CSE 560 Data Models and Query Languages - Project Report

TEJAS GAJANAN BHITLE, University at Buffalo, New York
PRASAD SHIRVANDKAR, University at Buffalo, New York
SHALINI AGARWAL, University at Buffalo, New York

ACM Reference Format:
Tejas Gajanan Bhitle, Prasad Shirvandkar, and Shalini Agarwal. 2022. CSE 560 Data Models and Query Languages - Project Report. 1, 1 (December 2022), 36 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 PROBLEM STATEMENT

A system where customers who are health conscious and who also want to have a good tasting and healthy food or anyone who wants some specific types of food with an exact amount of nutrients or calories in their food can query their particular type of requirements in the system to get recipes for that specific requirement specified.

A customer can have requirements for a low sodium or high protein diet, a low-calorie diet with low fat, a recipe that contains 1g of fat and is high in vitamins, or a low cholesterol diet. This project will help or provide solutions to any of these customer problems with a curated set of recipes dataset with nutritional information stored for quick access.

The dataset will be stored in database because it allows much higher flexibility with the query of data based on many criteria, which goes into much detailed querying, and the database will enable you to build relations between the different tables of your dataset like in our case, we link recipes with its nutritional information which relates with ingredients. A database offers much higher security with ACID properties to maintain the integrity and consistency of data and features like a rollback. In contrast, Excel files are made to work with a smaller amount of data than databases, mainly numerical data and statistical operations, but excel sheets could be better with filtering and querying data and miss features like imposing constraints on data and data validation. So there's a limit to how much data you can store in excel, which may be different based on sheets but still, there's a limit, whereas there is no such limit on data stored in a database.

2 TARGET USER

The database users are the customers who want to get their query-relevant food recipes from the database. E.g., A person trying to lose weight can search for low-calorie recipes on the database.

Anyone who wants to add recipes to the database to serve their customers is also a database user. E.g., Just like cooking shows, a nutritionist can put in a special recipe with its information.

Authors' addresses: Tejas Gajanan Bhitle, tejasgaj@buffalo.edu, University at Buffalo, New York; Prasad Shirvandkar, prasadda@buffalo.edu, University at Buffalo, Buffalo, New York; Shalini Agarwal, shalinia@buffalo.edu, University at Buffalo, Buffalo, New York.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
Manuscript submitted to ACM

```python
import requests
import pyperclip

if __name__ == '__main__':
    response = requests.get("https://api.edamam.com/api/recipes/v2?type=public&q=chinese&app_id=6a2bebb2&app_key"
                            "=7e733e7fb967899c37daa12f41c679ec")
    pyperclip.copy(str(response.content.decode("utf-8")).replace('\\', ''))
```

Fig. 1. Edamam REST API call

The database administrator could be us, the team members, who have the right to modify the database schema and manage and maintain how the data is stored and related.

3 DATABASE IMPLEMENTATION

We are using a REST-based API for the food database, which is a NoSQL JSON structure, and we are trying to make it structured data to make it relational.

3.1 Code Used for Scraping the JSON Data

- `Recipe Table`: : We have used Edamam REST API to populate the Recipes information in our schema. Fig1 illustrates the call to Edamam REST API. We attach the user token and make a request to their API. After the data is fetched, it is then passed on to the recipe parser 2, which parses individual recipes and then creates a corresponding insertion statements for the same. We were able to generate around 1.6K recipes using this.
- `Ingredient, Label, Nutrient Table`: For this three tables, we obtained the data directly from Edamam's website. As the data counts are relatively smaller (113, 34, 30 respectively), we generated their insert statements manually.
- `Pivot tables`: These tables correspond to the many-to-many mappings in the relations. We generated these mappings using python script as shown in Fig.3. We generated around 7K, 4K, 0.5K and 0.5K records for recipe_ingredient_pivot, recipe_labels_pivot, allergy_labels_pivot and ingredient_nutrient_pivot respectively.
- `Customer table`: We generated mock data for customers using an online fake data generator (https://www.mockaroo.com/). We generated around 3K customer records.

3.2 Database Schema

(1) **Table Recipe:** Stores recipe-related information with its meta-data and a short description of each.
Attributes:
- recipe_id_pk - Int/Number (Primary Key)
  Uniquely identifies a particular recipe. Auto-incremented based on an entry in the table
- name- VARCHAR(100)
  Defines the name of a recipe stored. E.g., Pan Fried Rice or Mashed Potatoes.

combination_generator.py U

Backend > combination_generator.py

```python
    starting = [
        "A simple, fresh-tasting to serve as a part of family buffet",
        "A family favorite ",
        "High Protein recipe",
        "A healthy and tasty recipe",
        "An absolute delicacy whose taste will last long",
        "A excellent savoury recipe",
        "A well-prepared delectable meal with great presentation and an interesting flavor you have never experienced before.",
        "The food of the gods",
        "This appetizing dish was prepared with great care and attention to detail",
        "A divine combination of different food elements came together to create this innovative dish",
        "The most luscious dishes",
        "A lip-smacking experience",
        "Add a whole new dimension to the taste with this recipe",
        "A recipe that will seriously make you hungry",
        "The recipe of your dreams",
        "An absolute delight to eat",
        "A pleasant looking recipe",
        "A exquisite recipe that will touch your hearts",
        "A homely beautiful recipe",
        "A wholesome and refreshing recipe"
    ]

    # Reading from file
    data = json.loads(f.read())

    SQL = 'INSERT INTO recipe(name, description, prep_time, meal_type, servings, img_url, url) VALUES'

    dda = data['hits']
    for d in dda:
        recipe = d['recipe']
        recipename = recipe['label']
        description = random.choice(starting)
        prep_time = random.choice(range(30, 130))
        meal_type = recipe['mealType'][0]
        servings = random.choice([1, 2, 4])
        img_url = recipe['images']['SMALL']['url']
        url = recipe['uri']
        print(
            "{SQL} ('{recipename}', '{description}', {preptime}, '{mealtype}', {serving}, '{imageurl}', '{url}');".format(
                recipename = recipename, description = description, preptime = prep_time, mealtype = meal_type, serving = servings, imageurl = img_url, url = url)
        )
```

Fig. 2. Recipe Parser

- description - VARCHAR(500)
  Contains a brief description of the named recipe.
- prep_Time - Int/Number (Not null, default value is 15)
  Tells the preparation required to cook a recipe, specified in minutes.
- meal_type - VARCHAR(20)
  Defines if the recipe is a Breakfast, Snack, Full-Meal or Dessert
- servings - Int/Number DEFAULT 1
  Defines how many people will be served using this recipe and the number of ingredients mentioned
- Img_url - VARCHAR(2048)
  Stores the image url of the recipe.

(2) **Table Label:** Stores the labels that are assigned to a recipe. The labels briefly describe the type of recipe. (Eg. Dairy-Free, Peanut-Free, Vegan etc.)
Attributes:

mapping_sql_generator.py U

Backend > mapping_sql_generator.py

```python
import random

recipe_id_pk_max = 1646
label_id_pk_max = 34
cust_id_pk_max = 1004
nutrient_id_pk_max = 30
ingredient_id_pk_max=113


def create_recipe_ingredient():
    sql = "INSERT INTO recipeingredientpivot(ingredient_id_fk, recipe_id_fk, num_of_units) VALUES"

    for i in range(1, recipe_id_pk_max+1):
        ingredients = []
        for _ in range(random.randint(0, 10)):
            ingredients.append(random.randint(1, ingredient_id_pk_max))

        ingredients = list(set(ingredients))
        for j in ingredients:
            queries.append(sql + "("+str(j)+", "+str(i)+", "+str(random.randint(1, 10))+");")

    queries.append("\n")


def create_ingredient_nutrient():
    sql = "INSERT INTO ingredientnutrientpivot(ingredient_id_fk, nutrient_id_fk, quantity) VALUES "

    for i in range(1, ingredient_id_pk_max+1):
        nutrients = []
        for _ in range(random.randint(0, 10)):
            nutrients.append(random.randint(1, nutrient_id_pk_max))

        nutrients = list(set(nutrients))
        for j in nutrients:
            queries.append(sql + "("+str(i)+", "+str(j)+", "+str(random.randint(1, 10))+");")

    queries.append("\n")


def create_recipe_label():
```

master* ⊗ 0 ⚠ 0

Fig. 3. Pivot Table Mapping Generator Snippet

- label_id_pk - Int/Number (Primary Key)
  Uniquely identifies a particular label. Auto-incremented based on an entry in the table.
- name - VARCHAR(30) NOT NULL
  Defines the name of the label. E.g., Balanced, Alcohol-Free, Dairy-Free, etc.

(3) **Table RecipeLabelsPivot:** Stores the Id's of the Recipe and Labels table to identify the labels assigned to the recipe.

Attributes:

- recipe_label_id_pk - Int/Number (Primary Key)

- recipe_id_fk- Int/Number (FOREIGN KEY on Recipe (Recipe_Id))
  Stores the Recipe ID of the recipe, which references to the id stored in the Recipe Table.
- label_id_fk - Int/Number (FOREIGN KEY on Label (Label_Id))
  Stores the Label ID for the recipe, which references to the id stored in the Label Table. It is a one-to-many relation with the Label table, meaning a recipe can contain multiple labels.

(4) **Table Ingredient:** Stores the food ingredients with the ingredient information such as calories and the measurement.
Attributes:
- ingredient_id_pk - Int/Number (Primary Key)
  Uniquely identifies a particular ingredient. Auto-incremented based on an entry in the table.
- name- VARCHAR(50) NOT NULL
  Defines the name of the ingredient. E.g., Onion, Cheese, Noodles.
- calorie_per_unit - Int NOT NULL DEFAULT 10
  Defines the number of calories contained in the ingredient
- measure_type - VARCHAR(10) NOT NULL
  Defines the measure type of an ingredient. For e.g., For any liquid ingredient, it could be ML, or for solid ingredients, it could be GMs or individual quantities.

(5) **Table RecipeIngredientPivot:** Stores the Ids of the Recipe and Ingredients table with the number of ingredient units required for that recipe.
Attributes:
- ingredient_id_fk - Int/Number (FOREIGN KEY on Ingredient (ingredient_id_pk))
  Stores the Ingredient ID of the ingredients, which references the id stored in the Ingredient Table.
- recipe_id_fk- Int/Number (FOREIGN KEY on Recipe (Recipe_Id))
  Stores the Recipe ID of the recipe, which references to the id stored in the Recipe Table.
- num_of_units - Int/Number NOT NULL DEFAULT 1
  Stores the number of units or quantity of ingredients required in the recipe

(6) **Table Nutrient:** Stores the total nutrients available for the recipe, including energy, fat, protein, carbohydrates, and vitamins.
Attributes:
- nutrient_id_pk - Int/Number (Primary Key)
  Uniquely identifies a particular nutrient. Auto-incremented based on an entry in the table.
- name- VARCHAR(30) NOT NULL UNIQUE
  Stores the unique name of the nutrient
- is_macro: - BOOLEAN DEFAULT FALSE
  Indicates if its a macro-nutrient or micro-nutrient.
- unit: VARCHAR(10) NOT NULL
  Stores the unit in which nutrients are represented. Eg mg, g, ml

(7) **Table IngredientNutrientPivot:** Stores the Ids of the Ingredient and Nutrient table, which contains the information about the nutrients of a particular ingredient.
Attributes:
- ingredient_nutrient_id_pk - Int/Number (Primary key)

- ingredient_id_fk- Int/Number (FOREIGN KEY on Ingredient(ingredient_id_pk))
  Stores the ID of the Ingredient, which references to the id stored in the Ingredient Table.
- nutrient_id_fk - (FOREIGN KEY on Nutrient (nutrient_id_pk))
  Stores the ID of the Nutrients, which references the id stored in the Nutrients Table.
- Quantity - Int/Number NOT NULL DEFAULT 1
  Defines the quantity of nutrients in the ingredient which is measured in the unit defined for the nutrient in the Nutrient table.

(8) **Table Customer:** Stores the customer information who has queried the database so that the system can recommend food recipes based on customers' choices/preferences.
Attributes:

- customer_id - Int/Number (PRIMARY KEY)
  Uniquely identifies a customer name. Auto-incremented based on an entry in the table.
- Fname - VARCHAR(30)
  First Name of the customer
- Lname - VARCHAR(30)
  Last Name of the customer
- Email - VARCHAR(30)
  Email Id of the customer
- PasswordHash - VARCHAR(50)
  Hash of the password (to be used for login/signup purpose)
- Choice - VARCHAR(10)
  Food type that the customer prefers. (Eg. quick, healthy, tasty)

(9) **Table AllergyLabel:** Stores the information about labels that the customer is allergic to, so that system won't recommend some recipes that the customer is allergic to.
Attributes:

- Allergy_id - Int/Number (PRIMARY KEY)
  Uniquely identifies a particular allergy. Auto-incremented based on an entry in the table.
- cust_id_fk- Int/Number (FOREIGN KEY) on Customer (cust_id_pk)
  Stores the ID of the Customer, which references the id stored in the Customers Table
- label_id - Int/Number (FOREIGN KEY) on Label (label_id_pk)
  Stores the ID of the label which customer is allergic to, which references the id stored in the Labels Table

(10) **Table Preferences:** Stores the information about the preferences based on prep time, ingredient count and calories count for the customers.
Attributes:

- pref_id_pk - int/number (PRIMARY KEY)
  Uniquely identifies the preference count. Auto- incremented based on an entry in the table
- cust_id_fk - int/number(FOREIGN KEY) on Customer (cust_id_pk)
  Stores the ID of the Customer, which references the id stored in the Customers Table
- prep_time - int/number DEFAULT 120
  Stores the preferred preparation time for a recipe

