Index

1. Introduction

WeFood is a Social Network where users can share their recipes and provide feedbacks about other users' recipes through comments and star rankings. It manages in a completely automatic way the calories of the recipes, so the users do not have to worry about that when they post a new recipe. Using the search engine, users can discover new top rated recipes to amaze their friends, filtering by ingredients or calories. Furthermore, users can follow other users and get suggestions about new users to follow.

2. Requirements

Describing the requirements it is important to distinguish between functional and non-functional requirements.

2.1. Functional Requirements

The main functional requirements for *WeFood* can be organized by the actor that is involved in the use case.

1. **Unregistered User**:
   1.1. Browse *recent* recipes;
   1.2. Sign Up.
2. **Registered User**:
   2.1. Log In;
   2.2. Log Out;
   2.3. Upload a Post (Recipe);
   2.4. Modify his/her Posts;
   2.5. Delete his/her Posts;
   2.6. Comment a Post;
   2.7. Modify his/her own comments;
   2.8. Delete his/her own comments or comments on his/her Posts;
   2.9. Evaluate by a star ranking a Post;
   2.10. Delete his/her own star rankings;
   2.11. View the Recipe of a Post;
   2.12. View the Total Calories of a Recipe;
   2.13. View the Steps of a Recipe;
   2.14. View the Ingredients of a Recipe;

2.15. View the Calories of an Ingredient;

2.16. Browse most recent Posts;

2.17. Browse most recent top rated Posts;

2.18. Browse most recent Posts by ingredients;

2.19. Browse most recent Posts by calories (minCalories and maxCalories);

2.20. View his/her own personal profile;

2.21. Modify his/her own personal profile (e.g. change Name, Surname, Password);

2.22. Delete his/her own personal profile;

2.23. Find a User by username;

2.24. View other Users' profiles;

2.25. Follow a User;

2.26. Unfollow a User;

2.27. View his/her Friends (i.e. the Users he/she follows and that follow him/her);

2.28. View his/her Followers;

2.29. View his/her Followed Users.

3. **Admin**:

   3.1 Log In;

   3.2 Log Out;

   3.3. Browse all the Users;

   3.4. Browse all the Posts;

   3.5. Ban a User;

   3.6. Unban a banned User;

   3.7. Delete a Post;

   3.8. Delete a Comment;

   3.9. See statistics about the usage of *WeFood*;

   3.10. Add a new Ingredient.

2.2. Non-Functional Requirements

The non-functional requirements for *WeFood* are as follows.

1. **Performance**: the overall system must be able to handle a request in less than `1.5` seconds, because the user experience would be negatively affected by a longer response time. Being a

social network, it is necessary to have a good performance in order to provide a good user experience.

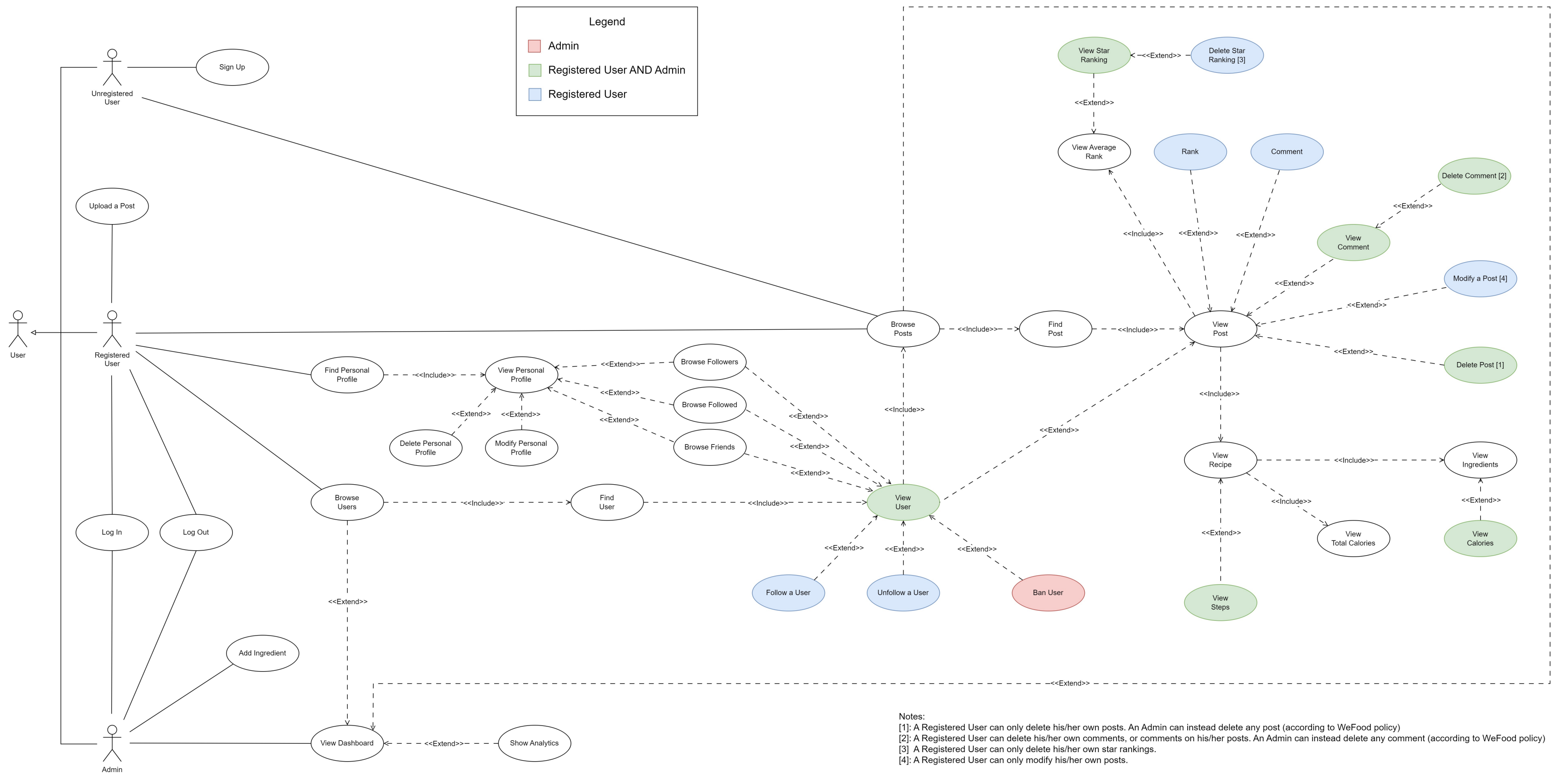
2. **Availability**: the system must be available 24/7 for allowing users to use it at any time.
3. **Security**: the system must be secure and protect users' data even from possible attacks. In particular, the information transmitted between client and server must be over HTTPS. Furthermore, the system must protect users' passwords by hashing them before the storing in the database.
4. **Reliability**: the system must be reliable and must not lose the information uploaded by the users. It must be caple of recovering from a crash and restore the data in a consistent state, exploiting the replicas of the database.
5. **Usability**: the GUI offered to the users must be easy to use and intuitive. Each user should be able to use the application without any training and in about 15 minutes.
6. The **Back-End** must be written in Java.

3. Design

After having defined the requirements, it is possible to proceed with the design of the system.

3.1. Use Case Diagram

Translating the requirements into a graphical representation we obtain the *UML Use Case Diagram* shown in Figure 1.



Notes:
[1]: A Registered User can only delete his/her own posts. An Admin can instead delete any post (according to WeFood policy)
[2]: A Registered User can delete his/her own comments, or comments on his/her posts. An Admin can instead delete any comment (according to WeFood policy)
[3] A Registered User can only delete his/her own star rankings.
[4]: A Registered User can only modify his/her own posts.

Figure 1: UML Use Case Diagram.

3.2. Class Diagram

The *UML Class Diagram* shown in Figure 2 represents the main entities of the system and their relationships.

