RecipeFinder

Yue Chen, Kaeli Kaymak-Loveless, Kailash Natarajan, Jacob Zimmerman

Introduction

In the contemporary digital landscape, users are often confronted with an overwhelming number of choices when searching for the ideal recipe. Websites like Food.com, for instance, host over 200,000 recipes, making it challenging for users to find a suitable dish unless they know precisely what they want. RecipeFinder aims to address this issue by providing a more personalized and efficient approach to recipe search and discovery. The primary motivation behind RecipeFinder is to simplify the recipe search process and cater to the unique preferences and dietary needs of each user. This platform is designed with the central goal of enabling users to find specific recipes based on customized parameters, thereby enhancing their overall culinary experience. RecipeFinder allows users to search for recipes using various parameters, such as ingredients, number of steps, preparation time, nutritional value, and more. By customizing the search criteria to match individual preferences and requirements, RecipeFinder facilitates the discovery of new dishes that align with users' distinct tastes. In addition to its core functionality, RecipeFinder encourages community engagement by providing users with the opportunity to share their creations, exchange suggestions, and connect with fellow food enthusiasts. This interactive environment fosters inspiration, creativity, and a shared passion for culinary exploration. RecipeFinder represents a significant advancement in recipe search and discovery by streamlining the process and personalizing it according to each user's unique tastes and preferences. By employing this innovative platform, users can overcome the challenges posed by an abundance of recipe choices and more efficiently navigate the vast culinary landscape.

Architecture

We used basic Python data frames to preprocess the raw data and save the new tables into CSVs. We created an Amazon AWS database to store the data from these tables and specify keys and foreign keys to relate the tables. We used React.js to interact with the database and create a server with a web interface that allows users to query the database themselves. We used a combination of javascript, HTML, and CSS to code our application.

Data

We found our datasets using Kaggle, a collaborative data science community with a large collection of datasets on various subjects. Our application uses two datasets, both of which can be accessed through this link:

https://www.kaggle.com/datasets/shuyangli94/food-com-recipes-and-user-interactions?select=RAW_recipes.csv

The two datasets that we utilize are *RAW_interactions.csv* and *RAW_recipes.csv.*

RAW_interactions.csv contains over one million reviews of recipes from specific users of Food.com. It has the following fields:

- user_id: the id of the user who created the review
- recipe_id: the id of the recipe that the user is reviewing
- date: the date that the review was written
- rating: a rating between 0 and 5 that the user gives the recipe under review
- review: the users review of the recipe

This dataset contains 5 fields (columns) and 1,125,284 unique reviews (rows). This dataset was used to allow our web application to filter recipes based on reviews. As an example, users can choose to only view recipes that have at least one rating that is at least as high as some specified value. By joining our two datasets on the id of the recipe, we can associate each recipe with all of its relevant reviews, allowing us to select recipes based on rating/review.

RAW_recipes.csv contains over 200,000 recipes from Food.com. It has the following fields:

- name: the name of the recipe
- id: the id of the recipe
- minutes: the number of minutes it takes to make the recipe
- contributor_id: the id of the user who submitted the recipe
- submitted: the date the recipe was submitted
- tags: a list of tags associated with the recipe (vegetarian, quick, easy, etc.)
- nutrition: a list containing number of calories, total fat (PDV), sugar (PDV), sodium (PDV), protein (PDV), saturated fat (PDV)
- n_steps: the number of steps in the recipe
- steps: Text for recipe steps, in order
- description: a description of the dish and recipe, created by the submitter
- ingredients: a list of the ingredients required for the recipe
- n_ingredients: the number of ingredients needed

This dataset contains 12 fields (columns) and 230,186 unique recipes (rows). This dataset gives us access to a large bank of recipes that can be displayed on a user's feed. With fields such as n_steps, nutrition, and ingredients, users can filter recipes based on how many steps the recipe requires, how nutritious the recipe is, and what ingredients the recipe requires. With that, users can avoid looking through a list of over 200,000 recipes, and simply view the recipes that match their specific requirements. As stated earlier, by joining our two datasets on the id of the recipe,

we can associate each recipe with all of its relevant reviews, allowing us to select recipes based on rating/review. This dataset is the core of our web application.

Database

RAW_recipes.csv and RAW_interactions.csv are downloaded from Kaggle and unzipped into a directory called "data." RAW_recipes.csv is loaded into a Python dataframe called df_recipe_raw. The tags, nutrition, steps, and ingredients column entries are each converted into arrays to allow easier access to the items within each of these entries, and the submitted column is converted to datetime to make calendar calculations easier. RAW_interactions.csv is loaded into a dataframe called df_rating_raw, and the date column is converted to datetime like before.

