NYC Dish Finder

Andrew Clark
Joseph Goodman
Arielle Stern
Vishal Tien

Abstract

Finding the right place to eat can be quite the challenge, especially when presented with many options in a city like New York City. Our web application aims to connect users in the New York area with the food they want in a fast and user-friendly way by allowing them to search off of many custom conditions. Built using MySQL, AWS, Node, and React our database and web application integrate and deliver restaurant and dish level data to our users. Through our schema normalization and query optimization, we ensure that users get results to their searches in an efficient and timely manner.

Introduction

Our project is a web application that provides more powerful searching capabilities for users interested in finding specific food dishes or interested in using other advanced search criteria to find restaurants within New York City (NYC). The main goal of our web application is to connect our users with the specific food that they want, in a fast and user-friendly way. We have found from using food delivery services and map services like Google Maps that it is often difficult to locate the exact food you want at a location nearby, especially at the granularity level beyond restaurant cuisines, for example, finding restaurants serving specific food dishes. Our application aims to simplify this process, allowing users to search for food by a dish, cuisine, or restaurant and conveniently view nearby results in detailed lists or geographic views. Further, since we limited our web application to restaurants in the New York City area, we are able to provide our users with detailed information about restaurant location such as the borough in which a restaurant is located, as well as the exact coordinates of the restaurant. We hope that our application helps our users learn more about the food scene around their location, compare similar restaurants, as well as discover new restaurants.

Data Sources

Data Sources
Allmenus API:
- Data access for restaurants and their menus
- Summary statistics: 41k+ restaurants (22k+ in NYC), millions of food dishes (2 million+ in NYC)

Kaggle's NYC Boroughs Dataset:
- Links zip codes to their associated borough in NYC
- Summary statistics: 240 zip codes, 5 boroughs

Images scraped from 123rf and unsplash.

- 8000+ images related to food dishes sampled from Allmenus API.

Data Access
To store the restaurant and food dish items, we queried the Allmenus API for 50k restaurants. Each key to the API was allowed to make 400 queries and we used 11 keys to query their database 44,000 times. Each query contained a page of 10 results from which we got each restaurant's menu, location, name, email, website, and phone.

We then saved the API results in csv files from which we imported into our database. To scrape images of food corresponding to our 2 million food items, we randomly sampled food names and used Selenium to search them on different free image sites.

For every image that had a result, we saved the link to the corresponding image under the name of the dish searched.

We sampled a total of 15k food items out of 2 million and have 5661 corresponding pictures of food.

Data Cleaning in Python
The process of converting the data we pulled from the API as described above into a SQL database involved transforming unstructured JSON data into a tabular format. In order to restructure our data, we used Python to loop through the nested JSON arrays present in our data and extract the desired information into a dictionary with consistent key values. With the dictionary in a consistent format with that required by relational databases, we were then able to convert the dictionaries into .csv files.

Technologies

React, JSX, and Node
To build our web app our team used React.js, JSX, and Node.js. React.js was used to create the different components of our website (like buttons, drop downs, text boxes, posts, etc.) and to provide interactive functionality for those components. We also made use of local storage via JavaScript functions in order to cache inputs from the user. We used JSX to define what our web app will actually look like visually, making use of state variables for conditional formatting. Our team used Express from Node.js to handle the routing (GET and POST requests) on our web app, which allows us to retrieve data from our database and display it on the appropriate pages.

MySQL
Our database is a MySQL database hosted through Amazon Web Services's (AWS) Relational Database Service (RDS). We accessed our database either through the command line or using the MySQL Workbench application.

Architecture

Home Search Page
The Home Search Page (which is the page that the user sees when first navigating to the web app) has various different inputs that the user can specify based on preference. Specifically, the user can opt for just a simple search by using the top search

bar. When the user inputs a dish or cuisine into the top search bar and clicks the "simple search" button the user will be routed to the results page, which contains various information about the restaurants that serve the cuisine or dishes that were entered. Refer to the *Results Page* section below for a more detailed description of the information that is provided on the results page. Another feature on the Home Search Page is the button that reads "Want Info on A Specific Restaurant? Click Here!". This button allows the user to input a specific restaurant name, and then returns on the results page the restaurant with the cheapest prices compared to other restaurants with the same name. The next feature on the Home Search page is the Advanced Search button. When this button is clicked, a text box to enter a cuisine, an interactive map that allows you to select your location, a select borough dropdown, and a select price dropdown appears. The user is able to specify any combination of these and when the user clicks the "Advanced Search" button the web app routes to the Results page which displays the information about the restaurants that match the inputs of the user. The last feature on the Home Search page is a very specific feature of the web app that allows a user to find the restaurant in she/he's selected borough in a certain price range that serves a certain cuisine of choice. The idea behind this feature is that if a user knows what kind of food she/he wants to eat and what her/his price range is, that user can find the restaurant in New York City with the most selection that matches her/his selections.

