What's Cooking

A project report presented to Professor Neamtu in fulfillment of the Final Project Requirement for CS 542 - Database Management Systems @ WPI

Sean Arackal*, Alexander Bell*, Spencer Greene*, Axel Luca*, Jamie Ortiz*
*Worcester Polytechnic Institute
{sarackal, abell, ssgreene, aluca, jcordova}@wpi.edu

***Abstract*—This web application allows users to view high-quality recipes so they can use them as inspiration to come up with their own recipes using ingredients from those original high-quality recipes. Utilizing scraped data from the popular cooking website "AllRecipes.com", our system stores and organizes recipe information, including ingredients, cooking time and more.**

We further add to this database by allowing users to sign up for our platform. We then allow users to save or like recipes as well as leave reviews and ratings on other recipes. We keep track of all this on our database backend. We keep users connected to recipes through our database so we can also track the average rating for each recipe as well as the total number of users that have saved the recipe.

Storing all this information helps us write queries, views, and triggers that allow us to sort the recipes in a certain way, prevent erroneous or repetitive data in our database as well as allowing users to view the information they need in the most effective way possible.

I. Introduction

"What's Cooking" will be a web application that serves as a community-driven forum for recipe creation.

We will start by scraping data related to recipes and ingredients from pre-existing cooking websites like Allrecipes [2]. Once we have this data, we will clean it and store it on our modeled database. The database will contain information such as ingredients, cook/preparation time, users, logins, their comments and ratings on recipes, saved recipes as well as existing recipes for users to take inspiration from. Our system's targeted user group are people looking for new recipes, or people who enjoy cooking.

Once we have set up our back end, we will begin creating and stylizing our web application itself, allowing opportunities for users to interact with the dataset. The website can be found at https://whats-cooking.fyi/.

The main data model includes users, recipes, ingredients, and reviews. Users will have unique profiles containing email addresses and passwords. Recipes are the main feature and they will consist of the recipeID, recipe name, prep time, cooking time, servings, calories, and directions. Ingredients will be separate from each other, with the ingredient name being the primary key.

Users can rate and comment on the recipes. Each review will be between the user and a specific recipe, using a many-to-many relationship between users and recipes. Users can interact with different recipes and each recipe can get feedback from all users. The DB will support queries such as retrieving the recipes that were previously saved, and the submitted recipes, and also to see the comments and ratings on each recipe.

It will also allow users to search for recipes by keywords, ingredients, and prep time. Users will also have the ability to create, read, update, and delete their own recipes and reviews. There will be a user interface and an admin interface.

II. Background Materials

In order to successfully complete, we decided to use MySQL as our backend. Some of the primary reasons for this choice were that MySQL is open-source, that it is used in multiple kinds of applications, its ease of setup, and the features it provides. To be more precise, the fact that MySQL is open-source is what allowed us to easily download and set up some of its products such as MySQLWorkbench, and make a MySQL server that could be shared remotely across different computers. In addition to this, examples of applications that rely on MySQL for functionality include Netflix, Uber, and Airbnb [1]. Moreover, MySQL provides several functionality features that we think are essential for our project such as views and triggers [3]. Along with those, some additional features we thought would be useful for that project that MySQL includes are a LIMIT clause that allows users to limit the amount of rows shown when running queries, and an IFNULL function that allows users to specify a default value to return if a desired result returns NULL [4, 5].

We are using React for this project because it is very flexible and it is not too complicated for us to use. Some of us are learning the technologies and the ones that already know them are helping others learn. [6]

We are using TypeScript with react because we thought we are going to benefit from type safety, preventing errors, and this way we also learn to use TypeScript appropriately. React combined with TypeScript is useful for us because we can do easy debugging and it has good performance.

We decided to use Express because it is simple and flexible, and it allows us to make a quick set up of routes. This way we have good communication between frontend and backend, and add some level of error handling.

We are also using Vite because it is efficient at bundling. We made use of Hot Module Replacement (hmr) which was really useful because we could instantly reflect the changes we made to the code on the web. We didn't have problems with it and it was decently fast.

The server's backend runs a Caddy web-server which does two things. One, it acts as a reverse proxy, serving the internally running node app to the external server address. Secondly, Caddy automatically generates HTTPS certificates so that the traffic on the website can use the more secure protocol. In terms of the internal node app, this is run using a service called pm2 which allows us to run a node application in the background forever.

III. Methods

A. Data Collection and Storage

We will start by scraping data related to recipes and ingredients from pre-existing cooking websites like AllRecipes. Once we have this data, we will clean it and store it on our modeled database. The database will contain information such as ingredients, cook/preparation time, users, logins, their comments and ratings on recipes, saved recipes as well as existing recipes for users to take inspiration from.

B. Backend Architecture

We built the app using Node.js with Express.js to handle HTTP requests and manage API endpoints. We set a MySQL connection pool using mysql2/promises, this will make it efficient and it can handle multiple simultaneous requests. API endpoints like /recipes are made to retrieve recipe data with ingredients and directions.

C. Testing Triggers & Views

We implement several triggers and views to ensure the success and validity of our application. These triggers and views are tested, and the results are shown here.

- Deleting a high quality recipe (a recipe with a rating over 4.0) fails. In Figures 1 and 2, we see the deletion failing, as well as the recipe being maintained in the database table respectively. Figure 3 shows the code for this trigger.

Fig. 1. Deleting a record with over a 4.0 average rating fails.

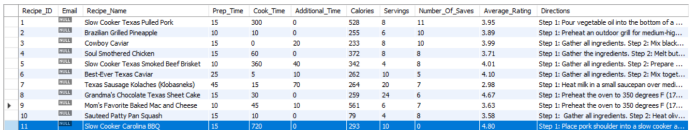

Fig. 2. Recipe with the ID 11 still exists in the tables.