To keep track of recipes, the id, name, contributor_id (renamed to user_id), submitted (renamed to date), description, minutes, n_steps, and n_ingredients from df_recipe_raw are loaded into a new dataframe called df_recipe. Then, to separate the items in each entry of the original nutrition columns, an enumerate object over the 6 items in every array entry of the nutrition column is created, and an apply function is used to load the corresponding items from each nutrition array into their new respective columns in df_recipe (instead of having one nutrition column containing arrays of calories, fat, etc. values, these values are now split into their own columns). Recipes with null names or minutes>600 are removed. Null descriptions are converted to blank strings, and the tokens in each recipe name are capitalized. The result is written to recipe.csv.

To keep track of ratings, df_rating is created by keeping all rows from df_rating_raw where the recipe_id value of df_rating_raw matches some id value in df_recipe. Null reviews are replaced with blank strings, and the result is written to rating.csv.

To keep track of ingredients, the ingredients column in df_recipe_raw is exploded into a list of all ingredients across all entries in this column, and duplicates are removed. The ingredients are lemmatized to find common root words (so ingredients that are the same but identified slightly differently in 2 separate ingredient lists will still be classified as the same ingredient), and the lemmatized versions are added to a list called ingredients_clean, with duplicates removed. The lemmatized versions are also added as values in a map called ingredients_clean_map, where the keys are the original ingredient names. The elements of ingredients_clean are enumerated and entered into a map where the key is the ingredient name and the value is the corresponding number from the enumerate object. Finally, df_ingredient is created with an "id" column that stores the map values and an "ingredient" column that stores the map keys. The result is written to ingredient.csv.

To keep track of ingredients in each recipe, a new dataframe called df_recipe_ingredient is created by exploding the ingredients list for each id so that each recipe now has one row per ingredient used. The ingredient names are mapped to ingredient ids using the maps created

above, the id column is renamed to recipe_id, and the ingredients column is renamed to ingredient_id. Finally, all rows where the recipe_id doesn't appear in df_recipe are removed, and the result is written to recipe_ingredient.csv.

To keep track of tags, the tags column from df_recipe_raw is exploded into a list, removing duplicates. The tags are mapped to numbers using an enumerate object, and a dataframe called df_tag is created with a column "id" containing the map's values (the numbers) and a column "tag" containing the map's keys (the names). The result is written to tag.csv.

To keep track of tags associated with each recipe, we perform the exact same process as for df_recipe_ingredient, but using the tags column and the tags map. The result is written to recipe_tag.csv.

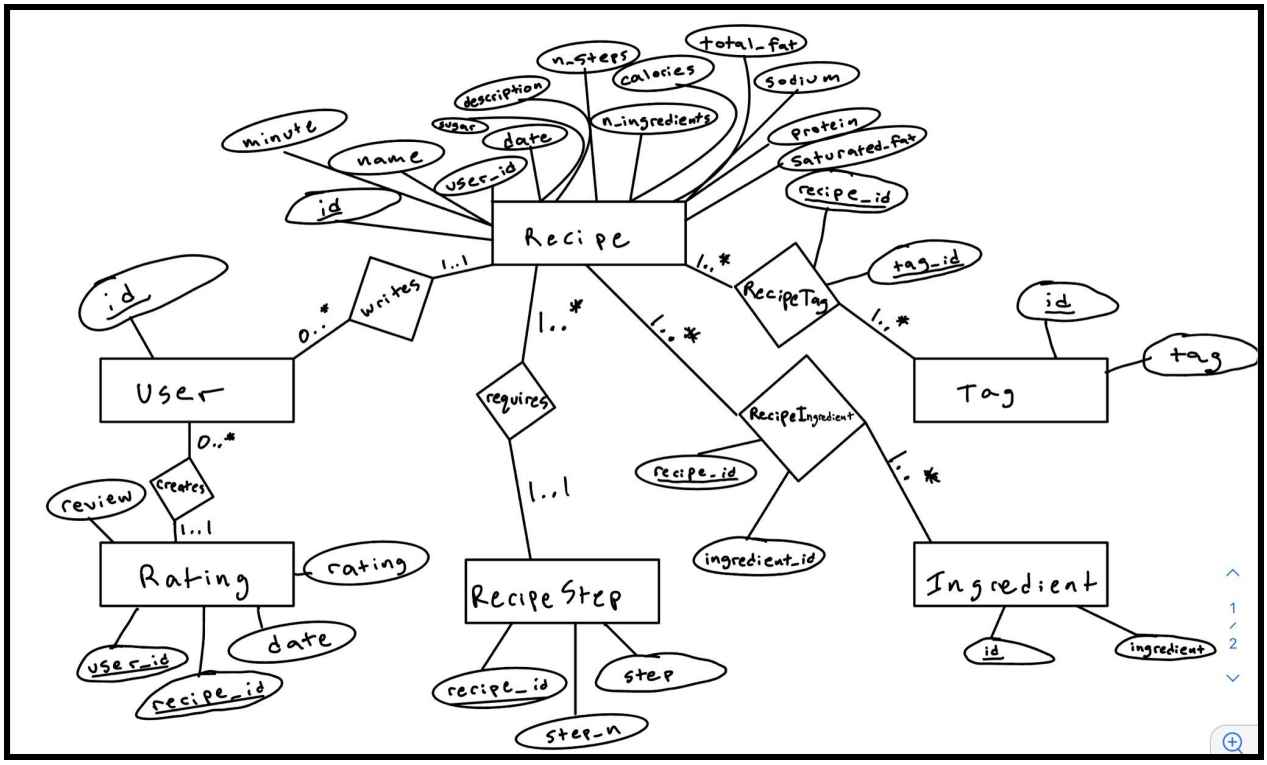
To keep track of recipe steps, we explode the steps entry associated with each recipe id (similar to how we exploded the ingredients column by id when created recipe_ingredient.csv) into a dataframe called df_recipe_step. Rows containing null steps or steps shorter than 3 characters are dropped, and a step_n column is created by using groupby and cumcount functions to assign an incremental count to each step in a recipe. The result is written to recipe_step.csv.