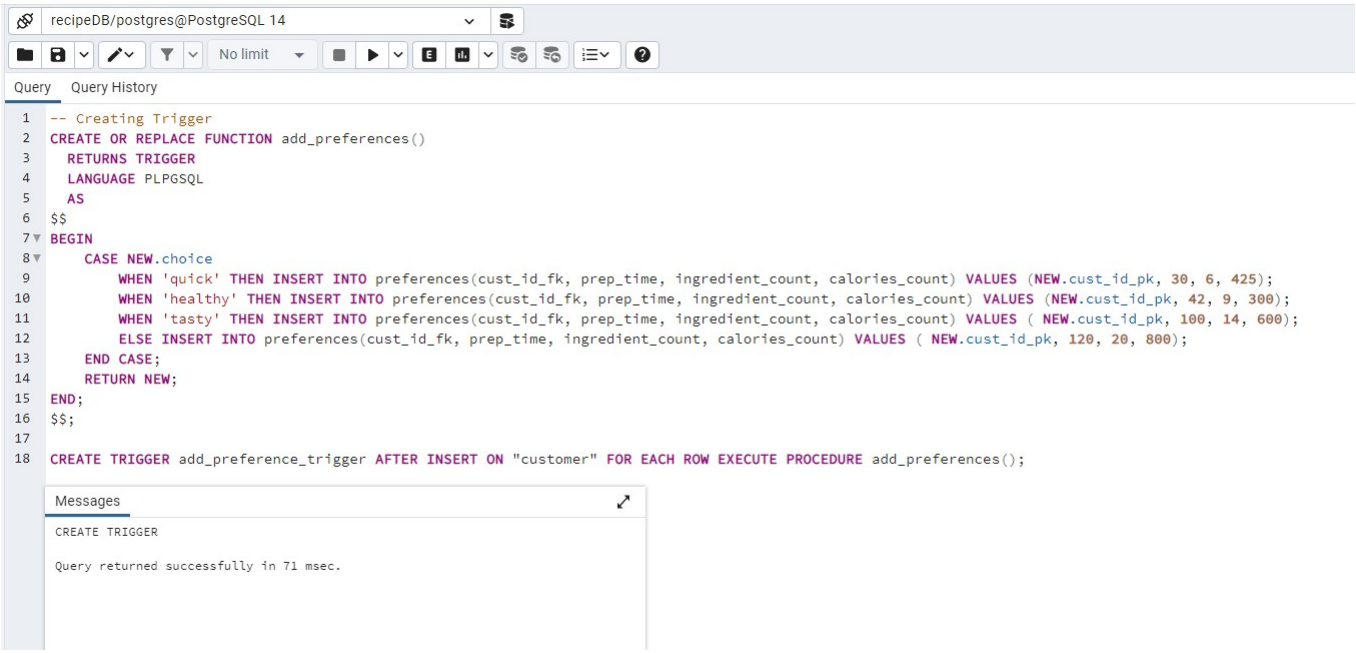

Fig. 4. Trigger Creation

- ingredient_count - int/number DEFAULT 20
  Stores the preferred number of ingredients for a recipe
- calories_count - int/number DEFAULT 800
  Stores the preferred number of calories for a recipe

3.3 Constraints in the database relations

- **Cascade** In all the foreign keys defined above, we have kept the action as CASCADE in case of updation and deletion of records, This is because, if a tuple from the Recipe Table is deleted there is no point in retaining its mapping with the associated tabled such as Ingredients and Labels. Similary, if a customer is deleted, then there is no need to retain the allergies of that particular customer.
- **Trigger** Whenever a new customer entry is added in the Customer table, we have a trigger in place that adds the corresponding preference entry in preferences table. This is illustrated in the figures- 4 and 5.
- **Miscellaneous** NOT NULL, DEFAULT and CHECK constraints are also in place wherever deemed necessary.

The detailed constraints are described below as well as mentioned in the Schema Creation section.

- Recipe
  - name attribute can't have null values.
  - prep_time attribute has a default value of 15 and it can't be null.
  - meal_type attribute must be one among these values- 'Breakfast', 'Snack', 'Full-Meal', 'Dessert'.
  - servings attribute has a default value of 1 and it can't be null.

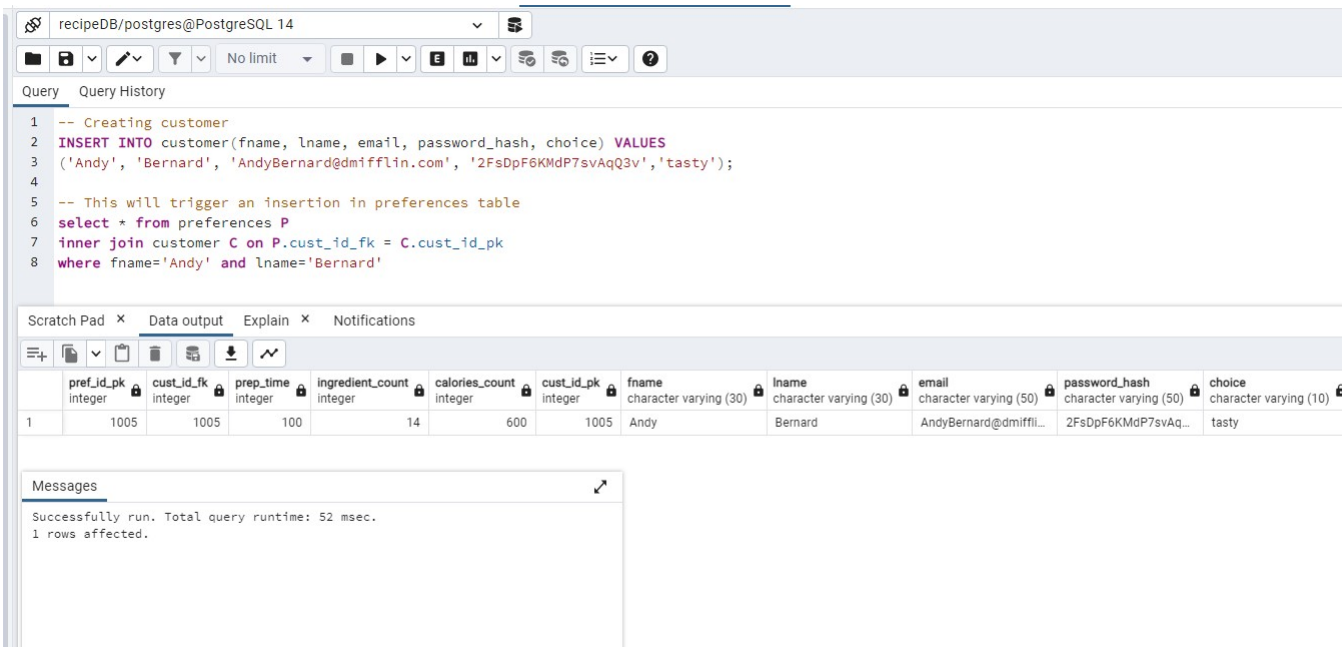

Fig. 5. Trigger Working

url attribute has a default value of null.

- Ingredient
  - calorie_per_unit attribute has a default value of 10 and it can't be null.
  - name and measure_type attribute can't be null value.

- Label
  - name attribute can't be of null value.

- Nutrient
  - name attribute can't have null values and should have unique values.
  - is_macro has default value as false.
  - unit attribute can't have null values.

- Customer
  - fname attribute can't have null values.
  - lname attribute can't have null values.
  - email attribute can't have null values.
  - password_hash attribute can't have null values.

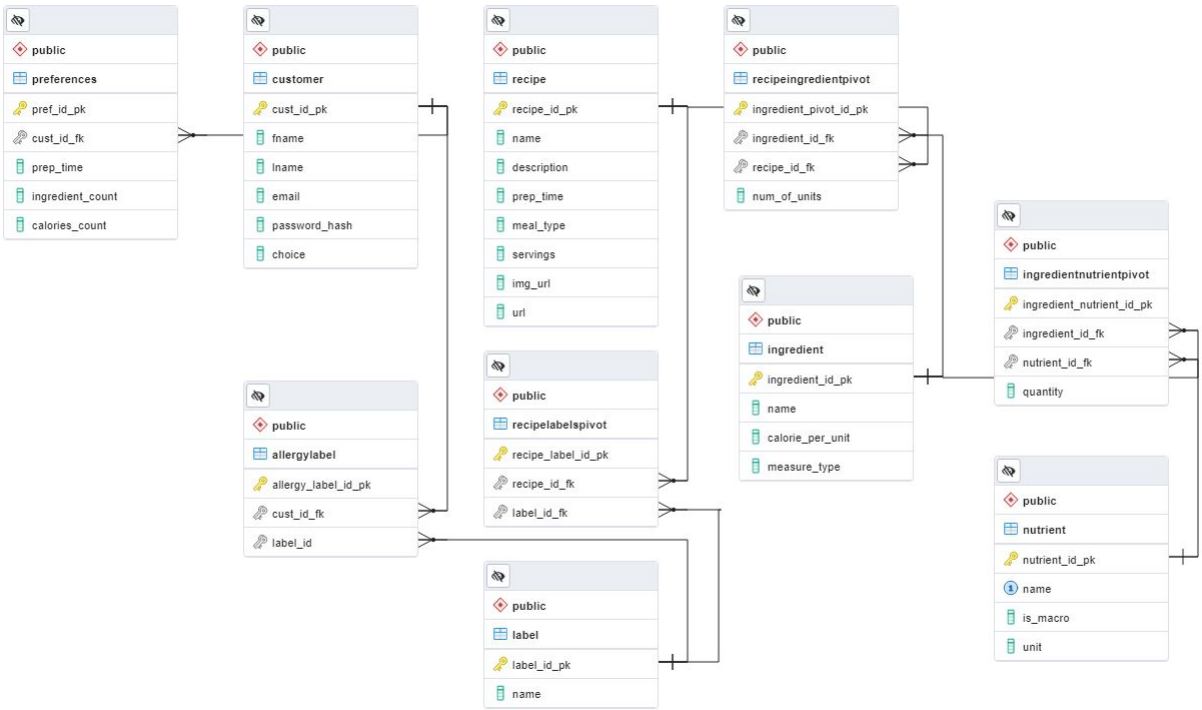

Fig. 6. ER Diagram

- AllergyLabel
  - cust_id_fk has a referential constraint which implies when the values gets updated or deleted in the referenced table, it should reflect the same in the referencing table.
  - label_id has a referential constraint which implies when the values gets updated or deleted in the referenced table, it should reflect the same in the referencing table.

3.4 E/R Diagram The E/R Diagram of our Database is as shown in Fig.6

.

3.5 Normalizing DB into BCNF

These are the relations we have in our database system with their corresponding functional dependencies:-

- Recipe (recipe_id, name, description, prep_time, meal_type, servings, img_url, url)
  recipe_id_pk →name, description, prep_time, meal_type, servings, img_url, url
- Label (label_id_pk, name)
  label_id_pk →name

- Ingredient (ingredient_id_pk, name, calorie_per_unit, measure_type)
  ingredient_id_pk →name, calorie_per_unit, measure_type
- Nutrient (nutrient_id_pk, name, is_macro, unit)
  nutrient_id_pk →name, is_macro, unit
- Customer (customer_id, Fname, Lname, Email, PasswordHash)
  customer_id →Fname, Lname, Email, PasswordHash
- Allergy (allergy_id, cust_id_fk, label_id)
  allergy_id →cust_id_fk, label_id

The definition of BCNF states that for table to be in BCNF form, for every functional dependency X Y, X should be the super key of the relation. If a table is in BCNF, then it automatically satisfies the condition of being in 3NF form as BCNF is an advanced version of 3NF. A super key for a relation is defined as the set of those attributes which are able to uniquely identify a tuple. We can see in the relations defined above that each of the relations have only one functional dependency which is from its primary key to all the other attributes of the relation. Since, each FD has primary key on the LHS, this automatically satisfies the criteria of X being a super key in the definition of BCNF. Thus, all the relations in our database are already in BCNF form and don't require any further decomposition.

3.6 Schema Creation

The fig.7 and fig.8 shows the creation of the database relations with the CREATE command.

The fig.9 shows the execution of the DDL script in PostgresSQL.

3.7 Record Insertion

The fig.10 shows the insertion of records in the relations.

The fig.11 shows the number of records for each of the relation in the database.

4 QUERY IMPLEMENTATION

4.1 Sample Queries

Please find the attached screenshots of the queries mentioned below.

- **INSERTION →fig.12, and fig.13**
- **DELETION →fig.14, fig.15**
- **UPDATION →fig.16, fig.17**
- **TRUNCATE** →fig.18
- **OTHER QUERIES**:-
  (1) Get top 5 quickest breakfast →Fig.19
  (2) Get count of high fibre recipes→Fig.20
  (3) Retrieving a recipe with least number of ingredients →Fig.21
  (4) Retrieving ingredient names that have carbs present →Fig.22

5 QUERY OPTIMIZATION

SQL performance can impact performance when it comes to scaling up the database solutions. It is important to have efficient queries executing so as to improve the throughput of any DB oriented systems.

ddl.sql

db_scripts_final > data_loading > ddl.sql

```sql
-- creating tables

CREATE TABLE RECIPE (
    recipe_id_pk SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    description VARCHAR(500),
    prep_time INTEGER NOT NULL DEFAULT 15,
    meal_type VARCHAR(20) CHECK(meal_type in ('Breakfast', 'Snack', 'Full-Meal', 'Dessert')),
    servings INTEGER NOT NULL DEFAULT 1,
    img_url VARCHAR(2048),
    url VARCHAR(2048) DEFAULT NULL
);

CREATE TABLE INGREDIENT(
    ingredient_id_pk SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    calorie_per_unit INTEGER NOT NULL DEFAULT 10,
    measure_type VARCHAR(10) NOT NULL
);

CREATE TABLE RecipeIngredientPivot (
    ingredient_pivot_id_pk SERIAL PRIMARY KEY,
    ingredient_id_fk INTEGER NOT NULL REFERENCES INGREDIENT (ingredient_id_pk)
                     ON UPDATE CASCADE
                     ON DELETE CASCADE,
    recipe_id_fk INTEGER NOT NULL REFERENCES RECIPE (recipe_id_pk)
                 ON UPDATE CASCADE
                 ON DELETE CASCADE,
    num_of_units INTEGER NOT NULL DEFAULT 1
);

CREATE TABLE LABEL (
    label_id_pk SERIAL PRIMARY KEY,
    name VARCHAR(30) NOT NULL
);


CREATE TABLE RecipeLabelsPivot (
    recipe_label_id_pk SERIAL PRIMARY KEY,
    recipe_id_fk INTEGER NOT NULL REFERENCES RECIPE (recipe_id_pk)
                 ON UPDATE CASCADE
                 ON DELETE CASCADE,
    label_id_fk INTEGER NOT NULl REFERENCES LABEL (label_id_pk)
                ON UPDATE CASCADE
                ON DELETE CASCADE
);

CREATE TABLE Nutrient (
```

Fig. 7. Creation of the Database Tables

The Performance optimizations made by us are as follows:

(1) Indexing: We have made sure that each of our primary keys have an index on them as all of our join queries use primary keys. (PostgreSQL by default creates indexes on all of the primary keys)
(2) Select: We are selecting only a required number of columns instead of running 'select *'.
(3) We have replaced most of our subqueries with joins as subqueries would search row by row, reducing the speed of the query.
(4) We have replaced 'having' clause with 'where' clause as WHERE filters the data at the initial stage while HAVING filters in in later stage.

