The Recipe Database: Final Project Report

Zoe Wang, Mohammed Mojtabaee-Zamani

Introduction

The motivation behind the "The Recipe Database" web app is to make cooking at home more accessible for our users. Cooking at home saves money and promotes healthier eating habits. To accomplish this goal, two different datasets were used–a Kaggle recipe dataset and the USDA nutritional dataset. Combining these two datasets together allowed us to provide the nutritional information for every recipe in our database! Additionally, our application provides the following functionality:

- Search recipes by name, ingredients used, and nutritional content
- Find recipes that adhere to the trending diets of keto, high protein, and low carb
- Generate balanced meal plans for an entire week of healthy eating
- The ability to build an ingredient list using food already at the user's home and then find recipes using only those ingredients.

Our hope is that the combination of all these features will empower our users to cook at home and make healthier eating choices.

Architecture

On the database side, we created a MySQL, a relational database, and hosted it on Amazon Web Services. On the web development side, we created our application using the code template provided to us in Homework_2. We used Node.js to create the server for the application. All of the routes in our API were loaded into the routes.js file. These routes contained the queries that would retrieve or insert data into our MySQL database. For the frontend we used React.js. React.js allowed us to use fetches to retrieve data from the server. React also had the added benefit of quickly updating the page in response to fetched data without reloading the entire page.

Data

We started off with a **Kaggle** recipe dataset that consists of a set of 2,231,142 recipes scraped from cookbooks.com. Each recipe has a title, ingredients with quantities, directions, a link to the recipe, how the information was sourced, and a list of unique ingredients. We used this dataset to extract ingredients, amounts, and portions for each recipe to determine the nutritional value of each recipe.

In order to get nutritional information of each recipe, we used a **USDA** dataset with a total size of 54M. This dataset is split into many subsets of information, of which we used 5 tables.

Table: food_names (usda_id, name)
This table includes the usda_id and names of 1,911,983 different foods and ingredients. We used this table to match usda_ids to the ingredients extracted from the recipe dataset using edit distance matching via Pandas.

Table: food_portions (usda_id, amount, weight in grams, measure_unit_id, portion_description, modifier))
This table contains information on the amount of an ingredient in grams relative to its portion size. Since portion descriptions can be vague and the dataset is a combined set of different subsets of foods the USDA researches, the table has three attributes (measure_unit_id, portion_description, modifier) that can be used to identify the portions of food. The former is a foreign key referencing the measure_unit table, and the other two are strings. For most foods, only one of these attributes was not null. For each usda_id, amount, and portion, there was a calculated weight in grams that was useful in determining the nutritional value of the food. This table has 39,113 tuples.

Table: measure_unit (id, name)
This table includes the measure id (referenced in the portions table) and names of 122 different measure units.

Table: nutrient_names (id, name, unit_name)
This table includes the nutrient id, names, and unit of the nutrient for 474 different nutrients.

Table: nutrient_food_ratio (usda_id, nutrient_id, amount of nutrient per 100 grams of usda food)
This table contains information on the amount of a nutrient for every 100 grams of the associated usda food. This table has 9,170,777 tuples.

***** Note: the schema described above for the tables are not the original schema. The original schema is very long for tables, and we only extracted the attributes we needed***

The food_portions, measure_unit, nutrient_names, and nutrient_food_ratio tables were used to calculate the nutritional value of each ingredient of each recipe. Since each recipe has its own unique set of ingredients, portions, and amounts, we had to :

- match the usda_id to the ingredient
- use the food_portions table to find the amount in grams of each ingredient (joined with measure_unit id in order to match the name of a measure descriptor)
- Use the nutrient_food_ratio table to calculate the amount of nutrients per ingredient
- Use the nutrient_names table to get the names of the nutrients

In addition to our Kaggle recipe dataset and the USDA nutrition dataset, we used the **UpSplash** photo search API. Our web application generates a unique webpage for each of our recipes whenever a user clicks on the recipe's title. On each of these unique web pages, we placed an image of the recipe at the top of the page using UpSplash. We accomplished this by passing the title of the recipe as a query parameter when fetching data from UpSplash. At first we were retrieving images that did not fit the description of the recipe. For example if we searched for an image of chicken, we might get a picture of an actual live chicken. We fixed this problem by using UpSplash's photo collections feature. We identified the top three image collections for food and added the collection ids to the API calls. Each collection had about 3,000 images.

