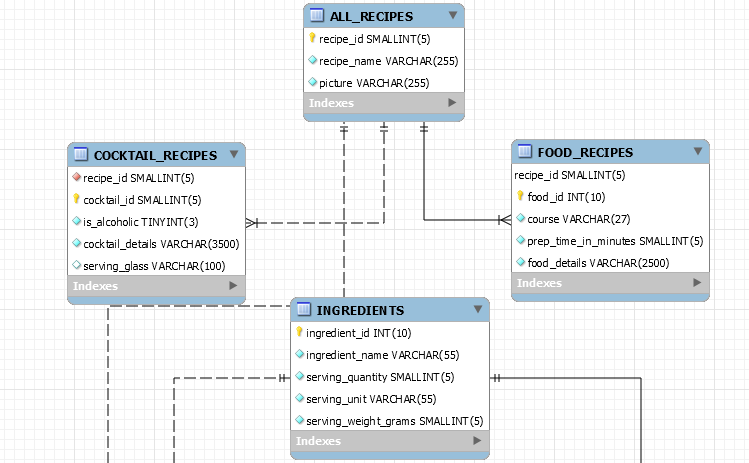
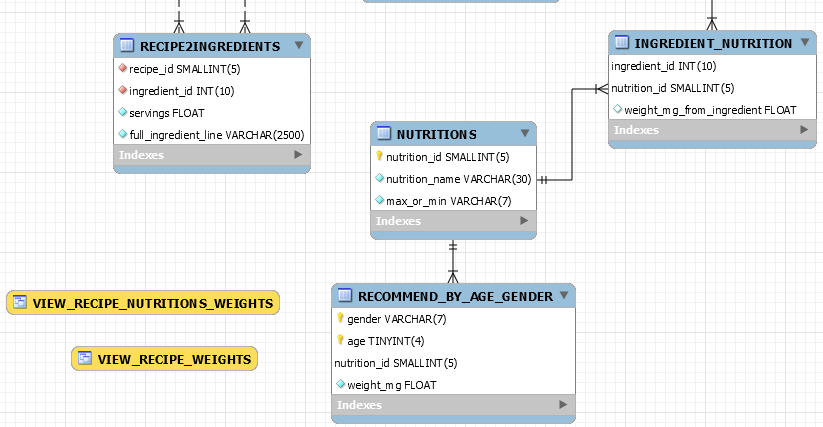
**MATKONIM – Software Documentation**

**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 "max\_or\_min" 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 | INDEX | food\_id |
| recipe\_id\_index\_food\_rec | INDEX | recipe\_id |
| course | 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 | TINYINT | V | - | 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 | PRIMARY | cocktail\_id |
| recipe\_id\_index\_cocktail | INDEX | recipe\_id |

**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 | PRIMARY | ingredient\_id |
| ingredient\_name\_index | 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 | INDEX | recipe\_id |
| ingredient\_id\_rec2ing | 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\_ingred\_nutr | INDEX | ingredient\_id |

**NUTRITIONS(nutirition\_id, nutrition\_name, max\_or\_min)**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.
* max\_or\_min: Defines whether the user's input should be the maximum amount or the minimum 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 | - | - | - | - |
| max\_or\_min | VARCHAR(7) | V | - | - | - | - |

|  |  |  |
| --- | --- | --- |
| Keys | | |
| Column Name | **Key type** | **Refers to** |
| nutrition\_id | PRIMARY | - |

|  |  |  |
| --- | --- | --- |
| Indexes | | |
| Index Name | **Index Type** | **Index Column** |
| nut\_id\_index | PRIMARY | nutrition\_id |
| nut\_name\_index | INDEX | nutrition\_name |

**RECOM****MENDED\_BY\_AGE\_GENDER(gender, 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:

* gender: The gender for which the recommended amounts are (either male or female).
* 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** |
| gender | VARCHAR(7) | 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** |
| nut\_id\_index\_recom\_gender\_age | INDEX | nutrition\_id |
| gender\_age\_index | INDEX | gender, age |
| weight\_mg\_index\_recom\_gender\_age | INDEX | weight\_mg |

**DATABASE VIEWS**

**VIEW\_RECIPE\_WEIGHTS(weight, recipe\_id)**This view calculates for each recipe its total weight. It sums up all of its ingredients to get the total weight of the recipe. Each Ingredient weight is calculated by its *serving\_weight\_grams* multiplied by the *servings* in *RECIPE2INGREDIENTS.*

The view query will be exaplained in “COMPLEX QUERIES AND DB OPTIMIZATION” section.

