REFERENCES `RecipesDB`.`tag` (`name`)
    ON DELETE CASCADE
    ON UPDATE CASCADE);
```

Tag

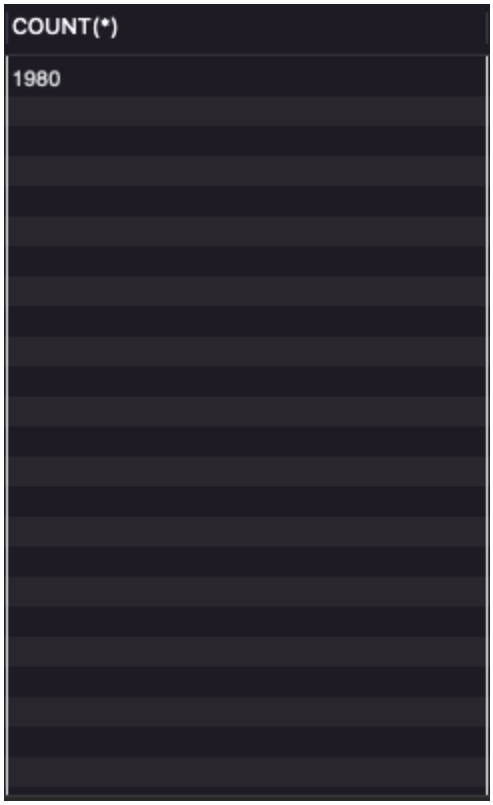
```
CREATE TABLE `RecipesDB`.`tag` (
  `name` VARCHAR(255) NOT NULL,
  `tag_id` INT NULL,
  PRIMARY KEY (`name`));
```

User

```
CREATE TABLE `RecipesDB`.`users` (
  `user_id` INT NOT NULL,
  `first_name` VARCHAR(255) NULL,
  `last_name` VARCHAR(255) NULL,
  `email` VARCHAR(255) NULL,
  PRIMARY KEY (`user_id`));
```

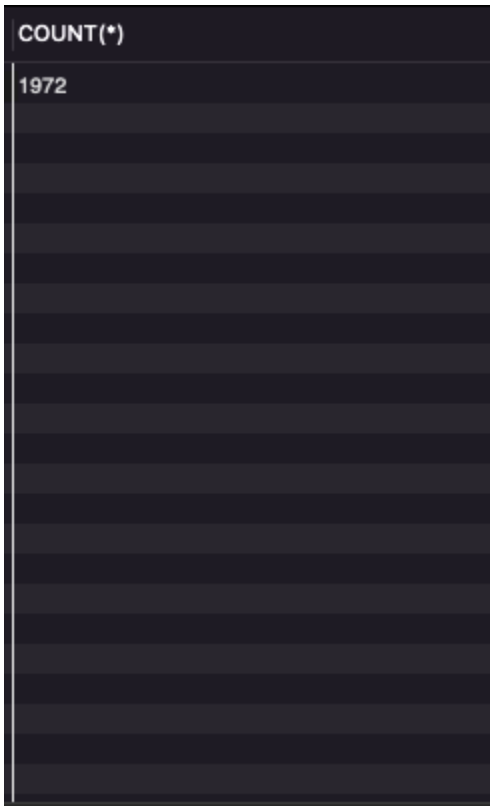
Data Entry:

Recipe: SELECT COUNT(*) FROM `RecipesDB`.`recipe`;



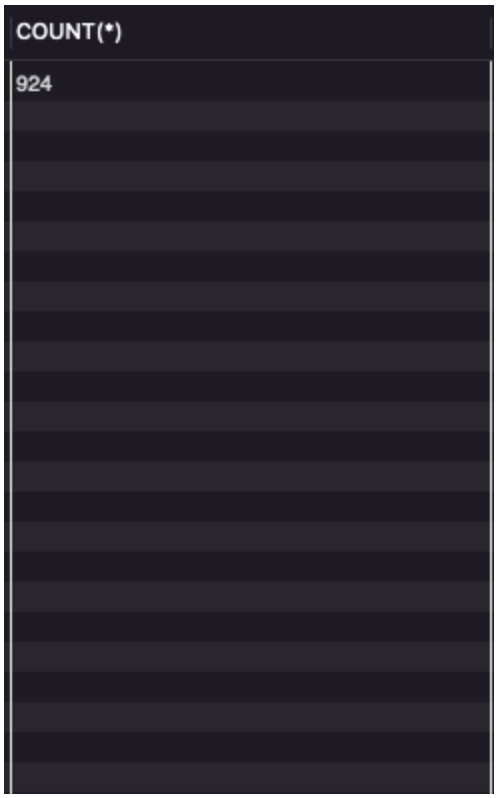
COUNT(*)
1980

Recipe_Tags: SELECT COUNT(*) FROM `RecipesDB`.`recipe_tags`;



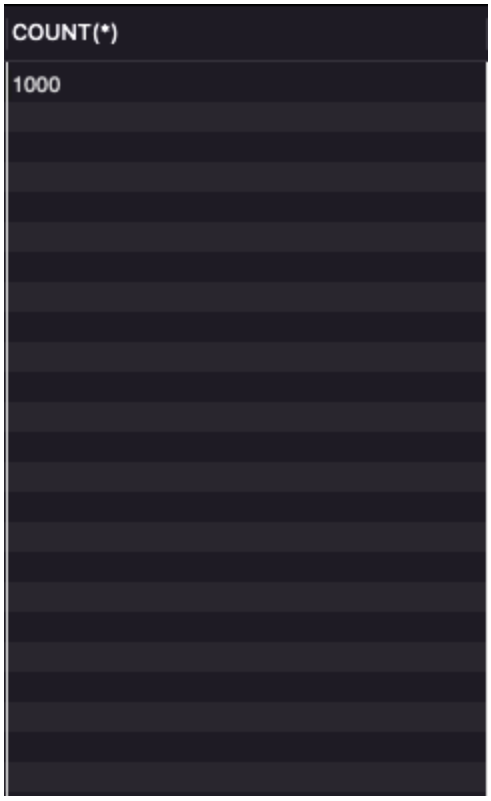
COUNT(*)
1972

Tag: SELECT COUNT(*) FROM `RecipesDB`.`tag`;



COUNT(*)
924

User: SELECT COUNT(*) FROM `RecipesDB`.`user`;



As a group, develop two advanced SQL queries related to the project that are different from one another. The two advance queries are expected to be part of your final application. The queries should each involve at least two of the following SQL concepts:

- Join of multiple relations
- Set operations
- Aggregation via GROUP BY
- Subqueries

Tables we have:
ingredient(name:VARCHAR(255)[PK], recipe_id:INT[PK][FK to recipe.recipe_id])
user(user_id:VARCHAR(30) [PK], first_name:VARCHAR(20), last_name:VARCHAR(20), email: VARCHAR(255))
recipe(recipe_id:INT [PK], title:VARCHAR(50), cooking_method:VARCHAR(255), cuisine:VARCHAR(30), image_path:VARCHAR(255), prep_time:INT, serving_number:INT)
tag(name:VARCHAR(20) [PK], tag_id:INT)
recipe_tags(recipe_tag_id:INT [PK] [FK to recipe.recipe_id], tag_name:VARCHAR(20) [PK] [FK to tag.name])

—----------

Query 1: Select all recipe id + titles that are tagged “vegan”

```
SELECT recipe_id, title
FROM `RecipesDB`.`recipe`
WHERE recipe_id IN
(SELECT recipe_tag_id
FROM `RecipesDB`.`recipe_tags` r JOIN `RecipesDB`.`tag` t ON r.tag_name = t.name
WHERE r.tag_name LIKE '%vegan%');
```

Has a subquery and join

—----------

Query 2: Find all recipe id + titles + user first name + user last name that are created by users with first name starting with ‘A’ and last name starting with ‘T’

```
SELECT recipe_id, title, u.first_name, u.last_name