- In this view, the top ten rated recipes are displayed and arranged by descending average rating.In 4 this view is tested. In testing this view, we also tested a trigger that updates the average user rating on a recipe as reviews are being left on the recipe. The code for both the view and the trigger are provided in 5 and 6 respectively.
- In this view, we tested that the top 10 recipes, arranged by descending number of saves, would display properly,

```
CREATE TRIGGER cannot_delete_high_quality_recipes
BEFORE DELETE ON Recipe
FOR EACH ROW
BEGIN
    IF OLD.Average_Rating >= 4.0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Cannot delete high quality recipe';
    END IF;
END//
```

Fig. 3. The code for this trigger

Recipe_ID	Recipe_Name	Average_Rating
11	Slow Cooker Carolina BBQ	4.80
8	Grandma's Chocolate Texas Sheet Cake	4.67
6	Best-Ever Texas Caviar	4.10
5	Slow Cooker Texas Smoked Beef Brisket	4.01
3	Cowboy Caviar	3.99
1	Slow Cooker Texas Pulled Pork	3.95
2	Brazilian Grilled Pineapple	3.89
4	Soul Smothered Chicken	3.71
9	Mom's Favorite Baked Mac and Cheese	3.63
10	Sauteed Patty Pan Squash	3.58

Fig. 4. This view shows the top 10 recipes by average rating.

```
CREATE OR REPLACE VIEW top_10_recipes_by_average_rating AS
SELECT Recipe_ID, Recipe_Name, Average_Rating
FROM Recipe
WHERE Average_Rating IS NOT NULL
ORDER BY Average_Rating DESC
LIMIT 10;
```

Fig. 5. The code for this view

```
CREATE TRIGGER update_average_rating
AFTER INSERT ON User_Review
FOR EACH ROW
BEGIN
    UPDATE Recipe
    SET Average_Rating = (SELECT AVG(Rating)
    FROM User_Review
    WHERE Recipe_ID = NEW.Recipe_ID)
    WHERE Recipe_ID = NEW.Recipe_ID;
END//
```

Fig. 6. The code for this trigger

as seen figure 7. This also checks that the trigger which increments number of saves for a recipe is working properly. The code for the view and trigger are provided in 8 and 9 respectively.

Recipe_ID	Recipe_Name	Number_Of_Saves
1	Slow Cooker Texas Pulled Pork	11
2	Brazilian Grilled Pineapple	10
3	Cowboy Caviar	10
4	Soul Smothered Chicken	8
5	Slow Cooker Texas Smoked Beef Brisket	8
10	Sauteed Patty Pan Squash	8
7	Texas Sausage Kolaches (Klobasneks)	7
9	Mom's Favorite Baked Mac and Cheese	7
8	Grandma's Chocolate Texas Sheet Cake	6
6	Best-Ever Texas Caviar	5

Fig. 7. This view shows the top 10 recipes by number of saves.

```
CREATE OR REPLACE VIEW top_10_recipes_by_number_of_saves AS
SELECT Recipe_ID, Recipe_Name, Number_Of_Saves
FROM Recipe
WHERE Number_Of_Saves > 0
ORDER BY Number_Of_Saves DESC
LIMIT 10;
```

Fig. 8. The code for this view

```
CREATE TRIGGER update_num_saves_when_user_saves_recipe
AFTER INSERT ON User_Saved_Recipe
FOR EACH ROW
BEGIN
    UPDATE Recipe
    SET Number_Of_Saves = Number_Of_Saves + 1
    WHERE Recipe_ID = NEW.Recipe_ID;
END//
```

Fig. 9. The code for this trigger

- In this view, we tested for the top 10 healthiest recipes arranged according to the least number of calories would be properly displayed. It is worth noting that these recipes are typically sauces or seasoning. These can be seen in figure 10. The code for this view can be seen in 11.
- Trigger tested for Review ID incrementing as new reviews are created. This trigger exists because due to MySQL limitations, we could not use an auto-increment function for them. See figure 12. There's also a very similar trigger for auto-incrementing the Recipe_ID's. The code for both of these trigger can be seen 13 for reviews, and 14 for recipes.
- Updating the number of saves for when a user unsaved a recipe. In this example, 2 different users unsaved recipe 10, dropping it from 8 saves to 6 saves See figure 7 for

Recipe_ID	Recipe_Name	Calories
200	The Famous Seafood Seasoning Recipe	1
143	Eastern North Carolina BBQ Sauce	4
229	Authentic Mexican Hot Sauce	5
491	Grandma Oma's Pickled Okra	10
144	Vinegar Based BBQ Sauce	11
246	Fresh California Salsa	12
735	Homemade Taco Seasoning Mix	12
160	Carolina-Style Mustard BBQ Sauce	13
479	Red Chile Paste	13
18	D's Famous Salsa	16

Fig. 10. This view shows the 10 lowest calorie dishes.

```
CREATE OR REPLACE VIEW top_10_healthiest_recipes AS
SELECT Recipe_ID, Recipe_Name, Calories
FROM Recipe
WHERE Calories IS NOT NULL
ORDER BY Calories ASC
LIMIT 10;
```

Fig. 11. The code for this view

Email	Recipe_ID	Review_ID	Comment	Rating
user1@example.com	1	1	Very good	4.50
user1@example.com	2	2	Delicious recipe!	5.00
user1@example.com	3	3	Not bad, but could improve.	3.00
user1@example.com	4	4	Loved it!	4.80
user1@example.com	5	5	Too salty for my taste.	2.00
user2@example.com	1	6	Very easy to make!	4.70
user2@example.com	2	7	Simple and tasty.	4.00
user2@example.com	3	8	Good for meal prep!	4.60
user2@example.com	4	9	Very filling.	4.30
user2@example.com	5	10	Great for parties!	4.60
user3@example.com	1	11	Average dish.	3.00
user3@example.com	2	12	Too sweet for my liking.	2.20

Fig. 12. The Review_ID is incremented for each review.

```
CREATE TRIGGER generate_next_review_id
BEFORE INSERT ON User_Review
FOR EACH ROW
BEGIN
    DECLARE next_review_id INT;

    SET next_review_id = 0;

    SELECT IFNULL(MAX(Review_ID), 0) INTO next_review_id
    FROM User_Review;

    SET NEW.Review_ID = next_review_id + 1;
END//
```

Fig. 13. The code for this trigger

```
CREATE TRIGGER generate_next_recipe_id
BEFORE INSERT ON Recipe
FOR EACH ROW
BEGIN
    DECLARE next_recipe_id INT;

    SET next_recipe_id = 0;

    SELECT IFNULL(MAX(Recipe_ID), 0) INTO next_recipe_id
    FROM Recipe;

    IF NEW.Recipe_ID IS NULL OR NEW.Recipe_ID = 0 THEN
        SET NEW.Recipe_ID = next_recipe_id + 1;
    END IF;
END//
```

Fig. 14. The code for this trigger

before. For after, see figure 15. For the code for when a user unsaves a recipe, look for figure 16

	Recipe_ID	Recipe_Name	Number_Of_Saves
▶	1	Slow Cooker Texas Pulled Pork	11
	2	Brazilian Grilled Pineapple	10
	3	Cowboy Caviar	10
	4	Soul Smothered Chicken	8
	5	Slow Cooker Texas Smoked Beef Brisket	8
	7	Texas Sausage Kolaches (Klobasneks)	7
	9	Mom's Favorite Baked Mac and Cheese	7
	8	Grandma's Chocolate Texas Sheet Cake	6
	10	Sauteed Patty Pan Squash	6
	6	Best-Ever Texas Caviar	5

Fig. 15. Updating the number of saves when a user unsaves a recipe.

```
CREATE TRIGGER update_num_saves_when_user_unsaves_recipe
AFTER DELETE ON User_Saved_Recipe
FOR EACH ROW
BEGIN
    UPDATE Recipe
    SET Number_Of_Saves = Number_Of_Saves - 1
    WHERE Recipe_ID = OLD.Recipe_ID;
END//
```

Fig. 16. The code for this trigger

IV. The Database

We present the iterations of our database, specifically how the entity-relation diagram evolved from initial draft to how it now appears.