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 | - | - | - | - |

**VIEW\_RECIPE\_NUTRITIONS\_WEIGHTS(recipe\_id, nutrition\_id, weight, precentage)**This view calculates for each recipe all of the nutritional values. Meaning, for each recipe and for each nutrition, this view calculates the nutrition weight (by summing up all of this nutrition weight in all of the recipe’s ingredients), and calculates its percentage by diving the recipe’s total weight. (Each ingredient is linked to all of the nutritions by the *INGREDIENT\_NUTRITION* table, and has its weight as *weight\_mg\_from\_ingredient*).

The view query will be exaplained in “COMPLEX QUERIES AND DB OPTIMIZATION” section.

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 | - | - | - | - |

**COMPLEX QUERIES AND DB OPTIMIZATIONS**

**VIEW 1 – VIEW\_RECIPE\_WEIGHTS**

CREATE VIEW VIEW\_RECIPE\_WEIGHTS AS

SELECT DISTINCT SUM(r2i.servings \* i.serving\_weight\_grams) as weight, r2i.recipe\_id as recipe\_id

FROM RECIPE2INGREDIENTS r2i

INNER JOIN INGREDIENTS i on i.ingredient\_id = r2i.ingredient\_id

GROUP BY r2i.recipe\_id

This query is also a view which explained in the views section.

This query gets for each recipe its total weight by summing up all of its ingredients weights. Each ingredient has *serving\_weight\_grams* value, and insde *RECIPE2INGREDIENTS* the *servings* 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.

Optimizations:

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 2 – VIEW\_RECIPE\_NUTRITIONS\_WEIGHTS**

CREATE VIEW VIEW\_RECIPE\_NUTRITIONS\_WEIGHTS AS

SELECT DISTINCT r2i.recipe\_id as recipe\_id, inn.nutrition\_id as nutrition\_id, SUM(r2i.servings \* inn.weight\_mg\_from\_ingredient) as weight,

(SUM(r2i.servings \* inn.weight\_mg\_from\_ingredient) / (vrw.weight \* 1000)) as precentage

FROM RECIPE2INGREDIENTS r2i

INNER JOIN INGREDIENT\_NUTRITION inn

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 query is also a view which explained in the views section.

This query gets for each recipe and nutrition the nutrition’s weight inside the recipe, and its percentage from the total recipe 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).

Optimizations:

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.

*In the following queries, we’ll present an example for them (because at the website we can set all of the parameters, so for every parameter the query looks a bit different. Note, at the server each query that is being executed is being printed also at the server, so you can view 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

fr.course = "Main Dishes"

AND fr.prep\_time\_in\_minutes <= 60

AND ( n.nutrition\_name = "sugar" OR

n.nutrition\_name = "protein" OR

n.nutrition\_name = "cholesterol" OR

n.nutrition\_name = "lactose" OR

n.nutrition\_name = "calories\_kcal" )

AND (n.nutrition\_name <> "sugar" OR vrnw.precentage <= 0.05)

AND (n.nutrition\_name <> "protein" OR vrnw.precentage >= 0.15)

AND (n.nutrition\_name <> "cholesterol" OR vrnw.precentage <= 0.05)

AND (n.nutrition\_name <> "lactose" OR vrnw.precentage = 0)

AND (n.nutrition\_name <> "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 gets all of the recipes which the following rules are applied upon them:

1. It’s lunch
2. The preparation time needed is less than 90 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 calories is 2000.

* Of course at the website we can choose other nutritions and other restrictions for the nutritions, and a course other than “lunch”, and a different preparation time or “don’t care” if you don’t care.

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.

Also 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.

The outer query then gets all of the recipes that has the number of restrictions, in this case it’s 5.