To keep track of users, a set (without duplicates) is created that combines the contributor_ids from df_recipe_raw and the user_ids from df_rating_raw. This is written to user.csv.

Each of the CSVs produced during this preprocessing match up directly with the tables defined in the schema.

We performed entity resolution during this preprocessing by making sure that ingredient ids and names were standardized through lemmatization (an ingredient will always be associated with the same id and standard name, no matter in what format it appeared in the original recipe). This makes sure that the ingredients in the Ingredient table and the RecipeIngredient table can be properly matched up.

ER Diagram



Schema

User (<u>id</u>)
# of instances: 236719 instances

Recipe (<u>id</u>, name, user_id, date, description, minute, n_steps, n_ingredients, calories, total_fat, sugar, sodium, protein, saturated_fat)

- user_id FOREIGN KEY REFERENCES User (id)

# of instances: 228172 instances

RecipeStep (<u>recipe_id</u>, step, step_n)

- recipe_id FOREIGN KEY REFERENCES Recipe (id)

# of instances: 696005 instances

Ingredient (<u>id</u>, ingredient)
# of instances: 13755 instances

RecipeIngredient (<u>recipe_id</u>, <u>ingredient_id</u>)

- recipe_id FOREIGN KEY REFERENCES Recipe (id)
- ingredient_id FOREIGN KEY REFERENCES Recipe (id)

# of instances: 2064986 instances

Tag (<u>id</u>, tag)
# of instances: 551 instances

RecipeTag (recipe_id, tag_id)
- recipe_id FOREIGN KEY REFERENCES Recipe (id)
- tag_id FOREIGN KEY REFERENCES Tag(id)

# of instances: 4078500 instances

Rating (user_id, recipe_id, date, rating, review)
- user_id FOREIGN KEY REFERENCES User (id)
- recipe_id FOREIGN KEY REFERENCES Recipe (id)

# of instances: 1114719 instances

Third Normal Form (3NF) Justification

The schema above is already in its third normal form. Each table adheres to the conditions required for 3NF. Recall that a non-prime attribute is an attribute that is a part of no candidate keys. In the case of the User table, there is only a single attribute (id), which serves as the primary key with no non-prime attributes. The Recipe table is also in 3NF because all non-prime attributes are fully functionally dependent on the primary key (id), and the foreign key user_id establishes a relationship with the User table without introducing any transitive dependencies. The RecipeStep table satisfies 3NF, as the primary key is recipe_id, and the other attributes (step, step_n) are dependent on the recipe_id without any transitive dependencies. Similarly, the Ingredient table adheres to 3NF because there are only two attributes (id and ingredient), with the primary key (id), uniquely determining the ingredient name. The RecipeIngredient table is in 3NF, as it is a linking table with a composite primary key (recipe_id and ingredient_id) that represents a many-to-many relationship between the Recipe and Ingredient tables, with no non-prime attributes or transitive dependencies. The Tag table is also in 3NF since there are only two attributes (id and tag), and the primary key (id) determines the tag name uniquely. The RecipeTag table follows 3NF for a reason similar to that of the RecipeIngredient table. Lastly, the Rating table is in 3NF, as it has a composite primary key (user_id and recipe_id), with non-prime attributes (date, rating, and review) being fully dependent on the primary key and no transitive dependencies present.