FROM `RecipesDB`.`recipe` rec JOIN `RecipesDB`.`user` u ON rec.recipe_id = u.user_id
WHERE recipe_id IN
(SELECT recipe_tag_id
FROM `RecipesDB`.`recipe_tags` r JOIN `RecipesDB`.`user` u2 ON r.recipe_tag_id =
u2.user_id
WHERE u2.first_name LIKE 'A%' and u2.last_name LIKE 'T%');
```

Has a subquery and a join

Query 1 results:

recipe_id	title
1843	Akara
1953	Allspiced Apple Butter
1943	All-Purpose Vinaigrette
1509	Mango Smoothie
1966	Almond Brittle
1773	Agua Fresca
1775	Agua Fresca
1647	AB's Chili Powder
1935	All-Purpose Baking Mix
1936	All-Purpose Baking Mix
1449	Blueberry Power Smo…
1776	Agua Fresca de Pepi…
1690	Achiote Paste
1632	A Vegetable Pie Filled…
1710	Acorn Squash with R…

Query 2 results:

recipe_id	title	first_name	last_name
14	Tex-Mex Migas	Ada	Turner
26	Low Carb Potato Salad	Ada	Tucker
95	Moroccan Meatballs	Ada	Thompson
176	Buffalo Chicken Egg Rolls	Arnold	Thompson
304	Asian Beef Zoodle Soup (Whole30)	Alen	Thompson
371	Chicken Rollatini with Sun Dried Tomato Brusch…	Adrian	Thomas
446	Honey Teriyaki Drumsticks	Arnold	Thompson
463	Broiled Miso Salmon	Abigail	Thompson
515	Avocado and Lump Crab Salad	Ada	Turner
532	Sticky Baked Chicken with Apricot, Sage and Le…	Alexia	Thomas
565	Coconut Obsessed Pie	Alina	Thompson
738	Tomato and Zucchini Frittata	Alina	Turner
741	Zucchini Pizza Bites	Aida	Thomas
786	Spicy Shrimp with Chilies	Adison	Thompson
793	Sauteed Julienned Summer Vegetables	Albert	Thompson

Indexing: Measuring the performance of advanced queries prior to adding indexes

Query 1: Select all recipe id + titles that are tagged “vegan”

```
EXPLAIN ANALYZE SELECT recipe_id, title
FROM `RecipesDB`.`recipe`
WHERE recipe_id IN
(SELECT recipe_tag_id
FROM `RecipesDB`.`recipe_tags` r JOIN `RecipesDB`.`tag` t ON r.tag_name = t.name
WHERE r.tag_name LIKE '%vegan%');
```

```
'-> Nested loop inner join  (cost=371.44 rows=214) (actual time=2.582..2.748 rows=65 loops=1)\n
 -> Table scan on <subquery2>  (cost=0.02..5.16 rows=214) (actual time=0.002..0.006 rows=65 loops=1)\n
-> Materialize with deduplication  (cost=291.56..296.70 rows=214) (actual time=2.558..2.566 rows=65 loops=1)\n
-> Nested loop inner join  (cost=270.19 rows=214) (actual time=0.122..2.527 rows=65 loops=1)\n