Query Optimizations:

1. INNER JOIN, we get only the rows which are connected by the nutrition\_id and by the recipe\_id.
2. The nutrition\_name has an index inside *NUTRITIONS* 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.
3. The recipe\_id and the nutrition\_id are indexed in *FOOD\_RECIPES and NUTRITIONS* 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 RECIPE2INGREDIENTS r2i on ar.recipe\_id = r2i.recipe\_id

INNER JOIN FOOD\_RECIPES fr on r2i.recipe\_id = fr.recipe\_id

WHERE

fr.course = "Soups"

AND ar.recipe\_id NOT IN (

SELECT DISTINCT r2i.recipe\_id

FROM RECIPE2INGREDIENTS r2i

INNER JOIN INGREDIENTS ing 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 RECIPE2INGREDIENTS r2i

INNER JOIN INGREDIENTS ing 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 RECIPE2INGREDIENTS r2i

INNER JOIN INGREDIENTS ing on r2i.ingredient\_id = ing.ingredient\_id

WHERE MATCH(ing.ingredient\_name) AGAINST("egg yolk")

)

GROUP BY ar.recipe\_id

**NEED TO BE EXPLAINED**

**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 FOOD\_RECIPES fr

INNER JOIN VIEW\_RECIPE\_NUTRITIONS\_WEIGHTS vrnw

on fr.recipe\_id = vrnw.recipe\_id

INNER JOIN RECOMMEND\_BY\_AGE\_GENDER rbag

on rbag.nutrition\_id = vrnw.nutrition\_id

INNER JOIN NUTRITIONS n on n.nutrition\_id = rbag.nutrition\_id

WHERE rbag.gender = "female" AND rbag.age = 3 AND

fr.course = "Main Dishes"

AND ( n.nutrition\_name = "iron" OR

n.nutrition\_name = "potassium" OR

n.nutrition\_name = "saturated" )

AND ( n.nutrition\_name <> "iron" OR (

(n.max\_or\_min = "min" AND vrnw.weight / rbag.weight\_mg >= 0.23)

OR

(n.max\_or\_min = "max" AND vrnw.weight / rbag.weight\_mg <= 0.23)

)

)

AND ( n.nutrition\_name <> "potassium" OR (

(n.max\_or\_min = "min" AND vrnw.weight / rbag.weight\_mg >= 0.55)

OR

(n.max\_or\_min = "max" AND vrnw.weight / rbag.weight\_mg <= 0.55)

)

)

AND ( n.nutrition\_name <> "saturated" OR (

(n.max\_or\_min = "min" AND vrnw.weight / rbag.weight\_mg >= 0.13)

OR

(n.max\_or\_min = "max" AND vrnw.weight / rbag.weight\_mg <= 0.13)

)

)

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 nutritions he would like to limit, and by what percent from the recommended daily intake by his/her age and gender specified at *RECOMMEND\_BY\_AGE\_GENDER*. 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 that we could search by the nutrition name, age and gender what is the recommended nutrition weight. We checked that if the nutrition name is “satuated” for example, than the percentage should be lower or greater than it, by the *max\_or\_min* column inside *NUTRITIONS.* 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.

Query Optimizations –

1. 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.
2. We used COUNT and GROUP BY using recipe\_id which also has index so the group by would be faster.

**Query 4 - Daily Meal Plan by Recommended Daily Intake**

**MAYBE CHANGE WITH ANOTHER ONE?**

**Query 5 – 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 RECIPE2INGREDIENTS r2i

INNER JOIN INGREDIENT\_NUTRITION inn

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 RECIPE2INGREDIENTS r2i

INNER JOIN INGREDIENT\_NUTRITION inn

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 questions we select a random recipe by a different small query. Then we calculate the main nutritional value using the above query – this query returns the nutrition which has the max weight inside this recipe. After that we use another query to get us 3 random differnet nutritions.*

Query explanation:

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 the *INGREDIENT\_NUTRITION* 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.

After that 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.

Then we inner join 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.

Then we join it with the *NUTRITIONS* table on the nutrition\_id so we get the wanted nutrition\_name.

Query Optimizations:

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 6 – 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 questions we select a random nutrition by a different small query. After that we select 4 random recipes by a different small 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*

Query explanation:

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)* by the weight per seving (specified in INGREDIENT\_NUTRITION) – 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.

Query Optimization:

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 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 ( n.nutrition\_name = "cholesterol" OR

n.nutrition\_name = "potassium" OR

n.nutrition\_name = "lactose")

AND (n.nutrition\_name <> "cholesterol" OR vrnw.precentage <= 0.05)

AND (n.nutrition\_name <> "potassium" OR vrnw.precentage = 0)

AND (n.nutrition\_name <> "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 gets all of the recipes which the following rules are applied upon them:

1. It’s alcoholic
2. The cholesterol percentage in the recipe is less than 5%
3. The ins’t potassium in the cocktail
4. There isn’t lactose in this recipe

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 whether it’s 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 added the *VIEW\_RECIPE\_WEIGHTS* to get the total weight and then divide 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.