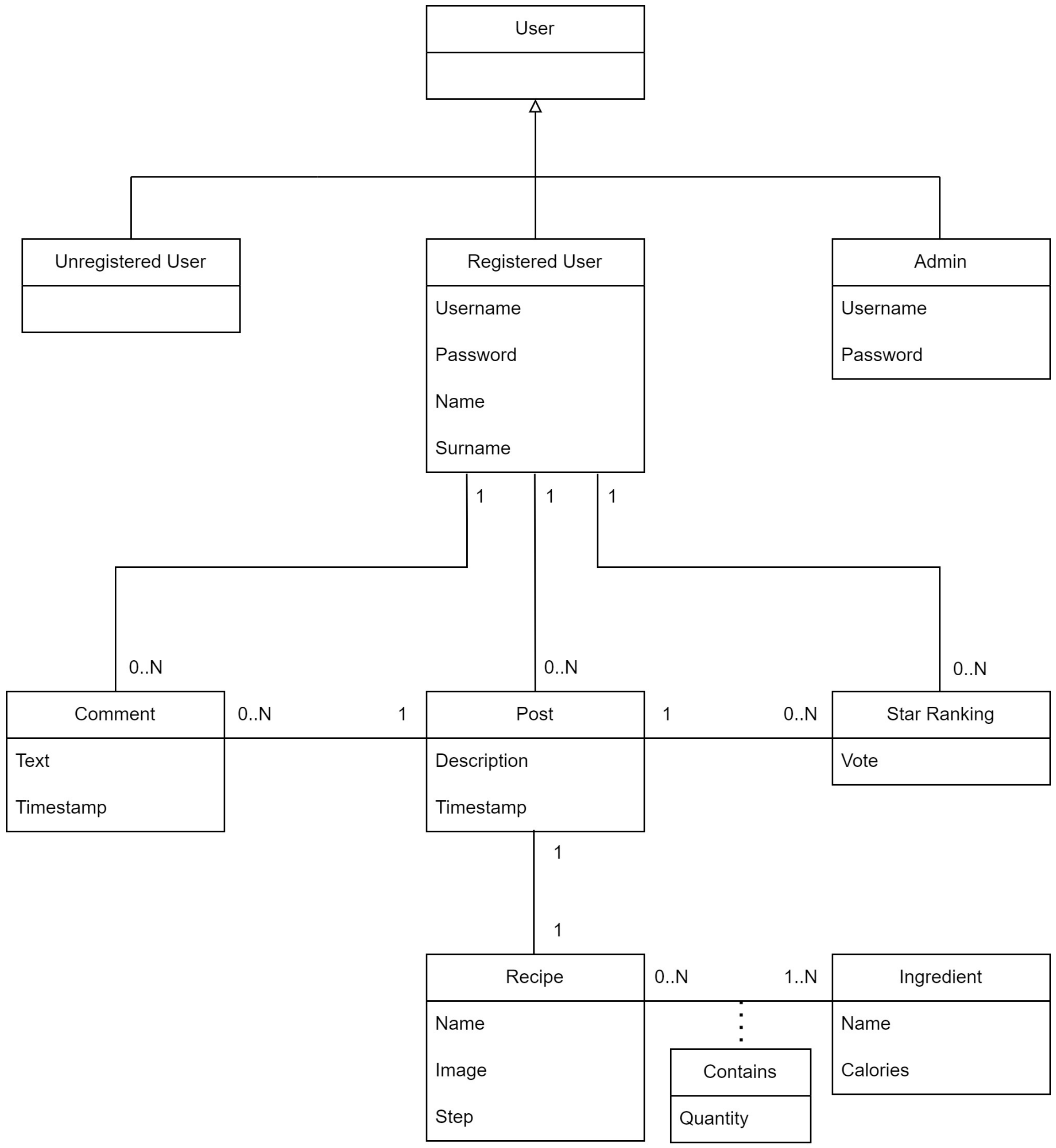


Figure 2: UML Class Diagram.

4. DataBases

Before cleaning and preparing the dataset needed to populate the databases, it is necessary to define the structure of the latter. In particular, two different databases will be used: a document DB and a graph DB.

4.1. Document DB

The entities managed by the document DB are the following.

- User (Unregistered User, Registered User, Admin);
- Post;
- Recipe;
- Comment;
- StarRanking;
- Ingredient.

4.1.1. Collections

The collections designed for storing the information inside the document DB are three: User, Post and Ingredient.

The structure of the User collection is as follows.

```
[User]:
{
    _id: ObjectId('...'),
    type: "Admin", # Applicable only for Admin
    username: String, [UNIQUE]
    password: String,
    name: String, # Not Applicable for Admin
    surname: String, # Not Applicable for Admin
    posts: [{
                idPost: ObjectId('...'),
                name: String, [REDUNDANCY]
                image: String [REDUNDANCY]
    }, ...] # Not Applicable for Admin
}
```

This collection is used both to store information about the Registered Users and the Admins. The field `type` is used to distinguish between the two types of Users, and it is set only for the Admins because they will be in minority compared to the Registered Users. The field `username` is required for the authentication of the Users and it is unique. So there cannot be two Users with the same username. The `password` is used for the authentication as well, and it contains the password provided by the User at the moment of the registration. For security reasons, the password is hashed before being stored in the database. The fields `name` and `surname` are used to store the name and the surname of the Registered Users and hence they are not applicable for the Admins for which it is not necessary to know their names and surnames. Lastly, the field `posts` is used to link (i.e. document linking) the Registered Users with their Posts. In particular, it contains a list of objects, each one containing the id of a Post and the `name` and the `image` of the Recipe of the Post. The fields `name` and `image` are redundant because they are already stored in the Post collection, but they are useful for the queries that involve the User collection because they avoid the need of joining the Post collection.

The structure of the Post collection is a little bit more complex because it manages the information about the Posts, the Recipes, the Comments and the StarRankings. The decision of storing a Post in his own collection instead of embedding it in the document of the User that created it (i.e. document embedding) is due to several reasons:

- a User can publish an unlimited number of Posts, and a Post can have an many Comments and StarRankings. So the size of the document of a User would have grown indefinitely and very quickly, and this would have lead to a degradation of the performance of the system;
- the Posts are the main entities of the system and they are used in many queries. So having them in a separate collection ensures better performance by limiting the size of the individual document and taking advantage of tailored indexes.

```
[Post]:
{
    _id: ObjectId('...'),
    idUser: ObjectId('...'),
    username: String, [REDUNDANCY]
    description: String,
    timestamp: Long,
    recipe: {
                name: String,
                image: String,
                steps: [String, ...],
                totalCalories: Double, [REDUNDANCY]
                ingredients: [{
                                name: String,
                                quantity: Double
                }, ...]
    },
    starRankings: [{
                    idUser: ObjectId('...'),
                    username: String, [REDUNDANCY]
                    vote: Double
    }, ...],
    avgStarRanking: Double, [REDUNDANCY]
    comments: [{
                idUser: ObjectId('...'),
                username: String, [REDUNDANCY]
                text: String,
                timestamp: Long
    }, ...]
}
```

Here the field `idUser` is used to link the Post with the Registered User that created it and `username` contains his/her username. The field `description` is used to store the description of the Post provided by the User. The field `timestamp` is used to store the timestamp of when the Post was uploaded. The field `recipe` is used to store the information about the Recipe contained in the Post. In particular, it contains the `name` and the `image` of the Recipe, the `steps` of the Recipe, the `totalCalories` of the Recipe and the list of `ingredients` of the Recipe together with the respective quantities in grams. It is important to notice that `image` does not contain the whole image, but just the URL of the image. The latter is stored or online or locally in the server. The field `starRankings` is used to store the information about the star rankings of the Post. In particular, it contains a list of objects, each one containing the id and the `username` of the Registered User that provided the star ranking and the `vote`, that represents the vote expressed by the Registerd User. The field `avgStarRanking` is used to store the average of the star rankings of the Post. The field `comments` is used to store the information about the comments of the Post. It contains a list of objects, each one containing the id and the `username` of the RegisteUser that provided the comment, the `text` and the `timestamp` of the comment.