Queries

Query #1:

```
WITH TopContributors AS (
  SELECT
      r.user_id
  FROM Recipe r
  JOIN Rating rt ON rt.recipe_id = r.id
  GROUP BY r.user_id
  HAVING COUNT(DISTINCT r.id)>=20
  AND AVG(rt.rating)>=4.5
)
SELECT
  r.id,
  r.name,
  r.description,
  r.minute,
  r.n_steps,
  r.n_ingredients,
  r.calories,
  r.average_rating,
  r.n_ratings,
  COUNT(DISTINCT tc.user_id) / (SELECT COUNT(user_id) FROM TopContributors) num_top_contributors
FROM Recipe r
JOIN Rating rt ON rt.recipe_id = r.id
JOIN TopContributors tc ON tc.user_id = rt.user_id
WHERE r.user_id NOT IN (SELECT tc.user_id FROM TopContributors)
GROUP BY r.id, r.name
HAVING COUNT(DISTINCT tc.user_id) >= 0.05 * (SELECT COUNT(user_id) FROM TopContributors)
ORDER BY average_rating DESC, num_top_contributors DESC
LIMIT ${numResults}
;
```

This query finds recipes rated by top contributors (user's who have submitted a large amount of recipes). We define top contributors as the top 1000 users who have submitted the largest number or recipes. We use this query to display recipes rated by top contributors. The CTE finds the top contributors, and the main query returns the id of the recipe, its name, average rating, number of ratings, and number of top contributors who gave it a review.

Query #2:

```
WITH GivenRecipe AS (
  SELECT DISTINCT *
  FROM Recipe r
  JOIN RecipeIngredient ri on r.id = ri.recipe_id
  WHERE r.id = ${recipe_id}
)
SELECT
  r.id,
  r.name,
  AVG(rt.rating) as avg_rating,
  r.calories,
  COUNT(DISTINCT ri.ingredient_id) AS num_common_ingredients
FROM Recipe r
JOIN RecipeIngredient ri on r.id = ri.recipe_id
JOIN Rating rt ON rt.recipe_id = r.id
WHERE r.id <> ${recipe_id}
AND r.calories BETWEEN (SELECT AVG(0.9 * calories) FROM GivenRecipe) AND (SELECT AVG(1.1 * calories) FROM GivenRecipe)
AND ri.ingredient_id IN (
  SELECT DISTINCT ingredient_id
  FROM GivenRecipe
)
GROUP BY r.id, r.name
ORDER BY num_common_ingredients DESC, avg_rating DESC
LIMIT 5
```

This query finds 5 similar recipes to a given recipe (specified by ${recipe_id}) based on common ingredients and calorie range while also considering their average ratings. The CTE GivenRecipe selects all distinct rows from the Recipe and RecipeIngredient tables where the recipe ID matches the provided ${recipe_id}. The main query filters the results with the following conditions - The recipe ID should not be equal to the provided ${recipe_id}, the recipe's calories should be within 90% and 110% of the given recipe's average calories, the recipe should have at least one common ingredient with the given recipe. This query is used when a user selects some recipe. On the page associated with that recipe, our application lists similar recipes, hence the need for this query.

Query #3:

```
WITH Tag AS (
    SELECT * FROM Tag ORDER BY Tag LIMIT 10
)
SELECT
    T.id,
    T.tag,
    COUNT(RT.recipe_id) AS num_recipe,
    AVG(R2.rating) AS avg_rating,
    COUNT(R2.rating) AS num_rating,
    AVG(R2.rating) * COUNT(R2.rating) / (SELECT COUNT(DISTINCT user_id) FROM Rating) AS score
FROM Tag T
JOIN RecipeTag RT ON T.id = RT.tag_id
JOIN Rating R2 ON R2.recipe_id = RT.recipe_id
GROUP BY T.id, T.tag
ORDER BY 6 DESC
LIMIT 10
;
```

This query finds the top 10 tags out of the first 10 alphabetically ordered tags, ranked by a calculated popularity score. The popularity score is based on the average rating of the recipes associated with each tag, the number of ratings for these recipes, and the total number of distinct users who have rated recipes. Simply, it finds the 10 most popular tags, along with the number of recipes associated with that tag. This query is used in the Recipe Search page, where this query gives users the option to filter by popular tags.