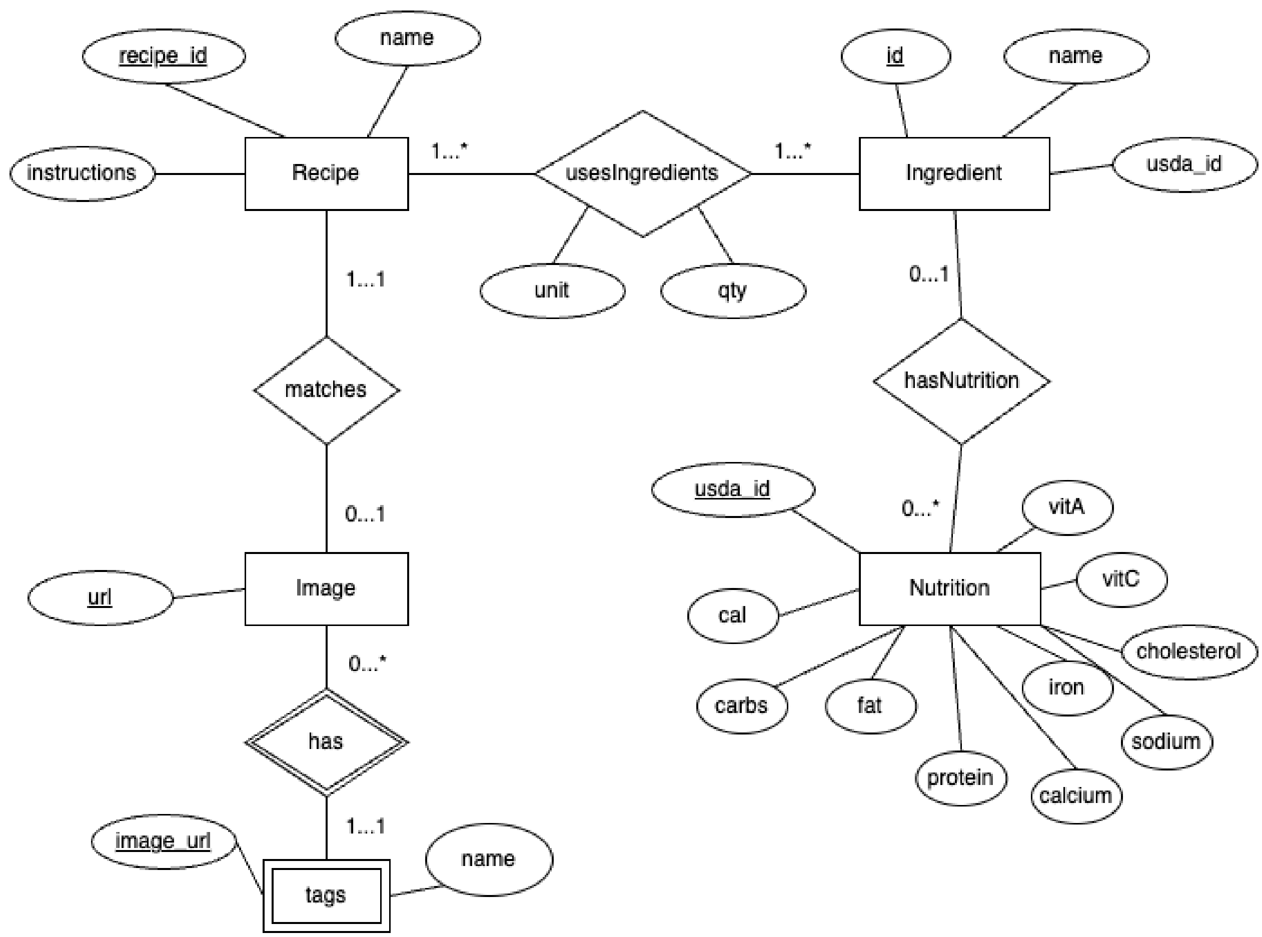
Database

We spent a lot of time cleaning the datasets. For the recipe dataset, we needed to extract the ingredients, amounts, and units of measure. The recipe dataset has a list of ingredients, so we used that information combined with regex matching to find the amount of units of the ingredients since the common format was (amount - unit - ingredient). However, extraction was not perfect as units are not always one word and sometimes units and amounts were after the ingredient. These factors resulted in nulls in our data after extraction. Instead of deleting the nulls, we replaced the null amounts with 1 and the null units with 'serving' since these were the standard and minimal amounts of an ingredient that had to be in the recipe. Moreover, when extracting amounts, a lot of recipe tuples were in the format of "ex: 2(2 1/2 cup cans)," so we had to use regex to match fractions and parentheses in strings, turn the extracted numbers into floats, multiply, and update the string with the new amount. The units in the recipe dataset were also abbreviated (ex: cups was written as c.). We normalized this to match the USDA set by replacing abbreviations with their full descriptors and also updating the abbreviations attribute of USDA measure_unit.

For the USDA dataset, since certain tables like nutrient_food_ratio were robust, we removed tuples that did not use the nutrients that we needed since we were only looking for certain nutrients for the recipes. We also removed attributes that were not needed from each table.

Lastly, we had to match the USDA-id with the ingredients that we extracted. To do this, we had to match each recipe ingredient name with the USDA food names and return the USDA-id of the best match. We used an edit distance function that found the best match for each ingredient. We had to modify the function so matches that were not ideal were not included. Additionally, since the USDA set has duplicate food names (with different portion descriptions), if there were multiple matches, we matched with the food that had the most information on portions. This helped maximize our opportunity to match an ingredient to its food portion descriptor. Some ingredients from our extracted recipe data had more than one word, which resulted in a difficult matching process and returned a lot of nulls. To address this issue, we split up ingredients with more than one word and matched the "important" strings, so as to not match filler strings such as "of" and "a".

Entity Relationship Diagram:



SQL RELATIONAL SCHEMA:

```
CREATE TABLE Recipe (
        recipe_id int,
        name varchar(255),
        Ingredients varchar(255),
        instructions varchar(255),
        PRIMARY KEY(recipe_id)
) ; 
```
1,451,270 tuples

```
CREATE TABLE Ingredient (
        id int,
        name varchar(255),
        usda_id int,
        PRIMARY KEY (id)
        FOREIGN KEY usda_id REFERENCES nutrition(usda_id)
) ; 
```
12,893 tuples

```
CREATE TABLE usesIngredients (
        recipe_id int,
        ingredient_id int,
        quantity int,
        unit varchar(255),
        PRIMARY KEY (recipe_id, ingredient_id)
        FOREIGN KEY recipe_id REFERENCES recipe(recipe_id)
        FOREIGN KEY ingredient_id REFERENCES ingredient(ingredient_id)
)  ; 
```
569,815 tuples

```
CREATE TABLE Nutrition (
        usda_id int,
        cal int,
        protein int,
        carbs int,
        fat int,
        vitA int,
        vitC int,
        cholesterol int,
        iron int,
        sodium int,
```

calcium int,

PRIMARY KEY (usda_id);

) ; ***366,205 tuples***

***** Note: recipe images were obtained through an API***
***** Note: The data from USDA tables were all combined into the Nutrition table***

Our data is in BCNF:
Recipe: recipe_id → name, instructions
Ingredient: id→name, usda-id
Nutrition: usda_id→qty, cal, carbs, fat, protein, calcium, iron, sodium, cholesterol, vitC, citA
There are no functional dependencies for image and tags.
In each of these relations, for X→A, C is a superset, so our data is in BCNF.