-> Filter: (RecipesDB.r.tag_name like \'%vegan%\')  (cost=195.45 rows=214) (actual time=0.101..2.252 rows=65 loops=1)\n
-> Index scan on r using tag_name_idx  (cost=195.45 rows=1922) (actual time=0.089..0.615 rows=1972 loops=1)\n
-> Single-row index lookup on t using PRIMARY (name=RecipesDB.r.tag_name)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=65)\n
-> Single-row index lookup on recipe using PRIMARY (recipe_id=`<subquery2>`.recipe_tag_id) (cost=53.48 rows=1) (actual time=0.003..0.003 rows=1 loops=65)\n'
```

Query 2: Find all recipe id + titles + user first name + user last name that are created by users with first name starting with ‘A’ and last name starting with ‘T’

```
EXPLAIN ANALYZE SELECT recipe_id, title, u.first_name, u.last_name
FROM `RecipesDB`.`recipe` rec JOIN `RecipesDB`.`user` u ON rec.recipe_id = u.user_id
WHERE recipe_id IN
(SELECT recipe_tag_id
FROM `RecipesDB`.`recipe_tags` r JOIN `RecipesDB`.`user` u2 ON r.recipe_tag_id =
u2.user_id
WHERE u2.first_name LIKE 'A%' and u2.last_name LIKE 'T%');
```

```
'-> Nested loop semijoin  (cost=114.46 rows=12) (actual time=0.128..0.679 rows=16 loops=1)\n
-> Nested loop inner join  (cost=110.14 rows=12) (actual time=0.117..0.609 rows=16 loops=1)\n
-> Nested loop inner join  (cost=105.82 rows=12) (actual time=0.107..0.552 rows=16 loops=1)\n
-> Filter: ((RecipesDB.u2.first_name like \'A%\') and (RecipesDB.u2.last_name like \'T%\'))
(cost=101.50 rows=12) (actual time=0.081..0.500 rows=16 loops=1)\n
-> Table scan on u2  (cost=101.50 rows=1000) (actual time=0.068..0.381 rows=1000 loops=1)\n
-> Single-row index lookup on u using PRIMARY (user_id=RecipesDB.u2.user_id)  (cost=0.26
rows=1) (actual time=0.003..0.003 rows=1 loops=16)\n
-> Single-row index lookup on rec using PRIMARY (recipe_id=RecipesDB.u2.user_id)
(cost=0.26 rows=1) (actual time=0.003..0.003 rows=1 loops=16)\n
-> Index lookup on r using PRIMARY (recipe_tag_id=RecipesDB.u2.user_id)  (cost=0.26
rows=1) (actual time=0.004..0.004 rows=1 loops=16)\n'
```

Indexing Strategy 1: Use recipe title as an index(Query 1 and 2)
This column would be good to index because it is being accessed and returned by both advanced queries we created. For this reason, it might be helpful to index and see if the query performance improves from doing this.