The last collection is the Ingredient collection, that is used to store the information about the Ingredients. The structure of the Ingredient collection is very simple because it contains only the `name` and the `calories` per 100 grams of the Ingredient. The `name` is unique because does not make sense to have two Ingredients with the same name.

```
[Ingredient]:
{
    _id: ObjectId('...'),
    name: String, [UNIQUE]
    calories: Double
}
```

4.2. Graph DB

The entities that are managed by the graph DB are just: User (Registered User), Recipe, Ingredient.

4.2.1. Nodes

The nodes designed for storing the information inside the graph DB are three, one for each entity.

The User node is used to store the information about the Registered Users. Each node contains the `_id` of the Registered User that is stored in the User collection of the document DB and his/her `username`.

```
(User):
    - _id: String
    - username: String [REDUNDANCY]
```

The Recipe node is used to store the information about the Recipes. Each node contains the `_id` of the Post that contains the Recipe, that is stored in the Post collection of the document DB and the `name` of the Recipe.

```
(Recipe):
    - _id: String
    - name: String [REDUNDANCY]
```

The Ingredient node is used to store the information about the Ingredients. Each node contains the `_id` of the Ingredient that is stored in the Ingredient collection of the document DB and the `name` of the Ingredient.

```
(Ingredient):
    - _id: String
    - name: String [REDUNDANCY]
```

4.2.2. Relationships

Between the described nodes are possible several relationships that are formally defined as follows.

```
(User)-[:FOLLOWS]->(User)
```

This relationship allows Users to follow other Users. Two Users become friends when they follow each other.

```
(User)-[:USED]->(Ingredient)
       (times: int) [REDUNDANCY]
```

This relationship, instead, allows to quickly retrieve the Ingredients that have been used by the Users in their Recipes. The `times` attribute, in addition to being used for counting the number of times that an Ingredient has been used by a User, is used to keep track of the fact that the relationship with the Ingredient still can exist in other Recipes after the deletion of a Recipe by a User. Only when `times` becomes 0 the relationship can be deleted.

```
(Ingredient)-[:USED_WITH]->(Ingredient)
              (times: int) [REDUNDANCY]
             [BIDIRECTIONAL]
```

This relationship allows to quickly retrieve the Ingredients that have been used together in the Users' Recipes. The `times` attribute is used in the same way as the previous relationship: it is used for counting the number of times that two Ingredients have been used together and to keep track of the fact that the relationship between two Ingredients still can exist in other Recipes after the deletion of a Recipe. Only when `times` becomes 0 the relationship can be deleted. This relationship is bidirectional because if an Ingredient `A` has been used with an Ingredient `B`, then also the Ingredient `B` has been used with the Ingredient `A`.

```
(Recipe)-[:CONTAINS]->(Ingredient)
```

This last relationship allows to retrieve the Ingredients that are contained in a Recipe.

5. Dataset

To populate the databases with a substantial volume of realistic data, datasets sourced from Kaggle were employed.

5.1. Raw Dataset

The intial raw datasets are related to the main functionalities of *WeFood*. In particular, datasets about recipes and ingredients were found.

- Calories per 100 grams in Food Items `[1]`
- Recipes and Interactions `[2]`
- Recipes and Reviews `[3]`

Contained in these datasets there are *almost* all the information needed to populate the databases. Indeed, in around 1 GB of raw data it is possible to find 2225 food items, over 500,000 recipes and 1,400,000 reviews. After a careful analysis, however, it was found that something was missing.

1. Personal Information about the Users: only the username of the Users was available (`AuthorName` in `recipes.csv` of `[3]`).
2. The quantity of each ingredient in the recipes: `RecipeIngredientQuantities` in `recipes.csv` of `[3]` contains some numbers that could be useful for this purpose, but they are not clear and there is no documentation about them. Indeed there is no way to understand if they are the quantities of the ingredients in grams or in other units of measure (e.g. just to have an idea there numbers like the following: 1, 1/2, 3, 5, etc). Furthermore, there are a lot of `NA` values in this column.

Everything else is in the datasets, and need only to be cleaned and appropriately merged to obtain the structure needed for the population of the databases.

5.2. Cleaning Process

There is the need to clean the datasets because in them there are plenty of information that are not useful for the purposes of *WeFood* and would result only in a waste of space. For achieving this goal, the datasets were analyzed in detail and the information that were not useful were discarded. The cleaning process was performed using Python and Jupyter Notebook.

Below a separate description of the cleaning process for each entity identified in the design phase is provided.

Ingredient: The starting point was: `ingr_map.pickle` of `[2]`. Here there are lots of fields useful for machine learning related tasks, but not for the purposes of *WeFood*. Only the fields strictly needed for the link with the recipes and with the dataset about the calories of the ingredients were kept. The final result is the following. To facilitate referencing in the subsequent merging process, each intermediate product generated during the cleaning process will be assigned a distinct name, starting with the following.

```
Ingredient_A: {
    "raw_ingr":"pretzels",
    "replaced":"pretzel",
    "id":5711
}
```

The first field `raw_ingr` contains the original text of the ingredient, the one inserted by the user in the recipe. The second field `replaced` contains the unique representation of the ingredient and the last field `id` contains the unique identifier of the ingredient.

At this point, the file `calories.csv` of [1] was analyzed. In this file in addition to the calories per 100 grams of each ingredient there are also Food Categories associated to them. These categories were really useful because they allowed to devise a plan for dealing with the lack of the quantity of each ingredient in the recipes in a simple but effective way. The idea was the following:

1. to associate to each `FoodCategory` two quantities, `quantity_min` and `quantity_max`, that are a realistic representation of the quantities used in real life for that specific `FoodCategory` (this labour intensive work was done with the support of an AI);
2. to generate a random quantity for each ingredient in each recipe in the range `[quantity_min, quantity_max]`.

This solution does not provide a precise quantity for each ingredient in each recipe, but it is a good approximation that won't produce unrealistic results. This means that there won't be a recipe where there are 500 grams of `salt`, because the maximum quantity of `salt` that can be used in a recipe is 10 grams (i.e. `Herbs&Spices`, the `FoodCategory` of `salt`, has `quantity_max` equal to 10 grams).

After having removed some duplicates from `calories.csv`, based on the `FoodItem` field, the fields that were not useful were discarded. An example can be of help for understanding the final structure of the file.

```
Ingredient_B: {
    "FoodCategory":"Pastries,Breads&Rolls",
    "FoodItem":"Pretzel",
    "Cals_per100grams":"338 cal",
    "quantity_min":100,
    "quantity_max":500
}
```

Recipe: After the conversion of `RAW_recipes.csv` in `json` to have a more readable format, the file was analyzed in detail. Here there were lots of fields that could be discarded. The structure after the cleaning is, as before, better described by an example object.

```
Recipe_A: {
    "name":"pretzel crust",
    "id":194491,
    "contributor_id":356062,
    "submitted":"2006-11-07",
    "steps":"['preheat oven to 350', 'crush pretzels in a blender', 'add sugar
    ↪  and butter and mix well', 'press into a 9 inch pie plate', 'bake at 350
    ↪  for 8 minutes and cool', 'add desired filling']",
    "description":"recipe for a basic pretzel crust i found in a magazine. i
    ↪  don't think i saw it posted here yet.",
    "ingredients":"['pretzels', 'sugar', 'butter']"
}
```

Here the `name` is the name of the Recipe, `id` is the unique identifier of the Recipe and will be useful for linking the recipe with the interactions (i.e. comments and star rankings) of the dataset [2]. The `contributor_id` is the unique identifier of the User that created the Recipe. The `submitted` field is the timestamp of when the Recipe was uploaded. The `steps` field contains the steps of the Recipe, the `description` field contains the description that will be used for the Post that contains the Recipe and the `ingredients` field contains the list of the ingredients of the Recipe. Observing carefully the `steps` and the `ingredients` it is clear that they must be transformed into an array of strings because at the moment they are just strings. So the next step is the latter.