5.1 DML Query Optimization

While inserting the records in the table, an optimization technique followed was that instead of inserting queries individually, insertion of bulk queries was performed. The fig.23 shows the execution of inserting records individually.

The fig.24 shows the execution of records in bulk.

ddl.sql × dml.sql

db_scripts_final > data_loading > ddl.sql

```sql

CREATE TABLE Nutrient (
    nutrient_id_pk SERIAL PRIMARY KEY,
    name VARCHAR(30) NOT NULL UNIQUE,
    is_macro BOOLEAN DEFAULT FALSE,
    unit VARCHAR(10) NOT NULL
);

CREATE TABLE IngredientNutrientPivot (
    ingredient_nutrient_id_pk SERIAL PRIMARY KEY,
    ingredient_id_fk INTEGER NOT NULL REFERENCES INGREDIENT (ingredient_id_pk)
                     ON UPDATE CASCADE
                     ON DELETE CASCADE,
    nutrient_id_fk INTEGER NOT NULL REFERENCES Nutrient (nutrient_id_pk)
                   ON UPDATE CASCADE
                   ON DELETE CASCADE,
    quantity INTEGER NOT NULL DEFAULT 1
);


CREATE TABLE CUSTOMER (
    cust_id_pk SERIAL PRIMARY KEY,
    fname VARCHAR(30) NOT NULL,
    lname VARCHAR(30) NOT NULL,
    email VARCHAR(50) NOT NULL,
    password_hash VARCHAR(50) NOT NULL,
    choice VARCHAR(10)
);

CREATE TABLE AllergyLabel (
    allergy_label_id_pk SERIAL PRIMARY KEY,
    cust_id_fk INTEGER NOT NULL REFERENCES CUSTOMER (cust_id_pk)
               ON UPDATE CASCADE
               ON DELETE CASCADE,
    label_id INTEGER NOT NULL REFERENCES LABEL (label_id_pk)
             ON UPDATE CASCADE
             ON DELETE CASCADE
);

CREATE TABLE Preferences (
    pref_id_pk SERIAL PRIMARY KEY,
    cust_id_fk INTEGER NOT NULL,
    prep_time INTEGER DEFAULT 120,
    ingredient_count INTEGER DEFAULT 20,
    calories_count INTEGER DEFAULT 800
);


```

Fig. 8. Creation of the Database Tables

5.2 Optimization through Indexing:-

The relations Recipe, Nutrient and Ingredient have the following indexes created as shown in Fig.25. The optimization through these is shown in the subsequent section.

6 QUERY EXECUTION ANALYSIS

The stats of the performance of the queries are as follows:-

6.1 Sample Query 1

Get recipes for a customer with his/her preferences and without his/her allergies.

The following three figures 26 27 28 shows the execution of the query and its analysis before indexing.

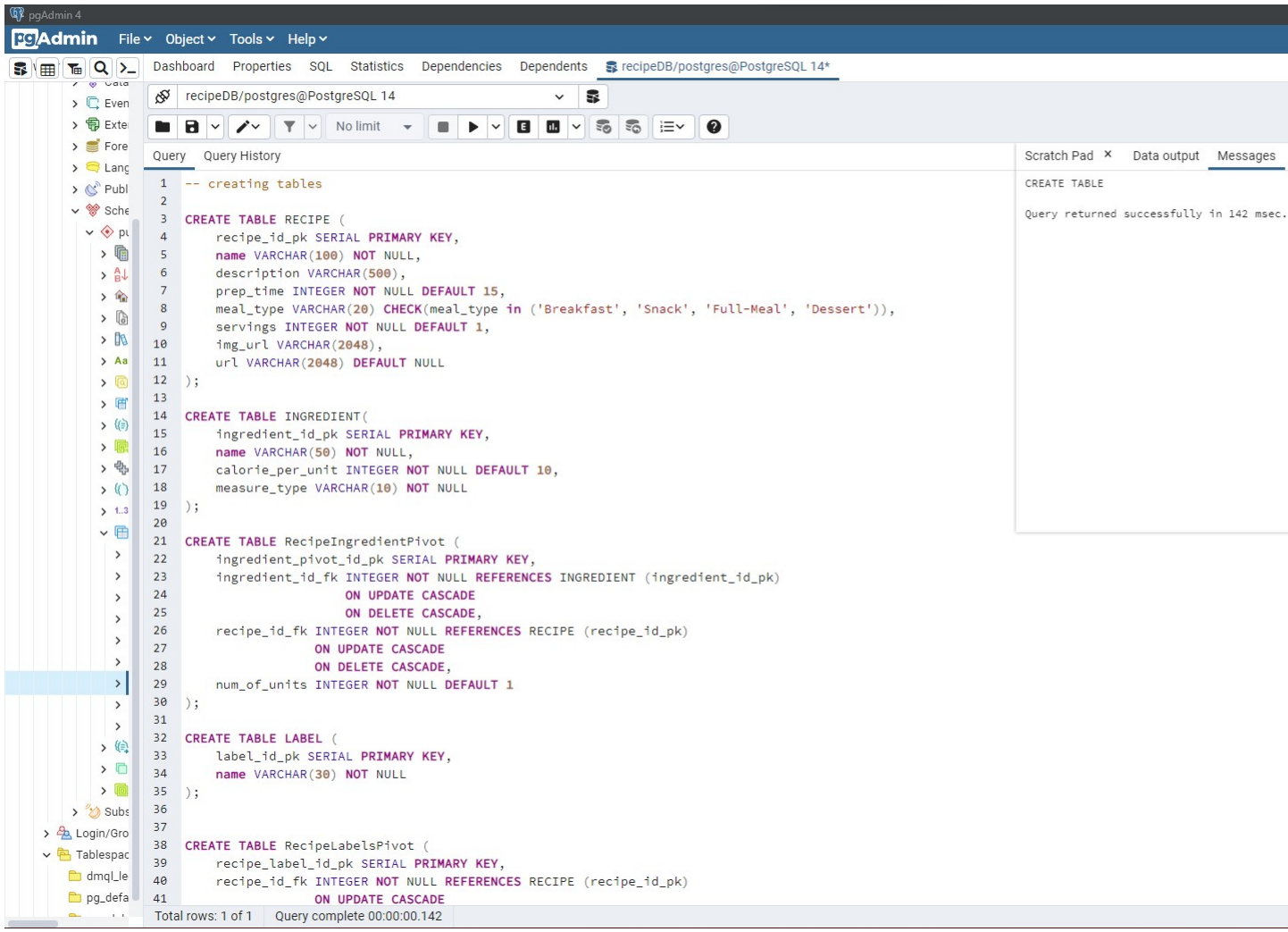

Fig. 9. DDL Execution

The next three figures 29 30 31 shows the execution of the query and its analysis after indexing.

6.2 Sample Query 2

Get recipe containing highest sugar content.

The following three figures 32 33 34 shows the execution of the query and its analysis before indexing.

The next three fig.35, fig.36 and fig.37 shows the execution of the query and its analysis after indexing.

Fig. 10. Tuple Insertion

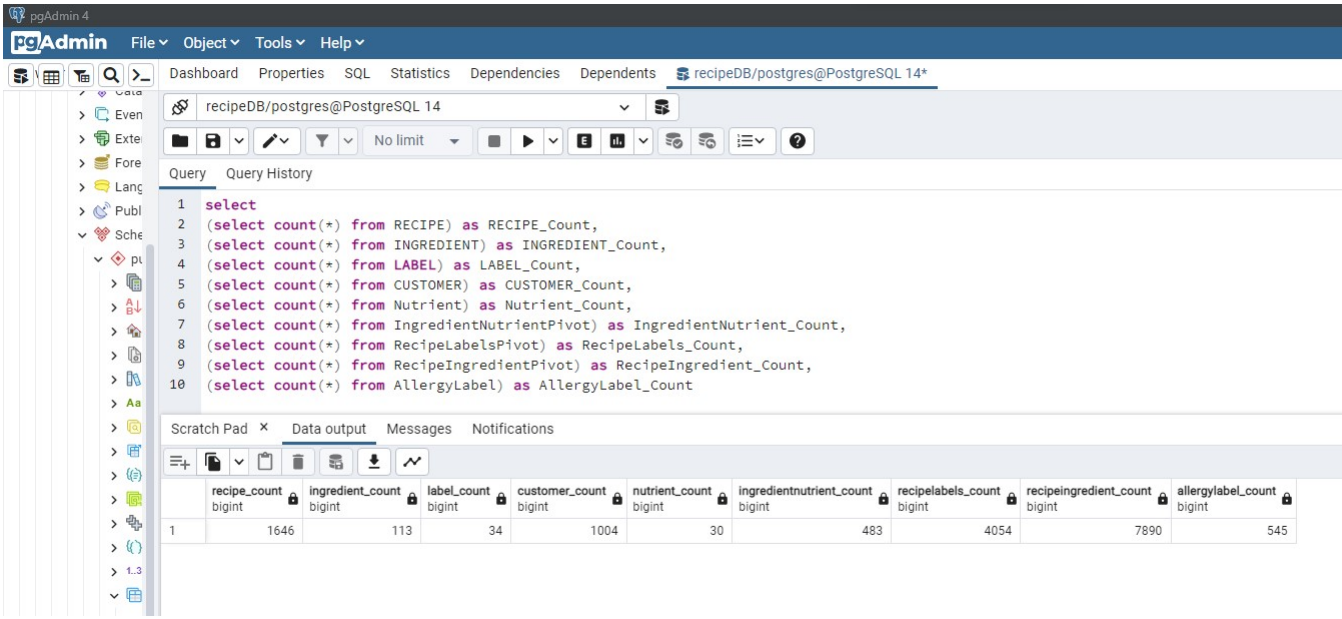

Fig. 11. DB Relation Entry Count

6.3 Sample Query 3

Get top 10 low calorie meals.

The following three fig.38, fig.39 and fig.40 shows the execution of the query and its analysis before indexing.

The next three fig.41, fig.42 and fig.43 shows the execution of the query and its analysis after indexing.

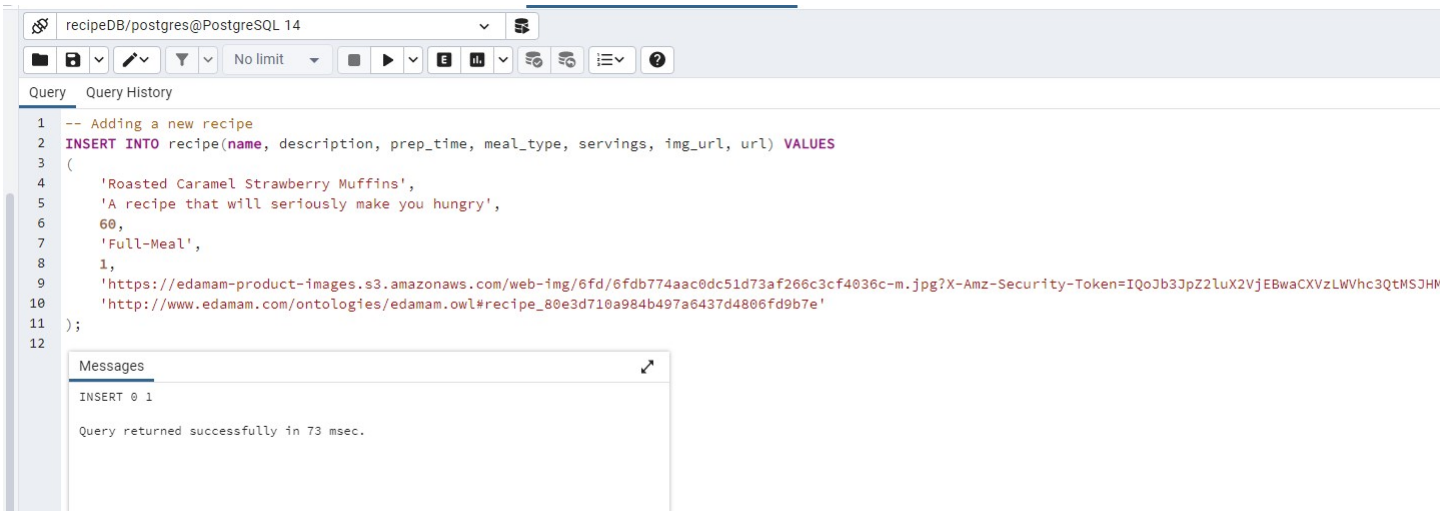

Fig. 12. Sample query for inserting into Recipe Table

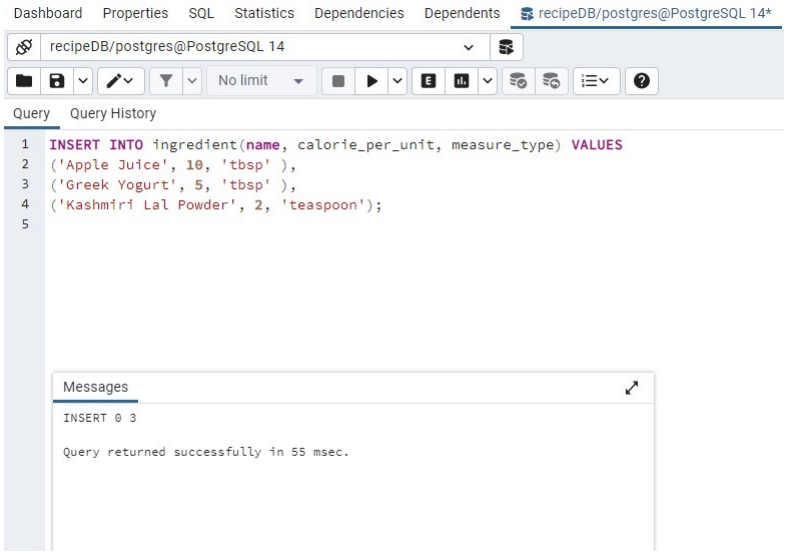

Fig. 13. Sample query for inserting into Ingredient Table

6.4 Sample Query 4

Get most common macro nutrient found in all ingredients.