CREATE INDEX title_idx ON `RecipesDB`.`recipe`(title);

Query 1: Rerun EXPLAIN ANALYZE query from above on advanced query 1
'-> Nested loop inner join (cost=371.44 rows=214) (actual time=2.391..2.523 rows=65 loops=1)\n
-> Table scan on <subquery2> (cost=0.02..5.16 rows=214) (actual time=0.001..0.005 rows=65 loops=1)\n
-> Materialize with deduplication (cost=291.56..296.70 rows=214) (actual time=2.377..2.384 rows=65 loops=1)\n
-> Nested loop inner join (cost=270.19 rows=214) (actual time=0.074..2.355 rows=65 loops=1)\n
-> Filter: (RecipesDB.r.tag_name like \'%vegan%\') (cost=195.45 rows=214) (actual time=0.057..2.110 rows=65 loops=1)\n
-> Index scan on r using tag_name_idx (cost=195.45 rows=1922) (actual time=0.047..0.523 rows=1972 loops=1)\n
-> Single-row index lookup on t using PRIMARY (name=RecipesDB.r.tag_name) (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=65)\n
-> Single-row index lookup on recipe using PRIMARY (recipe_id=`<subquery2>`.recipe_tag_id) (cost=53.48 rows=1) (actual time=0.002..0.002 rows=1 loops=65)\n'

Query 2: Rerun EXPLAIN ANALYZE query from above on advanced query 2
-> Nested loop semijoin (cost=114.46 rows=12) (actual time=0.299..1.189 rows=16 loops=1)
    -> Nested loop inner join (cost=110.14 rows=12) (actual time=0.127..0.859 rows=16 loops=1)
        -> Nested loop inner join (cost=105.82 rows=12) (actual time=0.113..0.788 rows=16 loops=1)
            -> Filter: ((RecipesDB.u2.first_name like 'A%') and (RecipesDB.u2.last_name like 'T%')) (cost=101.50 rows=12) (actual time=0.095..0.504 rows=16 loops=1)
                -> Table scan on u2 (cost=101.50 rows=1000) (actual time=0.081..0.380 rows=1000 loops=1)
            -> Single-row index lookup on u using PRIMARY (user_id=RecipesDB.u2.user_id) (cost=0.26 rows=1) (actual time=0.017..0.017 rows=1 loops=16)
        -> Single-row index lookup on rec using PRIMARY (recipe_id=RecipesDB.u2.user_id) (cost=0.26 rows=1) (actual time=0.004..0.004 rows=1 loops=16)
    -> Index lookup on r using PRIMARY (recipe_tag_id=RecipesDB.u2.user_id) (cost=0.26 rows=1) (actual time=0.020..0.020 rows=1 loops=16)

Conclusion: Based on the results from running the EXPLAIN ANALYZE query prior to adding the index for recipe title from the recipes table and after adding the index, we determined that this indexing strategy does not help the query performance significantly. After further investigation, we can see that the costs for query 1 and query 2 both remain the same following the creation of the index for recipe title. One explanation for this with respect to both queries is that since the recipe title is not anywhere in a filter condition, indexing it will not have an affect on the query performance overall. Additionally, one point to note specific to query 1 is that the filter in this query is based on a foreign key constraint, which are already indexed by default, thus making it impossible to index anything for query 1 to improve its performance drastically.

Indexing Strategy 2: Use user first name as an index(Query 2)

This column would be particularly useful to index for query 2 since it is part of the filter condition in that query. For this reason, we choose to use user first name as an index and determine the impact of this on the performance of query 2.