From `recipes.csv` of [3], instead, it is possible to retrieve the URLs of the images of the Recipes. Thus only the fields `RecipeId` and `Images` are retained. Note that not all the Recipes have an image, and some of them have more than one image. Where no image is available, a default image will be applied. Viceversa, if multiple images are present, only the first image will be utilized.

```
Recipe_B: {
    "RecipeId":194491,
    "Image":"https:\/\/img.sndimg.com\/food\/image\/upload\/
    ↪  w_555,h_416,c_fit,fl_progressive,q_95\/v1\/img\/recipes
    ↪  \/19\/44\/91\/3xjO4aXTeiZpOajAsBRX_0S9A6246.jpg"
}
```

Comment: In `RAW_interactions.csv` of [2] the reviews (i.e. comments of *WeFood*) and the ratings (i.e. star rankings of *WeFood*) are stored together, because to each review there is associated a rating. In *WeFood* it is not the same, because a Registered User can leave a comment without providing a star ranking and viceversa. So the first step was to separate the two types of interactions.

```
Comment_A: {
    "user_id":430471,
    "recipe_id":194491,
    "timestamp":"2007-04-09"",
    "text":"This tasted great, however, I couldn't get it to hold together good.
    ↪  Either I didn't crush the pretzels small enough or I should have used a
    ↪  little more butter. But either way it was easy to make and tasted great.
    ↪  I used it as a base for a cheesecake pudding with fresh strawberries on
    ↪  the top."
}
```

Star Ranking: As previously mentioned, also the star rankings were stored in `RAW_interactions.csv` of [2]. For this reason the cleaning process was similar as the one used for the comments.

```
StarRanking_A: {
    "user_id":254614,
    "recipe_id":194491,
    "vote":4
}
```

Post: Posts are an abstraction of the Recipes that was introduced in *WeFood*. In the datasets there is no information about them, so they will be created from scratch in the next merging process.

User: As previously noted, another lack of the datasets was the absence of personal information about the Users. Indeed, only the username of the Users was available. For not having an inconsistent situation where the Users have a name and a surname that are not coherent with their username, the username was dropped as well. In this way, it was possible to generate all the information needed for the Users using the Python library `Faker`. In particular, the name and the surname of the Users were generated randomly, and the username was computed accordingly using the following structure:

```
username = f"{name.lower()}_{surname.lower()}_{num}"
```

Where `num` is a random number in the range `[1, 99]`. All this is made paying attention to the fact that the username must be unique. By employing this approach, it was possible to create a User for each `contributor_id` of the Recipes (i.e. the Users who uploaded at least one Recipe). Users who did not upload any Recipe (i.e. who only appear in interactions) were excluded as a sufficient number of users was already available. The passwords (currently in plain) were generated randomly as well.

```
User_A: {
    "contributor_id":356062,
    "name":"Justin",
    "surname":"Alexander",
    "username":"justin_alexander_34",
    "password":"e6FMX30hGu"
}
```

5.2. Merging Process

After having cleaned the datasets, and having generated the missing information, it was possible to proceed with the merging process. This step was necessary to recreate the structure needed for the population of the databases. Also the merging process was performed using Python and Jupyter Notebook.

Because the merging process aims to produce the final structure of the documents that will be stored in the document DB, the latter will follow a different partitioning compared to the one used in the cleaning process. Here the partitioning will be based on the collections of the document DB.

Ingredient: Obtaining the final structure for the `Ingredient` collection is straightforward if `Ingredient_B` is considered. Indeed, it is sufficient to:

- rename `FoodItem` to `name`;
- rename `Cals_per100grams` to `calories`;
- take the float value of `calories`;
- discard `FoodCategory`, `quantity_min` and `quantity_max`.

```
{
    name: "Pretzel",
    calories: 338.0
}
```

The `_id` field will be automatically generated at the moment of the insertion in the database.

Post: Being the main collection of *WeFood*, the Post collection was also the one that takes more time to be merged. For this reason it is necessary to describe a step at a time for not complicating too much the explanation.

1. Merge of Recipe_A and Recipe_B on on id and RecipeId respectively for including the URLs of the images inside the Recipes. In this way Recipe_AB is obtained.

2. The subsequent task involves converting the ingredients array of strings within Recipe_A into an array of objects. Each of these objects will include in the final structure the ingredient's name and its corresponding quantity expressed in grams. It is crucial to emphasize that the name of the ingredient must match an entry in the Ingredient collection. For achieving this:

   a. Firstly, Ingredient_A and Ingredient_B are matched on replaced and FoodItem respectively. For maximizing the number of matches, the matching was performed with the support of a Python Library called fuzzywuzzy which merges strings by likelyhood using *Levenshtein Distance*, which is a metric used in information theory, linguistics, and computer science for measuring the difference between two sequences. Thanks to this approach, no Ingredient_A was left unmatched with an Ingredient_B. An example of the matching is the following.

   ```
   Ingredient_AB: {
       "raw_ingr":"pretzels",
       "replaced":"pretzel",
       "id":5711,
       "FoodCategory":"Pastries,Breads&Rolls",
       "FoodItem":"Pretzel",
       "Cals_per100grams":"338 cal",
       "quantity_min":100,
       "quantity_max":500
   }
   ```

   b. By noticing that the raw_ingr field of Ingredient_AB contains the name of the ingredients as they are inserted by the users in the recipes, it possible to do a further match (this time using exact equality) between the latter and the strings contained in ingredients of Recipe_AB.

   c. At this point, ingredients will be an array of objects of the type of Ingredient_AB. For each of this object, it is possible to add a field quantity that is a random number in the range [quantity_min, quantity_max]. This is the quantity of the ingredient in grams that will be used in the recipe. After removing the no longer needed fields and making other minor modifications, here's the new structure.

   ```
   Recipe_C: {
       "name": "pretzel crust",
       "id": 194491,
       "contributor_id": 356062,
       "submitted": "2006-11-07",
       "steps": [
           "preheat oven to 350",
           "crush pretzels in a blender",
   ```

```
        "add sugar and butter and mix well",
        "press into a 9 inch pie plate",
        "bake at 350 for 8 minutes and cool",
        "add desired filling"
    ],
    "description": "recipe for a basic pretzel crust i found in a
    ↪  magazine. i don't think i saw it posted here yet.",
    "ingredients": [
        {
            "foodItem": "Pretzel",
            "Cals_per100grams": 338.0,
            "quantity": 176
        },
        {
            "foodItem": "Sugar",
            "Cals_per100grams": 405.0,
            "quantity": 102
        },
        {
            "foodItem": "Butter",
            "Cals_per100grams": 720.0,
            "quantity": 43
        }
    ],
    "Image": "https://img.sndimg.com/food/image/upload/
    ↪  w_555,h_416,c_fit,fl_progressive,q_95/v1/img/recipes/
    ↪  19/44/91/3xjO4aXTeiZpOajAsBRX_0S9A6246.jpg"
}
```

d. Now, exploiting the field `Cals_per100grams`, which has not been discarded yet, it is possible to compute the `totalCalories` of the recipe. This is done by executing the following operations for each recipe.

```
totalCalories = 0
for ingredient in ingredients:
    totalCalories += ingredient["quantity"] *
↪  (ingredient["Cals_per100grams"]/100)
```

e. In the end, the desired structure for `ingredients` was obtained.

```
Recipe_D: {
    "name": "pretzel crust",
    "id": 194491,
    "contributor_id": 356062,
    "submitted": "2006-11-07",
    "steps": [
        "preheat oven to 350",
        "crush pretzels in a blender",
        "add sugar and butter and mix well",
        "press into a 9 inch pie plate",
        "bake at 350 for 8 minutes and cool",
        "add desired filling"
    ],
    "description": "recipe for a basic pretzel crust i found in a
     ↪  magazine. i don't think i saw it posted here yet.",
    "ingredients": [
        {
            "foodItem": "Pretzel",
            "quantity": 176
        },
        {
            "foodItem": "Sugar",
            "quantity": 102
        },
        {
            "foodItem": "Butter",
            "quantity": 43
        }
    ],
    "totalCalories": 1317.58,
    "Image": "https://img.sndimg.com/food/image/upload/
        ↪  w_555,h_416,c_fit,fl_progressive,q_95/v1/img/recipes/
        ↪  19/44/91/3xjO4aXTeiZpOajAsBRX_0S9A6246.jpg"
    }
```

```
[Post]:
{
    _id: ObjectId('...'),
    idUser: ObjectId('...'),
    username: String, [REDUNDANCY]
    description: String,
    timestamp: Long,
    recipe: {
                name: String,
                image: String,
                steps: [String, ...],
                totalCalories: Double, [REDUNDANCY]
                ingredients: [{
                                name: String,
                                quantity: Double
                }, ...]
    },
    starRankings: [{
```

```
                        idUser: ObjectId('...'),
                        username: String, [REDUNDANCY]
                        vote: Double
    }, ...],
    avgStarRanking: Double, [REDUNDANCY]
    comments: [{
                idUser: ObjectId('...'),
                username: String, [REDUNDANCY]
                text: String,
                timestamp: Long
    }, ...]
}
```

