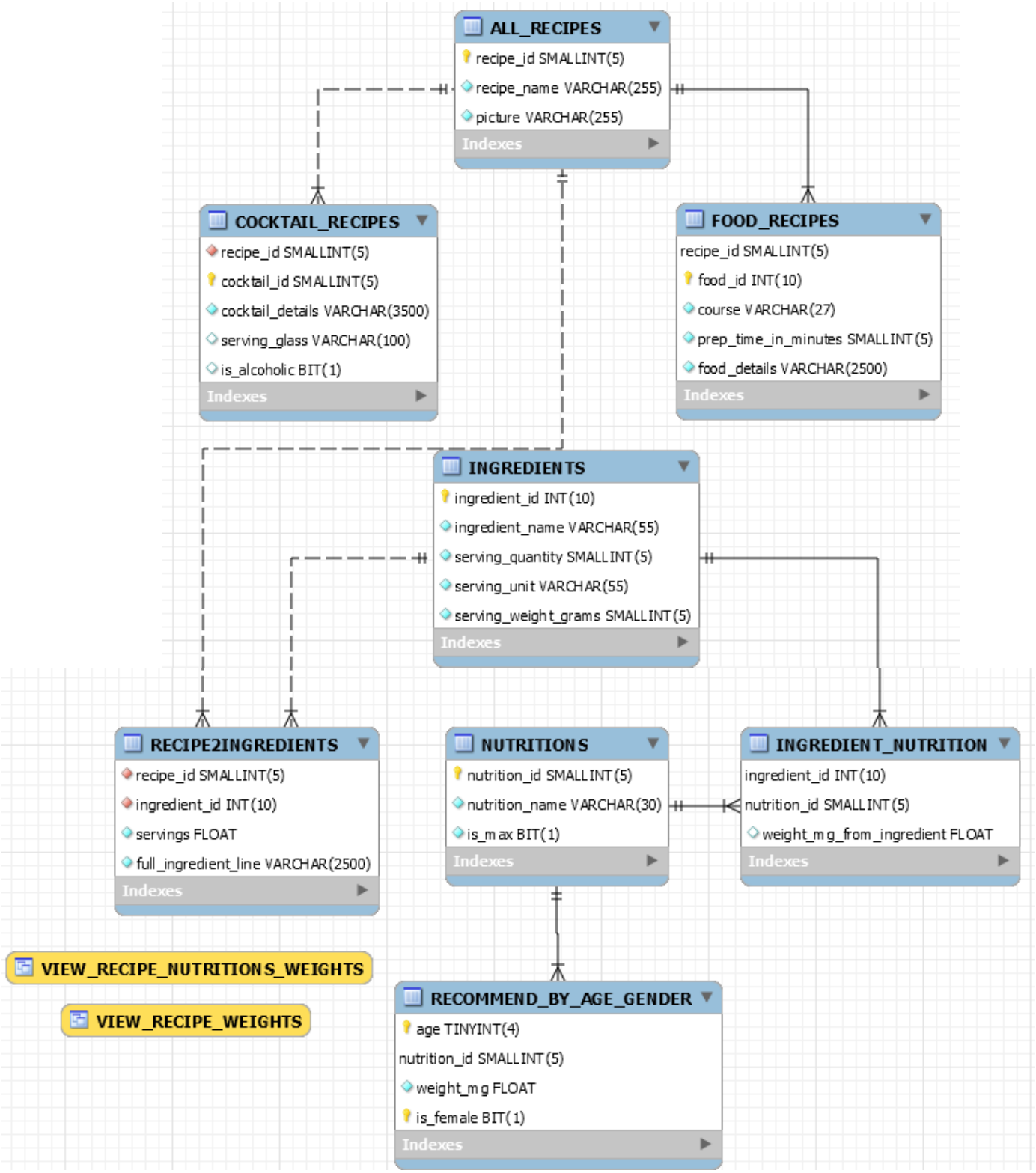
MATKONIM – Software Documentation

EER DIAGRAM:



DATA-BASE DESIGN PROCESS

From the beginning of the database design process, we knew that we needed different tables for food recipes, cocktail recipes, and ingredients. While progressing with the application's functionality and database designing, we made several decisions regarding the database's structure:

1. We realized food recipes and cocktail recipes had some common attributes. Therefore, we decided to add a more general table, ALL_RECIPES, that will include those common attributes and a unique recipe ID. COCKTAIL_RECIPES and FOOD_RECIPES tables are referring to that table using foreign keys, and each of those tables include attributes that relate to the specific type of recipe (food/cocktail).
2. We understood that each recipe contains a non-constant number of ingredients, and also each ingredient can appear in many recipes. Therefore, to avoid a many-to-many relation, we decided on a new table, RECIPE2INGREDIENTS, which connects each recipe to its ingredients and amounts of it.
3. At the beginning of the design process, we considered storing for each ingredient in each recipe its quantity and unit. Later on we realised that the conversion between different units is going to be harder than we thought, so instead we did this conversion before inserting the data into the DB, and stored the amounts as a factor of the serving size of each ingredient.
4. At first we thought about storing the nutritional values amounts of each ingredient in the INGREDIENTS table, but then we realized this will not be efficient. We then decided on 2 new tables – NUTRITIONS and INGREDIENT_NUTRITION. The first table includes the details of each nutritional value, while the second connects each ingredient to its nutritional values amounts. INGREDIENT_NUTRITION refers to both INGREDIENTS and NUTRITIONS tables by foreign keys. The reason for this 2 separate tables is keeping the database's consistency, and enabling simple updates and inserts to the database, if needed.
5. While writing queries that use NUTRITIONS table, we noticed that some of the nutritional values are considered "bad" (e.g. cholesterol, saturated fat), while others are considered "good" (e.g. iron, potassium). When the user inputs an amount of a nutritional value, we needed a way to know if this input is the maximal or minimal amount of the value. Meaning, for a "bad" nutritional value we would like recipes with at most user's input, and for a "good" nutritional value, we would like recipes with at least that amount. To solve this problem we added a "is_max" column which defines whether user's input is the maximal or minimal amount of this nutritient.
6. We contemplated on the best way to save the recommended daily intake for each age and gender, and finally decided on creating a new table – RECOMMENDED_BY_AGE_GENDER, which includes data we manually retrieved from the internet. We thought about what would be the best way to store different age groups, and eventually decided on creating a "virtual dictionary", in which every value 0-6 is representing a different age group.

DATABASE SCHEME STRUCTURE, TABLES AND OPTIMIZATIONS

ALL_RECIPES(recipe_id, recipe_name, picture)

This table lists information regarding all recipes in the database (both food and cocktail).
This table is referred to by a foreign key in FOOD_RECIPES, COCKTAIL_RECIPES and RECIPE2INGREDIENTS tables.