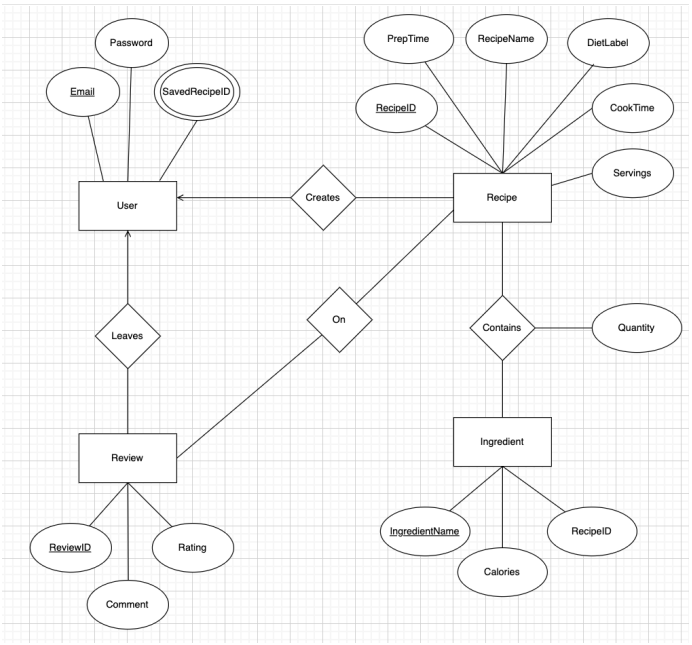

Fig. 17. Our initial ERD.

A. The Initial ERD

The initial ERD is found in. Figure 17 While this ER diagram supports all of the functionality we wish to include in our application, it introduces some redundancy. Fortunately, after conducting further analysis, we realized that we could minimize this redundancy while maintaining all of the required functionality of our app with very small changes to our ER diagram. To be more precise, we first removed our diagram's "Review" entity along with any relationships that connect to it. Once that was done, we created an additional many-to-many relationship called "Reviews" between our diagram's "User" and "Recipe" entities, and then added all the attributes of the original "Review" entity to this new relationship.

B. A Revised ERD

This schema, found in 18 is an improvement from our first one, we cleaned recipe, recipe content, and ingredient tables, and thought about what kind of functionality we will need in the future. We once again modified our database schema to be as seen in the next section.

C. Final ERD

The final entity-relation diagram is seen in Figure 19. The first modification we did on our schema was moving the "Calories" ingredient from the "Ingredient" table to the "Recipe" table. The reason why we did this is because when scraping the recipe data needed to make our application functional, all of the recipes had a total calorie count rather than one calorie count for each of their individual ingredients. Something else we noticed while scraping the data was that there were some recipes that used the same ingredients with the same units and quantities more than once. Therefore, to

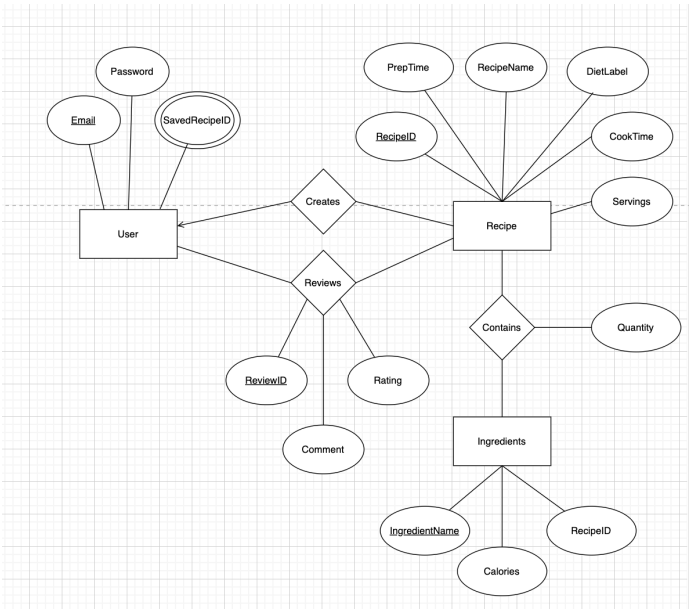

Fig. 18. Our revised ERD.

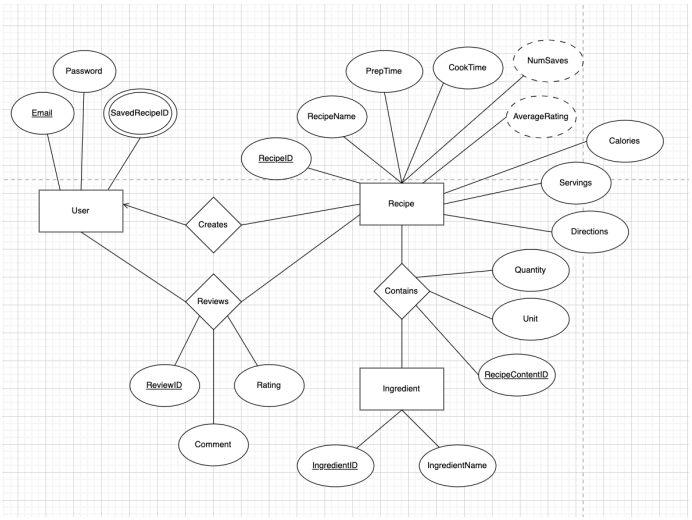

Fig. 19. Our third, and most recent version of the ERD.