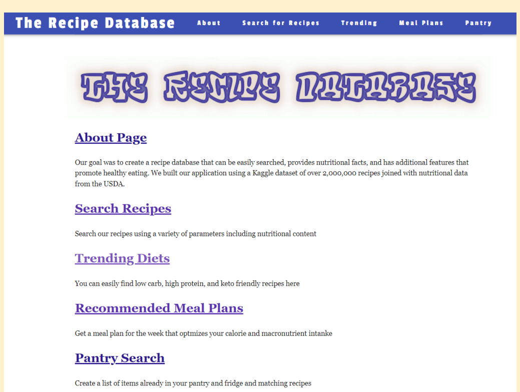
Web App Description

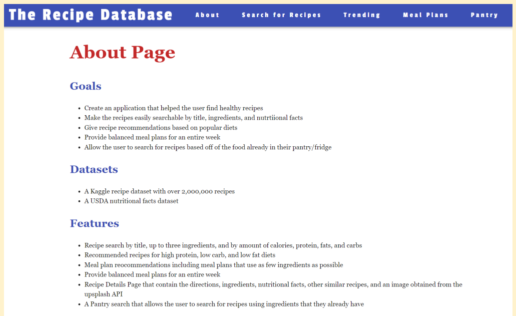
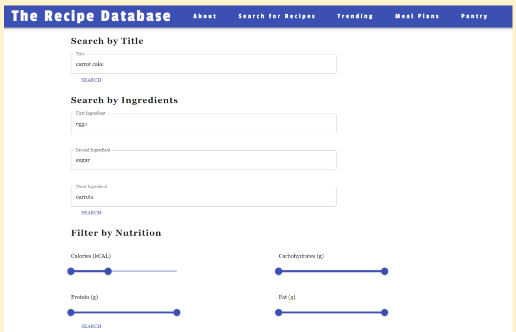
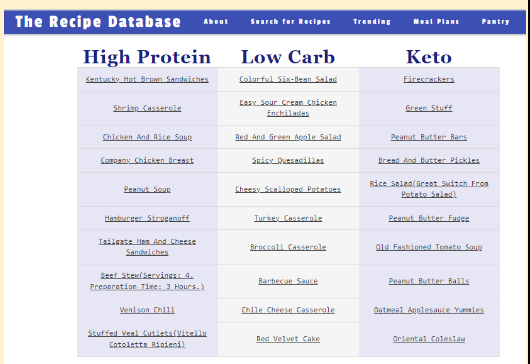
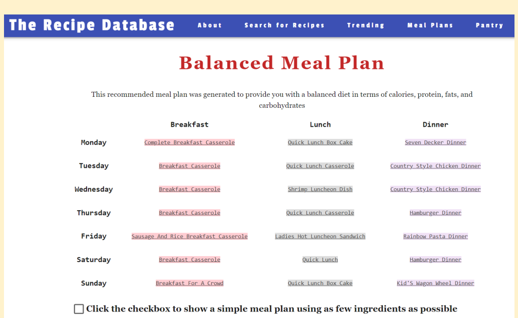
Brief on Pages and their Functionality:

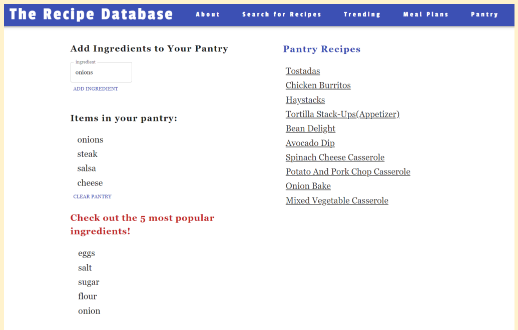
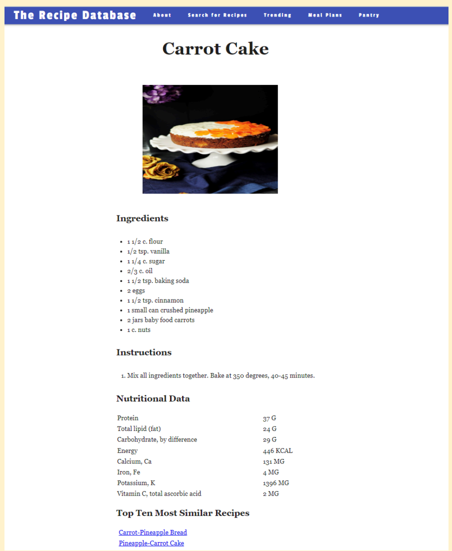
We built our web application using the code template provided in Homework 2 as a starting point. The application uses the Node.js framework for the server and React.js for the client-side code. The app has the following pages:

1. Homepage
2. About page
3. Search Recipes Page
4. Trending Diets Page
5. Balanced Meal Plan Page
6. Pantry Page (allows the user to store the ingredients already in their pantry using a temporary table)
7. Detail recipe Info pages for each recipe in the database

A more detailed look at how each page functions

Page name	Image	Description
The Homepage		The Homepage provides an overview of the web app's content, giving a brief description of each page. It also has hyperlinks to each page.