Columns:

- recipe_id: A surrogate primary key used to uniquely identify each recipe in the table.
- recipe_name: The recipe's name.
- picture: A URL to the recipe's picture.

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
recipe_id	SMALLINT	V	V	V	V	-
recipe_name	VARCHAR(255)	V	-	-	-	-
picture	VARCHAR(255)	V	-	-	-	-

Keys		
Column Name	Key type	Refers to
recipe_id	PRIMARY	-

Indexes		
Index Name	Index Type	Index Column
recipe_index_ar	PRIMARY	recipe_id

FOOD_RECIPES(food_id, recipe_id, course, prep_time_in_minutes, food_details)

This table lists all details regarding food recipes.

This table refers to ALL_RECIPES table by a foreign key.

Columns:

- food_id: A key used to identify each food recipe in the table.
- recipe_id: A foreign key identifying the recipe in ALL_RECIPES table.
- course: The course of the food recipe.
- prep_time_in_minutes: The amount of minutes required to prepare this recipe.
- food_details: A URL to the full recipe preparation instructions.

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
food_id	INT	V	-	V	-	-
recipe_id	SMALLINT	V	-	V	V	-
course	VARCHAR(27)	V	-	-	-	-
prep_time_in_minutes	SMALLINT	V	-	V	-	-
food_details	VARCHAR(2500)	V	-	-	-	-

Keys		
Column Name	Key type	Refers to
food_id, recipe_id	PRIMARY	-
recipe_id	FOREIGN	ALL_RECIPES.recipe_id

Indexes		
Index Name	Index Type	Index Column
food_id_fr	INDEX	food_id
recipe_id_index_fr	INDEX	recipe_id
Course_name_fr	FULLTEXT	course

COCKTAIL_RECIPES(cocktail_id, recipe_id, is_alcoholic, serving_glass, cocktail_details)

This table lists all details regarding cocktail recipes.

This table refers to ALL_RECIPES table by a foreign key.

Columns:

- cocktail_id: A primary key used to uniquely identify each cocktail recipe in the table.
- recipe_id: A foreign key identifying the recipe in ALL_RECIPES table.
- is_alcoholic: Identifying if the cocktail is alcoholic or not.
- serving_glass: The recommended serving glass of the cocktail recipe.
- cocktail_details: Full cocktail preparation instructions.

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
cocktail_id	SMALLINT	V	V	V	-	-
recipe_id	SMALLINT	V	V	V	-	-
is_alcoholic	BIT	V	-	-	-	-
serving_glass	VARCHAR(100)	-	-	-	-	NULL
cocktail_details	VARCHAR(3500)	V	-	-	-	-

Keys		
Column Name	Key type	Refers to
cocktail_id	PRIMARY	-
recipe_id	FOREIGN	ALL_RECIPES.recipe_id

Indexes		
Index Name	Index Type	Index Column
cocktail_id_cr	PRIMARY	cocktail_id
recipe_id_cr	INDEX	recipe_id
Is_alcoholic_cr	INDEX	is_alcoholic

INGREDIENTS(ingredient_id, ingredient_name, serving_quantity, serving_unit, serving_weight_grams)

This table lists all details regarding all ingredients that appear in food or cocktail recipes in the DB. This table is referred to by foreign keys in RECIPE2INGREDIENTS and INGREDIENT_NUTRITION tables.

Columns:

- ingredient_id: A surrogate primary key used to uniquely identify each ingredient in the table.
- Ingredient_name: The name of the ingredient.
- serving_quantity: The quantity of "units" in a single serving of the ingredient.
- serving_unit: The size of a "unit" in a single serving of the ingredient.
- serving_weight_grams: The weight (in grams) of a single serving of the ingredient.

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
ingredient_id	INT	V	V	V	V	-
ingredient_name	VARCHAR(55)	V	-	-	-	-
serving_quantity	SMALLINT	V	-	V	-	-
serving_unit	VARCHAR(55)	V	-	-	-	-
serving_weight_grams	SMALLINT	V	-	V	-	-

Keys		
Column Name	Key type	Refers to
ingredient_id	PRIMARY	-

Indexes		
Index Name	Index Type	Index Column
ingredient_id_ing	PRIMARY	ingredient_id
ingredient_name_ing	FULLTEXT	ingredient_name

RECIPE2INGREDIENTS(recipe_id, ingredient_id, servings, full_ingredient_line)

This table connects each recipe to its ingredients and amounts.

This table refers to ALL_RECIPES and INGREDIENTS tables by foreign keys.

Columns:

- recipe_id: A foreign key identifying the recipe in ALL_RECIPES table.
- ingredient_id: A foreign key identifying the recipe in INGREDIENTS table.
- servings: How many servings of the ingredient are present in the recipe.
- full_ingredient_line: The full text line of the ingredient to present to the user.

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
recipe_id	SMALLINT	V	-	V	-	-
ingredient_id	INT	V	-	V	-	-
servings	FLOAT(7)	V	-	V	-	-
full_ingredient_line	VARCHAR(2500)	V	-	-	-	-

Keys		
Column Name	Key type	Refers to
recipe_id	FOREIGN	ALL_RECIPES.recipe_id
ingredient_id	FOREIGN	INGREDIENTS.ingredient_id

Indexes		
Index Name	Index Type	Index Column
recipe_id_r2i	INDEX	recipe_id
ingredient_id_r2i	INDEX	ingredient_id

INGREDIENT_NUTRITION(ingredient_id, nutrition_id, weight_mg_from_ingredient)

This table connects each ingredient to the amount of each nutritional value in one serving of it.
This table refers to INGREDIENTS and NUTRITIONS tables by foreign keys.

Columns:

- ingredient_id: A foreign key identifying the ingreidnet in INGREDIENTS table.
- nutrition_id: A foreign key identifying the nutritional value in NUTRITIONS table.
- weight_mg_from_ingredient: The weight (in mg) of the nutritional value in one serving of the ingredient.

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
ingredient_id	INT	V	-	V	-	-
nutrition_id	SMALLINT	V	-	V	-	-
weight_mg_from_ingredient	FLOAT(3)	-	-	-	-	0

Keys		
Column Name	Key type	Refers to
ingredient_id, nutrition_id	PRIMARY	-
ingredient_id	FOREIGN	INGREDIENTS.ingredient_id
nutrition_id	FOREIGN	NUTRITIONS.nutrition_id

Indexes		
Index Name	Index Type	Index Column
ingredient_id_in	INDEX	ingredient_id
Nutrition_id_in	INDEX	nutrition_id

NUTRITIONS(nutirition_id, nutrition_name, is_max)

This table lists all details regarding all nutritional values present in the DB.
This table is referred to by foreign keys in INGREDIENT_NUTRITION and RECOMMENDED_BY_AGE_GENDER tables.