CREATE INDEX fname_idx ON `RecipesDB`.`user`(first_name);

Query 2: Rerun EXPLAIN ANALYZE query from above on advanced query 2

'-> Nested loop semijoin (cost=122.09 rows=24) (actual time=0.219..0.875 rows=16 loops=1)\n
-> Nested loop inner join (cost=113.73 rows=24) (actual time=0.204..0.599 rows=16 loops=1)\n
-> Nested loop inner join (cost=105.37 rows=24) (actual time=0.195..0.546 rows=16 loops=1)\n
-> Filter: (RecipesDB.u2.last_name like \'T%\') (cost=97.01 rows=24) (actual time=0.186..0.513 rows=16 loops=1)\n
-> Index range scan on u2 using fname_idx, with index condition: (RecipesDB.u2.first_name like \'A%\') (cost=97.01 rows=215) (actual time=0.183..0.479 rows=215 loops=1)\n
-> Single-row index lookup on u using PRIMARY (user_id=RecipesDB.u2.user_id) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=16)\n
-> Single-row index lookup on rec using PRIMARY (recipe_id=RecipesDB.u2.user_id) (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=16)\n
-> Index lookup on r using PRIMARY (recipe_tag_id=RecipesDB.u2.user_id) (cost=0.25 rows=1) (actual time=0.017..0.017 rows=1 loops=16)\n'

Conclusion: The action in which the index is used is the filter condition, where it is used to do an index range scan on the users table. For this action, the cost of completing this task without the index was 101.5 and took 0.38 seconds in total. After the creation of the index for first names, the cost of the task was 97.01 and took 0.479 seconds in total. From this, we can see that there was a decent decrease in the cost of the action, though the time had increased between the two scenarios. From this, we can say that adding the index was helpful, even if it took slightly longer to run the query due to its presence.

Indexing Strategy 3: Use user last name as an index(Query 2)

Similar to Strategy 2, indexing the user last name will attempt to improve the query performance of query 2 specifically, since it is part of the filter condition of query 2. For this reason, we choose to index the user last name column of the user table and determine whether this helps query performance.

CREATE INDEX lname_idx ON `RecipesDB`.`user`(last_name);

Query 2: Rerun EXPLAIN ANALYZE query from above on advanced query 2

'-> Nested loop semijoin (cost=32.56 rows=6) (actual time=0.189..0.301 rows=16 loops=1)\n
-> Nested loop inner join (cost=30.34 rows=6) (actual time=0.177..0.246 rows=16 loops=1)\n
-> Nested loop inner join (cost=28.13 rows=6) (actual time=0.170..0.202 rows=16 loops=1)\n
-> Filter: (RecipesDB.u2.first_name like \'A%\') (cost=25.91 rows=6) (actual time=0.162..0.177 rows=16 loops=1)\n
-> Index range scan on u2 using lname_idx, with index condition: (RecipesDB.u2.last_name like \'T%\') (cost=25.91 rows=57) (actual time=0.157..0.168 rows=57 loops=1)\n
 -> Single-row index lookup on u using PRIMARY (user_id=RecipesDB.u2.user_id) (cost=0.27 rows=1) (actual time=0.001..0.001 rows=1 loops=16)\n
-> Single-row index lookup on rec using PRIMARY (recipe_id=RecipesDB.u2.user_id) (cost=0.27 rows=1) (actual time=0.003..0.003 rows=1 loops=16)\n
 -> Index lookup on r using PRIMARY (recipe_tag_id=RecipesDB.u2.user_id) (cost=0.27 rows=1) (actual time=0.003..0.003 rows=1 loops=16)\n'

Conclusion: The filter condition for query 2 contains a check on the last name of a user, which can be seen in the action corresponding to index range scan on the users table. Prior to adding the index, we see a cost of 101.5 which took 0.38 seconds to complete in total for that specific action. Then, after we add the index for last name, we see a cost of 25.91 which took 0.168 seconds in total to complete for that action. We can see that the query performance significantly improves after adding this index, and the time taken for this query also decreases a decent amount as well. For this reason, we can say that the addition of this index is extremely influential on the query performance.

Indexing Strategy 4: Use recipe tag name as an index(Query 1)
Since the filter condition in query 1 is already indexed by default due to the fact that it is a foriegn key constraint, we do not add any indexes in this strategy, we simply see how the query performs, without any comparison.