Query #4:

```
  SELECT
    R.id,
    R.name,
    ROUND(RR.avg_rating, 1) AS avg_rating,
    RR.num_rating
  FROM Recipe R
  JOIN RecipeRating RR
    ON RR.id = R.id
  WHERE RR.name LIKE '${name}'
  `;
if (ingredientsCount > 0) {
  query += `
    AND R.id IN (
      SELECT DISTINCT recipe_id
      FROM RecipeIngredient
      WHERE ingredient_id IN (
        SELECT id
        FROM Ingredient
        WHERE ingredient IN (${ingredients.map((i) => `'${i}'`).join(',')})
      )
      GROUP BY recipe_id
      HAVING COUNT(DISTINCT ingredient_id) = ${ingredientsCount}
    )
  `;
}
query += `
  AND R.n_ingredients <= ${maxNumIngredient}
  AND R.n_steps <= ${maxNumSteps}
  AND R.minute <= ${maxTime}
  AND R.calories <= ${maxCalories}
  AND RR.avg_rating >= ${minRating}
  AND RR.num_rating >= ${minNumRatings}
  ORDER BY n_ratings DESC, average_rating DESC
  LIMIT ${pageSize}
`;
```

This query searches for recipes based on multiple criteria, such as name, ingredients, maximum number of ingredients, steps, time, and calories, as well as minimum average rating and number of ratings. The query is constructed dynamically based on the input values provided. It filters and sorts the results based on these criteria, with the option to include specific ingredients if desired. The final output is limited by a given page size. This query aims to provide a flexible search functionality for users to find recipes matching their preferences and requirements. This query is used in our Recipe Search page, where users can specify parameters regarding the number of ingredients in the recipe, the number of steps, how long it takes to make, etc. This is a core part of our web application and simplifies searching for recipes.

Query #5

```
SELECT
    id,
    name,
    ROUND(avg_rating, 1) AS avg_rating,
    num_rating
FROM RecipeRating
ORDER BY score DESC
LIMIT 10
```

This query finds the id, name, average rating, and number of ratings of the top 10 highest rated recipes (by average). This query is essential to our top recipes page, where we display the most popularly rated recipes.

Performance Evaluation

We designed most of our queries very efficiently to begin with, but there was one query in particular that was causing us issues: the top contributors query (used in general search functionality). This query first finds "top contributors" by filtering for user_ids from Recipe that appear at least 20 times and have an average rating of at least 4.5 across all of the recipes they have submitted. Then, it selects all recipes that have been rated by at least 5% of these "top contributors." Since the average_rating column in Recipe was not associated with an index, it was taking longer than necessary to sort through the recipes to find the ones with average ratings above 4.5. Similarly, there wasn't an index on user_id, so it was taking longer than necessary to match up user_ids between Recipe, Rating, and the TopContributors CTE. As such, we added 2 indexes: one for each of these columns. These datasets are absolutely massive, so even with the optimization performed, this query is still quite slow. However, it improved from about a 14 second runtime to around a 10 second runtime. This improvement arose from now being able to avoid full table scans and use index scans (eg. Index Range Scan on the average_rating index) to filter data.

Technical Challenges

One of the toughest challenges that we faced was in regards to entity resolution. The issue was that some versions of the same ingredient were not recognized as representing the same entity (perhaps due to misspellings or uncommon descriptors), so issue arose when querying for ingredients. For example, two recipes that contained the same ingredient may not be returned as such because one of the recipes may have misspelled the ingredient, causing it to be associated with a different ingredient id. To overcome this issue, we used lemmatization while

pre-processing our data. Lemmatization is the process of grouping together different forms of the same word, or in this case, ingredient. This also ensured that the ingredients in the Ingredient table and the RecipeIngredient table will properly match.

Another challenge we faced was in regards to query speed and optimization. Given that one of our datasets contains over 200,000 rows, and the other dataset contains over 1 million rows, some of the queries that scanned the entirety of a table took up to 10 seconds. Obviously, this is not acceptable for our web application, so we created indexes to reduce the amount of unnecessary scanning, subsequently decreasing the runtimes of our queries. As an example, one page of our application displays highly rated recipes from the top contributors (those who have submitted a large amount of recipes). To speed up this query, we created two indexes: one on user_id and one on average_rating.

Finally, our group lacked experience in front-end development, specifically React. This made it difficult to enhance the UI aspect of our project, as well as properly display the results of our queries. In an effort to gain a better understanding of front-end development, we relied on youtue explanations, the examples provided on Canvas, and other resources such as StackExchange and GeeksForGeeks.