Columns:

- nutrition_id: A surrogate primary key uniquely identifying the nutritional value in the DB.
- nutrition_name: The name of the nutritional value.
- Is_max: Defines whether the user's input should be the maximum (1/True) amount or the minimum(0/False) amount of this nutritional value. For example if the nutritional value is a bad one (e.g. cholesterol), user's input will be the maximum amount of this nutrient. If the nutritional value is a good one (e.g. iron), user's input will be the minimum amount of this nutrient.

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
nutrition_id	SMALLINT	V	V	V	V	-
nutrition _name	VARCHAR(30)	V	-	-	-	-
Is_max	BIT	V	-	-	-	-

Keys		
Column Name	Key type	Refers to
nutrition_id	PRIMARY	-

Indexes		
Index Name	Index Type	Index Column
nut_id_nut	PRIMARY	nutrition_id
Is_max_nut	INDEX	Is_max
nut_name_nut	FULLTEXT	nutrition_name

RECOMMENDED_BY_AGE_GENDER(is_female, age, nutrition_id, weight_mg)

This table lists all recommended daily intake amounts of each nutritional value, per age and gender. This table refers to NUTRITIONS table by a foreign key.

Columns:

- is_female: The value 1/True will represent a female gender for which the recommended amounts are (male – 0/False).
- age: The age group for which the recommended amounts are, by this dictionary:
    - 0 is ages 14-18
    - 1 is ages 19-30
    - 2 is ages 31-40
    - 3 is ages 41-50
    - 4 is ages 51-60
    - 5 is ages 61-70
    - 6 is ages 71+
- nutrition_id: A foreign key identifying the nutritional value in NUTRITIONS table.
- weight_mg: The recommended intake of the nutritional value for this age and gender.

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
Is_female	BIT	V	-	-	-	-
age	TINYINT	V	-	-	-	-
nutrition_id	SMALLINT	V	-	V	-	-
weight_mg	FLOAT(3)	V	-	-	-	0

Keys		
Column Name	Key type	Refers to
gender, age, nutrition_id	PRIMARY	-
nutrition_id	FOREIGN	NUTRITIONS.nutrition_id

Indexes		
Index Name	Index Type	Index Column
nutrition_id_rbag	INDEX	nutrition_id
weight_mg_rbag	INDEX	weight_mg
if_female_age_rbag	INDEX	Is_female, age

VIEWS

VIEW_RECIPE_WEIGHTS(weight, recipe_id)

```
CREATE VIEW VIEW_RECIPE_WEIGHTS AS
SELECT DISTINCT ROUND(SUM(r2i.servings * i.serving_weight_grams), 2) as weight, r2i.recipe_id as recipe_id
FROM            INGREDIENTS i
INNER JOIN      RECIPE2INGREDIENTS r2i on i.ingredient_id = r2i.ingredient_id
GROUP BY r2i.recipe_id
```

This view gets for each recipe its total weight by summing up all of its ingredients weights. Each ingredient has *serving_weight_grams* value, and the *servings* attribute from RECIPE2INGREDIENTS table, tells us how much sevings of this ingredient are insde this recipe, so the query multiply these values, and summing them up using the "SUM" aggregation, when it's GROUPED BY the recipe id – so we get for each recipe all of its ingredients.

Columns:

- weight: The total weight of the recipe in grams
- recipe_id: The recipe id from ALL_RECIPES

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
recipe_id	SMALLINT(5)	V	-	-	-	-
Weight	DOUBLE	V	-	-	-	-

Optimizations made for this view:

1. INNER JOIN - we get only the rows which are connected by the ingredient_id. It's better than joining all of the rows together and inserting (where i.ingredient_id=r2i.ingredient_id).
2. The recipe_id is an index for the RECIPE2INGREDIENTS table which improves the GROUP BY operation.

VIEW_RECIPE_NUTRITIONS_WEIGHTS(recipe_id, nutrition_id, weight, precentage)

```
CREATE VIEW VIEW_RECIPE_NUTRITIONS_WEIGHTS AS
SELECT DISTINCT r2i.recipe_id as recipe_id, inn.nutrition_id as nutrition_id, ROUND(SUM(r2i.servings *
inn.weight_mg_from_ingredient), 2) as weight,
ROUND((SUM(r2i.servings * inn.weight_mg_from_ingredient) / (vrw.weight * 1000)),2) as percentage
FROM                INGREDIENT_NUTRITION inn
INNER JOIN          RECIPE2INGREDIENTS r2i
                    on inn.ingredient_id = r2i.ingredient_id
INNER JOIN   VIEW_RECIPE_WEIGHTS vrw on vrw.recipe_id = r2i.recipe_id
GROUP BY r2i.recipe_id, inn.nutrition_id
```

This view gets for each recipe the weight of each nutritional value weight inside the recipe, and its percentage from the total recipe weight. Meaning, for each recipe and for each nutritional value, this view calculates the nutritient's weight (by summing up all of this nutritient's weight in all of the recipe's ingredients), and calculates its percentage by diving the recipe's total weight. There isn't a table which connect between recipes and nutritions, but there is one which connect between ingredients and nutritions: INGREDIENT_NUTRITION. So, in order to get the total weight of a certain nutrition inside the recipe, this query gets the nutrition weight for each this recipe's ingredients by *weight_mg_from_ingredient* and multiply it by the amount of the ingredient's servings inside this recipe by *servings.* This calculation is for the nutrition weight inside the ingredient. To get the percentage for this nutrition inside the recipe the query divids the nutrition weight by the recipe weight which is gathered from *VIEW_RECIPE_WEIGHTS.* (The summing is grouped by both recipe_id and nutrition_id, because we want each nutrition value for each recipe).

Columns:

- recipe_id: The recipe id from ALL_RECIPES
- nutrition_id: The nutrition id from NUTRITIONS
- weight: The weight of the current nutrition inside the current recipe
- percentage: The percentage of the current nutrition inside the current recipe

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
recipe_id	SMALLINT(5)	V	-	-	-	-
nutrition_id	SMALLINT(5)	V	-	-	-	-
weight	FLOAT	V	-	-	-	-
precentage	FLOAT	V	-	-	-	-

Optimizations made for this query:

1. INNER JOIN, we get only the rows which are connected by the ingredient_id and by the recipe_id. It's better than joining all of the rows together and inserting (where i.ingredient_id=r2i.ingredient_id).
2. The ingredient_id is an index inside the *RECIPE2INGREDIENTS,* INGREDIENT_NUTRITION table which improves the inner join operation and the group by.
3. The recipe_id is an index inside the RECIPE2INGREDIENTS which improves the inner join and the group by.

COMPLEX QUERIES AND DB OPTIMIZATIONS

In the following queries, only an example of the query will be presented, since user's input sets the query's parameters, and for every parameter the query looks a bit different. Note: in the server's program, each query that is being executed is printed by the server to stdout, so you can see them.