Results Page

The results page shows restaurants meeting search criteria in a cards format. For each matching restaurant, we display the restaurant's name and location. This view allows users to quickly see restaurants based on their custom criteria. If a certain restaurant peaks a user's interest, they can view more detailed information about the restaurant's menu by clicking on the card associated with that restaurant to open the menu page.

Map Page

In addition to displaying the restaurant results returned by a given search condition in a list of cards format, we also decided to include an interactive map with pop-up markers displaying the locations of the restaurants returned on the results page. This allows users to see with a quick glance exactly where the restaurants that are returned by their search are located, which is an important piece of information when deciding where to eat out. Furthermore, when markers on the map are clicked, the map zooms in and pans to center the clicked marker, and info windows pop up that display the name of the restaurant, the price range (indicated by the number of "$" signs), and a hyperlink to the menu for that restaurant.

Menu Page

The menu page lists the price for each item on the menu for a specific restaurant with the percentage price differential between each item and the average among all similar items at other restaurants. For instance, if a

hamburger costs five dollars at one restaurant where the average is four, there will be a +25% indicator on the image of a hamburger on the menu.

In addition to listing the stored menu from our database, we also linked pictures of food to the title and description of each item using the images we scraped. For each item, we found the picture with the most similar key using a heuristic similarity score based off of the tf-idf metric.

Tailwind CSS Template

For our project we utilized (with our project TA's approval) a React + CSS template from: https://treact.owaiskhan.me/#. This template used Tailwind CSS and just provided our team with styling and code for the basic outline of pages. We then went ahead and added components like (but not limited to) buttons, dropdowns, text boxes, page links, maps, and images based on fuzzy string matches of food item names. Additionally, we connected our database hosted on AWS to the frontend of our web app and linked our pages together when buttons are clicked.

Database

Database Population

To populate our database, we uploaded the .csv files generated from our Python data cleaning. Since the data upload through MySQL proved to be prohibitively slow, we opted to use the command line for ingesting data into our database. After uploading the data, we had the following tables in the database: Restaurant, FoodItem, Cuisine, NYzips. Once the data was in the database, we used SQL for further data cleaning.

Database Cleaning

For the FoodItem and Cuisine tables, we focused on removing instances with no useful information. For the FoodItem table, we noticed that several dishes had names or descriptions that were both the empty string, a space, a dot, a dash, or some combination of these characters. We removed the dishes that had any of these characters as both the name and description from the table as there was no useful information to dictate what the dish was. For dishes where the name was one of these characters, but the descriptions provided useful information about the dish, we copied the description over to the name. For the Cuisine table, we removed instances in which the cuisine was the empty string.

For the Restaurant table, we focused on normalizing the data in the table. Particularly, we trimmed extra white space in the restaurant name, city, longitude and latitude. Additionally, we wanted to normalize city names to make it easier for users to filter by city. Since the city field had varying capitalization, we decided to set all the city names to be upper case. Lastly, we decided to limit our database to restaurants in New York City, defined as being in one of the city's five boroughs. We made this decision so that we could best visualize results geographically. We used restaurant's zip codes to filter the data to those only in New York City. Originally, we had data from 41,450 restaurants. After filtering to restaurants in New York City, we had 22,342 restaurants remaining. Once all the data remaining was for restaurants in New York

City, we added a borough column to the Restaurant field. We then eliminated the NYzips table from the database as its contents were incorporated into the Restaurant table.

Table Sizes

After data cleaning, we were left with a large amount of tuples in each table. The table below shows the number of instances in each table in the database

Table	**Number of Instances**
Restaurant	22,342
FoodItem	2,659,021
Cuisine	28,563

Table Schemas

Restaurant(restaurant_id, restaurant_name, restaurant_phone, restaurant_website, hours, price_range, price_range_num, city, state, postal_code, street, formatted, lat, lon, borough)
FoodItem(name, description, rest_id, price, section_name)
-FK rest_id references Restaurant.restaurant_id
Cuisine(cuisine, restaurant id)
-FK restaurant_id references Restaurant.restaurant_id

Schema Normal Form

Our schema is in Boyce-Codd Normal Form (BCNF). For the Restaurant table, restaurant_id functionally determines all other attributes in the table. This relationship holds because restaurant_id dictates the restaurant's name, logistic features such as hours and website, and location. Since restaurant_id is the key of the Restaurant table, it is a superkey of itself, so this relationship satisfies BCNF. For FoodItem, name, description, and rest_id functionally determine price. This relationship holds since the combination of the item being served and the restaurant serving the item dictates the features about the item. Note that description is needed to determine price since we found that it is possible for a restaurant to serve several items with the same name, but with different descriptions and prices. Since name, description, and rest_id is the key of the FoodItem table, it is a superkey of itself, so this relationship satisfies BCNF. Lastly, for the Cuisine table, restaurant_id and cuisine together functionally determine cuisine. Note that restaurant_id on its own does not determine cuisine as a restaurant can be associated with more than one cuisine. Since restaurant_id and cuisine is the key of the Cuisine table, it is a superkey of itself, so this relationship satisfies BCNF. Therefore, all relations in the schema satisfy BCNF.