//Aggiungere dimensione finale del dataset//

Recipes Merging: At this point, we merged the two files, to associate the images to the recipes. The file contains 231323 recipes. We then saved the ingredients not just as an array of string, but using the structure that we decided for our posts in the Post collection, generating for each of them the random quantity in the range we previously defined.

Ingredients Merging: We then removed duplicates from calories.csv, based on FoodItem field. At the end we merged ingredients.json with calories.csv using "replaced" and "FoodItem", to create a fixed structure of the possible ingredients that can be used in our social network. This association was made with a library called fuzzy wuzzy which merges string by likelyhood using Levenshtein Distance which is a metric used in information theory, linguistics, and computer science for measuring the difference between two sequences.

A questo punto, utilizzando il seguente script, abbiamo effettuato il matching per verosimiglianza, tra i file Ingredients.json e cleaned_calories.csv effettuando il matching tra le replaced (Ingredients.json) e FoodItem (cleaned_calories.csv).

Mappatura dei valori 'replaced': Utilizza fuzzywuzzy per mappare ciascun valore di replaced nel DataFrame degli ingredienti al valore più simile nel DataFrame delle calorie.

Comments Merging: We then eliminated all the comments of users that did not create a post (generated users which will be discussed later), to mantain a clean structure, ... //Da rivedere// The final result was a file that contained: user_id recipe_id timestamp text

5.4. Merging Process

After having cleand the datasets we started to merge them in order to obtain the information needed for populating the databases.

Here it is worth noticing that the data about the Users, because incomplete in the datasets we found (i.e. it was only provided the username of the users) was generated randomly using the Python library Faker. In this way the name and surname of the users are coherent with their username. The passwords are generated randomly too.

See Data... (# Load Estimation)

6. Statistics and Queries

6.1. CRUD operations

6.1.1. Create

- Create a new user
- Create a new post / recipe
- Create a new comment
- Create a new star ranking
- Create a new ingredient
- Create a new following relationship
- Create a new recipe-ingredient relationship
- Create a new user-ingredient relationship
- Create a new ingredient-ingredient relationship

6.1.1.1. MongoDB

- Create a new user

```
db.User.insertOne({
    username: String,
    password: hashedString,
    name: String,
    surname: String
})
```

- Create a new post

```
db.Post.insertOne({
    idUser: #,
    username: String,
    description: String,
    timestamp: Timestamp,
    recipe: {
                name: String,
                image: String_URL,
                steps: [String, ...],
                totalCalories: Double,
                ingredients: [{
                                name: String,
                                quantity: Double,
                }, ...]
    }
})
```

- Create a new comment

```
db.Post.updateOne({
    _id: #,
}, {
    $push: {
        comments: {
            idUser: #,
            username: String,
            text: String,
            timestamp: Timestamp
        }
    }
})
```

- Create a new star ranking

```
db.Post.updateOne({
    _id: #,
}, {
    $push: {
        starRankings: {
            idUser: #,
            username: String,
            vote: Double
        }
    }
})
```

- Create a new ingredient

```
db.Ingredient.insertOne({
    name: String,
    calories: Double,
})
```

6.1.1.2. Neo4j

- Create a new user

```
CREATE (u:User {
    _id: #,
    username: String
})
```

- Create a new recipe

```
CREATE (r:Recipe {
    _id: #,
    name: String
})
```

- Create a new ingredient

```
CREATE (i:Ingredient {
    _id: #,
    name: String
})
```

- Create a new following relationship

```
MATCH (u1:User {username: String}), (u2:User {username: String})
CREATE (u1)-[:FOLLOWS]->(u2)
```

- Create a new recipe-ingredient relationship

```
MATCH (r:Recipe {_id: #}), (i:Ingredient {name: String})
CREATE (r)-[:CONTAINS]->(i)
```

- Create a new user-ingredient relationship
  - Check if the relationship already exists
  - If it exists, increment the times attribute
  - If it doesn't exist, create the relationship

  ```
  MATCH (u:User {username: String}), (i:Ingredient {name: String})
  MERGE (u)-[r:USED]->(i) ON CREATE SET r.times = 1 ON MATCH SET r.times =
  ↪  r.times + 1
  ```

(Done after the creation of a Recipe) - Create a new ingredient-ingredient relationship - Check if the relationship already exists - If it exists, increment the times attribute - If it doesn't exist, create the relationship

```
MATCH (i1:Ingredient {name: String}), (i2:Ingredient {name: String})
MERGE (i1)-[r:USED_WITH]->(i2) ON CREATE SET r.times = 1 ON MATCH SET r.times =
↪  r.times + 1
```

Dobbiamo farlo in entrambi i versi!

6.1.2. Read

- Show users
- Shows posts
- Show comments of a Post
- Show star ranking of a Post
- Show ingredients
- Find ingredient by name
- Show friends
- Show followers
- Show followings
- Show calories of an ingredient
- Show steps of a Recipe
- Show recipe of a Post
- Show ingredients of a Recipe
- Show recipes filtering by the ingredients

- Show recipes filtering by the calories (lower bound and upper bound) (if the totalCalories is the same, we show the ones with the highest star ranking)
- Show the most/least recent posts (timestamp)

6.1.2.1. MongoDB

- Show users

```
db.User.find()
```

- Show posts

```
db.Post.find()
```

- Show comments

```
db.Post.find({
    _id: #,
}, {
    comments: 1
})
```

- Show star ranking of a Post

```
db.Post.find({
    _id: #,
}, {
    starRankings: 1
})
```

- Show ingredients

```
db.Ingredient.find()
```

- Find ingredient by name

```
db.Ingredient.find({
    name: String,
})
```

- Show calories of an ingredient (DA NON METTERE IN DB)

```
db.Ingredient.find({
    name: String,
}, {
    calories: 1
})
```

- Show steps of a Recipe

```
db.Post.find({
    _id: #,
}, {
    recipe: {
        steps: 1
    }
})
```

- Show recipe of a Post

```
db.Post.find({
    _id: #,
}, {
    recipe: {
        name: 1,
        image: 1,
        steps: 1,
        totalCalories: 1,
        ingredients: 1
    }
})
```

- Show ingredients of a Recipe

```
db.Post.find({
    _id: #,
}, {
    recipe: {
        ingredients: 1
    }
})
```

- Show recipes filtering by the calories (lower bound and upper bound) (if the totalCalories is the same, we show the ones with the highest star ranking)

```
db.Post.find({
    "recipe.totalCalories": {
        $gte: lowerBound,
        $lte: upperBound
    }
}).sort({
    avgStarRanking: -1
})
```

- Show the most recent posts (timestamp)

```
db.Post.find().sort({
    timestamp: -1
})
```

- Show the least recent posts (timestamp)

```
db.Post.find().sort({
    timestamp: 1
})
```

- Browse most recent top rated Posts

```
db.Post.find({
    timestamp: {
        $gte: Timestamp
    }
}).sort({
    avgStarRanking: -1
}).limit(10)
```