<table>
<tr><td>About Page</td><td>
</td><td>The about page is purely informational. It provides an outline of the project goals, the datasets used, and the features of the web app.</td></tr>
<tr><td>Search for Recipes Page</td><td>
</td><td>This page allows the user to search through the entire database of recipes. They can search by any combination of the parameters recipe title, ingredients (up to three), and the macronutrients of calories, protein, fats, and carbohydrates. The search parameters are inputted using a mixture of text fields and sliders. After the user clicks the search button, results matching the search will appear at the bottom of the page. Each recipe in the results has a link that will take the user to a page with more detailed information specific to that recipe, including its nutritional facts.</td></tr>
<tr><td>Trending Page</td><td>
</td><td>The “Trending” diets page utilizes three separate queries to the database to find recipes that are appropriate for the popular high protein, low carb, and keto diets. Each recipe has a link that will take the user to a page with more information specific to that recipe, including its nutritional facts.</td></tr>
<tr><td>Meal Plans Page</td><td>
</td><td>The “Meal Plan” page provides the user with a nutritionally balanced meal plan for the entire week. The plan is produced by a query that randomly selects a combination of recipes that balances total calories, protein, carbs, and fats for the week. There is also a bar graph that visually displays how the meal plan is balanced.At the bottom of the page, the user can use a checkbox to switch to a meal plan that provides all the required nutrition using a minimal amount of ingredients.</td></tr>
</table>

Pantry Page		The “Pantry Page” allows the user to select ingredients that they already have in their pantry and/or refrigerator. These ingredients are added to a temporary table and then used to identify recipes using only those ingredients. The user can select an unlimited number of ingredients and they can also clear the pantry of ingredients using a “clear” button.
Detail Recipe Info Page		Every recipe displayed on our web app can be clicked on and the user will be taken to an auto generated page that contains a picture, ingredients list, nutritional facts, and list of similar recipes.

API Specification:

Route 1

Route: /recipe/:recipe_id
Description: Returns all information about a recipe
Route Parameter(s): recipe_id (string)
Query Parameter(s): None
Route Handler: recipe(req, res)
Return Type: JSON Object
Return Parameters: [rid (string), title (string), ingredients (list), instructions (list), link (string), source (string), ner (list)
Expected Behavior: If a valid recipe_id is provided, all the return parameters above will be returned from the database. Otherwise, an empty object will be returned.

Route 2

Route: /get_nutrition
Description: gets the nutrition of a recipe
Route Parameter(s): None
Query Parameter(s): recipe_id (string)

Route Handler: get_nutrition(req, res)
Return Type: JSON Object
Return Parameters: [name (string), unit_name (string), total_ntr_per_recipe (float)
Expected Behavior: If a valid recipe_id is provided, all the return parameters above will be returned from the database. Otherwise, an empty object will be returned.

Route 3
Route: /get_balanced_meal_plan
Description: get a balanced meal plan for the week
Route Parameter(s): None
Query Parameter(s): None
Route Handler: get_balanced_meal_plan(req, res)
Return Type: JSON Object
Return Parameters: [breakfast_rid (string), lunch_rid (string), dinner_rid (float), total_calories(float), breakfast_carbs(float), lunch_carbs(float), dinner_carbs(float), breakfast_fat(float), lunch_fat(float), dinner_fat(float), breakfast_protein(float), lunch_protein(float), dinner_protein(float)
Expected Behavior: Returns the above parameters

Route 4
Route: /get_top_ten_most_similar
Description: given a recipe_id, returns the top ten most similar recipes based off of a natural join on the recipe description
Route Parameter(s): None
Query Parameter(s): recipe_id (string)
Route Handler: get_top_ten_most_similar(req, res)
Return Type: JSON Object
Return Parameters: [rid (string), title (string), ingredients (list), instructions (list), ner(list)]
Expected Behavior: If a valid recipe_id is provided, all the return parameters above will be returned from the database. Otherwise, an empty object will be returned.

Route 5
Route: /get_top_five_ingredients
Description: gets the top 5 most used ingredients
Route Parameter(s): None
Query Parameter(s): None
Route Handler: get_top_five_ingredients(req, res)
Return Type: JSON Object
Return Parameters: [id (string), ingredient (string),
Expected Behavior: returns the top 5 most used ingredients.

Route 6
Route: /get_recipes_under_ingredient_amount
Description: given an ingredient amount, returns recipes that have at most that amount of ingredients
Route Parameter(s): None
Query Parameter(s): ingredAmt (int)
Route Handler: get_recipes_under_ingredient_amount(req, res)
Return Type: JSON Object

Return Parameters: [rid (string), title (string), ingredients (list), instructions (list), ner(list)]
Expected Behavior: If a valid ingredAmt is provided, all the return parameters above will be returned from the database. Otherwise, an empty object will be returned.

Route 7

Route: /get_recipes_by_parameters
Description: Returns recipes based off of ingredient names and nutritional parameters of carbs, protein, fats, and calories
Route Parameter(s): None
Query Parameter(s): ingredient1 (string), ingredient2 (string), ingredient3 (string), caloriesLow (int), caloriesHigh (int), fatsLow (int), fatsHigh (int), carbsLow (int), carbsHigh (int), fatsLow (int), fatsHigh (int)
Route Handler: get_recipes_by_parameters(req, res)
Return Type: JSON Object
Return Parameters: [rid (string), title (string), ingredients (list), instructions (list), ner(list)]
Expected Behavior: If valid input parameters are provided, all the return parameters above will be returned from the database. Otherwise, an empty object will be returned.