The following three fig.44, fig.45 and fig.46 shows the execution of the query and its analysis before indexing.

The next three figures fig.47, fig.48 and fig.49 shows the execution of the query and its analysis after indexing.

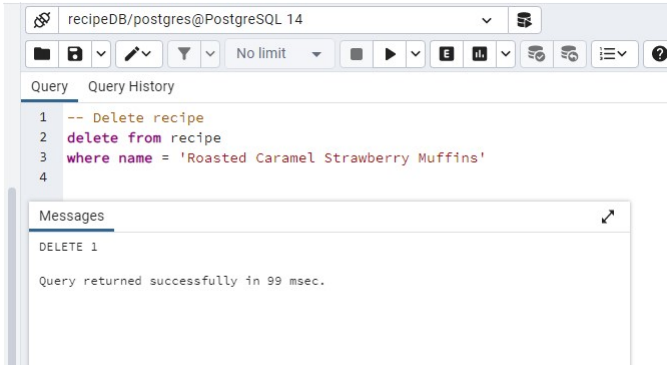

Fig. 14. Sample query for deleting from Recipe Table

Fig. 15. Sample query for deleting from Customer Table

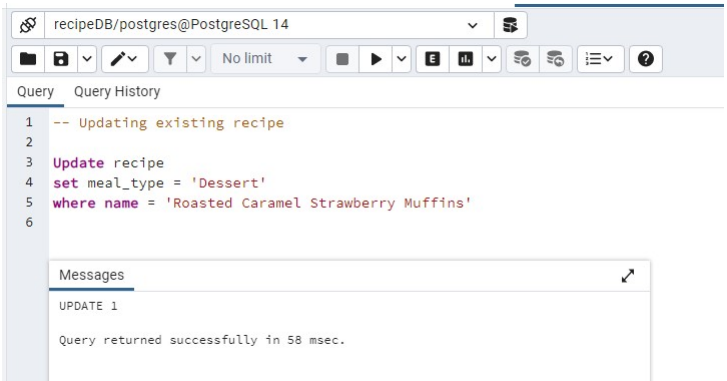

Fig. 16. Sample query for updating record into Recipe Table

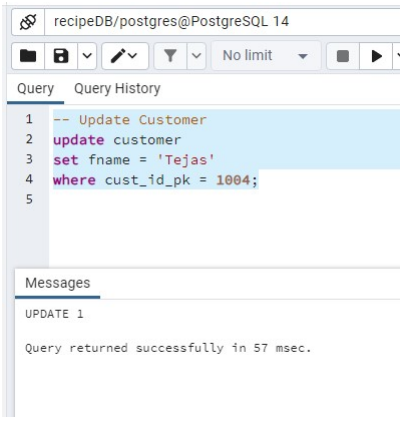

Fig. 17. Sample query for updating record into Customer Table

7 DEMO WEBSITE SETUP (BONUS TASK)

We have also setup a demo website that queries our PostgreSQL Server inorder to fetch the data. We have used a React application on our frontend side and a Flask application on the backend side. The button clicks events on the react application make API calls to the Flask server, which inturn calls the PostgreSQL server in order to obtain the data. For demo purposes, we have setup this website locally, but we can always host this on any domain if needed. The screenshots of our website are as shown in Fig.50 and Fig.51. The sample screenshots of the codebase are also as mentioned in Fig.52 and Fig.53.