Query Optimizations:

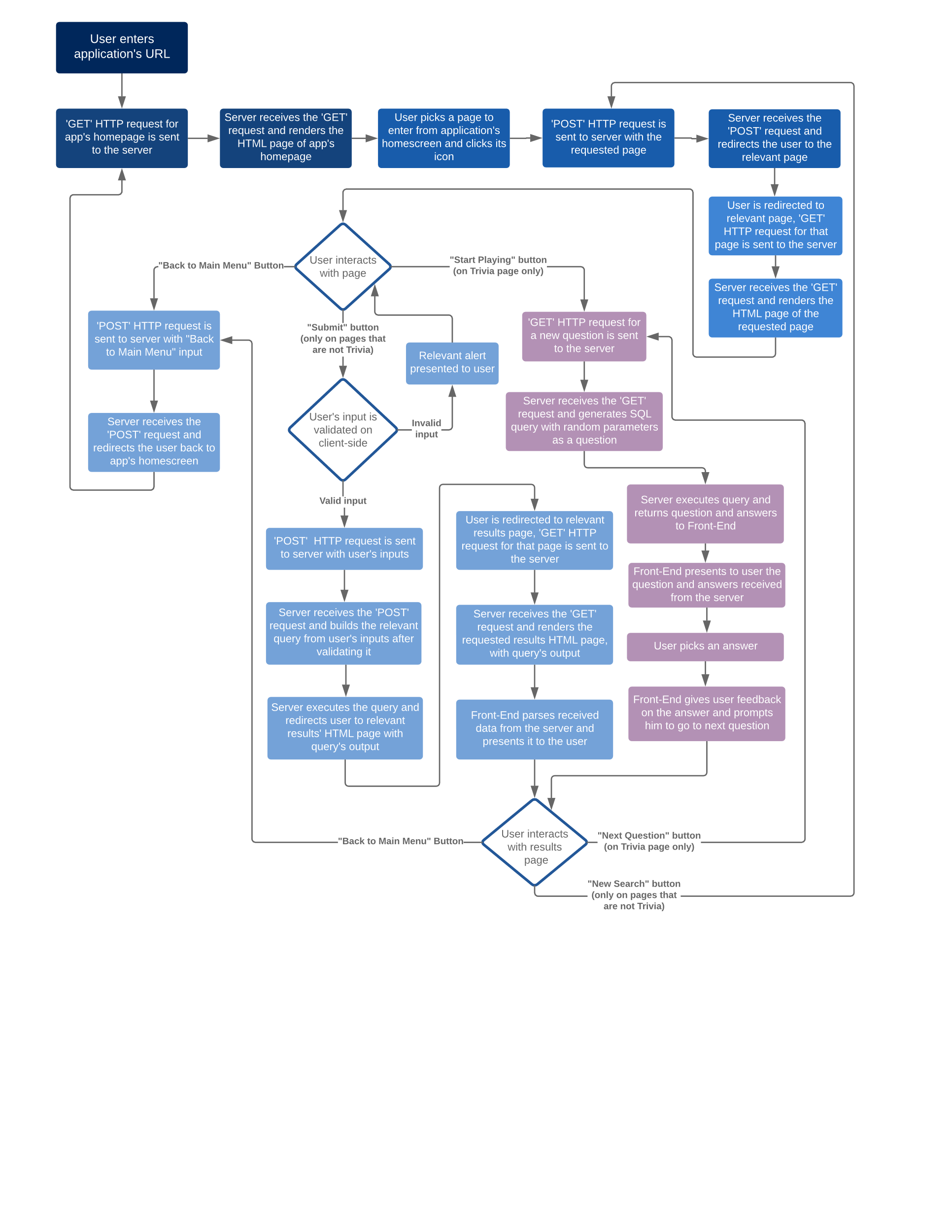
1. INNER JOIN, we get only the rows which are connected by the nutrition\_id and by the recipe\_id.
2. The nutrition\_name has an index inside *NUTRITIONS* so it’s faster. We didn’t want to search by the nutrition\_id which would be faster because than we would have to save the mapping in our server.
3. The recipe\_id and the nutrition\_id are indexed in *COCKTAIL\_RECIPES and NUTRITIONS* 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: Yummly, CocktailDB and Nutritionix.

**Yummly API**

This is the main API we used. It was used for retrieving ~40,000 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 11 different courses.
* 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.

1. **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.