help deal with this, we decided to add a primary attribute called "RecipeContentID" to the "Contains" relationship between the "Recipe" and "Ingredient" tables that would properly distinguish such recipe ingredient (content) data instances and thus allow them to be inserted in the database even if they appeared multiple times within the same recipe with the same quantity and units. In addition to these modifications, we also decided to add an "IngredientID" attribute to our "Ingredient" table. The main reason for this is that, while user emails usually have a consistent structure, the structure of ingredient names can vary much more in terms of their length and structure. Therefore, using the newly created "IngredientID" attribute to identify attributes provides a more efficient way to identify them in our database regardless of the length and structure of their names. Once this was done, we removed the "RecipeID" attribute from the "Ingredient" table due to the fact that the "Contains" relationship between the "Recipe" and "Ingredient" tables already captures the information needed regarding which ingredients are used in which recipes. Once we finished modifying the "Contains" relationship and the "Ingredient" table to make them more accommodating with the data we scraped and to remove any redundant attributes from them, we decided that some changes to our schema's "Recipe" table would also be beneficial. To begin, while we knew that we had all of the information needed regarding the ingredients associated with recipes, we thought it would also be beneficial for users to have access to step-by-step instructions they can use and follow to help them successfully create the dishes resulting from those recipes. When scraping those instructions, we realized that while we needed to keep track of multiple things regarding ingredients associated with recipes, that was not the case for each recipe's instructions to follow. Thanks to this, we determined that we'd be able to store each recipe's instructions to follow in the form of one single TEXT attribute called "Directions" within the "Recipe" table. Along with this attribute, we also decided to create derived attributes within our "Recipe" table called "AverageRating" to keep track of a recipe's average rating based on associated customer reviews, and 'NumSaves' to track the number of times a recipe is currently saved by users. While we realize that this information can be obtained by joining the appropriate tables, we intend to fetch it frequently to keep track of the current recipes with the most user saves and the highest average ratings. Additionally, as the tables in this database grow, queries to retrieve this information will take longer to fully execute. Therefore, because this is information that we often need to refer to for our project, we decided to store this information as derived attributes in our 'Recipe' table, along with triggers that automatically update the related values to ensure faster query retrievals compared to using joins across multiple tables.

D. Queries

1) Data Retrieval Queries
   - These queries are used to fetch specific data from the database. For example, retrieving all records of students enrolled in a particular course.
   - Our application fetches data from our database to display information like recipes, users, ingredients and more.
2) Aggregation Queries
   - These queries perform calculations on data, such as summing, averaging, or counting. For example, calculating the average grade of students in a course.
   - We have added statistics like average review rating for each recipe which will help users find high quality recipes.
3) Update Queries
   - These queries modify existing data in the database. For example, updating the contact information of a

student.
- Recipe and User data is frequently updated throughout the use of the application, when new recipes are created as well as reviews being left on recipes.

4) Join Queries
   - These queries combine data from multiple tables based on a related column. For example, retrieving student information along with their course details.
   - This is fulfilled on the recipe page, which retrieves the recipe's data as well as information on the recipe contents and ingredients.
5) Filter Queries
   - These queries apply conditions to filter data. For example, retrieving all students who scored above a certain grade in a course.
   - Our recipe builder page includes a filter for users to parse through our database based on different criteria. For example, Ingredients, Cook/Prep Time, Servings, etc.

E. Normalization Analysis

In our database, we have modeled a total of 6 tables using our ER diagram. We will now conduct a normalization analysis for each one of those tables.

1) User Table: In this table, as each email is unique and is associated with a password, a functional dependency for this table is Email → Password. In this table, as all of the functional dependencies have a super key on their left-hand side, this table is in BCNF form. See figure 20 for the full format of the table in the SQL database.

```
CREATE TABLE User (
Email VARCHAR(250),
Password varchar(250),
CONSTRAINT PK_USER PRIMARY KEY (Email)
);
```

Fig. 20. The format of the user table in the SQL database

2) Recipe Table: In this table, as each Recipe_ID is unique and is associated with all of the other attributes for an individual row, functional dependencies for this table are Recipe_ID → all subsets of the attributes Email, Recipe_Name, Prep_Time, Cook_Time, Additional_Time, Calories, Servings, Number_Of_Saves, Average_Rating, Directions. Now, as there is no guarantee that the value of any other set of attributes in this table will uniquely map to the value of a different set of attributes in this table, there are no other functional dependencies for this table. In this table, as all of the functional dependencies have a super key on their left-hand side, this table is in BCNF form. See figure 21 for information on the recipe table in the database.

```
CREATE TABLE Recipe (
    Recipe_ID INT,
    Email VARCHAR(250),
    Recipe_Name VARCHAR(250),
    Prep_Time INT,
    Cook_Time INT,
    Additional_Time INT,
    Calories INT,
    Servings INT,
    Number_Of_Saves INT,
    Average_Rating DECIMAL(10,2),
    Directions TEXT,
    CONSTRAINT PK_Recipe PRIMARY KEY(Recipe_ID),
    CONSTRAINT FK_Recipe FOREIGN KEY(Email) REFERENCES User (Email) ON DELETE CASCADE,
    CONSTRAINT CHK_Prep_Time_Not_Negative CHECK (Prep_Time >= 0),
    CONSTRAINT CHK_Cook_Time_Not_Negative CHECK (Cook_Time >= 0),
    CONSTRAINT CHK_Additional_Time_Not_Negative CHECK (Additional_Time >= 0),
    CONSTRAINT CHK_Calories_Not_Negative CHECK (Calories >= 0),
    CONSTRAINT CHK_Servings_Not_Negative CHECK (Servings >= 0)
);
```

Fig. 21. The format of the recipe table in the SQL database

3) User_Saved_Recipe Table: For this table, as we went on the premise that each user can save multiple recipes, we decided to model this as a composite attribute on our ER diagram, which became the table with the composite primary key seen above with as per the corresponding rules used for mapping from an ER diagram to the relational model. In this table, the candidate key is the combination of the Email and Recipe_ID attributes, which happens to already be the full set of attributes for this table. Because of this, there is no guarantee that the value of any other set of attributes in this table will uniquely map to the value of a different set of attributes in this table. Hence, this table has no functional dependencies and is in BCNF form. See figure 23 for the format of this table in the SQL database.

```
CREATE TABLE User_Saved_Recipe (
Email VARCHAR(250),
Recipe_ID INT,
CONSTRAINT PK_User_Saved_Recipes PRIMARY KEY(Email, Recipe_ID),
CONSTRAINT FK1_User_Saved_Recipes FOREIGN KEY(Email) REFERENCES User (Email) ON DELETE CASCADE,
CONSTRAINT FK2_User_Saved_Recipes FOREIGN KEY (Recipe_ID) REFERENCES Recipe (Recipe_ID) ON DELETE CASCADE
);
```