Query 1: Run EXPLAIN ANALYZE query from above on advanced query 1
'-> Nested loop inner join (cost=371.44 rows=214) (actual time=2.582..2.748 rows=65 loops=1)\n
 -> Table scan on <subquery2> (cost=0.02..5.16 rows=214) (actual time=0.002..0.006 rows=65 loops=1)\n
-> Materialize with deduplication (cost=291.56..296.70 rows=214) (actual time=2.558..2.566 rows=65 loops=1)\n
-> Nested loop inner join (cost=270.19 rows=214) (actual time=0.122..2.527 rows=65 loops=1)\n
-> Filter: (RecipesDB.r.tag_name like \'%vegan%\') (cost=195.45 rows=214) (actual time=0.101..2.252 rows=65 loops=1)\n
-> Index scan on r using tag_name_idx (cost=195.45 rows=1922) (actual time=0.089..0.615 rows=1972 loops=1)\n
-> Single-row index lookup on t using PRIMARY (name=RecipesDB.r.tag_name) (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=65)\n
-> Single-row index lookup on recipe using PRIMARY (recipe_id=`<subquery2>`.recipe_tag_id) (cost=53.48 rows=1) (actual time=0.003..0.003 rows=1 loops=65)\n'

Conclusion: The filter condition being a foriegn key constraint means we have no comparison to make between the query performance prior to adding the index and afterwards. For this reason, we simply notice that the cost of running the action for the scan on recipe_tags using the foreign key constraint index is 195.45 and runs in 0.615 seconds in total.

Indexing Strategy 5: Use tag name as an index(Query 1)

Since the tag name is in the JOIN clause of the subquery, we create an index for the tag name from the tag table because we join on this column. We determine if this improves query performance in comparison to the cost and time taken by the query prior to adding the index.

Query 1: Run EXPLAIN ANALYZE query from above on advanced query 1

'-> Nested loop inner join (cost=375.50 rows=214) (actual time=3.884..5.033 rows=65 loops=1)\n
-> Table scan on <subquery2> (cost=0.02..5.16 rows=214) (actual time=0.002..0.009 rows=65 loops=1)\n
-> Materialize with deduplication (cost=295.63..300.77 rows=214) (actual time=3.796..3.807 rows=65 loops=1)\n
-> Nested loop inner join (cost=274.25 rows=214) (actual time=0.357..3.745 rows=65 loops=1)\n
-> Filter: (RecipesDB.r.tag_name like \'%vegan%\') (cost=199.51 rows=214) (actual time=0.228..2.752 rows=65 loops=1)\n
-> Index scan on r using tag_name_idx (cost=199.51 rows=1922) (actual time=0.216..1.224 rows=1972 loops=1)\n
-> Single-row index lookup on t using PRIMARY (name=RecipesDB.r.tag_name) (cost=0.25 rows=1) (actual time=0.014..0.015 rows=1 loops=65)\n
 -> Single-row index lookup on recipe using PRIMARY (recipe_id=`<subquery2>`.recipe_tag_id) (cost=53.48 rows=1) (actual time=0.018..0.018 rows=1 loops=65)\n'

Conclusion: Prior to adding the index for tag name in query 1, we will look at the cost and time taken for the location where the index is actually applied, which is in the nested loop inner join for the subquery. The cost of this action prior to adding the index is 270.19 and the time taken is 0.122 seconds. After adding the index, we see that the cost is 274.25 and the time taken is 0.357. From this, it is obvious that adding this index in fact hurts the query performance instead of helping.

Final Strategy Selection and Justification

Query 1: Based on the three strategies that were targeted to improve the query performance of query 1(1, 4, and 5), we select indexing strategy 4 as the most efficient and cost effective strategy for query 1. This is also the strategy including the default index that is added for the foreign key constraint on recipe tag name from the recipe tags table. The reason we do not select strategy 1 was because its performance mirrors strategy 4 but requires us to perform the extra step of creating the index for recipe title, so it would just be better to use the default index that is created for us. Strategy 5, on the other hand, causes the performance of our query to be worse since the use of the tag name column from tag is in a JOIN condition. For this reason, our best option is using the default indexing strategy specified by strategy 4.

Query 2: Based on the three strategies that we created to improve the performance of query 2(1, 2, and 3), we select indexing strategy 3 as the best strategy for query 2. As mentioned in our analysis for strategy 3 above, it is clear that this strategy not only decreases the cost(-75.59) but decreases the time taken(-0.212 seconds), which cannot be said for either of the two other strategies that were created for query 2. Strategy 1, for example, provides no improvement to query performance and we see the same cost and time taken for actions involving the column being indexed. Additionally, while strategy 2 leads to a decrease in the cost of the query, however minimal it may be, the time taken actually increases, meaning that this is not as good as strategy 3 for our purposes. For this reason, our best strategy of the ones tested is using strategy 3 to index by user last name.