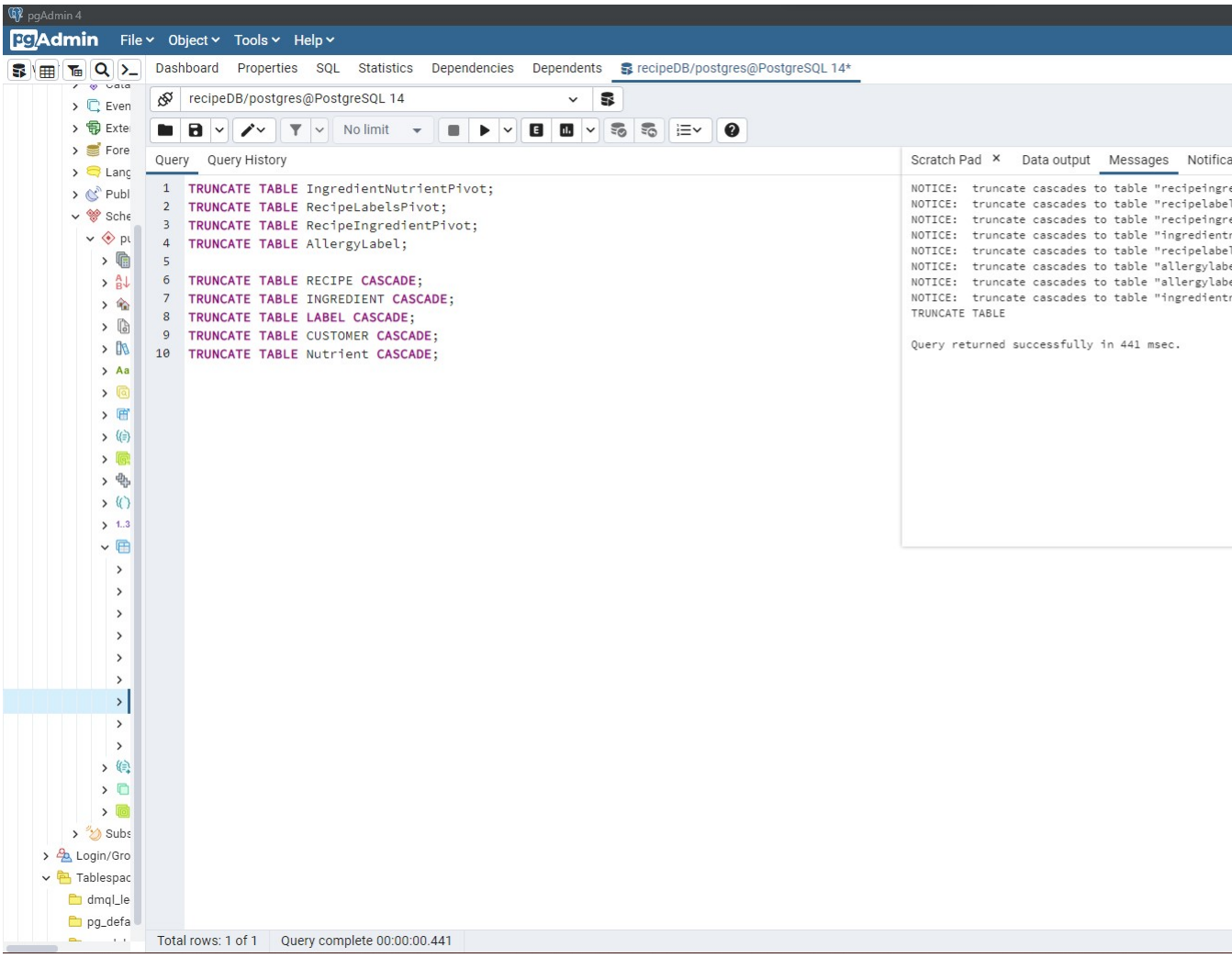

Fig. 18. Sample query for truncation from the relations

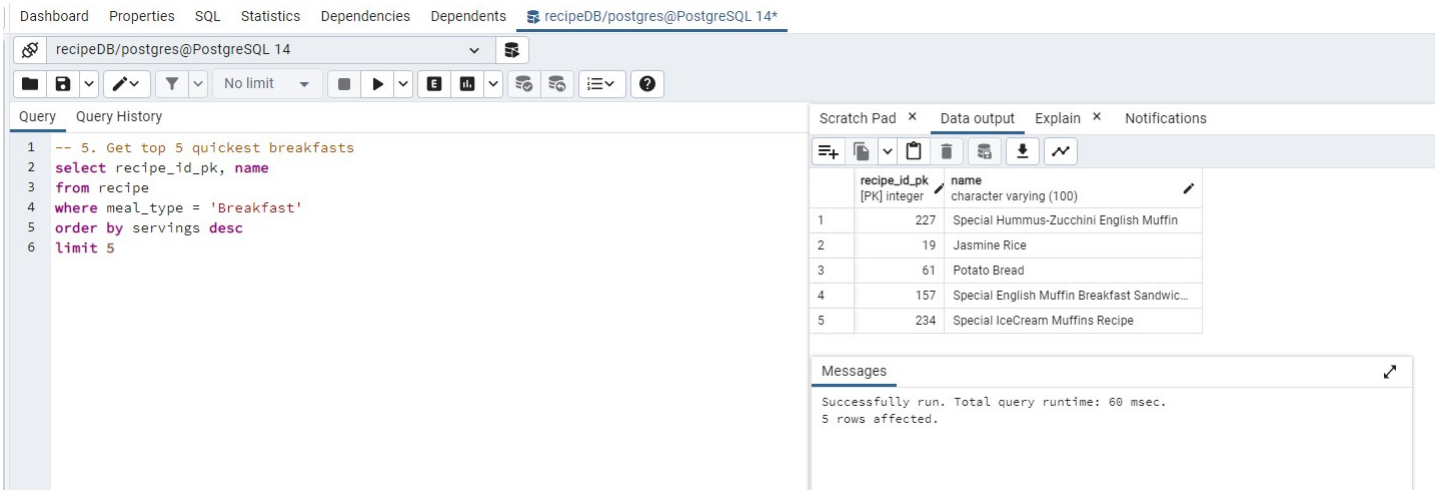

Fig. 19. Sample query for getting top 5 quickest breakfast

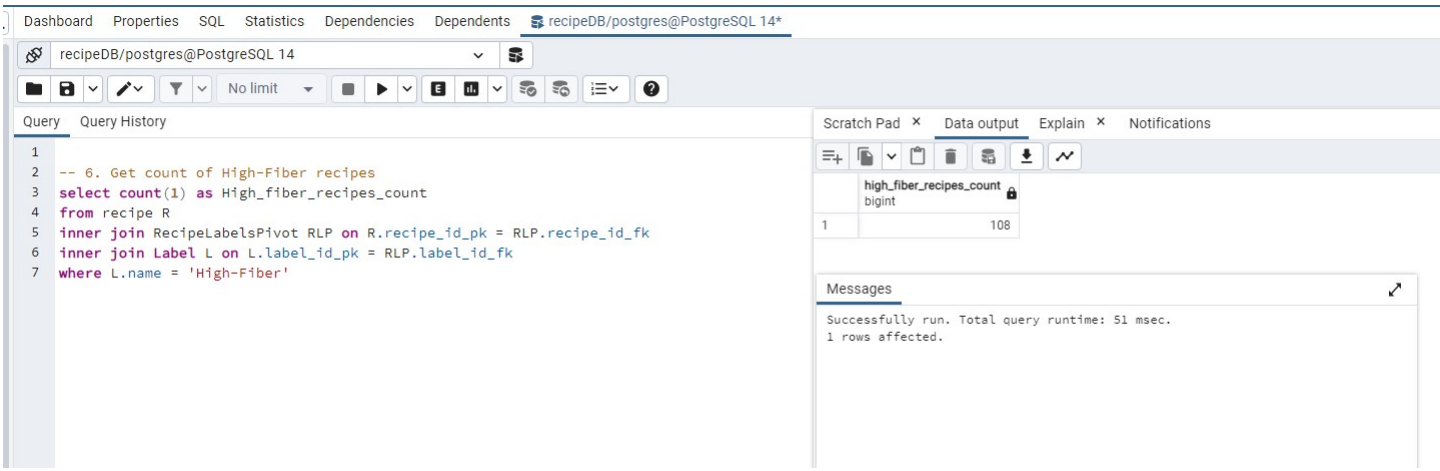

Fig. 20. Sample query for getting count of high fibre recipes

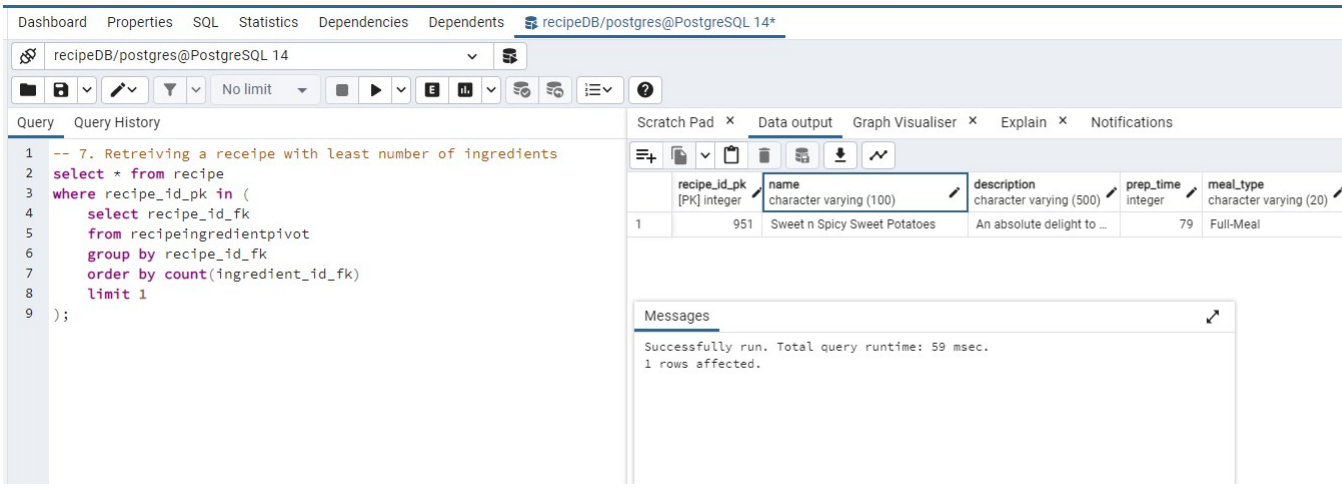

Fig. 21. Sample query for getting recipe with least number of ingredients

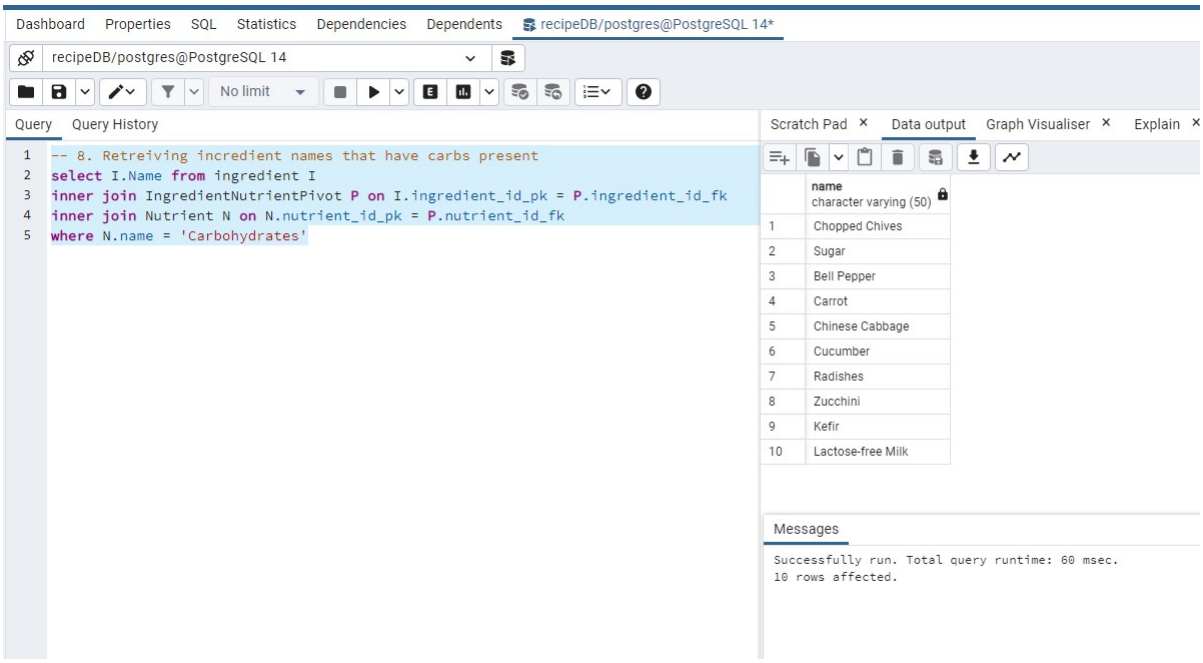

Fig. 22. Sample query for getting ingredient names which has carbs present

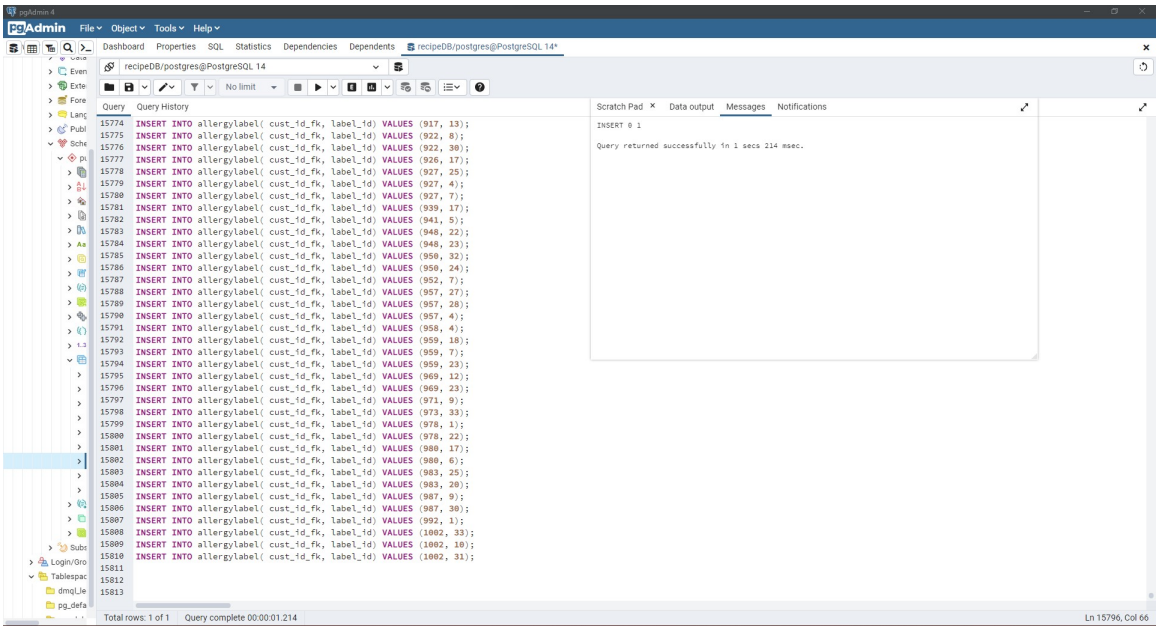

Fig. 23. Individual Record Insertion; Time taken 1 sec 214 msec

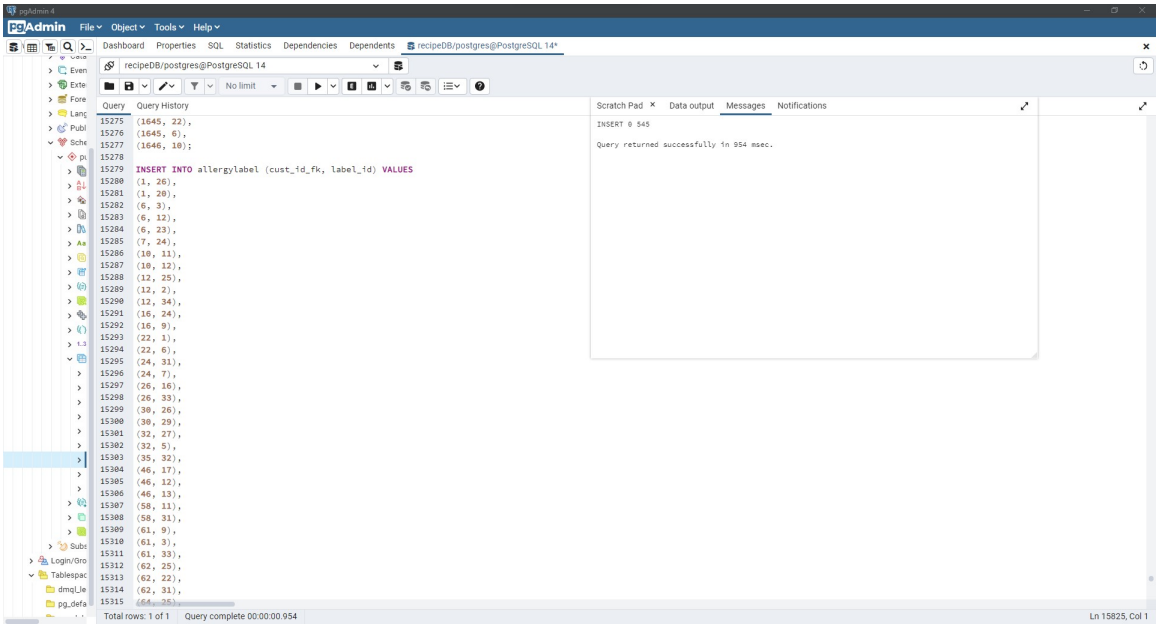

Fig. 24. Bulk Record Insertion; Time taken 954 msec

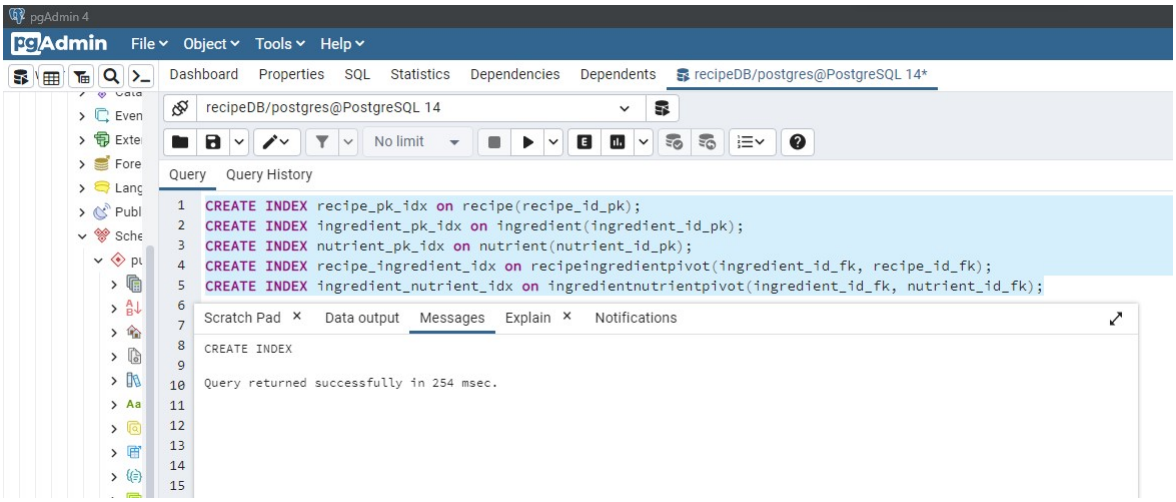

Fig. 25. Index Creation

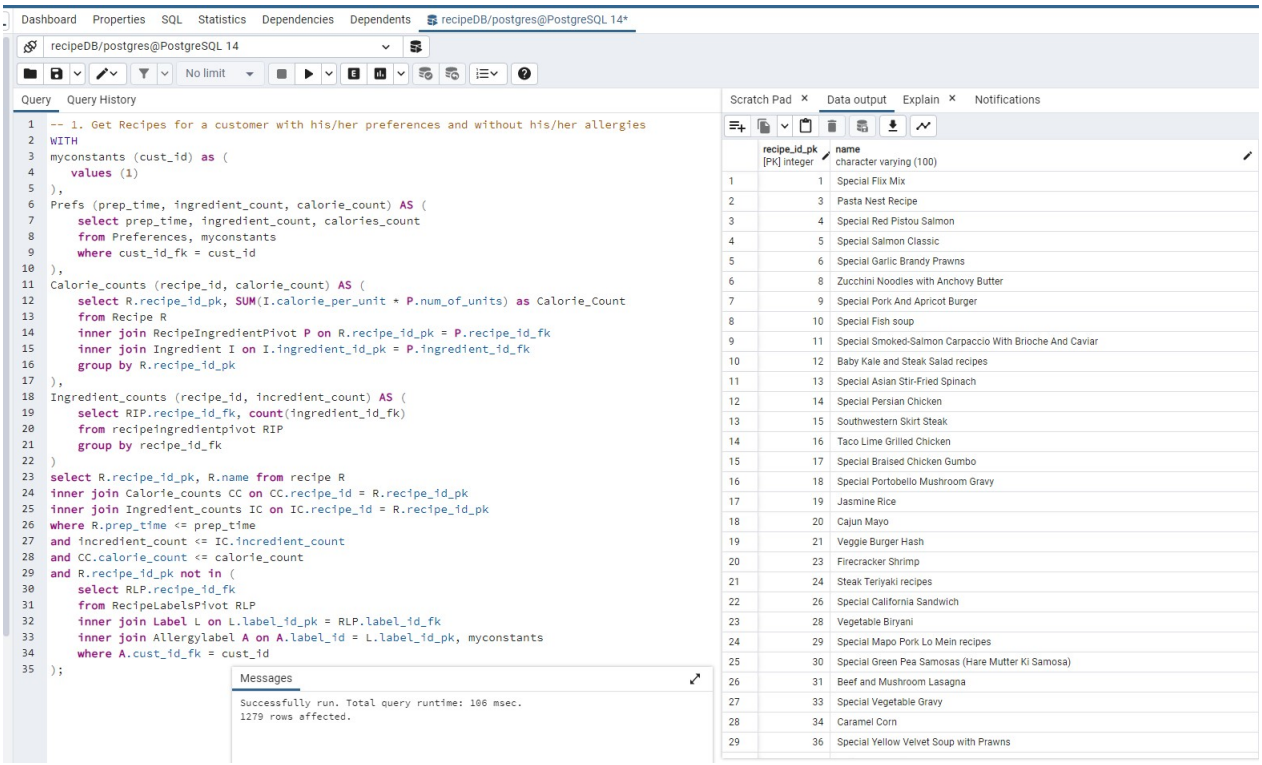

Fig. 26. Query execution before indexing

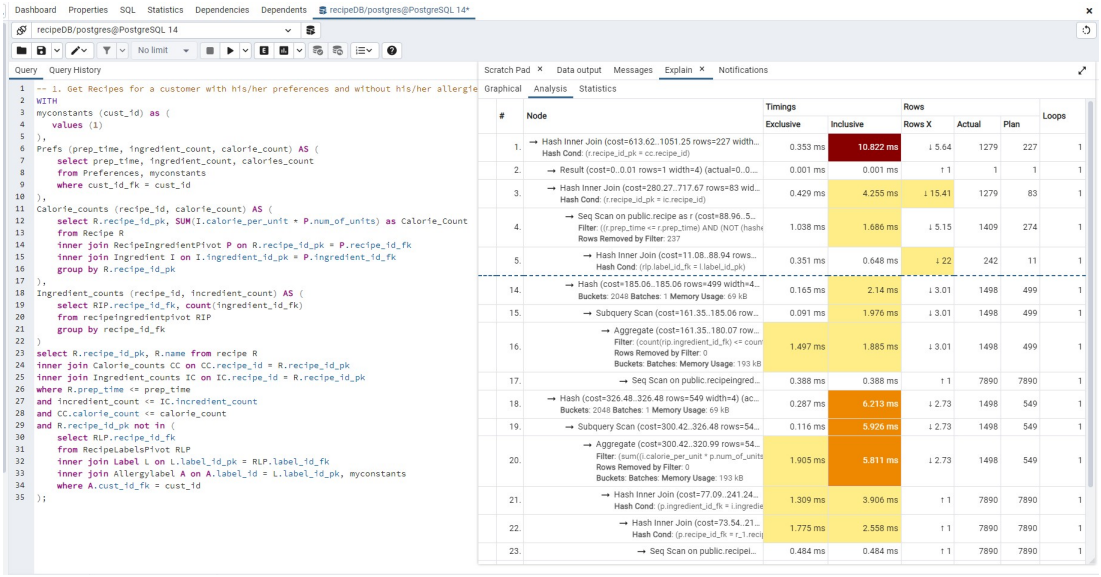

Fig. 27. Analysis of the query execution before indexing

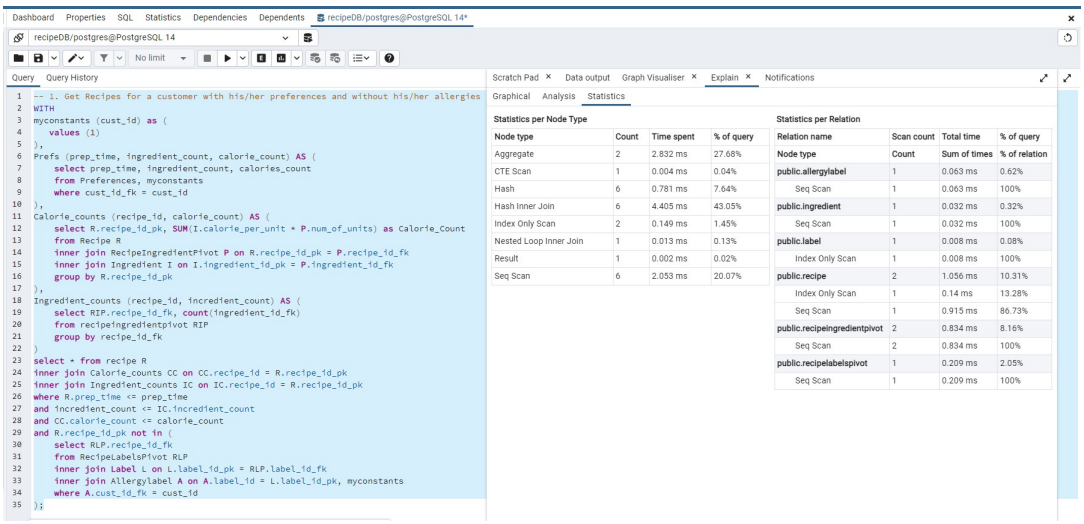

Fig. 28. Statistics of the query execution before indexing

Dashboard Properties SQL Statistics Dependencies Dependents recipeDB/postgres@PostgreSQL 14*