Query 1 – Recipe by nutritional values

```
SELECT DISTINCT fr.recipe_id
FROM FOOD_RECIPES as fr
INNER JOIN (
        SELECT DISTINCT COUNT(fr.recipe_id) as cnt, fr.recipe_id as recipe_id
        FROM FOOD_RECIPES fr
        INNER JOIN VIEW_RECIPE_NUTRITIONS_WEIGHTS vrnw
                on fr.recipe_id = vrnw.recipe_id
        INNER JOIN NUTRITIONS n
                on vrnw.nutrition_id = n.nutrition_id
        WHERE
        MATCH(fr.course) AGAINST("Main Dishes")
        AND fr.prep_time_in_minutes <= 60
        AND ( MATCH(n.nutrition_name) AGAINST("sugar")          OR
                MATCH(n.nutrition_name) AGAINST("protein")      OR
                MATCH(n.nutrition_name) AGAINST("cholesterol") OR
                MATCH(n.nutrition_name) AGAINST("lactose")      OR
                MATCH(n.nutrition_name) AGAINST("calories_kcal")   )
        AND
        (NOT MATCH(n.nutrition_name AGAINST("sugar") OR vrnw.precentage <= 0.05)
        AND
        (NOT MATCH(n.nutrition_name) AGAINST("protein") OR vrnw.precentage >= 0.15)
        AND
        (NOT MATCH(n.nutrition_name) AGAINST("cholesterol") OR vrnw.precentage <= 0.05)
        AND
        (NOT MATCH(n.nutrition_name) AGAINST("lactose") OR vrnw.precentage = 0)
        AND
        (NOT MATCH(n.nutrition_name) AGAINST("calories_kcal") OR vrnw.weight <= 2000)
        GROUP BY fr.recipe_id
) RECIPE_COUNTERS on RECIPE_COUNTERS.recipe_id = fr.recipe_id
WHERE RECIPE_COUNTERS.cnt = 5
```

This query finds all the recipes which apply to user's chosen course, nutritional percentages preferences, maximal preparation time, and maximal calories. In the above example:

1. It's a Main Dish.
2. The preparation time needed is less than 60 minutes.
3. The sugar percentage in the recipe is less than 5%.
4. The protein percentage in the recipe is more than 15%.
5. The cholesterol percentage in the recipe is less than 5%.
6. There isn't lactose in this recipe.
7. The maximum amount of calories is 2000.

Of course in the application the user can choose other nutritional values and other restrictions for them, as well as a different course, maximal preparation time, etc. A value not chosen by the user is defaulted as "Don't Care" and isn't included in the above query by the server.
This query has a inner query inside it named *RECIPE_COUNTERS.* In this sub-query we check how much of the restrictions are applied. We check that the recipe course is the chosen one, and the preparation time limit (if defined). After that we care about only the specific nutritions that we chose, so we checked that the nutrition_name is one of the chosen nutritions. After that we made a little trick: (Reminder from discrete mathematics: A=>B ⇔ (not A) or B). And we wanted to check that if the ingredient is (for example) sugar so the percentage of the current nutrition is less than 5% so we checked: "not sugar or percentage less than 5%", and we count the amount of restrictions exists by counting the results for every recipe and group by the recipe_id. We now got the precentages for each nutritional value from VIEW_RECIPE_WEIGHTS by getting the total weight and then dividing the calculated nutrition weight by it. The outer query then gets all of the recipes that has the number of restrictions, in this case it's 5.

<u>Optimizations made for this query:</u>

1. In order to get the precentages for each nutrition we added the VIEW_RECIPE_WEIGHTS to get the total weight and then divide the calculated nutrition weight by it.
2. INNER JOIN, we get only the rows which are connected by the nutrition_id and by the recipe_id.
3. The nutrition_name has an index inside NUTRITIONS table for a faster lookup. We didn't want to search by the nutrition_id which would be faster because than we would have to save the mapping in our server.
4. The recipe_id and the nutrition_id are indexed in FOOD_RECIPES and NUTRITIONS tables, so the inner join is faster.

Query 2 – Recipe by Allergies

```
SELECT DISTINCT ar.recipe_id AS recipe_id, ar.recipe_name AS recipe_name
FROM        ALL_RECIPES AS ar
INNER JOIN  FOOD_RECIPES fr on ar.recipe_id = fr.recipe_id
WHERE
MATCH(fr.course) AGAINST("Lunch")
AND ar.recipe_id NOT IN (
        SELECT DISTINCT r2i.recipe_id
        FROM        INGREDIENTS ing
        INNER JOIN RECIPE2INGREDIENTS r2i on r2i.ingredient_id = ing.ingredient_id
        WHERE MATCH (ing.ingredient_name) AGAINST("squid")
        )
AND ar.recipe_id NOT IN (
        SELECT DISTINCT r2i.recipe_id
        FROM        INGREDIENTS ing
        INNER JOIN RECIPE2INGREDIENTS r2i on r2i.ingredient_id = ing.ingredient_id
        WHERE MATCH(ing.ingredient_name) AGAINST("milk")
        )
AND ar.recipe_id NOT IN (
        SELECT DISTINCT r2i.recipe_id
        FROM        INGREDIENTS ing
        INNER JOIN RECIPE2INGREDIENTS r2i on r2i.ingredient_id = ing.ingredient_id
        WHERE MATCH(ing.ingredient_name) AGAINST("egg yolk")
        )
GROUP BY ar.recipe_id
```

This query gets all of the food recipes where the course is user's chosen course (in the above example, "lunch") that do not contain user's input ingredients, or ingredients that have similar names. In the above example those are: "Squid", "Milk", and "Egg Yolk".
The user inputs if he wants a food recipe(and chooses the course the he wants), a cocktail, or the default which chooses randomly between a cocktail or a food of some course. The user also inputs 1 to 3 ingredients that are chosen with help from the auto complete feature, that will be discluded from the recipe that will be chosen eventually.
This query has an inner query inside it called “alg_query”, which selects from RECIPE2INGREDIENTS table all of the ingredients (for all recipes) that have a name resembling one of the input ingredients from the user. Our query then checks that the recipes we select are not inside this query result – meaning that it does not contain an ingredient like the one the user has requested to leave out.

Optimizations made for this query:

1. INNER JOIN, we get only the rows that are connected by the recipe_id’s for cocktails or food(according to what the user chose).
2. The search for ingredients names uses full text search in order to enhance the performance of the search.