- Browse most recent top rated Posts by set of ingredients. It is necessary that the recipe contains all the ingredients of the set.

```
db.Post.find({
    timestamp: {
        $gte: Timestamp
    },
    "recipe.ingredients.name": {
        $all: [String, ...]
    }
}).sort({
    avgStarRanking: -1
}).limit(10)
```

- Browse most recent posts by minCalories and maxCalories

```
db.Post.find({
    timestamp: {
        $gte: Timestamp
    },
    "recipe.totalCalories": {
        $gte: minCalories,
        $lte: maxCalories
    }
}).sort({
    timestamp: -1
}).limit(10)
```

- Find post by id

```
db.Post.find({
    _id: #,
})
```

6.1.2.2. Neo4j

- Show followings / followed

```
MATCH (u1:User {username: String})-[:FOLLOWS]->(u2:User)
RETURN u2
```

- Show followers

```
MATCH (u1:User)-[:FOLLOWS]->(u2:User {username: String})
RETURN u1
```

- Show friends

```
MATCH (u1:User {username: String})-[:FOLLOWS]->(u2:User)-[:FOLLOWS]->(u1)
RETURN u2
```

- Show recipes filtering by the ingredients

```
MATCH (r:Recipe)-[:CONTAINS]->(i:Ingredient)
WHERE i.name IN [String, ...]   # List of ingredients
RETURN r
```

6.1.3. Update

- Update user's information
  - [username] : we have to mantain the consistency
  - password
  - name
  - surname
- Update post
  - description
- [Update recipe] : we have to mantain the consistency
- Update comment
- [Update user-ingredient relationship] (it is done in the create operation)
- [Update ingredient-ingredient relationship] (it is done in the create operation)

6.1.3.1. MongoDB

- Update user's information
- Update password

```
db.User.updateOne({
    _id: #,
}, {
    $set: {
        password: hashedString,
    }
})
```

- Update name

```
db.User.updateOne({
    _id: #,
}, {
    $set: {
        name: String,
    }
})
```

- Update surname

```
db.User.updateOne({
    _id: #,
}, {
    $set: {
        surname: String,
    }
})
```

- Update password, name, surname

```
db.User.updateOne({
    _id: #,
}, {
    $set: {
        password: hashedString,
        name: String,
        surname: String
    }
})
```

- Update post
- description

```
db.Post.updateOne({
    _id: #,
}, {
    $set: {
        description: String,
    }
})
```

- Update comment

```
db.Post.updateOne({
    _id: #,
    comments: {
        $elemMatch: {
            idUser: #,
            timestamp: Timestamp,
        }
    }
}, {
    $set: {
        "comments.$.text": String,
    }
})
```

6.1.4. Delete

- [Delete user] : we give the possibility to the user to delete his/her own profile.

When the user deletes his/her own profile, we have to delete all the posts of the user (calling the delete post operation) + we set "delete" = True

- Delete post
- Delete comment
- Delete star ranking
- Delete following relationship
- [Delete ingredient] : no because otherwise we would lose all the information about the ingredient (e.g. recipes, etc.)
- Delete user-ingredient relationship (see delete a post)
- Delete ingredient-ingredient relationship (see delete a post)

6.1.4.1. MongoDB

- Delete user Before deleting a user we have to call delete post for each post of the user But before we can have to delete the recipes inside the posts of the user from Neo4j (calling the delete recipe operation) After this operation we can delete the user from Neo4j

```
MATCH (u:User {username: String})
DELETE u
```

Now we can delete evry post of the user from the Post collection (delete post operation)

At this point we can delete all the fields from the user collection, leaving only the username

```
db.User.updateOne({
    _id: #,
}, {
    $unset: {
        password: "",
        name: "",
        surname: "",
        posts: "",
    }
})
```

Now we have to add a field called deleted to the user collection

```
db.User.updateOne({
    _id: #,
}, {
    $set: {
        deleted: true,
    }
})
```

- Delete post Before deleting a post we have to mantain the consistency of the DBs

We need to delete the post from the User collection

```
db.User.updateOne({
    _id: #,
}, {
    $pull: {
        posts: {
            idPost: #,
        }
    }
})
```

Delete Post

```
db.Post.deleteOne({
    _id: #,
})
```

We need to delete all the relationships of the recipe contained in the post from Neo4j

```
MATCH (r:Recipe {_id: #})
DETACH DELETE r
```

We need to update the times attribute of the relationships of the ingredients used together in the recipe contained in the post from Neo4j

~~~
MATCH (i1:Ingredient {name: String})-[r:USED_WITH]->(i2:Ingredient {name:
↪  String})
SET r.times = r.times - 1
IF r.times = 0 THEN
    DELETE r
END IF
``` [DA PROVARE]

We need to update the relationships among the user and the ingredients used in
↪  the recipe contained in the post from Neo4j
```javascript
MATCH (u:User {username: String})-[r:USED]->(i:Ingredient {name: String})
SET r.times = r.times - 1
~~~

~~~
IF r.times = 0 THEN
    DELETE r
END IF
``` [DA PROVARE]

Now we can delete the post from the Post collection
```javascript
db.Post.deleteOne({
    _id: #,
})
~~~

- Delete comment

```
db.Post.updateOne({
    _id: #,
}, {
    $pull: {
        comments: {
            idUser: #,
            timestamp: Timestamp,
        }
    }
})
```

- Delete star ranking

```
db.Post.updateOne({
    _id: #,
}, {
    $pull: {
        starRankings: {
            idUser: #
        }
    }
})
```

6.1.4.2. Neo4j

- Delete following relationship

```
MATCH (u1:User {username: String})-[r:FOLLOWS]->(u2:User {username: String})
DELETE r
```

6.1.5. Query

6.1.5.1. Analytics

- Show most active users (da vedere) (DA ELIMINARE)
- Show most followed users
- [Show post with most comments] (Ci serve veramente?)

- Show posts with the highest/lowest star ranking
- Show most/least used ingredients
- Show most/least used ingredients by a user
- Show total amount of calories of a recipe
- Show the average of the avgStarRanking of a User's posts
- Average amount of grams of ingredients used in equal set of ingredients.
- Find recipes filtering the name
- To add others…

6.1.5.1.1. MongoDB

- Show posts with the highest star ranking (if the avgStarRanking is the same, we show the most recent one)

```
db.Post.find().sort({
    avgStarRanking: -1,
    timestamp: -1
})
```

- Show total amount of calories of a recipe

```
db.Post.find({
    _id: #,
}, {
    recipe: {
        totalCalories: 1
    }
})
```

- Show the average of the avgStarRanking of a User's posts

```
db.Post.aggregate([
    {
        $match: {
            idUser: #,
        }
    },
    {
        $group: {
            _id: null,
            avgOfAvgStarRanking: {
                $avg: "$avgStarRanking"
            }
        }
    }
])
```

- Average amount of grams of ingredients used in equal set of ingredients. (Da implementare)

- Find Posts by recipes filtering the name

```
db.Post.find( { "recipe.name": { $regex: "pork", $options: "i" } } )
```

6.1.5.1.2. Neo4j

- Show most followed users

```
MATCH (u1:User)-[:FOLLOWS]->(u2:User)
RETURN u2, COUNT(u1) AS followers
ORDER BY followers DESC
LIMIT 5
```

- Show most used ingredients

```
MATCH (u:User)-[r:USED]->(i:Ingredient)
RETURN i, SUM(r.times) AS times
ORDER BY times DESC
LIMIT 5
```

- Show least used ingredients

```
MATCH (u:User)-[r:USED]->(i:Ingredient)
RETURN i, SUM(r.times) AS times
ORDER BY times ASC
LIMIT 5
```

- Show most used ingredients by a user

```
MATCH (u:User {username: String})-[r:USED]->(i:Ingredient)
RETURN i, r.times AS times
ORDER BY times DESC
LIMIT 5
```

(Not interesting) - Show least used ingredients by a user

```
MATCH (u:User {username: String})-[r:USED]->(i:Ingredient)
RETURN i, r.times AS times
ORDER BY times ASC
LIMIT 5
```