recipeDB/postgres@PostgreSQL 14 No limit

Query Query History

```sql
-- 1. Get Recipes for a customer with his/her preferences and without his/her allergies
WITH
myconstants (cust_id) as (
   values (1)
),
Prefs (prep_time, ingredient_count, calorie_count) AS (
    select prep_time, ingredient_count, calories_count
    from Preferences, myconstants
    where cust_id_fk = cust_id
),
Calorie_counts (recipe_id, calorie_count) AS (
    select R.recipe_id_pk, SUM(I.calorie_per_unit * P.num_of_units) as Calorie_Count
    from Recipe R
    inner join RecipeIngredientPivot P on R.recipe_id_pk = P.recipe_id_fk
    inner join Ingredient I on I.ingredient_id_pk = P.ingredient_id_fk
    group by R.recipe_id_pk
),
Ingredient_counts (recipe_id, incredient_count) AS (
    select RIP.recipe_id_fk, count(ingredient_id_fk)
    from recipeingredientpivot RIP
    group by recipe_id_fk
)
select R.recipe_id_pk, R.name from recipe R
inner join Calorie_counts CC on CC.recipe_id = R.recipe_id_pk
inner join Ingredient_counts IC on IC.recipe_id = R.recipe_id_pk
where R.prep_time <= prep_time
and incredient_count <= IC.incredient_count
and CC.calorie_count <= calorie_count
and R.recipe_id_pk not in (
    select RLP.recipe_id_fk
    from RecipeLabelsPivot RLP
    inner join Label L on L.label_id_pk = RLP.label_id_fk
    inner join Allergylabel A on A.label_id = L.label_id_pk, myconstants
    where A.cust_id_fk = cust_id
);
```

Messages

```
Successfully run. Total query runtime: 67 msec.
1279 rows affected.
```

Scratch Pad Data output Explain Notifications

	recipe_id_pk [PK] integer	name character varying (100)
1	1	Special Flix Mix
2	3	Pasta Nest Recipe
3	4	Special Red Pistou Salmon
4	5	Special Salmon Classic
5	6	Special Garlic Brandy Prawns
6	8	Zucchini Noodles with Anchovy Butter
7	9	Special Pork And Apricot Burger
8	10	Special Fish soup
9	11	Special Smoked-Salmon Carpaccio With Brioche And Caviar
10	12	Baby Kale and Steak Salad recipes
11	13	Special Asian Stir-Fried Spinach
12	14	Special Persian Chicken
13	15	Southwestern Skirt Steak
14	16	Taco Lime Grilled Chicken
15	17	Special Braised Chicken Gumbo
16	18	Special Portobello Mushroom Gravy
17	19	Jasmine Rice
18	20	Cajun Mayo
19	21	Veggie Burger Hash
20	23	Firecracker Shrimp
21	24	Steak Teriyaki recipes
22	26	Special California Sandwich
23	28	Vegetable Biryani
24	29	Special Mapo Pork Lo Mein recipes
25	30	Special Green Pea Samosas (Hare Mutter Ki Samosa)
26	31	Beef and Mushroom Lasagna
27	33	Special Vegetable Gravy
28	34	Caramel Corn
29	36	Special Yellow Velvet Soup with Prawns