Route 8

Route: /get_ingredient_id_by_ingredient_name
Description: get ingredient_id by name of ingredient and add to pantry
Route Parameter(s): None
Query Parameter(s): ingredient_name (string),
Route Handler: /get_ingredient_id_by_ingredient_name(req, res)
Return Type: JSON Object
Return Parameters: None
Expected Behavior: get each ingredient input by user and input into table

Route 9

Route: /get_recipes_by_ingredients_in_fridge
Description: gets recipes that can be made from the ingredients in the fridge
Route Parameter(s): None
Query Parameter(s): None
Route Handler: get_recipes_by_ingredients_in_fridge(req, res)
Return Type: JSON Object
Return Parameters: [rid (string), title (string), ingredients (list), instructions (list), ner(list)]
Expected Behavior: Will return recipes that can be made from ingredients in the refrigerator

Route 10

Route: /get_ingredients_from_recipe_id
Description: gets the ingredients list from a recipe_id
Route Parameter(s): None
Query Parameter(s): recipe_id
Route Handler: get_ingredients_from_recipe_id(req, res)
Return Type: JSON Object
Return Parameters ingredients(list)
Expected Behavior: Will return list of ingredients when a valid recipe_id is provided. Otherwise an empty object is returned.

Route 11
Route: /get_meal_plan_few_ingredients
Description: creates a meal plan using as few ingredients as possible
Route Parameter(s): None
Query Parameter(s): None
Route Handler: get_meal_plan_few_ingredients(req, res)
Return Type: JSON Object
Return Parameters: breakfast_rid (string), lunch_rid(string), dinner_rid(string)
Expected Behavior: return the above return parameters of three meals that use the fewest amount of ingredients

Route 12
Route: /get_low_carb
Description: returns ten low carb recipes
Route Parameter(s): None
Query Parameter(s): None
Route Handler: get_low_carb(req, res)
Return Type: JSON Object
Return Parameters: [rid (string), title (string), ingredients (list), instructions (list), link (string), source (string), ner (list)
Expected Behavior: Returns the above parameters for ten high protein recipes

Route 13
Route: /get_keto
Description: returns ten keto recipes
Route Parameter(s): None
Query Parameter(s): None
Route Handler: get_keto(req, res)
Return Type: JSON Object
Return Parameters: [rid (string), title (string), ingredients (list), instructions (list), link (string), source (string), ner (list)
Expected Behavior: Returns the above parameters for ten keto recipes

Route 14
Route: /get_high_protein
Description: returns ten high protein recipes
Route Parameter(s): None
Query Parameter(s): None
Route Handler: get_high_protein(req, res)
Return Type: JSON Object
Return Parameters: [rid (string), title (string), ingredients (list), instructions (list), link (string), source (string), ner (list)
Expected Behavior: Returns the above parameters for ten high protein recipes

Queries:

Balanced- before optimization:

```
with rids as (
SELECT distinct *
from (SELECT distinct *
from (select distinct m1.rid as breakfast_rid, m2.rid as lunch_rid, m3.rid as           dinner_rid,
round(m1.total_calories + m2.total_calories + m3.total_calories, 0) as total_meal_calories,
round(m1.total_carbs, 0) as breakfast_carbs,
round(m2.total_carbs,0) as lunch_carbs,
round(m3.total_carbs,0) as dinner_carbs,
round(m1.total_carbs + m2.total_carbs + m3.total_carbs,1) as total_meal_carbs,
round(m1.total_fat,0) as breakfast_fat,
round(m2.total_fat,0) as lunch_fat,
round(m3.total_fat,0) as dinner_fat,
round(m1.total_fat + m2.total_fat + m3.total_fat,1) as total_meal_fat,
round(m1.total_protein,0) as breakfast_protein,
round(m2.total_protein,0) as lunch_protein,
round(m3.total_protein,0) as dinner_protein,
round(m1.total_protein + m2.total_protein + m3.total_protein,1) as total_meal_protein
from breakfast_recipes_nutrition m1, lunch_recipes_nutrition m2 , dinner_recipes_nutrition m3
where m1.total_calories + m2.total_calories + m3.total_calories >= 1800 and m1.total_calories + m2.total_calories + m3.total_calories <= 2200)
as all_meal_plans)as meals
where total_meal_protein * 4 >= total_meal_calories * .20 AND total_meal_protein * 4 <= total_meal_calories * .40
AND total_meal_fat * 9 >= total_meal_calories * .20 AND total_meal_fat * 9 <= total_meal_calories * .40
AND total_meal_carbs * 4 >= total_meal_calories * .40 AND total_meal_carbs * 4 <= total_meal_calories * .50
ORDER BY RAND()
LIMIT 7
), breakfastTitle as (
select ri.*, r.title as breakfast_title
from rids ri join recipes r on ri.breakfast_rid = r.rid
), lunchTitle as (
select bt.*, r.title as lunch_title
from breakfastTitle bt join recipes r on bt.lunch_rid = r.rid
)
select lt.*, r.title as dinner_title
from lunchTitle lt join recipes r on lt.dinner_rid = r.rid;
```

This query was used to return a randomly ordered balanced weekly meal plan such that daily intake carbs, protein, and fats were all within a certain ratio of daily caloric intake that was defined as "balanced" by nutritional standards. We had to match breakfast, lunch, and dinner to get a valid meal plan. This was complex and used in the meal plan page.