Fig. 22. The format of the user_saved_recipe table in the SQL database

4) User_Review For this table, as we went on the premise that users can review multiple recipes, we decided to model this as the central point of a many-to-many our User and Recipe tables, which became the table with both the Email and Recipe_ID in it with as per the corresponding rules used for mapping from an ER diagram to the relational model. However, due to the fact we wanted to let users make multiple reviews for the same recipe, we realized that we needed to add an Review_ID attribute to the table's primary key as the table's original primary key would not allow this. In this table, as the combination of the Email, Recipe_ID, and Review_ID attributes is unique within each row and associated with the table's other attributes within each row, functional dependencies for this table are Email, Recipe_ID, Review_ID → all subsets of the attributes Comment, Rating. However, as the Review_ID

attribute on its own is unique within each row, it is this table's candidate key. This means that other functional dependencies for this table are Review_ID → all subsets of the attributes Email, Recipe_ID, Comment, Rating. Here, as all the functional dependencies of this table have a super key on their left-hand side, this table is in BCNF form. See figure 23 For an in-depth look at the formatting of the table in the SQL database.

```
CREATE TABLE User_Review (
Email VARCHAR(250),
Recipe_ID INT,
Review_ID INT,
Comment VARCHAR(2000),
Rating DECIMAL(10,2),
CONSTRAINT PK_User_Reviews PRIMARY KEY(Email, Recipe_ID, Review_ID),
CONSTRAINT FK1_User_Reviews FOREIGN KEY(Email) REFERENCES User(Email) ON DELETE CASCADE,
CONSTRAINT FK2_User_Reviews FOREIGN KEY(Recipe_ID) REFERENCES Recipe(Recipe_ID) ON DELETE CASCADE,
CONSTRAINT CHK_Rating_Values CHECK (Rating BETWEEN 0 AND 5)
);
```

Fig. 23. The format of the user_review table in the SQL database

5) Ingredient Table: In this table, as each Ingredient_ID is unique and is associated with an Ingredient_Name a functional dependency for this table is Ingredient_ID → Ingredient_Name. In this table, as all of the functional dependencies have a super key on their left-hand side, this table is in BCNF form. See figure 24 to view how this table is formatted in our SQL database.

```
CREATE TABLE Ingredient (
Ingredient_ID INT AUTO_INCREMENT,
Ingredient_Name VARCHAR(250),
CONSTRAINT PK_Ingredient PRIMARY KEY (Ingredient_ID)
);
```

Fig. 24. The format of the ingredient table in the SQL database

6) Recipe_Content Table: For this table, as we knew that recipes could have multiple ingredients in them, we decided to model this as the central point of a many-to-many our User and Recipe tables, which became the table with both the Recipe_ID and Ingredient_ID in it with as per the corresponding rules used for mapping from an ER diagram to the relational model. However, as we were analyzing the recipes we originally obtained, we saw that it was possible for an ingredient to be present with either different or the exact same instructions for the quantity and units to use for the recipe, we realized that we needed to add an Recipe_Content_ID attribute to the table's primary key as the table's original primary key would not allow such scenarios. In this table, as the combination of the Recipe_Content_ID, Recipe_ID, and Ingredient_ID attributes is unique within each row and associated with the table's other attributes within each row, functional dependencies for this table are Recipe_Content_ID, Recipe_ID, Ingredient_ID → all subsets of the attributes Quantity, Unit. However, as the Review_Content_ID attribute on its own is unique within each row, it is this table's candidate key. This means that other functional dependencies for this table are Review_Content_ID → all subsets of Recipe_ID, Ingredient_ID, Quantity, Unit. Here, as all the functional dependencies of this table have a super key on their left-hand side, this table is in BCNF form. See figure 25 for a full look at the formatting of this table in the database.

```
CREATE TABLE Recipe_Content (
Recipe_Content_ID INT AUTO_INCREMENT,
Recipe_ID INT,
Ingredient_ID INT,
Quantity DECIMAL(10,3),
Unit VARCHAR(250),
CONSTRAINT PK_Recipe_Content PRIMARY KEY (Recipe_Content_ID, Recipe_ID, Ingredient_ID),
CONSTRAINT FK1_Recipe FOREIGN KEY(Recipe_ID) REFERENCES Recipe (Recipe_ID) ON DELETE CASCADE,
CONSTRAINT FK2_Recipe FOREIGN KEY(Ingredient_ID) REFERENCES Ingredient (Ingredient_ID) ON DELETE CASCADE
);
```

Fig. 25. The format of the recipe_content table in the SQL database

F. Query Optimization

In our database, our table with the most amount of rows is the Recipe_Content table with over 15000 rows inside. In addition to this, a common query that we use to the entire information of each recipe is the one shown in figure 26.

```
SELECT rc.Recipe_ID, r.Recipe_Name, r.Prep_Time, r.Cook_Time, r.Calories, r.Servings,
       i.Ingredient_Name, rc.Quantity, rc.Unit, r.Directions
FROM Recipe_Content rc
JOIN Recipe r ON rc.Recipe_ID = r.Recipe_ID
JOIN Ingredient i ON rc.Ingredient_ID = i.Ingredient_ID;
```

Fig. 26. A very common query run on the database

Due to both the number of rows that our Recipe_Content table had and the fact that this query joins multiple tables together, it is clear that this query uses a heavy amount of I/Os when it is executed, which can be seen by using the command seen in figure 27.

```
EXPLAIN FORMAT = TREE
SELECT rc.Recipe_ID, r.Recipe_Name, r.Prep_Time, r.Cook_Time, r.Calories, r.Servings,
       i.Ingredient_Name, rc.Quantity, rc.Unit, r.Directions
FROM Recipe_Content rc
JOIN Recipe r ON rc.Recipe_ID = r.Recipe_ID
JOIN Ingredient i ON rc.Ingredient_ID = i.Ingredient_ID;
```

Fig. 27. Modified query to see how the query is being ran

We were able to see a hierarchical representation of the query's execution plan, as seen in figure 28.

```
/*
'-> Nested loop inner join  (cost=11316 rows=15584)
    -> Nested loop inner join  (cost=5862 rows=15584)
        -> Table scan on i  (cost=407 rows=4037)
        -> Index lookup on rc using FK2_Recipe (Ingredient_ID=i.Ingredient_ID)  (cost=0.965 rows=3.86)
    -> Single-row index lookup on r using PRIMARY (Recipe_ID=rc.Recipe_ID)  (cost=0.25 rows=1)'
*/
```