Query 3 – Recipe by Recommended Daily Intake

```
SELECT DISTINCT fr.recipe_id
FROM FOOD_RECIPES as fr,
INNER JOIN (
SELECT DISTINCT COUNT(fr.recipe_id) as cnt, fr.recipe_id as recipe_id
FROM NUTRITIONS n
INNER JOIN RECOMMEND_BY_AGE_GENDER rbag
                on n.nutrition_id = rbag.nutrition_id
INNER JOIN VIEW_RECIPE_NUTRITIONS_WEIGHTS vrnw
                on rbag.nutrition_id = vrnw.nutrition_id
INNER JOIN FOOD_RECIPES fr
                on fr.recipe_id = vrnw.recipe_id
WHERE rbag.is_female = 1 AND rbag.age = 3 AND
MATCH(fr.course) AGAINST "Main Dishes"
AND (           MATCH(n.nutrition_name) AGAINST("iron")  OR
                MATCH(n.nutrition_name) AGAINST("protein")  OR
                MATCH(n.nutrition_name) AGAINST("saturated")
)
AND   ( NOT MATCH(n.nutrition_name) AGAINST("iron") OR (
                (n.is_max = 0 AND vrnw.weight / rbag.weight_mg >= 0.23)
                OR
                (n.is_max = 1 AND vrnw.weight / rbag.weight_mg <= 0.23)
        )
)
AND   ( NOT MATCH(n.nutrition_name) AGAINST("protein") OR (
                (n.is_max = 0 AND vrnw.weight / rbag.weight_mg >= 0.55)
                OR
                (n.is_max = 1 AND vrnw.weight / rbag.weight_mg <= 0.55)
        )
)
AND   ( NOT MATCH(n.nutrition_name) AGAINST("saturated") OR (
                (n.is_max = 0 AND vrnw.weight / rbag.weight_mg >= 0.1)
                OR
                (n.is_max = 1 AND vrnw.weight / rbag.weight_mg <= 0.1)
        )
)
GROUP by fr.recipe_id
) RECIPE_COUNTERS on RECIPE_COUNTERS.recipe_id = fr.recipe_id
WHERE RECIPE_COUNTERS.cnt = 3
```

In this query the user chooses what nutritional values he would like to limit, by percentages from the recommended daily intake bsed on his/her age and gender, as specified in RECOMMEND_BY_AGE_GENDER table. In order to do so we Inner joined the tables NUTRITIONS, RECOMMEND_BY_AGE_GENDER, FOOD_RECIPES, VIEW_RECIPE_NUTRITIONS_WEIGHTS by the appropriate keys which have indexes for each.

By doing that, we were able to search by the nutrition name, age and gender what is the recommended nutrition weight. We checked that if the nutrition name is "iron" for example, than the percentage should be lower or greater than it, according to the *is_max* column inside NUTRITIONS table. This column specifies if the daily intake should be greater than the value specified or less than it. After that at the outer query we checked the amout of restrictions that were applied.

Optimizations made for this query:

1. We used VIEW_RECIPE_NUTRITIONS_WEIGHTS view, which gets for each recipe the weight of each nutritional value weight inside the recipe, and its percentage from the total recipe weight.
2. Inner join the tables by a certain key which has index – so we get only the rows we want and it faster due to the index.
3. We used COUNT and GROUP BY using recipe_id which also has index so the group by would be faster.

Query 4 – First Trivia Question

"In <RECIPE_NAME>, which of the following is the main nutritional value?"

```
SELECT DISTINCT n.nutrition_id, n.nutrition_name
FROM NUTRITIONS n
INNER JOIN
(
	SELECT DISTINCT r2i.recipe_id as recipe_id, inn.nutrition_id as nutrition_id, SUM(r2i.servings *
	inn.weight_mg_from_ingredient) as weight
	FROM		INGREDIENT_NUTRITION inn
	INNER JOIN
			on inn.ingredient_id = r2i.ingredient_id
	WHERE r2i.recipe_id = <RECIPE_ID>
	GROUP BY r2i.recipe_id, inn.nutrition_id
) vrnw on n.nutrition_id = vrnw.nutrition_id
INNER JOIN(
	SELECT DISTINCT v.recipe_id, MAX(v.weight) as max_weight
	FROM (
		SELECT DISTINCT r2i.recipe_id as recipe_id, inn.nutrition_id as nutrition_id,
		SUM(r2i.servings * inn.weight_mg_from_ingredient) as weight
		FROM		INGREDIENT_NUTRITION inn
		INNER JOIN RECIPE2INGREDIENTS r2i
			on inn.ingredient_id = r2i.ingredient_id
		WHERE r2i.recipe_id = <RECIPE_ID>
		GROUP BY r2i.recipe_id, inn.nutrition_id
	 ) AS v
	GROUP BY v.recipe_id
) max_nut_table on vrnw.weight = max_nut_table.max_weight
```

In order to get the trivia question, the server randomly selects a recipe using a separate non-complex query. Then we calculate the main nutritional value for this recipe using the above query – this query returns the nutritional values which has the max weight inside this recipe. Then, we use another query to get us 3 random differnet nutritions.

We made a sub query (vrnw) that calculates all of the nutritions weights for the specified <RECIPE_ID>. It uses the RECIPE2INGREDIENTS table in order to get all of the recipe ingredients, and then uses INGREDIENT_NUTRITION table to get all of the nutritions values for all of the ingredients. Then it sums up their weight by multiplying the amount of servings (specified in RECIPE2INGREDIENTS) by the weight of the nutritional values inside the ingredient (specified in INGREDIENT_NUTRITION). In order to do so for all of the nutritions inside the recipe we group by the recipe_id. Then, we made another sub query (max_nut_table) that gets the maximum nutrition weight along with the recipe_id, to do that it uses the same sub query from above. We inner joined the vrnw which contains all of the nutritions weights for the recipe with the max_nut_table which contains the maximum nutrition weight, and we joined it on the weight so we get only the line with the correct nutrition_id. We join it with the NUTRITIONS table on the nutrition_id so we get the wanted nutrition_name.

Optimizations made for this query:

1. We used inner queries with keys that have indexes so the join is faster.
2. The group by is on columns which has indexes so it's faster too.
3. We could have used the VIEW_RECIPE_NUTRITIONS_WEIGHTS but we didn't because we didn't care about all of the recipes, so we re-written it and limit it to only the recipe that we wanted – so it won't calculate all of the recipes.
4. We search by recipe_id so we didn't need to do an inner join with ALL_RECIPES table which would take time as we have a lot of recipes.