Getting nutritional info of a recipe

```
select *
from
(SELECT distinct rid, id, name, unit_name, total from (
with nutrient_amt_per_recipe AS (
select *, ROUND(((qty/amount)*avg_weight/100 * amt), 0) AS ntr_amt_per_recipe
from (
select *, avg(gram_weight) OVER(PARTITION BY rid, ingredient) AS avg_weight
from (
SELECT rid, ingredient, i.usda_id as usda_id, qty, amount, gram_weight
from ingredient_names i
join used_in on id = iid
join food_portions on i.usda_id = food_portions.usda_id
left join measure_unit on measure_unit_id = measure_unit.id
```

```
WHERE (name LIKE concat('%', unit, '%')
OR abbrev LIKE concat('%', unit, '%')
OR modifier LIKE concat('%', unit, '%')
OR portion_description LIKE concat('%', unit, '%') )
order by rid) as portions_per_recipe) ppr
natural join nutrient_food_ratio nfr
JOIN nutrient_names nn on nfr.nutrient_id = nn.id)

SELECT DISTINCT *, sum(ntr_amt_per_recipe)
OVER(PARTITION BY rid, id) AS total
FROM nutrient_amt_per_recipe) as total_ntr_per_recipe) as nutrition
WHERE rid = {$rid};
```

This query was used to return the nutrition information of a recipe: the name of the nutrient, the amount, and the unit of measurement. Although this seemed complex, after using it to only return information for one recipe, the run time was very fast (less than 2 sec). This was used in the recipe page and used to get nutritional information for the balanced meal plan.

Unoptimized Minimal meal plan

```
create view num_ingredients_per_day as (
with num_ingredients as(
select rid, count(*) as num_ingredients
from used_in
group by rid)

select distinct breakfast_rid, lunch_rid, dinner_rid, (n1.num_ingredients + n2.num_ingredients + n3.num_ingredients) as ingredients_per_meal
from (select distinct m1.rid as breakfast_rid, m2.rid as lunch_rid, m3.rid as dinner_rid,
m1.total_calories + m2.total_calories + m3.total_calories as total_meal_calories,
round(m1.total_carbs + m2.total_carbs + m3.total_carbs,1) as total_meal_carbs,
round(m1.total_fat + m2.total_fat + m3.total_fat,1) as total_meal_fat,
round(m1.total_protein + m2.total_protein + m3.total_protein,1) as total_meal_protein
from breakfast_recipes_nutrition m1, lunch_recipes_nutrition m2 , dinner_recipes_nutrition m3
where m1.total_calories + m2.total_calories + m3.total_calories >= 1800 and m1.total_calories + m2.total_calories + m3.total_calories <=
2200)as all_meal_plans
join num_ingredients n1 on breakfast_rid = n1.rid join num_ingredients n2 on lunch_rid = n2.rid join num_ingredients n3 on dinner_rid = n3.rid
order by ingredients_per_meal);

DROP temporary table IF EXISTS minimal_meal_plan;
create temporary table minimal_meal_plan (
breakfast_rid int unique,
lunch_rid int unique,
dinner_rid int unique);

insert ignore into minimal_meal_plan (breakfast_rid, lunch_rid, dinner_rid)
select m2.breakfast_rid, m2.lunch_rid, m2.dinner_rid
from num_ingredients_per_day m2;
select * from minimal_meal_plan
limit 7;
```

This query was used to return a randomly ordered minimal ingredient weekly meal plan such that recipes displayed had the least amount of ingredients and also did not repeat any recipes. To ensure that recipes were not repeated, we had to insert into a temporary table with unique constraints since SELECT DISTINCT for a meal plan would return distinct meal plans, but not meals. We had to match with breakfast, lunch, and dinner so we wouldn't be recommending steak for breakfast. This query was complex and used on the meal plan page.

Unoptimized Low carb

```
with with_carbs as (
select rid, id, name, unit_name, total_calories, total_carbs from total_ntr_per_recipe natural join (select rid, total_ntr_per_recipe as total_calories
from (with nutrient_amt_per_recipe AS (select *, ROUND(((qty/amount)*avg_weight/100 * amt), 1) AS ntr_amt_per_recipe
from (temp_portions) ppr natural join nutrient_food_ratio nfr JOIN nutrient_names nn on nfr.nutrient_id = nn.id)

SELECT DISTINCT rid, id, name, unit_name, sum(ntr_amt_per_recipe) OVER(PARTITION BY rid, id) AS total_ntr_per_recipe
FROM nutrient_amt_per_recipe)
)
WHERE id = 1008 and total_ntr_per_recipe != 0) as calories natural join (select rid, total_ntr_per_recipe as total_carbs from total_ntr_per_recipe
WHERE id = 1005 and total_ntr_per_recipe != 0) as carbs
WHERE id IN (1005, 1008))
select recipes.rid, recipes.title from recipes natural join (select distinct rid
from with_carbs
where total_carbs * 4 <= total_calories * 0.25
order by total_carbs/total_calories
limit 75
) as low_carb
where title not like '%dressing%'
order by rand()
limit 10;
```

This query was used to return 10 recipes of a certain set of recipes we extracted that best matched the diet criteria for low-carb diets in terms of carb to caloric ratio. We first had to join tables in order to get the total nutrition of each recipe based on ingredient amounts, portions, and name and finding the sum of nutrition value per nutrient per ingredient per recipe. After getting the nutrition, we selected recipes with low carb ratios. We had similar queries from keto and high-protein diets. This was complex and was used in the trending page of our webpage.

Unoptimized Top 10 similar recipes

```
select recipes.rid, recipes.title
from (select * from recipes) as recipes
natural join (select *
from used_in
where iid in (select iid from used_in where rid = ${recipe_id})
and rid != ${recipe_id}
group by rid
order by count(*) desc)
as top_ten_similar
group by title
limit 10;
```

This query was used to return the top 10 similar recipes of a given recipe. We found recipes that shared the most ingredients with the recipe, joined on recipe id with the recipe table to get the title, and grouped by title so duplicate titles would not show up. This query was complex and used in the recipe info page for each recipe.