Fig. 29. Query execution after indexing

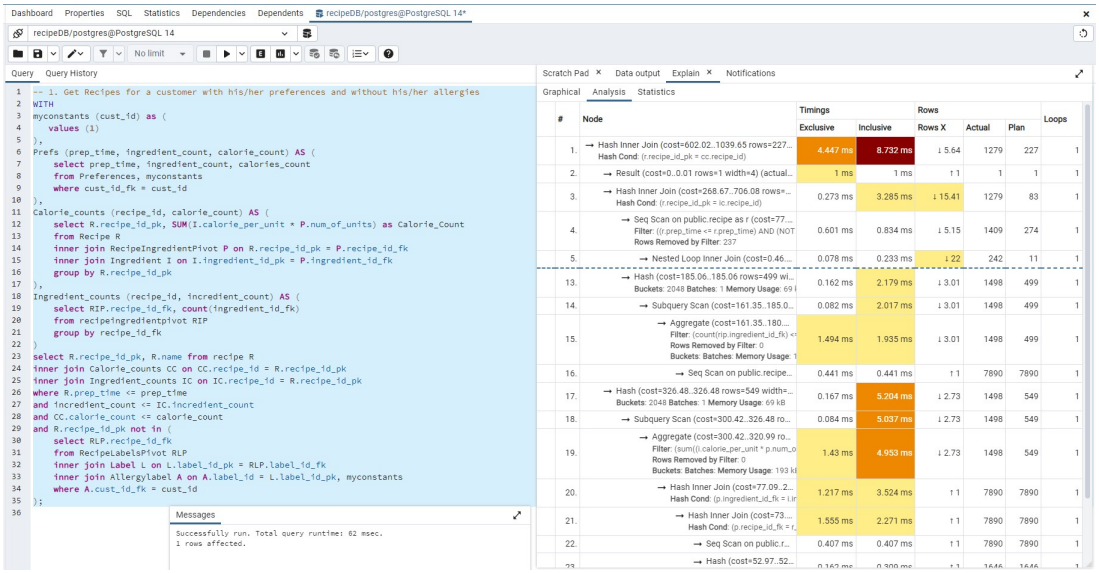

Fig. 30. Analysis of the query execution after indexing

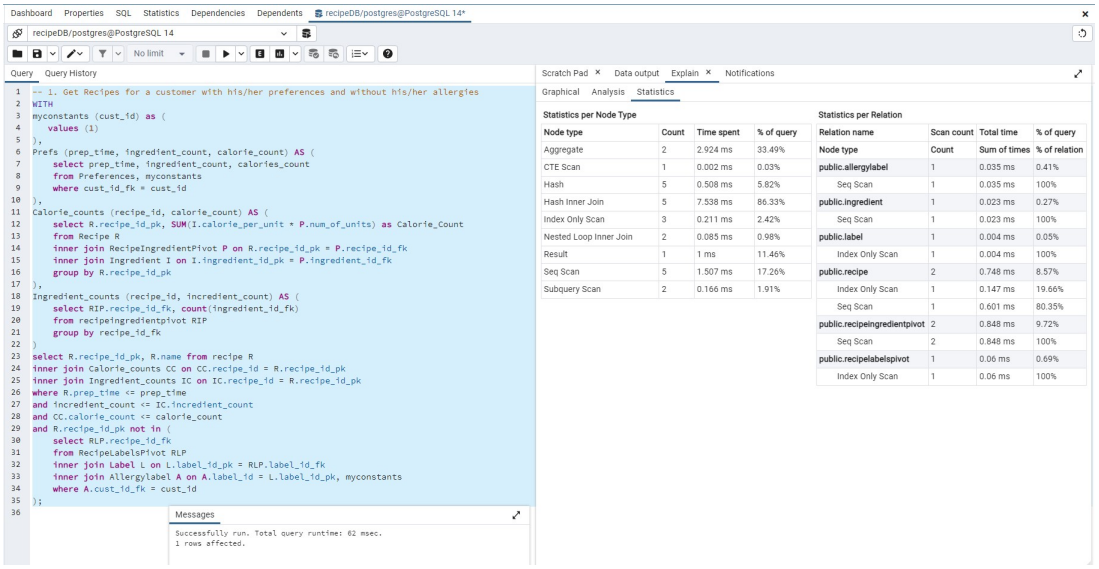

Fig. 31. Statistics of the query execution after indexing

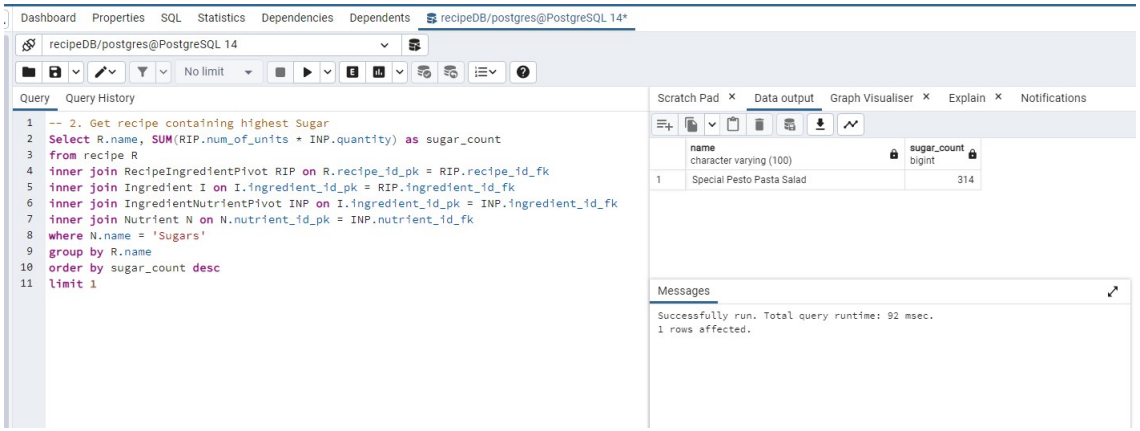

Fig. 32. Query execution before indexing

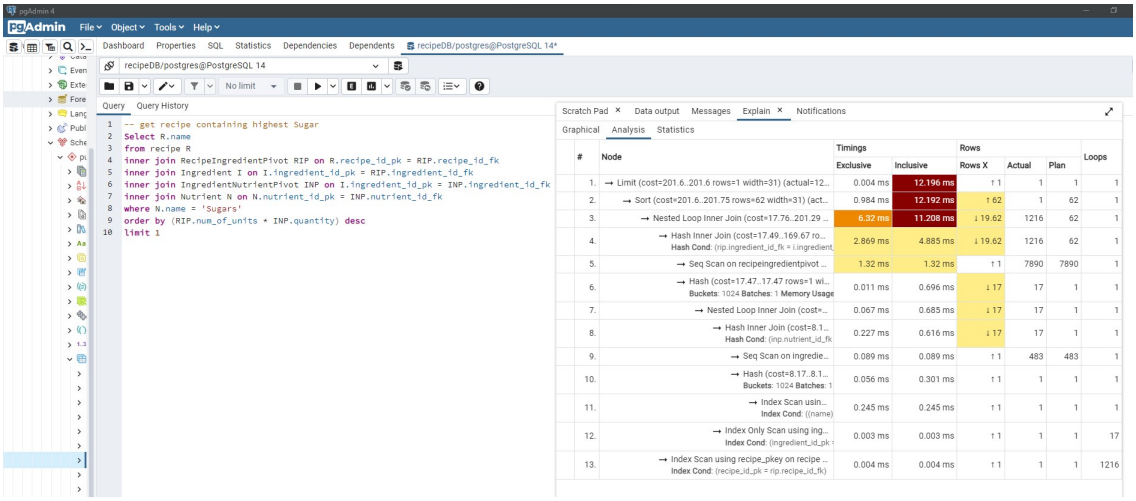

Fig. 33. Analysis of the query execution before indexing

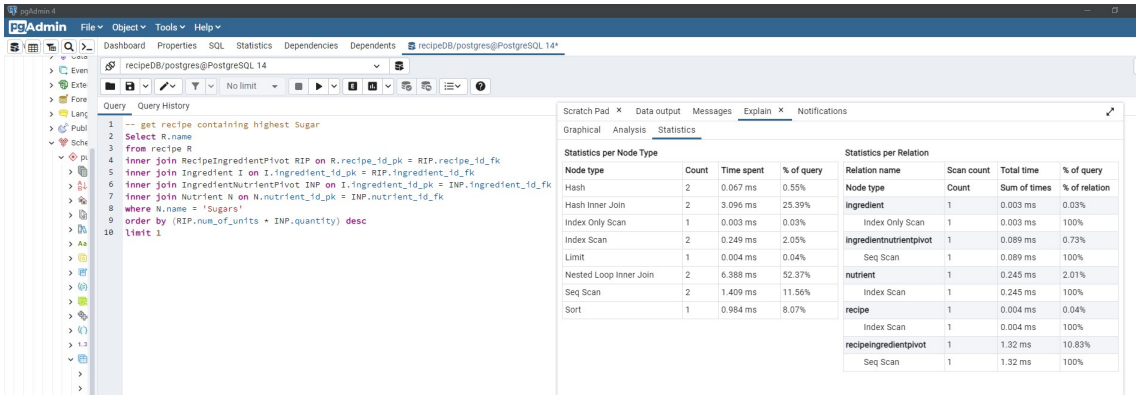

Fig. 34. Statistics of the query execution before indexing

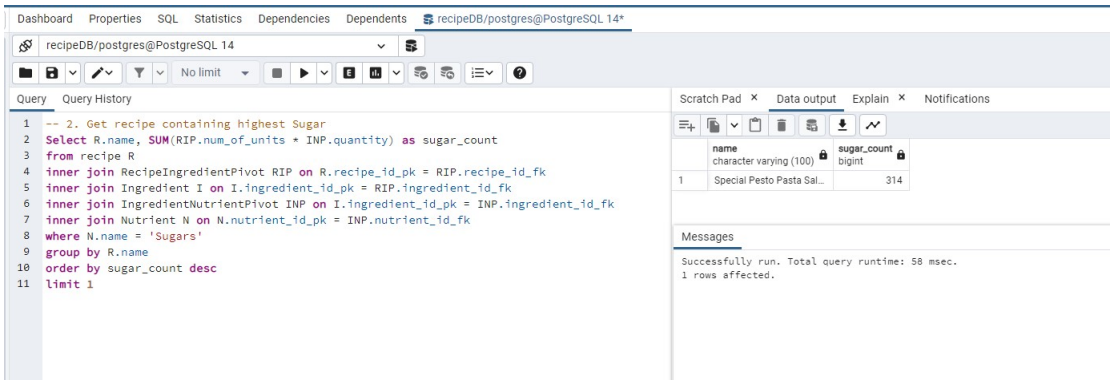

Fig. 35. Query execution after indexing

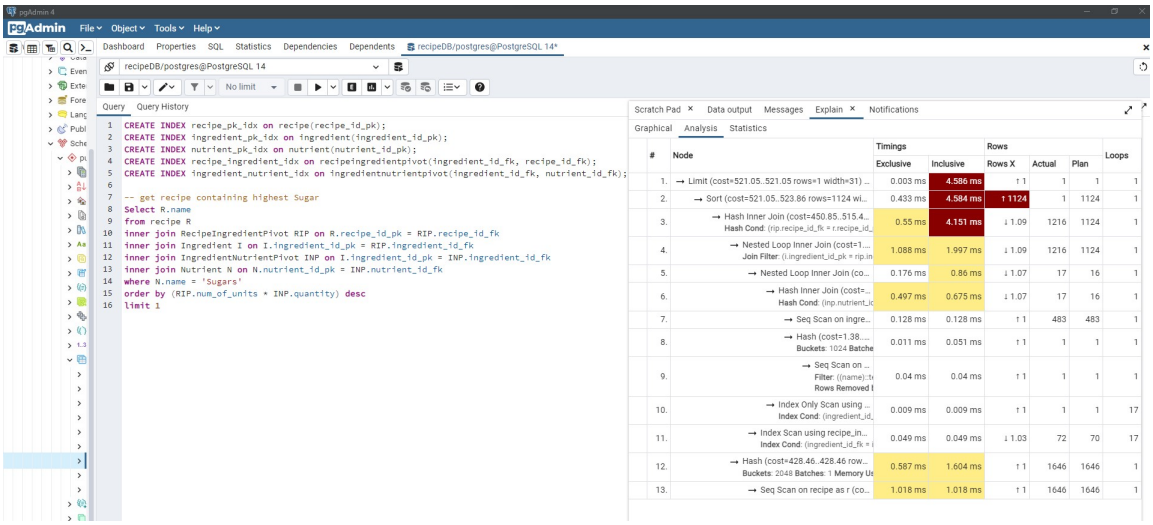

Fig. 36. Analysis of the query execution after indexing

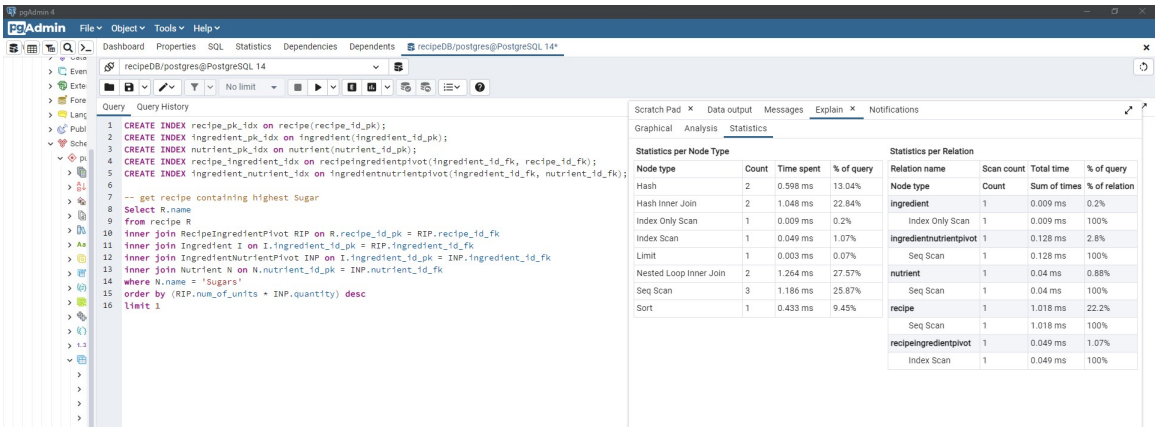

Fig. 37. Statistics of the query execution after indexing

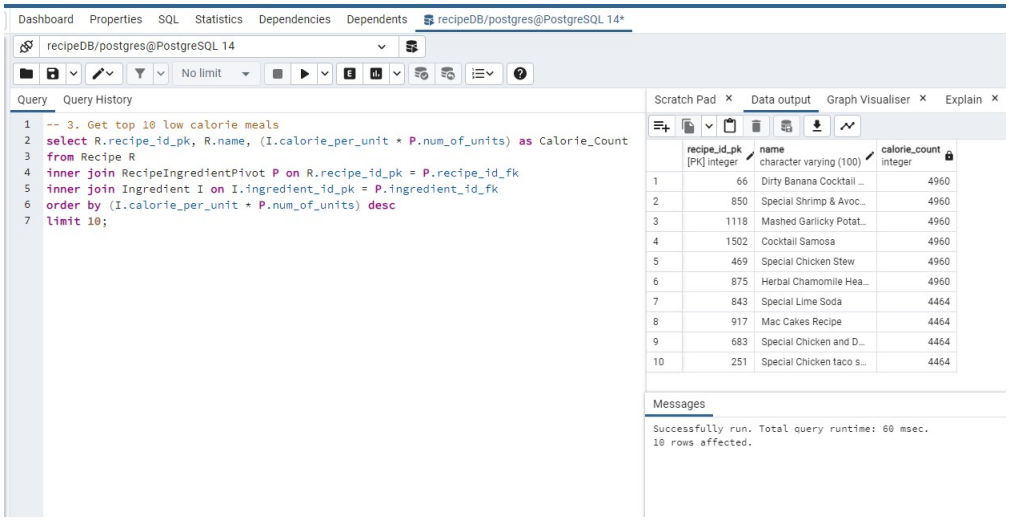

Fig. 38. Query execution before indexing

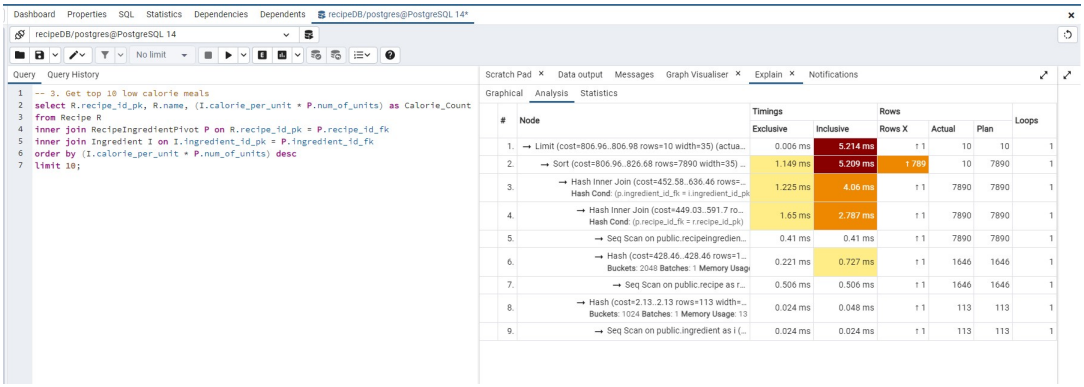

Fig. 39. Analysis of the query execution before indexing

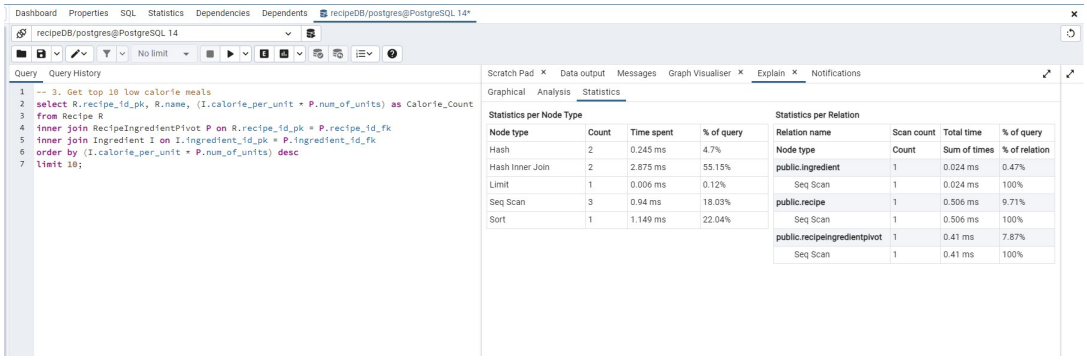

Fig. 40. Statistics of the query execution before indexing

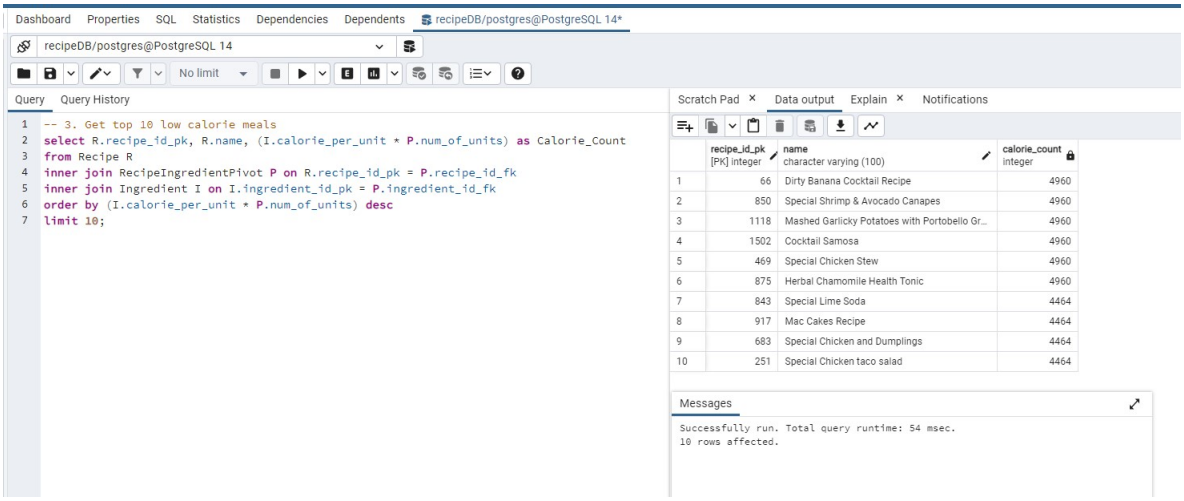

Fig. 41. Query execution after indexing

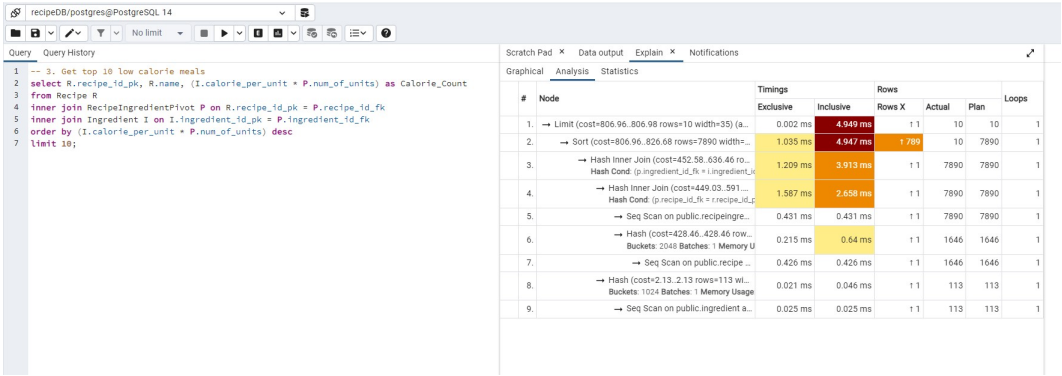

Fig. 42. Analysis of the query execution after indexing

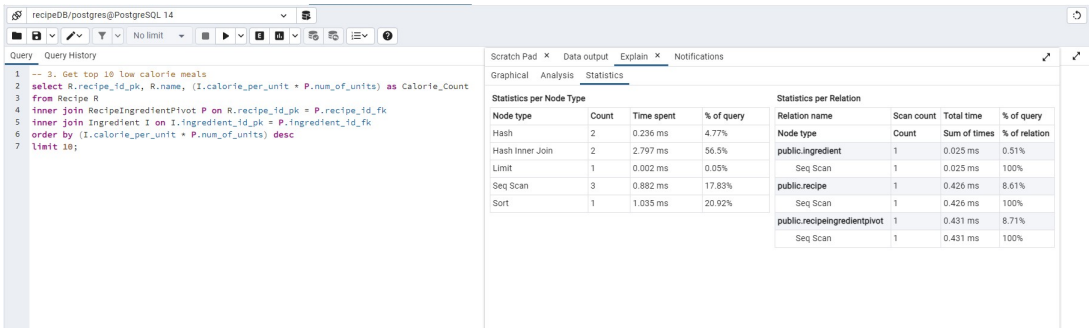

Fig. 43. Statistics of the query execution after indexing

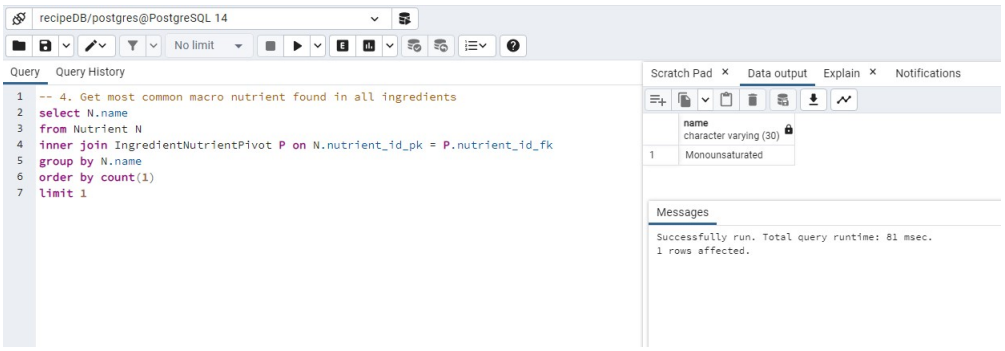

Fig. 44. Query execution before indexing

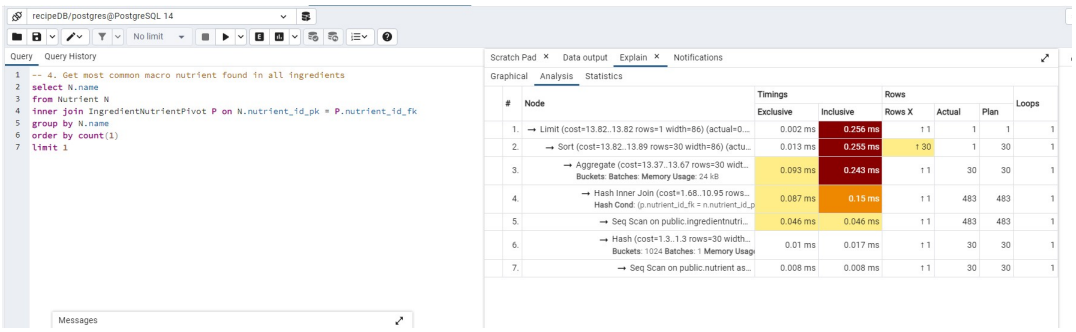

Fig. 45. Analysis of the query execution before indexing

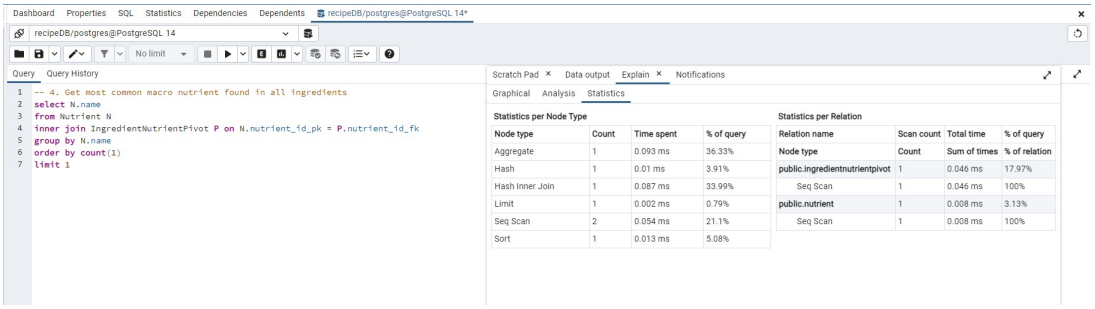

Fig. 46. Statistics of the query execution before indexing

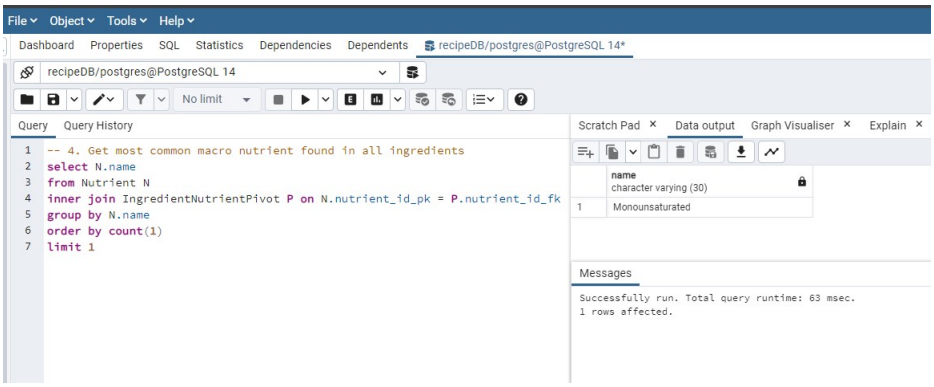

Fig. 47. Query execution after indexing

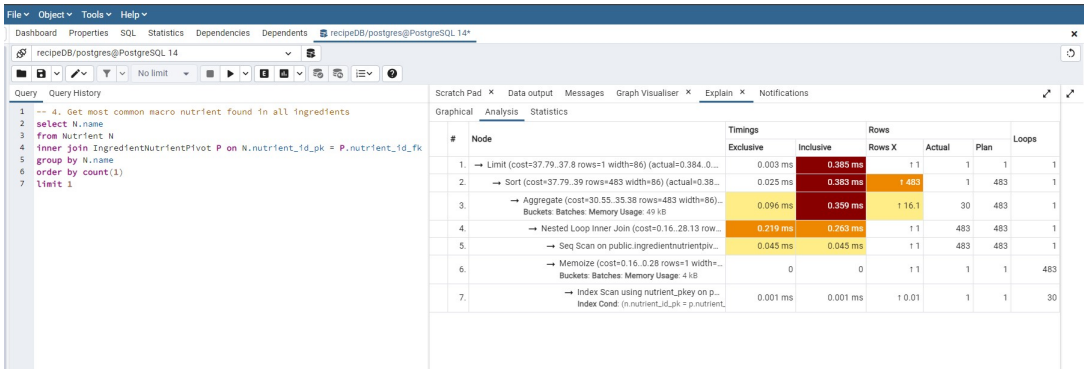