Query 5 – Second Trivia Question

"Which of the following recipes contains the most <NUTRITION_NAME>?"

```
SELECT DISTINCT     r2i.recipe_id as recipe_id, inn.nutrition_id as nutrition_id,
                    SUM(r2i.servings * inn.weight_mg_from_ingredient) as weight
FROM            RECIPE2INGREDIENTS r2i
INNER JOIN INGREDIENT_NUTRITION inn
                    on inn.ingredient_id = r2i.ingredient_id
WHERE
inn.nutrition_id = <NUTRITION_ID>
AND
(
        r2i.recipe_id = <RECIPE_ID1> OR
        r2i.recipe_id = <RECIPE_ID2> OR
        r2i.recipe_id = <RECIPE_ID3> OR
        r2i.recipe_id = <RECIPE_ID4>
)
GROUP BY    r2i.recipe_id, inn.nutrition_id
HAVING weight >= ALL
(
        SELECT DISTINCT SUM(r2i.servings * inn.weight_mg_from_ingredient) as weight
        FROM            RECIPE2INGREDIENTS r2i
        INNER JOIN INGREDIENT_NUTRITION inn
                        on inn.ingredient_id = r2i.ingredient_id
        WHERE
        inn.nutrition_id = <NUTRITION_ID> AND
        (
                r2i.recipe_id = <RECIPE_ID1> OR
                r2i.recipe_id = <RECIPE_ID2> OR
                r2i.recipe_id = <RECIPE_ID3> OR
                r2i.recipe_id = <RECIPE_ID4>
        )
        GROUP BY r2i.recipe_id, inn.nutrition_id
)
```

In order to get the trivia question, the server randomly selects a nutritional value using a separate non-complex query, then it randomly selects 4 random recipes using a separate non-complex query. Then, we calculate the nutritional weight for all of the chosen recipes using the above query – this query returns the recipe which has the max weight of the specified nutrition.

The part before the "having" calculates the weight of the nutrition for all of the given recipes, multiplies the amount of servings (specified in RECIPE2INGREDIENTS table) by the weight per serving (specified in INGREDIENT_NUTRITION table). This is because we need to get all of the nutritions weights from all of the ingrdients in the recipes. We made an inner join between this table by the ingredient_id key.

Then we checked that we calculate for the recipes that we want and for the nutrition that we want. And we grouped by recipe_id and nutrition_id so it would be unique value for every recipe and nutrition.

The "having" part inner query does the same – calculates the nutrition weight for all of the recipes. Then the "having" part chooses for us the recipe,nutrition row which its weight is >= from ALL of the results in the inner query so we get only the line which has the maximum weight and we can extract the recipe_id from it.

Optimizations made for this query:

1. The inner join is done on columns which have indexes so it much faster.
2. We could have used the VIEW_RECIPE_NUTRITIONS_WEIGHTS but we wanted only the specified nutrition for the specified recipes so we re-written it in order to not calculate the weights for all of the recipes and nutritions so it would run faster.
3. The having and sum is grouped by keys which has indexes so the grouping is done faster.

Query 6 – Third Trivia Question

"Which of the following ingredients that is at least <PRECENTAGE>% <NUTRITION_NAME> from the daily intake of <GENDER> at ages <AGE>, appears in the most recipes?(clue: more than <MIN_RECIPES> recipes)"

```
SELECT i.ingredient_name, i.ingredient_id
FROM INGREDIENTS i
INNER JOIN (
        SELECT r2i.ingredient_id as ingredient_id, count(r2i.recipe_id) as cnt
        FROM NUTRITIONS n
        INNER JOIN INGREDIENT_NUTRITION inn
                        on n.nutrition_id = inn.nutrition_id
        INNER JOIN RECOMMEND_BY_AGE_GENDER rbag
                        on rbag.nutrition_id = n.nutrition_id
        INNER JOIN RECIPE2INGREDIENTS r2i
                        on r2i.ingredient_id = inn.ingredient_id
        WHERE
        n.nutrition_id = <NUTRITION_ID>
        AND
        rbag.age = <AGE> AND rbag.is_female = <GENDER>
        AND
        inn.weight_mg_from_ingredient / rbag.weight_mg >= <RANDOM_PRECENTAGE>
        GROUP BY r2i.ingredient_id, r2i.recipe_id
        HAVING cnt >= <MIN_RECIPES>
        ORDER BY cnt DESC
) INGREDIENTS_COUNT on i.ingredient_id = INGREDIENTS_COUNT.ingredient_id
LIMIT 1
```

This query receives an ingredient, which:

- For the given age, gender, nutrition and percentage, this ingredient contains at least the specified percentage of recommended daily intake for those age and gender of the given nutrition.
- From all of the other ingredients it appears in the maximum number of recipes.
- It appears in at least <MIN_RECIPES> recipes(min_recipes is also given).

The sub query (INGREDIENTS_COUNT) searches for all the ingredients which for the given nutrition, age, gender and percentage are at least <PRECENTAGE> of the daily intake for <GENDER> at ages <AGE> of the given <NUTRITION>. For example: all the ingredients which are at least 16% from the sugar's daily intake for female in ages 21-30. It does that by inner joining the RECIPE2INGREDIENTS table (in order to get the number of recipes which this ingredient appears at) with INGREDIENT_NUTRITION (in order to get the nutrition values of that ingredient), and with RECOMMEND_BY_AGE_GENDER (in order to get the daily intake of <GENDER> and <AGE>) and then it checks it with the given <PRECENTAGE>. Later using the "having" it gets all of these ingredients which appears in more than <MIN_RECIPES> recipes.

Then in order to get the ingredient which appears in most of the recipes we order by the count of the recipes (which the ingredient appears at).

Then at the outer query we inner join this sub query with INGREDIENTS table in order to get the name of the result ingredient. And then we LIMIT 1 in order to get the max ingredient which appears at the max number of recipes amongst all of the result ingredients from the sub query.

Optimizations made for this query:

1. We made inner join with indexed keys so the join is done faster, and we get only the rows which we want.
2. We chose only the fields that we want so it's more compact.
3. The count is done with a GROUP BY over an indexed columns so it's faster too.

Query 7 – Cocktail by nutritional values

```
SELECT DISTINCT cr.recipe_id
FROM COCKTAIL_RECIPES cr
INNER JOIN (
	SELECT DISTINCT COUNT(cr.recipe_id) as cnt, cr.recipe_id as recipe_id
	FROM COCKTAIL_RECIPES cr
	INNER JOIN VIEW_RECIPE_NUTRITIONS_WEIGHTS vrnw
			on cr.recipe_id = vrnw.recipe_id
	INNER JOIN NUTRITIONS n
			on vrnw.nutrition_id = n.nutrition_id
	WHERE
	cr.is_alcoholic = 1
	AND ( MATCH(n.nutrition_name) AGAINST("cholesterol") OR
			MATCH(n.nutrition_name) AGAINST("sugar")  OR
			MATCH(n.nutrition_name) AGAINST("lactose"))
	AND   (NOT MATCH(n.nutrition_name) AGAINST("cholesterol") OR vrnw.precentage <= 0.05)
	AND   (NOT MATCH(n.nutrition_name) AGAINST("sugar") OR vrnw.precentage >= 0.15)
	AND   (NOT MATCH(n.nutrition_name) AGAINST("lactose") OR vrnw.precentage = 0)
	GROUP BY cr.recipe_id
) RECIPE_COUNTERS on RECIPE_COUNTERS.recipe_id = cr.recipe_id
WHERE RECIPE_COUNTERS.cnt = 3
```

This query finds all the cocktails which apply to user's chosen nutritional percentages preferences and alcohol inclusion. In the above example:

1. The cocktail is alcoholic.
2. The cholesterol percentage in the recipe is less than 5%.
3. The sugar percentage in the recipe is more than 15%.
4. There isn’t lactose in this recipe.