Performance Evaluation

#	Route Name	Original Performance (seconds)	Performance After Optimization (seconds)
1	get_balanced_meal_plan	15	2
2	get_top_ten_most_similar	6	2.5
3	get_meal_plan_few_ingredients	34	2
4	get_keto	47	2
5	get_low_carb	35	2
6	get_high_protein	42	2

After optimization, balanced

```
create table all_meal_plans_with_nutrition as(
SELECT distinct *
from (select distinct m1.rid as breakfast_rid, m2.rid as lunch_rid, m3.rid as dinner_rid,
round(m1.total_calories + m2.total_calories + m3.total_calories, 0) as total_meal_calories,
round(m1.total_carbs, 0) as breakfast_carbs,
round(m2.total_carbs,0) as lunch_carbs,
round(m3.total_carbs,0) as dinner_carbs,
round(m1.total_carbs + m2.total_carbs + m3.total_carbs,1) as total_meal_carbs,
round(m1.total_fat,0) as breakfast_fat,
round(m2.total_fat,0) as lunch_fat,
round(m3.total_fat,0) as dinner_fat,
round(m1.total_fat + m2.total_fat + m3.total_fat,1) as total_meal_fat,
round(m1.total_protein,0) as breakfast_protein,
round(m2.total_protein,0) as lunch_protein,
round(m3.total_protein,0) as dinner_protein,
round(m1.total_protein + m2.total_protein + m3.total_protein,1) as total_meal_protein
from breakfast_recipes_nutrition m1, lunch_recipes_nutrition m2 , dinner_recipes_nutrition m3
where m1.total_calories + m2.total_calories + m3.total_calories >= 1800 and m1.total_calories + m2.total_calories + m3.total_calories <= 2200)
as all_meal_plans
);

with rids as (
SELECT distinct *
from (all_meal_plans_with_nutrition)
where total_meal_protein * 4 >= total_meal_calories * .20 AND total_meal_protein * 4 <= total_meal_calories * .40
AND total_meal_fat * 9 >= total_meal_calories * .20 AND total_meal_fat * 9 <= total_meal_calories * .40
AND total_meal_carbs * 4 >= total_meal_calories * .40 AND total_meal_carbs * 4 <= total_meal_calories * .50
ORDER BY RAND()
LIMIT 7
), breakfastTitle as (
select ri.*, r.title as breakfast_title
from rids ri join recipes r on ri.breakfast_rid = r.rid
), lunchTitle as (
select bt.*, r.title as lunch_title
from breakfastTitle bt join recipes r on bt.lunch_rid = r.rid
)
select lt.*, r.title as dinner_title
from lunchTitle lt join recipes r on lt.dinner_rid = r.rid;
```

Prior to optimizing balanced meal plans, we had to utilize two cross products of recipes (breakfast recipes x lunch recipes x dinner recipes) to create a daily meal and then query for a balanced meal. Run time for running cross products is high, so instead of doing this each time, we created a table that displayed all possible meal plans and their nutritional data. Doing so helped us retrieve both balanced meal plans and minimal meal plans more quickly because the meal plans were already calculated and we just selected meal plans with nutritional requirements that matched our query and joined with recipes to get the titles.

Optimized minimal

```
create table minimal_meal_plan (
breakfast_rid int unique,
lunch_rid int unique,
dinner_rid int unique);
insert ignore into minimal_meal_plan (breakfast_rid, lunch_rid, dinner_rid)
select m2.breakfast_rid, m2.lunch_rid, m2.dinner_rid
from (with num_ingredients as(
select rid, count(*) as num_ingredients
from used_in
group by rid)
select distinct breakfast_rid, lunch_rid, dinner_rid, (n1.num_ingredients + n2.num_ingredients + n3.num_ingredients)
as ingredients_per_meal
from all_meal_plans_with_nutrition
join num_ingredients n1 on breakfast_rid = n1.rid
join num_ingredients n2 on lunch_rid = n2.rid
join num_ingredients n3 on dinner_rid = n3.rid
order by ingredients_per_meal) m2;

with rids as (
SELECT distinct breakfast_rid, lunch_rid, dinner_rid
from minimal_meal_plan
LIMIT 7
), breakfastTitle as (
select ri.*, r.title as breakfast_title
from rids ri join recipes r on ri.breakfast_rid = r.rid
), lunchTitle as (
select bt.*, r.title as lunch_title
from breakfastTitle bt join recipes r on bt.lunch_rid = r.rid
)
select lt.*, r.title as dinner_title
from lunchTitle lt join recipes r on lt.dinner_rid = r.rid;
```

For the minimal meal plan, we wanted to find all meal plans using minimal ingredients and distinct recipes. Because of that, before optimization, we had to insert each meal plan (from lowest number of ingredients) into a temporary table that had unique constraints so there would be no repeated recipes. However, inserting tuples into tables is slow, so we created a table with minimal meal plans that was ordered in ascending order of num ingredients used. The improved query just needed to retrieve the top 7 recipes, as recipes with minimal ingredients were already sorted to be at the top, and join with the recipe table to get titles of the weekly recipes and return.