Fig. 48. Analysis of the query execution after indexing

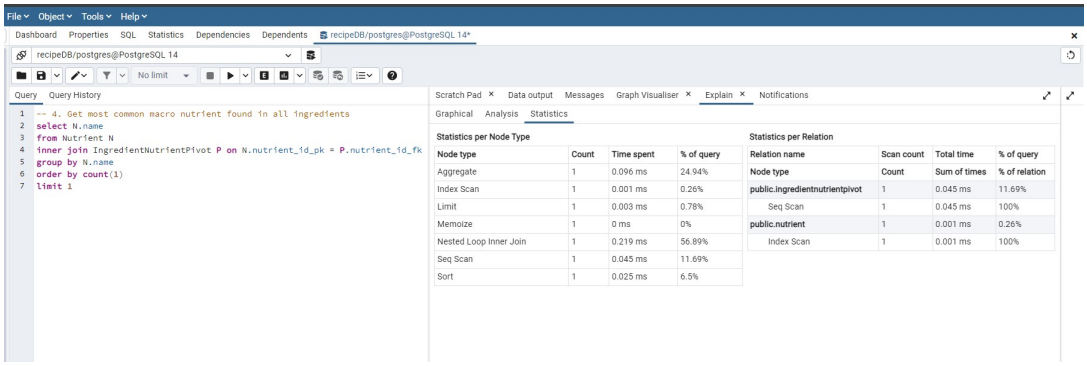

Fig. 49. Statistics of the query execution after indexing

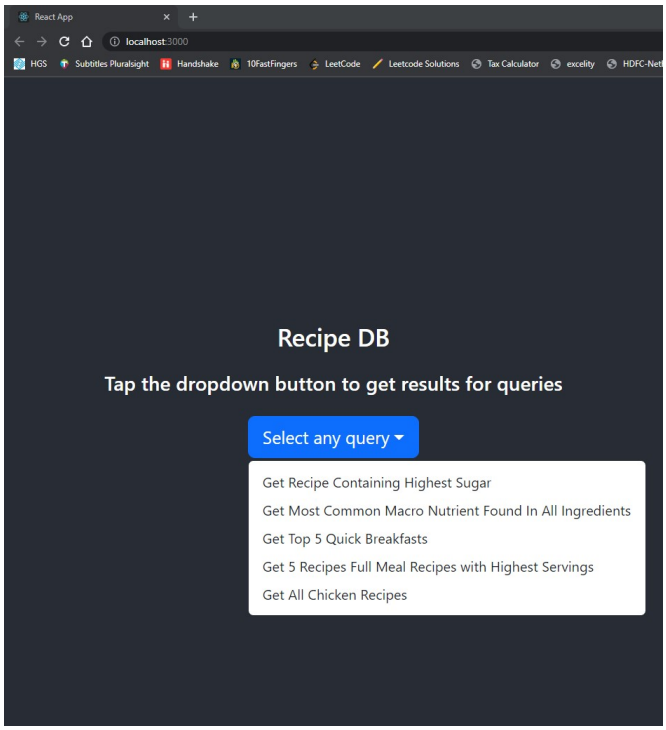

Fig. 50. Demo Frontend - Img1

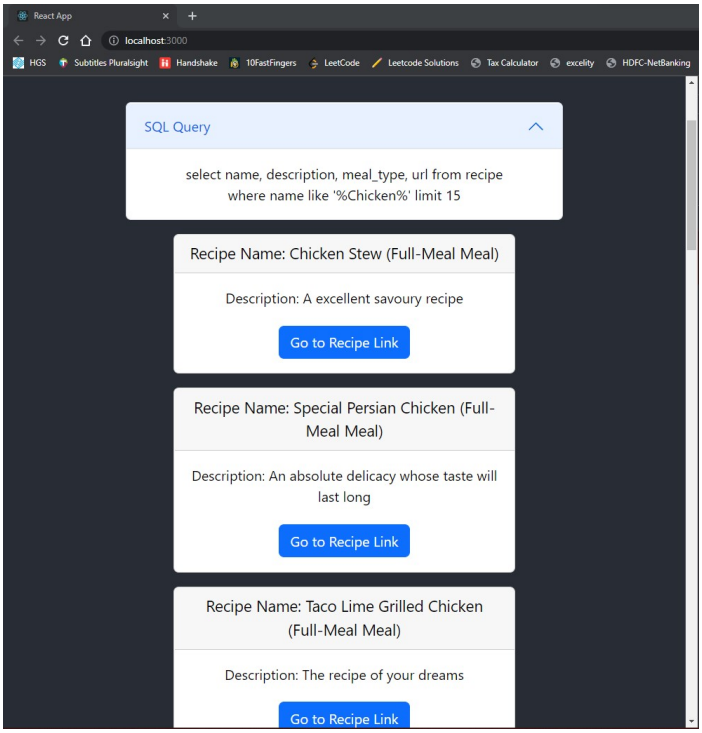

Fig. 51. Demo Frontend - Img2

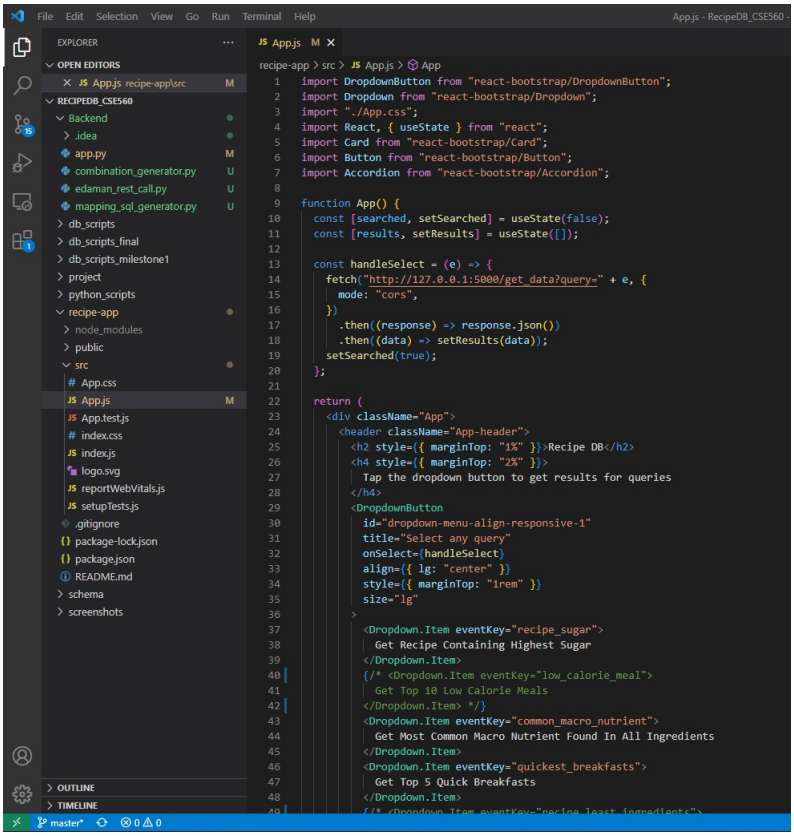

Fig. 52. React Application

app.py M ×

Backend > app.py

```python
from flask import Flask, request
import psycopg2
from psycopg2.extras import RealDictCursor
from flask_cors import CORS, cross_origin

app = Flask(__name__)
cors = CORS(app)
app.config['CORS_HEADERS'] = 'Content-Type'

conn = psycopg2.connect(
    host="localhost",
    database="recipeDB",
    user="postgres",
    # password="pexplorer"
    password="admin"
)


def execute_query(query1):
    curr = conn.cursor(cursor_factory=RealDictCursor)
    curr.execute(query1)
    return curr.fetchall()


@app.route('/hello')
@cross_origin()
def get_data1():
    return "Hello World"


@app.route('/get_data')
@cross_origin()
def get_data():
    query_arg = request.args.get("query")
    final_result = []

    if query_arg == 'recipe_sugar':
        sqlquery = """
        Select R.name, SUM(RIP.num_of_units * INP.quantity) as sugar_count
        from recipe R
        inner join RecipeIngredientPivot RIP on R.recipe_id_pk = RIP.recipe_id_fk
        inner join Ingredient I on I.ingredient_id_pk = RIP.ingredient_id_fk
        inner join IngredientNutrientPivot INP on I.ingredient_id_pk = INP.ingredient_id_fk
        inner join Nutrient N on N.nutrient_id_pk = INP.nutrient_id_fk
        where N.name = 'Sugars'
        group by R.name
        order by sugar_count desc
        limit 1
        """
```

master* ⊗ 0 ⚠ 0

Fig. 53. Flask Application