Of course in the application the user can choose other nutritional values and other restrictions for them, as well as a non-alcoholic cocktail, maximal calories, etc. A value not chosen by the user is defaulted as "Don't Care" and isn't included in the above query by the server.

This query has an inner query inside, named RECIPE_COUNTERS. In this sub-query we check how much of the restrictions are applied, and whether the cocktail is alcoholic. After that we care about only the specific nutritions that we chose, so we checked that the nutrition_name is one of the chosen nutritions. And we wanted to check that if the ingredient is (for example) potassium so the percentage of the current nutrition is 0. We count the amount of restrictions exists by counting the results for every recipe and group by the recipe_id. Also, in order to get the precentages for each nutrition we user the VIEW_RECIPE_WEIGHTS by getting the total weight and then dividing the calculated nutrition weight by it. The outer query then gets all of the recipes that has the number of restrictions, in this case it’s 3.

Optimizations made for this query:

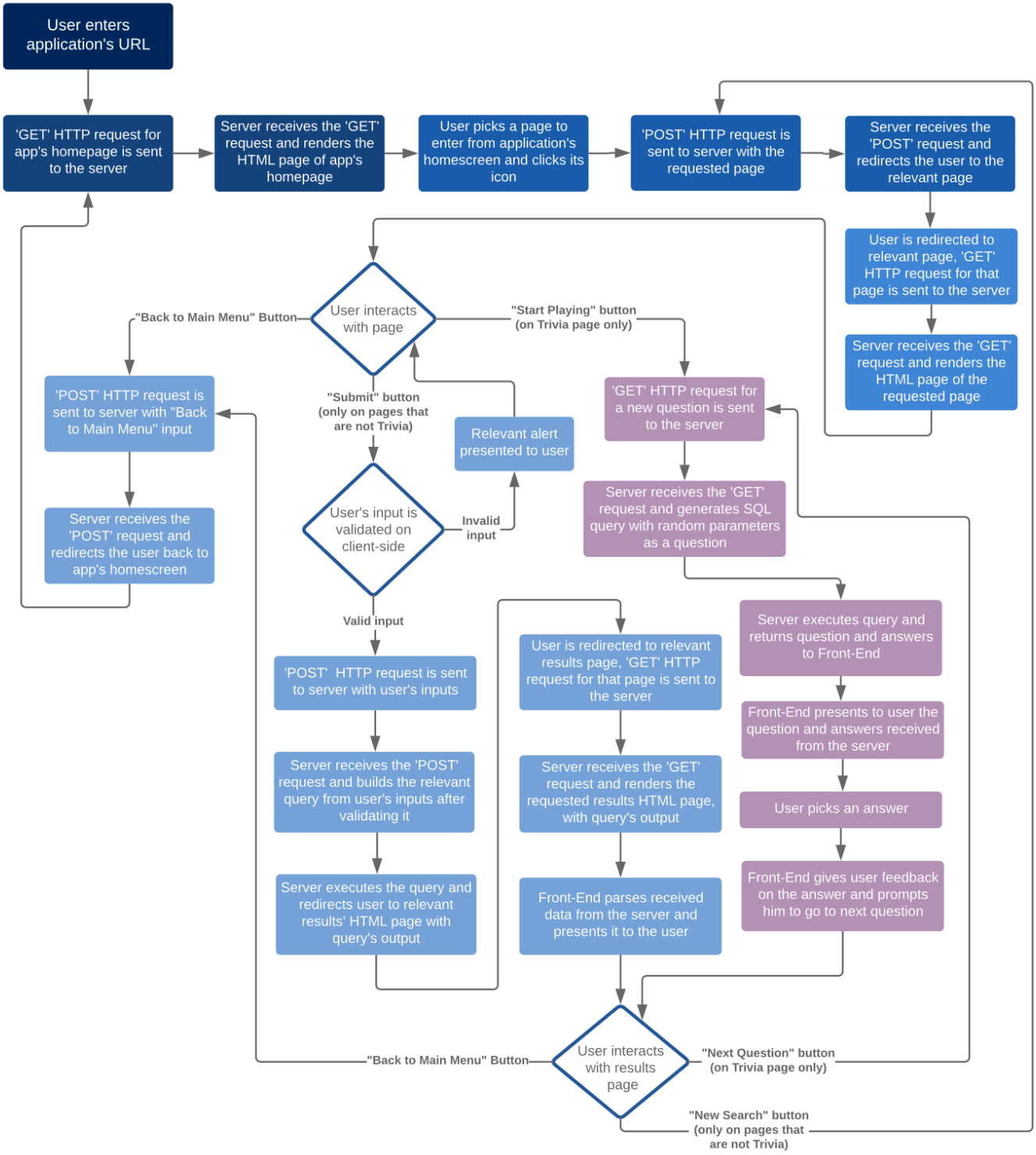
1. In order to get the precentages for each nutrition we added the VIEW_RECIPE_WEIGHTS to get the total weight and then divide the calculated nutrition weight by it.
2. INNER JOIN, we get only the rows which are connected by the nutrition_id and by the recipe_id.
3. The nutrition_name has an index inside NUTRITIONS table, so it's faster. We didn't want to search by the nutrition_id which would be faster because than we would have to save the mapping in our server.
4. The recipe_id and the nutrition_id are indexed in COCKTAIL_RECIPES *and* NUTRITIONS tables, so the inner join is faster.