6.1.5.2. Suggestions

- Suggest users to follow (based on the user's friends)
- Suggest users to follow (based on common ingredients)
- Suggest most popular combination of ingredients
- Suggest new ingredients based on friends' usage
- Suggest most followed users
- Suggest Users by Ingredient usage

6.1.5.2.1. Neo4j

- Suggest users to follow (based on the user's friends)

```
MATCH (u1:User {username: String})-[:FOLLOWS]->(u2:User)-[:FOLLOWS]->(u3:User)
WHERE (u2)-[:FOLLOWS]->(u1)
AND NOT (u1)-[:FOLLOWS]->(u3)
RETURN u3
```

(NON SI implementa più) - Suggest users to follow (based on common ingredients). r1.times and r2.times must be both greater than threshold.

```
MATCH (u1:User {username: String})-[r1:USED]->(i:Ingredient)<-[r2:USED]-(u2:User)
WHERE NOT (u1)-[:FOLLOWS]->(u2)
AND r1.times > threshold
AND r2.times > threshold
RETURN u2
```

- Suggest most popular combination of ingredients

```
MATCH (i1:Ingredient)-[r:USED_WITH]->(i2:Ingredient)
RETURN i1, i2, r.times AS times
ORDER BY times DESC
LIMIT 5
```

- Suggest new ingredients based on friends' usage

```
MATCH (u1:User {username: String})-[:FOLLOWS]->(u2:User)-[r:USED]->(i:Ingredient)
WHERE (u2)-[:FOLLOWS]->(u1)
AND NOT (u1)-[:USED]->(i)
RETURN i, r.times AS times
ORDER BY times DESC
LIMIT 5
```

- Suggest most followed users

```
MATCH (u1:User)-[:FOLLOWS]->(u2:User)
RETURN u2, COUNT(u1) AS followers
ORDER BY followers DESC
LIMIT 5
```

- Suggest Users by Ingredient usage

```
MATCH (u:User)-[r:USED]->(i:Ingredient {name: String})
RETURN u, i, r.times AS times
ORDER BY times DESC
LIMIT 10
```

6.1.6. Aggregations

- (#1)Show the ratio of interactions (number of comments / number of Posts and number of star rankings / number of Posts) and the average of avgStarRanking distinguishing among posts with and without images (i.e. no field image inside recipe)

```
db.Post.aggregate([
    {
        $project: {
            _id: 1,
            hasImage: {
                $cond: {
                    if: {
                         $eq: [{ $type: "$recipe.image" }, "missing"]
                    },
                    then: false,
                    else: true
                }
            },
            comments: {
                $size: {
                    $ifNull: ["$comments", []]
                }
            },
            starRankings: {
                $size: {
                    $ifNull: ["$starRankings", []]
                }
            },
            avgStarRanking: {
                $ifNull: ["$avgStarRanking", 0]
            }
        }
    },
    {
        $group: {
            _id: "$hasImage",
            numberOfPosts: {
                $sum: 1
            },
            totalComments: {
                $sum: "$comments"
            },
            totalStarRankings: {
                $sum: "$starRankings"
            },
            avgOfAvgStarRanking: {
                $avg: "$avgStarRanking"
            }
        }
    },
    {
        $project: {
            _id: 0,
            hasImage: "$_id",
            ratioOfComments: {
                $divide: ["$totalComments", "$numberOfPosts"]
            },
            ratioOfStarRankings: {
                $divide: ["$totalStarRankings", "$numberOfPosts"]
            },
            avgOfAvgStarRanking: 1
        }
    }
])
```

- (#2)Given a User, show the number of comments he/she has done, the number of star rankings he/she has done and the average of this star rankings

```
db.Post.aggregate([
    {
        $match: {
            $or: [
                {"comments.username": "cody_cisneros_28"},
                {"starRankings.username": "cody_cisneros_28"}
            ]
        }
    },
    {
        $project: {
            filteredComments: {
                $filter: {
                    input: "$comments",
                    as: "comment",
                    cond: {$eq: ["$$comment.username", "cody_cisneros_28"]}
                }
            },
            filteredStarRankings: {
                $filter: {
                    input: "$starRankings",
                    as: "starRanking",
                    cond: {$eq: ["$$starRanking.username", "cody_cisneros_28"]}
                }
            }
        }
    },
    {
        $group: {
            _id: null,
            avgOfStarRankings: {
                $avg: {$sum: "$filteredStarRankings.vote"}
            },
            numberOfStarRankings: {
                $sum: {$size: "$filteredStarRankings"}
            },
            numberOfComments: {
                $sum: {$size: "$filteredComments"}
            }
        }
    },
    {
        $project: {
            _id: 0,
            numberOfComments: 1,
            numberOfStarRankings: 1,
            avgOfStarRankings: 1
        }
    }
])
```

- (#3)After filtering recipes by name, retrieve the average amount of calories of first 10 recipes ordered by descending avgStarRanking

```
db.Post.aggregate([
    {
        $match: {
            "recipe.name": { $regex: "pork", $options: "i" }
        }
    },
    {
        $sort: {
            avgStarRanking: -1
        }
    },
    {
        $limit: 10
    },
    {
        $group: {
            _id: null,
            avgOfTotalCalories: {
                $avg: "$recipe.totalCalories"
            }
        }
    },
    {
        $project: {
            _id: 0,
            avgOfTotalCalories: 1
        }
    }
])
```

We can use this to show when a post is shown, the average amount of calories of the recipes with similar name

Others…

- Given a user, show the average totalCalories of recipes publishished by him/her.

```
db.Post.aggregate([
    {
        $match: {
            username: "cody_cisneros_28"
        }
    },
    {   $project: {
            recipeCalories: "$recipe.totalCalories"
        }
    },
    {   $group: {
            _id: null,
            avgCalories: {
                $avg: "$recipeCalories"
            }
        }
    }
])
```

7. Redundancies

Ridondanze segnate come [REDUNDANCY] nella sezione database

Table 1: Redundancies introduced into the model.

(1) `DocumentDB:User:posts:name` **Reason**: To avoid joins. **Original/Raw Value**: `DocumentDB:Post:recipe:name`
(2) `DocumentDB:User:posts:image` **Reason**: To avoid joins. **Original/Raw Value**: `DocumentDB:Post:recipe:image`
(3) `DocumentDB:Post:username` **Reason**: To avoid joins. **Original/Raw Value**: `DocumentDB:User:username`

(4) `DocumentDB:Post:recipe:totalCalories`
Reason: To avoid joins and to avoid computing the total calories of a recipe every time a post is shown.
Original/Raw Value: It is possible to compute the total calories of a recipe by summing the calories of the ingredients contained in the recipe In particular the precise formula is the following: $\sum_i \left(quantity_i \cdot \frac{calories100g_i}{100} \right)$ where $quantity_i$ is the quantity of the i-th ingredient contained in the recipe and $calories100g_i$ is the amount of calories contained in 100 grams of the i-th ingredient that can be retrieved from the `Ingredient` collection.

---

(5) `DocumentDB:Post:avgStarRanking`
Reason: To avoid computing the average star ranking of a post every time is shown.
Original/Raw Value: It is possible to compute the average star ranking of a post by averaging the values contained in `DocumentDB:Post:starRankings:vote`

---

(6) `DocumentDB:Post:starRankings:username`
Reason: To avoid joins.
Original/Raw Value: `DocumentDB:User:username`

---

(7) `DocumentDB:Post:comments:username`
Reason: To avoid joins.
Original/Raw Value: `DocumentDB:User:username`

---

(8) `GraphDB:(User):username`
Reason: To avoid joins with the DocumentDB.
Original/Raw Value: `DocumentDB:User:username`

---

(9) `GraphDB:(Recipe):name`
Reason: To avoid joins with the DocumentDB.
Original/Raw Value: `DocumentDB:Post:recipe:name`

---

(10) `GraphDB:(Ingredient):name`
Reason: To avoid joins with the DocumentDB.
Original/Raw Value: `DocumentDB:Ingredient:name`

---

(11) `GraphDB:(User)-[:USED]->(Ingredient):times`
Reason: To avoid computing the total number of times that a User used an Ingredient.
Original/Raw Value: It is possible to compute the total number of times that a User used an Ingredient by counting the number of times that the User used that Ingredient in his/her recipes (information that can be retrieved from the DocumentDB).