Fig. 28. The tree formatting of the query, before indexing

As you can see, it would seem that when this query executes, it uses a combination of nested loop joins, index lookups, and table scans. As you can see, both the nested loop inner joins are the main cause of this query's high I/O

cost and number of rows examined.. Hence, in an attempt to reduce both the query's cost and number of rows examined at each step, we decided to look into how we could use indexes to help. After conducting further analysis on the 3 tables involved in the join, we found out that the Ingredient table of our schema already has an index on Ingredient_ID that is the field that serves as the table's primary key. The same is true for the other 2 tables involved in our query's join. The Recipe table of our schema already has an index on Recipe_ID that is the field that serves as the table's primary key, and the Recipe_Content table of our schema already has an index on Recipe_Content_ID that is the field that serves as the table's primary key. However, while the Recipe_ID and Ingredient_ID columns are indexed on the Recipe and Ingredient tables, they are not indexed on the Recipe_Content table. Hence, as the query's join involves both of those columns on the Recipe_Content table as well, we thought it would be a good idea to create a composite index for both of those columns for that table as well like the one shown in figure 29.

```
CREATE INDEX idx_recipe_content ON Recipe_Content(Recipe_ID, Ingredient_ID);
```

Fig. 29. Creation of the index for the recipe_content table

Now when we run the "EXPLAIN FORMAT = TREE" command on our query to fetch all of the recipes with all of their information, the hierarchical representation of the query's execution plan is as seen in figure 30

```
/*
'-> Nested loop inner join  (cost=10631 rows=14933)
    -> Nested loop inner join  (cost=5404 rows=14933)
        -> Table scan on r  (cost=177 rows=1532)
        -> Index lookup on rc using idx_recipe_content (Recipe_ID=r.Recipe_ID)  (cost=2.44 rows=9.75)
    -> Single-row index lookup on i using PRIMARY (Ingredient_ID=rc.Ingredient_ID)  (cost=0.25 rows=1)'
*/
```

Fig. 30. The tree formatting of our query, after using indexing

As you can see, thanks to our proposed index, the cost of both the nested inner loop joins were reduced by several hundreds, and the same is true for the number of rows they examined. Furthermore, our proposed index reduced the cost of the table scan on the Recipe table by a few hundreds and the number of rows examined by a couple thousands. One more change that our proposed index caused on the query's execution plan is that instead of using the Recipe_Content foreign key referencing the Recipe table, it uses our proposed index instead as seen by the words "idx_recipe_content" on the Index lookup line of the query's execution plan. While this gave this step of the query a slight increase in the number of rows examined, it slightly decreased the I/O cost of this step of the query.

V. The Website Layout

Upon opening the website for the first time, you are prompted to login or register. A new user can then create an email and password, which can be immediately used to login after. Once a user is logged in, they are brought to a screen which will show all the recipes and various filtering options. See figure 31 for a look at the screen. Some of the filters included are search by recipe name, filter by calories, filter by a minimum rating, filter by ingredients used in the recipe, and lastly the ability to sort by recipe name, highest rating, and calories.

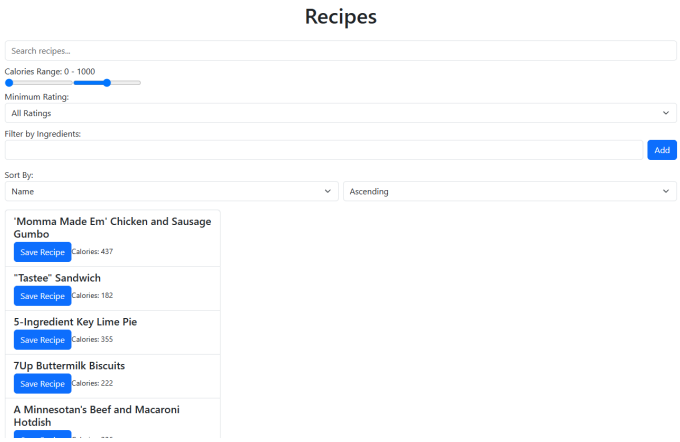

Fig. 31. A look at the Recipe page, with no filters applied

This is also the screen where users can save recipes, and if they click on a recipe to expand it's information, they can also leave a review of the recipe. The review is a simple text entry box, paired with a selection for how you'd rate the recipe on a scale from 1 to 5. Once the submit button is pressed, the review is created and a confirmation is given to the user to confirm that the review was created successfully. A similar confirmation is given when a user saves or unsaves a recipe. See figure 32 for a look at the information provided by the recipe, as well as what the review form looks like.

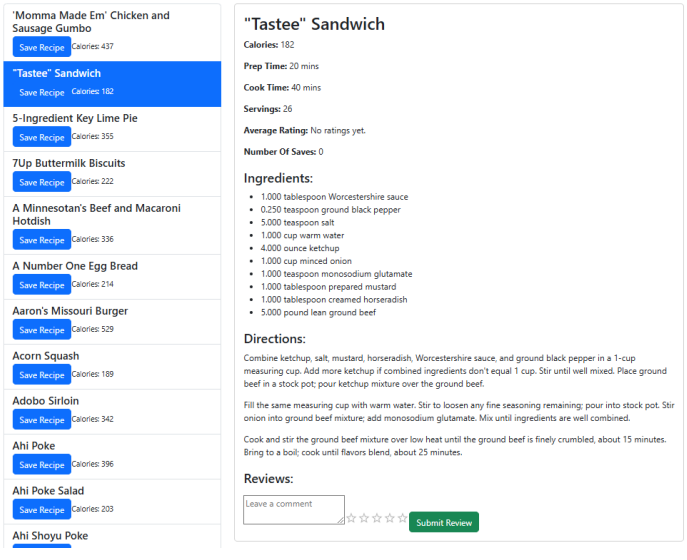

Fig. 32. A look at the Recipe page when a recipe has been selected

A user can also click on a link at the top of the page to view the recipe builder page. The recipe builder page looks

like a typical form, where you can enter information such as recipe name, prep time, cook time, servings prepared, calories per serving, directions, and which ingredients are used in what quantity for the recipe. The user can add as many ingredients as they'd like, and even use the same ingredient multiple times if they are needed in different quantities for different parts of the recipe. See figure 33 to see the full form, and figure 34 to see how to add ingredients to the recipe. Note that for a given user, each recipe name has to be unique.