CODE STRUCTURE

Matkonim-master:

- /SRC
  - /API-DATA-RETRIVAL
    - /yummly_retrival.py
    - /cocktailDB_retrival.py
    - /nutritionix_retrival.py
    - /retrival_utils.py
    - /dataFromRecipes.py
    - /dataFromCocktails.py
    - /dataFromIngredients.py
    - /dataFromDaily.py
    - /commonFile.py
    - /main.py
    - /recipes.csv
    - /cocktails.csv
    - /ingredients.csv
    - /daily_intake.csv
  - /APPLICATION-SOURCE-CODE
    - /static
    - /templates
      - homepage.html
      - recipes_by_nutritional.html
      - recipes_by_nutritional_results.html
      - cocktails_by_nutritional.html
      - cocktails_by_nutritional_results.html
      - daily_meal_plan.html
      - daily_meal_plan_results.html
      - recipes_by_allergies.html
      - recipes_by_allergies_results.html
      - nutrivia.html
    - /queries.py
    - /server.py
    - /utils.py
  - /CREATE-DB-SCRIPT.sql
- /DOCUMENTATION
  - /URL-TO-THE-APP.txt
  - /NAMES-AND-IDS.txt
  - /USER-MANUAL.pdf
  - /SOFTWARE-DOCS.pdf
  - /MYSQL-USER-AND-PASSWORD.txt

GENERAL APP FLOW



EXTERNAL PACKAGES / LIBRARIES USED

BACK-END

- Flask – a micro web framework for Python, used for development of the server-side of the application and communication with client-side.
- Python Requests – library for sending HTTP requests, used for API retrieval.
- WSGIServer - library for creating the HTTP server and run it at our server.

FRONT-END

- Bootstrap – a free open-source front-end framework, used for development of the user interface of the application.

API USAGE, RETRIEVAL AND INTEGRATION PROCESS

In our application we are using 3 API's:

1. Yummly
2. CocktailDB
3. Nutritionix

Yummly API

This is the main API we used. It was used for retrieving over 10,000 different food recipes. The retrieval from this API was done with the Python script "yummly_retrival.py" which does the following:

- Sends a single request to the API for a list of all possible courses. The response is received as a JSON dictionary. Overall we used 5 different courses out of the possible 13 Yummly offers.
- For each course in the abovementioned dictionary, it sends 20 consecutive requests, each request for metadata of 200 recipes of that course. Therefore, metadata of 4000 recipes for each course was retrieved. This metadata was written into a JSON file.
    - The reason this is done in 20 requests instead of one request for 4000 recipes, is that Yummly tends to compress large responses, in a way that Python's Requests library was unable to deal with. Requesting for a relatively small amount of recipes kept the response un-zipped and so we were able to parse it.
- For each recipe's metadata dictionary in the created metadata JSON file, it sends a single request to the API, for the recipe's details, using its ID. For each recipe retrieved, it excludes irrelevant information from the JSON returned, and writes it to a JSON file of all recipe details.
    - Yummly API has a limit of 30,000 API calls per user per lifetime, so we used multiple accounts for the retrieval process, which were switched by the script while running.
- The script parses the recipes details JSON file into a CSV file containing all the relevant information that we need for the DB, for each recipe.
- Other issues we encountered:
    - Some recipes included non-latin UNICODE characters which we had to ASCII-encode.
    - A recipe's metadata JSON includes an array of its ingredients, and the recipe's details JSON includes an array of ingredient full-description line. The amounts of each ingredient needed to be parsed and converted from those 2 arrays by us.
    - Some ingredients' amounts were inconsistent (e.g. "two and a half", "2.5", "2 1/2", etc.) and we had to parse them to eliminate as many inconsistencies as we could.

CocktailDB API

This is the second API we used. It was used for retrieving ~500 different cocktail recipes. The retrieval from this API was done with the Python script "cocktailDB_retrival.py" which does the following:

- Sends a single request to the API for a list of all possible ocktail categories. The response is received as a JSON dictionary.
- For each category in the abovementioned dictionary, it sends a single request for metadata of all cocktails in that category. The results are saved in a JSON array.
- For each cocktail's metadata dictionary in the created metadata JSON array, it sends a single request to the API, for the cocktail's details, using its ID. For each cocktail retrieved, it excludes irrelevant information from the JSON returned, and writes it to a CSV file of all cocktail details.
- Other issues we encountered:
    - Some cocktails included non-latin UNICODE characters which we had to ASCII-encode.
    - The amount of each ingredient in the cocktails had to be parsed and converted. Some ingredient names were written inconsistently, and had to be parsed and replaced.

Nutritionix API

This is the third API we used. It was used for retrieving ~3500 different ingredients. The retrieval from this API was done with the Python script "nutritionix_retrival.py" which does the following:

- It retrieves a list of ingredients from the food and cocktail CSV/JSON files. It eliminates duplicates and left us with only the unique ingredients we needed to retrieve from the API.
- Given the abovementioned array of ingredients, it sends a request to the API for details of 10 ingredients each time. The results are saved in a JSON file of all ingredients details.
    - The reason we retrieved 10 ingredients each time, is that we wanted to retrieve as many ingredients in one request as possible, but we saw that when requesting too many ingredients, the retrieval fails to find results for some of them.
    - Nutritionix API has a limit of 100 API calls per user per day, so we used multiple accounts for the retrieval process, which were switched by the script while running.
- The script parses the ingredients details JSON file into a CSV file containing all the relevant information that we need for the DB, for each ingredient.
- Other issues we encountered:
    - Some nutritional values were in grams, while others while in mg, so we had to convert.
    - The amount of each ingredient in the cocktails had to be parsed and converted.

Fixing the data

After retrieving the ingredients from Nutritionix API, we noticed that not all ingredients were found by the API. Therfore, we had to eliminate the recipes that contained ingredients that weren't retrieved. This was done with a Python script called "retrival_utils.py" which does the following:

- It goes over the cocktails and food CSV files and eliminates recipes that include ingredients not retrieved.
- It fixes the amounts of ingredients in each recipe to be the number of servings, according to the size of a single serving found in the ingredients CSV file.
- It outputs results to fixed cocktails and food CSV files, which are later used to construct the DB.

BONUS

1. INFORMATION SECURITY:

   While building our application, we put information security as a high priority. Firstly, the DB is only accessible by the server-side of the app, and the client-side has no communication with the DB. Secondly, we planned our application to be as immune to SQL Injection attacks as possible. Most user inputs are received through buttons and drop-down menus (so the users have a close choice between possible inputs and cannot construct one on their own). All the fields that allow free-text input, are validated in client-side and blocks input of illegal characters (depending on the field and its meaning, for example only digits allowed in "calories" field, only latin letters allowed in "allergies" field). We also implemented auto-complete feature for the allergies fields, and blocked the possibility to input text that is not part of the fixed auto-completed ingredient list. In addition to the client-side validation mentioned above, we performed input validation in server side, and checked that the inputs given are part of a pre-declared set of possible inputs, before executing any query. This was done to prevent a user from sending a malicious HTTP request directly to our server and bypassing the client-side input validation.

2. USER INTERFACE:

   Our application's user interface is based on Bootstrap libraries. It was originally designed by us and was created from scratch without a template. The use of Bootstrap makes the app comfortable to use in many platforms. Each feature in the application is accompanied by a short explanation paragraph, in order to clarify to the user all the different possibilities. We used JQuery to make our user interface interactive and responsive to user's actions. Furthermore, our user interface is (objectively, of course) very pretty.