---

(12) `GraphDB:(Ingredient)-[:USED_WITH]->(Ingredient):times`
Reason: To avoid computing the number of times that an ingredient is used with another one.
Original/Raw Value: It is possible to compute the number of times that an ingredient is used with another one by counting the number of times that all the users used these two ingredients together in their recipes (information that can be retrieved from the DocumentDB).

---

8. CONSISTENCY, AVAILABILITY AND PARTITION TOLERANCE [X]

8.1. Distributed Database Design

According to the non functional requirements expressed before, we should guarantee Availability and Partition tolerance, while consistency constraints can be relaxed. Indeed the application that we are designing is a social network, where the users are the main actors. We orient the design to the AP intersection of the CAP theorem ensuring eventual consistency. Indeed is important to always show some data to the user, even if it is not updated. For example, if a user is not able to see the latest posts of his friends, he will be a little disappointed but at the end it won't be the such a big problem because eventually it will be able to see them.

8.1.1. Replicas

We deployed mongoDB and neo4j with the following configuration: - MongoDB: 3 nodes (1 primary and 2 replicas DA VEDERE SE FARE 3 repliche con stessi poteri) - Neo4j: 1 node (1 primary) (we didn't implement the replicas because we would have needed the enterprise edition)

Read Operations: In WeFood, as in every Social Network, read operations are the most frequent and critical operations. For this reason we have to guarantee the lowest response time possible even if data is not updated to the latest version. So we decided to provide a response from the first available replica.

Write Operations: To ensure the low latency that we discussed before, write operations are considered successful when just a replica node (da sistemare dopo che si è scelta configurazione) wrote the data.

8.1.2. Handling inter database consistency

Using two different databases implemented redundancy of data, so for this reason any fail in insertion/up-date/deletion of data can cause inconsistencies between these to DBs, for this reason, in case of exceptions during write operations on one of the databases causes a rollback. If the operation succeeds on MongoDB, a success response is sent to the user, and the graph db becomes eventually consistent: if an exception occurs after this phase, a rollback operation starts bringing back the DBs in a state of consistency. Check write operations in the Replicas!

8.2. Sharding

In our application as it is implemented it is not useful to design the sharding approach. The main reason behind this decision is that we give the users the possibility to find the posts using completely uncorrelated filters that are not linked by a particular relationship (i.e. such as a common field). Indeed if for example we decided to shard the post collection by the timestamp field, we would have latency issues when we have to find the posts using other filters (e.g. by totalCalories). Indeed we would have to query all the shards and then merge the results. This would be a very inefficient approach. For this reason we decided to not implement the sharding approach.

For example if WeFood were implemented by considering a category for each recipe, where the category could be chosen from a fixed set of categories (e.g. geographic area, culture, ..), we could have decided to shard the post collection by the category field of the recipe. In this way we would have reached a good balancing of the load among the shards. But we decided not to proceed in this direction because we wanted to give the users the possibility to explore different recipes without any constraint.

We do not consider the sharding approach for the User collection because we do not expect the users to grow as much as the posts. Indeed we expect that the number of users will be much lower than the number of posts. For this reason we decided to not implement the sharding approach at all.

8.3. Configuration of MongoDB (ReplicaSet)

j = false (we do not handle sensitive data and there is no need to wait for the journal to be written to disk) w = 1 (write concern) wtimeout = 0 (handled in Server Java code)

Read Preferences

// Read Preferences at client level MongoClient mongoClient = MongoClients.create(“mongodb: //localhost: 27018, localhost: 27019, localhost: 27020/” + “?readPreference=nearest”);

NEAREST because in this way we can access the node with the lowest latency (Read from any member node from the set of nodes which respond the fastest. (Responsiveness measured in pings))

// Read Preferences at DB Level MongoDatabase db = mongoClient.getDatabase(s: “LSMDB”) .withReadPreference(ReadPreference. secondary ());

// Read Preferences at collection level MongoCollection myColl = db.getCollection(s: “students”) .withReadPreference(ReadPreference. secondary ());

8.4. Indexes and Constraints

8.4.1. MongoDB

Possible Indexes:

- Ingredient: name

db.Ingredient.createIndex({ “name”: 1 }, { unique: true })

In this way we also ensure the unique constraint on the name field

Find Ingredient by name

Index	nReturned	executionTimeMillis	totalKeysExamined	totalDocsExamined
No	1	4	0	1911
Yes	1	0	1	1

We can clearly see that by adding an index on the name field we can speed up the find operation. We do not have any disadvantage in adding this index because the writing operations are not frequent on the Ingredient collection. Because the admin is the only one that can add new ingredients and we do not expect that the admin will add new ingredients very frequently.

- Post: timestamp

Find the 100 most recent posts

Index	nReturned	executionTimeMillis	totalKeysExamined	totalDocsExamined
No	100	109	0	231323
Yes	100	3	100	100

Reason: we expect to have more read operations than write operations. And furthermore the writing operations are already ordered because posts are published in a chronological order.

- Post: recipe: totalCalories

Index	nReturned	executionTimeMillis	totalKeysExamined	totalDocsExamined
No	100	144	0	231323
Yes	100	7	100	100

- User: username

db.User.createIndex({ “username”: 1 }, { unique: true })

In this way we also ensure the unique constraint on the username field

Index	nReturned	executionTimeMillis	totalKeysExamined	totalDocsExamined
No	1	78	0	27901
Yes	1	2	1	1

Reason: since the user is the actor of the social network that performs the most number of operations and most of these operations are find operations (e.g. showing posts with different filters) we decided to implement indexes on the above fields. In this way we can speed up the find operations. We estimate that the number of find operations done by the admin will be negligible compared to the number of find operations done by the users.

try { // Prova ad inserire il documento collection.insertOne(nuovoDocumento); } catch (MongoWriteException e) { // Controlla se è un errore di duplicazione chiave if (e.getError().getCategory() == ErrorCategory.DUPLICATE_KEY) { System.out.println(“Impossibile inserire il documento: il valore è già presente.”); } else { System.out.println(“Errore durante l’inserimento del documento:” + e.getMessage()); } }

8.5. Indexes of Neo4j

Da discutere dopo aver caricato tutti i dati ed eventualmente prevedere indice per ricette che sono davvero tante

9. System Architecture

MVC

9.1. Backend

9.2. General description

9.3. Frameworks and components

10. Implementation

More details on the classes and their attributes are as follows.

Admin:

- username: `String`
- password: `String` (hashed)

RegisteredUser:

- username: `String`
- password: `String` (hashed)
- name: `String`
- surname: `String`

Post:

- user: `RegisteredUser`
- description: `String`
- timestamp: `Date`
- comments: `List<Comment>`
- starRankings: `List<StarRanking>`
- recipe: `Recipe`

Comment:

- user: `RegisteredUser`
- text: `String`
- timestamp: `Date`

StarRanking:

- user: `RegisteredUser`
- vote: `Double`

Recipe:

- name: `String`
- image: `String`
- steps: `List<String>`
- ingredients: `Map<Ingredient, Double>`

Ingredient:

- name: `String`
- calories: `Double`

11. PERFORMANCE TEST

12. USER MANUAL

13. References

[1] Calories in Food Items (per 100 grams) - https://www.kaggle.com/datasets/kkhandekar/calories-in-food-items-per-100-grams - Accessed: December 2023.

[2] Food.com Recipes and Interactions - https://www.kaggle.com/datasets/shuyangli94/food-com-recipes-and-user-interactions?select=PP_recipes.csv - Accessed: December 2023.

[3] Food.com - Recipes and Reviews - https://www.kaggle.com/datasets/irkaal/foodcom-recipes-and-reviews?select=reviews.csv - Accessed: December 2023.