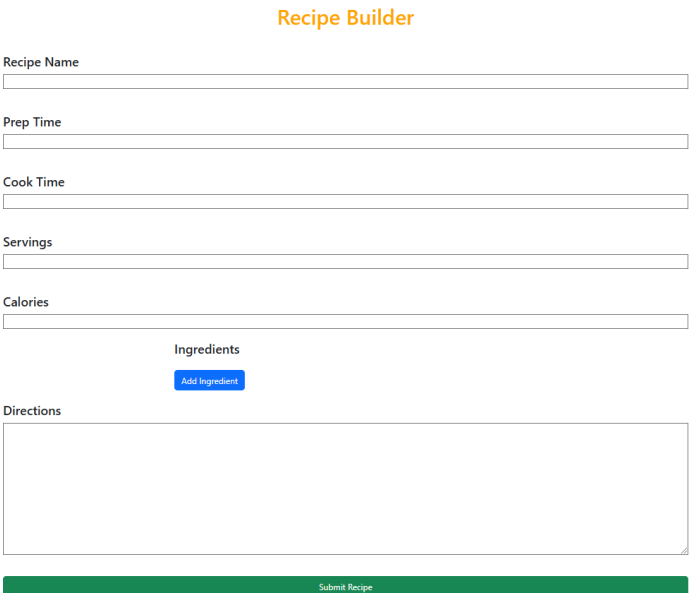

Fig. 33. A look at the Recipe Builder page

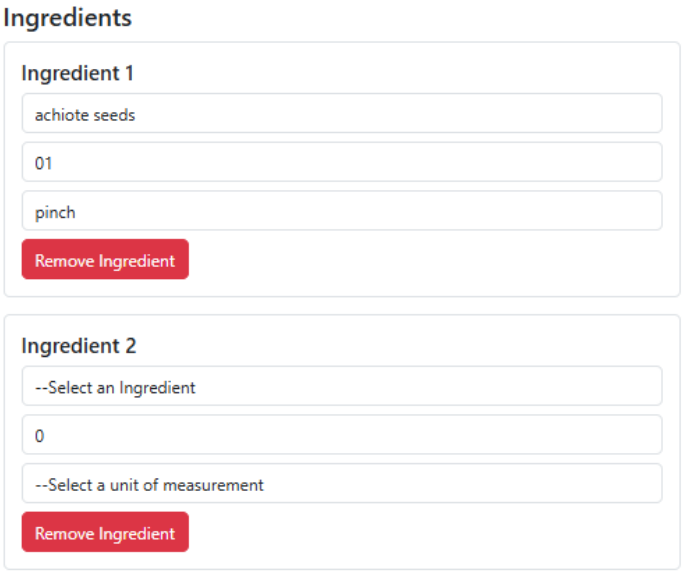

Fig. 34. A look at adding ingredients, Ingredient 1 shows an example of a filled in ingredient, Ingredient 2 shows an empty example

There's also a page which displays the top 10 recipes of three different categories, which are highest average review score, healthiest, and most saves. These three views are dynamically updated, so they may change depending on when you visit the webpage. Screenshots of the three views can be seen in figures 35, 36, and 37. Information about the views in the database can be found in the section "Testing Triggers & Views".

Top 10 Recipes by Average Rating:

Recipe ID	Recipe Name	Average Rating
3	Cowboy Caviar	5.00
5	Slow Cooker Texas Smoked Beef Brisket	5.00
694	'Momma Made Em' Chicken and Sausage Gumbo	5.00
784	Tiramisu Cheesecake	5.00
964	Amish Meatloaf	5.00
1807	HIGH QUALITY RECIPIE	5.00
1809	Hot Dogs	5.00
1	Slow Cooker Texas Pulled Pork	4.28
10	Sauteed Patty Pan Squash	4.00
2	Brazilian Grilled Pineapple	3.50

Fig. 35. Screenshot of the highest average rating view as seen on the website

Top 10 Recipes by Number of Saves:

Recipe ID	Recipe Name	Number Of Saves
1	Slow Cooker Texas Pulled Pork	3
2	Brazilian Grilled Pineapple	2
5	Slow Cooker Texas Smoked Beef Brisket	2
3	Cowboy Caviar	1
6	Best-Ever Texas Caviar	1
7	Texas Sausage Kolaches (Klobasneks)	1
9	Mom's Favorite Baked Mac and Cheese	1
11	Slow Cooker Carolina BBQ	1
15	Texas Cowboy Stew	1
16	Spicy Fish Soup	1

Fig. 36. Screenshot of the most saves view as seen on the website

The last page that the user can visit is their personal profile. From there, it will show the user's created recipes, saved recipes, and reviews left on recipes. This is also the page where users can delete their own recipes, in case they were not happy with it's quality or they wanted to make major changes to it. See figure 38

A. *Testing Website*

We individually tested various pages and features of the website to make sure that they work as intended. While it

Fig. 37. Screenshot of the healthiest recipe view as seen on the website

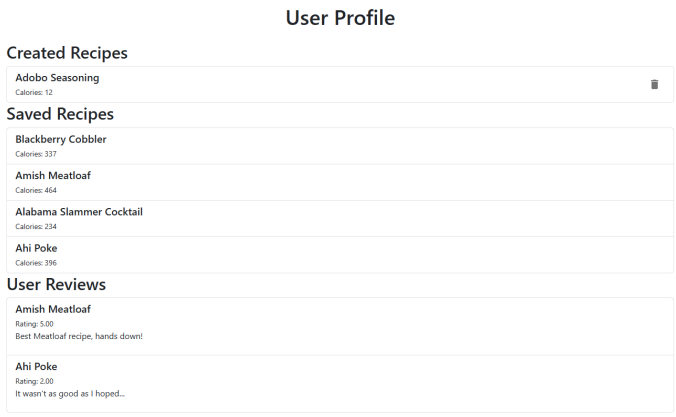

Fig. 38. A look at the user profile screen

is possible that we all missed a bug or unintended effect from our approach, we did our best to ensure that we all tested thoroughly on the website, which included tests such as making sure links work as intended, users are prompted with errors when appropriate, and that the database is updated properly when certain functions are called. We did uncover a few minor bugs that have since been fixed, such as links being broken, and information not being displayed properly. While this isn't an ideal approach to testing the frontend of a system, given our time frame it is what we could manage, and we are confident that there are no major bugs or flaws in the program.

VI. Conclusions

We accomplished all of the things we initially set out to do with this project, including creation of a website accessible to anyone on campus, creation of a robust database, and integration between those two parts. We used React with TypeScript to build the website, with the help of a few other tools as described in the Background Materials section. The React interface allows for users to interact with the database by viewing recipes, creating their own recipes, and leaving reviews and saving recipes that they like. This is all updated and saved on a database that is online nearly 24/7.