Optimized low carb/high protein/keto

```
create table total_ntr_per_recipe_table as (
SELECT rid,total_calories, total_carbs, total_fat, total_protein, ((total_carbs*4)/total_calories) as carbs_ratio,
((total_protein*4)/total_calories) as protein_ratio,
((total_fat*9)/total_calories) as fat_ratio
FROM total_ntr_per_recipe
```

```
NATURAL JOIN (
SELECT rid, total_ntr_per_recipe AS total_calories
FROM total_ntr_per_recipe
WHERE id = 1008 AND total_ntr_per_recipe != 0
) AS calories
NATURAL JOIN (
SELECT rid, total_ntr_per_recipe AS total_carbs
FROM total_ntr_per_recipe
WHERE id = 1005 AND total_ntr_per_recipe != 0
) AS carbs
NATURAL JOIN (
SELECT rid, total_ntr_per_recipe AS total_fat
FROM total_ntr_per_recipe
WHERE id = 1004 AND total_ntr_per_recipe != 0
) AS fat
NATURAL JOIN (
SELECT rid, total_ntr_per_recipe AS total_protein
FROM total_ntr_per_recipe
WHERE id = 1003 AND total_ntr_per_recipe != 0
) AS protein
);
create index carb_ratio on total_ntr_per_recipe_table(carbs_ratio);
create index protein_ratio on total_ntr_per_recipe_table(protein_ratio);
create index fat_ratio on total_ntr_per_recipe_table(fat_ratio);

SELECT recipes.rid, recipes.title
FROM recipes
NATURAL JOIN (
SELECT DISTINCT rid
FROM total_ntr_per_recipe_table
WHERE total_ntr_per_recipe_table.carbs_ratio<= 0.25
LIMIT 75
) AS low_carb
WHERE title NOT LIKE '%dressing%'
order by rand()
LIMIT 10;

#optimized protein
select title, rid
from recipes
natural join (select distinct rid
from total_ntr_per_recipe_table
where total_ntr_per_recipe_table.protein_ratio>= .30
order by total_protein desc
limit 30) as high_protein
order by rand()
limit 10;

#optimized keto
select title, rid
from recipes
natural join (select distinct rid
from total_ntr_per_recipe_table
where total_ntr_per_recipe_table.fat_ratio>= .70 AND total_ntr_per_recipe_table.fat_ratio<= .80
limit 50
) as high_fat
WHERE title NOT LIKE '%cookie%' AND title NOT LIKE '%cake%'
order by rand()
limit 10;
```

Prior to optimization, the query ran a complex join that calculated the nutritional data of all recipes. We decided instead to store that information into a table since nutritional info of recipes was not going to change much, thus it made sense to add them to a table instead of calculating them each time. Additionally, it was useful to use that table over multiple queries. We also limited cardinality by taking only the top 30-75 of recipes since we only want the recipes that best match the criteria for these diets. Moreover, we precalculated the ratios of carb/calorie, fat/calorie, protein/calorie for each recipe and stored that in the table we created. Afterwards we added an index for macro to caloric ratio. With the index, now when we selected the top 30-75 recipes, the data was already indexed by the ratios, lowering the cost of reading tuples.

Optimized top 10

```
create index title_idx on recipes(title);

select recipes.rid, recipes.title
from (select rid, title from recipes) as recipes
natural join (select rid
from used_in
where iid in (select iid from used_in where rid = ${recipe_id})
and rid != ${recipe_id}
group by rid
order by count(*) desc)
as top_ten_similar
group by title
limit 10;
```

Since there were recipes with different ids but with duplicate titles in the recipes table, when finding similar recipes, we would get repeat titles (though different recipes). Originally, to prevent that from showing up, we grouped by title. Thus, creating an index on title helped lower run time. Additionally, we removed any attributes that were not needed in the joins/selections/projections, maximizing the number of tuples per block.

Technical Challenges:

We ran into some difficulties during data cleaning using Google Colab. Since our dataset is so large, performing matchings of USDA-ids to recipe ingredients and extracting ingredients, quantities, and units were both very time consuming, and we were only able to extract 102,000 recipes given the time constraints of Google Colab and the project deadline. Moreover, a lot of the cleaning process required complex regex matching, which was difficult to get a grasp of in the beginning of the process.

Github came in handy for the sharing of code between partner and version control. Towards the end of the project, we changed the schema of a table to improve performance, but this had the unintended effect of breaking one of our routes. We didn't know why the route was no longer at the time, but we were able to narrow down the possibilities by tracking the revisions made to the code on GitHub. After realizing that the route had never been changed, we knew that it must be an issue with the database itself.

During the web development phase of the project, we had trouble narrowing down why some pages of the web app were not working properly. Was the problem faulty code, an issue with query performance, or were simply not getting a response from the server at all? One tool that came in handy was using the inspector on Google chrome to monitor the network traffic. By looking at the fetches, we were able to see whether there was an error in the code, a pending fetch due to slow query performance, or a response from the server that was formatted differently than we expected. This really helped to speed up debugging.

Another issue that we ran into during the web development phase was the timing of fetches. For example, on the recipe info pages, the recipe's information is fetched using the recipe_id. The title of the newly fetched recipe information must then be used to search for an image on the UpSplash API. The image fetch must therefore occur after the fetch that retrieves the recipe information. Initially we had each fetch running concurrently and sometimes the page would load and other times we would get an error. It took a while for us to diagnose this problem and the solution, after a lot of tinkering, was to nest the image fetch inside of the recipe fetch in order to control the timing.

Overall, we found that SQL schema structure was very powerful in quickly matching foreign key referencing primary keys, which was helpful in lowering the runtime of our queries. However, we found that MySQL is rather limited in natural language processing and edit distance since it is a database language. The schema also proved to be limiting to some of the queries we wanted to perform, such as finding recipes that had the least number of distinct ingredients, which might require a NOSQL language such as Neo4j that stores relationships.