Although we ran into issues at times, we worked together to overcome what we could in order to provide a better project and experience for users. Some of the larger issues we experienced are described in the Lessons Learned subsection, however we all ran into our own set of smaller issues when programming, particularly when programming the frontend we all ran into our own issues and problems that we couldn't necessarily solve on our own. Ultimately, we are happy with the project we've created in the time we had given.

A. Lessons Learned

While we did learn a lot from working on the Database, we learned even more by trying to connect it to our frontend. It was a lot more time consuming trying to figure out bugs when dealing with both a frontend and backend that could be causing issues. This leads nicely into our second lesson learned, that we need to leave more time for debugging and make more realistic assumptions about how long a task will take. We frequently spent more time trying to figure out what was going wrong with our code than we expected. In addition, maintenance of a smaller web server proved problematic at times, as we had to frequently reboot the server, however we later upgraded our server that hosted the website and database so that it wouldn't crash as often. Our last lesson learned would be that more frequent, shorter meetings would generally be more helpful than one longer meeting each week, especially if some team members aren't able to be present for all meetings. This would allow us to give more updates throughout the week, and we'd be able to see where team members are falling behind and what tasks they are doing well in.

B. Future Work

There are still some features we were unable to implement in the given time frame, that given more time we would have loved to implement. The biggest change we would have liked to implement would be creation of new ingredients. Currently, in the recipe builder portion of the website, it limits you to choosing from the list of 4000 ingredients that exist in the default recipes we got from the original dataset. Another thing we could have added with enough time would have been viewing other user's profiles. This would have allowed users to identify other users who have similar taste based on their reviews or the recipes they contain. There is also the issue that a user can leave multiple reviews on the same recipe, when in most systems the way this would be handled is to update an existing review, or create one if one does not exist. Other than that, there are a few other minor issues that exist in the site. One such example would be that when a user leaves a review, the review doesn't show up unless the user select a different recipe and then go back to the recipe the user left the review on.

CONTRIBUTIONS

- **Sean Arackal**: Tested triggers and views created by other members with Alex. Wrote up the abstract for the project progress. Reviewed and edited the paper as needed. Helped write a majority of functionality related to features such as "Saving and Unsaving Recipes", "Adding Reviews", and "Displaying Reviews" on recipes with help from Axel and Alex. Wrote the entirety of the User Profile page. Improved UI gently throughout the project.
- **Alexander Bell**: Set up team meetings, tested triggers and views not already covered by current data with the help of Sean, and helped write the Project Overview section in the report. Helped create the recipe builder page with Axel, and connected the webpage form to the database with INSERT statements that also prevent simple SQL Injection attacks by using prepared statements. Helped Sean debug features related to displaying reviews and accessing user profiles. In addition, wrote the sections "The Website Layout", "Testing Triggers & Views", and "Conclusions" including it's subsections, and helped review, edit, and format the paper, including subsections "Queries", "Normalization Process", "Query Optimization", and made minor changes to other sections as needed. Main author of the Project Final Presentation slide deck.
- **Spencer Greene**: Backend specialist. Managed the dedicated server in the Cloud, including installing all software on this server to run necessary services, and created the infrastructure to support the domain being hosted. This involved pointing the domain to the web server, and creating a reverse proxy that serves the running client/server instance to the web. Additionally, installed and deployed MySQL on the server, allowing a 24/7 running instance of our data to be accessible by the team. Assisted Jamie with backend/frontend development (in developing the website with Node.JS, React, and Express). Developed various features on the website - i.e. the login/registration page by linking and connecting them to the database. Documentalist (formatted the report in IEEE format with LaTeX).
- **Axel Luca**: Primary editor of the project intent and proposal and wrote up the whole section about the ER diagram for both documents. Made most of the presentation for the project proposal. Scraped and cleaned all of the data we need for our project with help from scripts provided by Jaime. Main editor of our project's ER diagram. Provided the initial MySQL server we used for the project. Translated our application's ER diagram to a MySQL schema. Came up with and wrote up the MySQL views and triggers we plan to use for our project. Provided the default schema needed for the MySQL cloud server provided by Spencer that we're using for our project now. Expanded on the ER diagram section for this project progress report, wrote up the project progress report's MySQL background, and wrote up part of the project abstract for this project progress report along with the normalization analysis and query optimization as well. As for the website, I made the entire page displaying the 3 views created in our database inside visible tables on the GUI. Additionally, I also worked with Sean to implement some of the functionality related to features such as "Saving and Unsaving Recipes", "Adding Reviews", and "Displaying Reviews" on recipes. Moreover, I also collaborated with Alex to implement the functionality required for users to be able to create recipes and add their ingredients.
- **Jamie Ortiz**: Created python scripts to retrieve data that was not already in our database, including all the recipe instructions for all the recipes in the database, we couldn't find a database with all the information needed so I needed to query google with a python script that retrieves the instructions based on all the information we have until now about that recipe (from the same website Allrecipes.com). Together with Spencer we created the backend and frontend for the website, we connected it to MySQL. Our website queries the recipes from the database with all the information from them, we plan to include a user login and a filter option for recipes. After that we will include a system to leave reviews. Helped write all the paper like the rest of the team, documented things like react, things we used for the web. Used React.js, Node.js, express, etc. Language used Typescript, Javascript, CSS. Connection pool to MySQL done with mysql2/promise. Tested triggers like same recipe name with same users, using dummy data.

REFERENCES

[1] Oracle, "What is MySQL?," Oracle, 2021. https://www.oracle.com/mysql/what-is-mysql/

[2] Allrecipes, "Allrecipes — Food, friends, and recipe inspiration," Allrecipes, 2019. https://www.allrecipes.com/

[3] Thomsen, C. (2002). Using Stored Procedures, Views, and Triggers: How to Use Stored Procedures, Views, and Triggers. In Database Programming with C# (pp. 367-412). Berkeley, CA: Apress.

[4] Siahaan, V., & Sianipar, R. H. (2019). The Self-Taught Coder: The Definitive Guide to Database Programming with Python and MySQL (Vol. 1). SPARTA Publishing.

[5] Stephens, J., & Russell, C. (2004). Optimizing Queries with Operators, Branching, and Functions. In Beginning MySQL Database Design and Optimization: From Novice to Professional (pp. 171-238). Berkeley, CA: Apress.

[6] Komperla, Varun & Pratiba, Deenadhayalan & Ghuli, Poonam & Pattar, Ramakanthkumar. (2022). React: A detailed survey. Indonesian Journal of Electrical Engineering and Computer Science. 26. 1710. 10.11591/ijeecs.v26.i3.pp1710-1717.