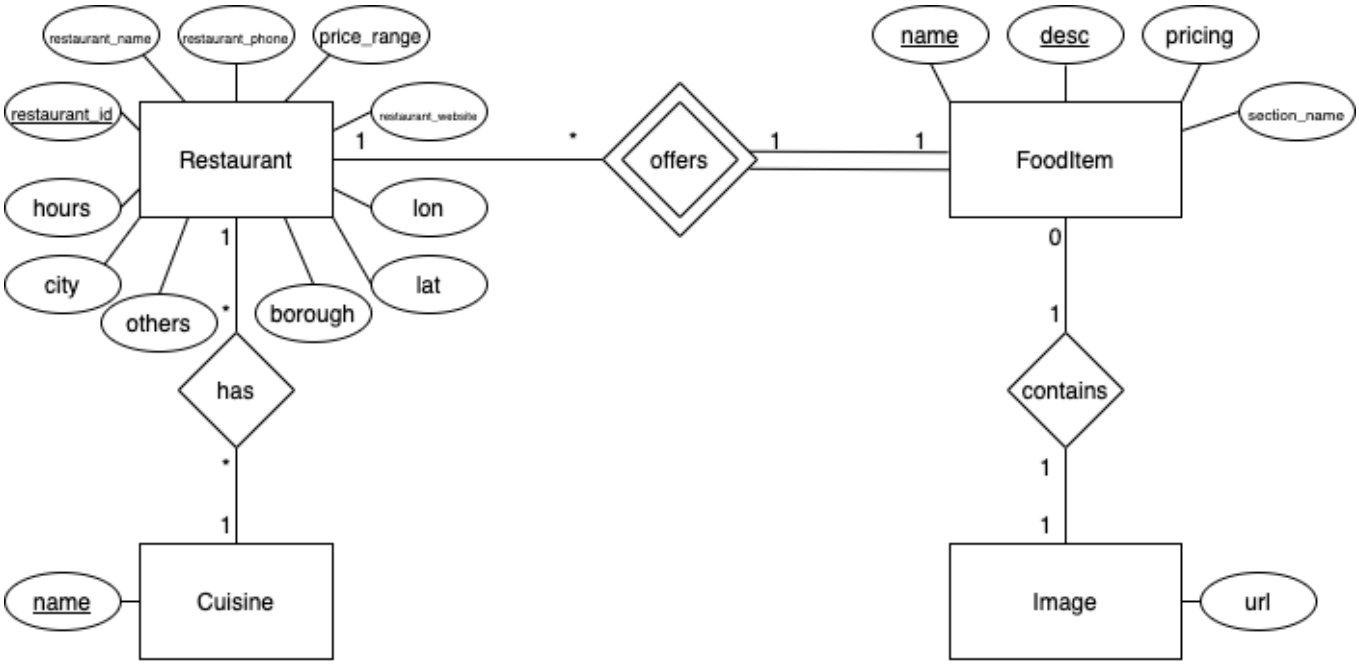
Entity Relationship Diagram

The figure on the following page shows our ER diagram. Since the restaurant table has many attributes, we grouped some of them together into "other" for the sake of the visualization. The attributes not shown on the diagram include price_range_num, state, postal_code, street, and formatted.

Performance Evaluation

Reducing Data Redundancy

Originally, we noticed that our data was redundant as the cuisine information was stored along with the restaurant information.



This setup was not ideal as a restaurant can be associated with multiple cuisines. Therefore, we separated the cuisine information from the restaurant information in order to store our data more efficiently.

Query Restructuring

Throughout writing queries, we focused on reducing intermediate table sizes and joining data in optimal orders. As an example, consider the following two versions of the same query:

a.
```
WITH joined AS (
SELECT r.*, cuisine
FROM Restaurant r
JOIN Cuisine c ON
r.restaurant_id =
c.restaurant_id
),
rest_ids AS (
SELECT restaurant_id FROM
joined),
num_dishes AS (
SELECT COUNT(name) AS
num_dishes, FoodItem.rest_id
FROM FoodItem
JOIN rest_ids ON
FoodItem.rest_id =
rest_ids.restaurant_id
GROUP BY FoodItem.rest_id
)
SELECT num_dishes,
restaurant_name,
restaurant_website,cuisine FROM
num_dishes
JOIN joined ON
num_dishes.rest_id =
joined.restaurant_id
WHERE cuisine LIKE '%Chinese%'
AND price_range <= '$$' AND
city = 'STATEN ISLAND' AND
LENGTH(restaurant_website) >0
ORDER BY num_dishes DESC LIMIT
1;
```

b.
```
WITH rests AS (
  SELECT *
  FROM Restaurant
```

```
  WHERE price_range <=
'${inputPrice}' AND borough
LIKE '%${inputBorough}%' AND
LENGTH(restaurant_website) >0
),
  cuisines AS(
  SELECT * FROM Cuisine
  WHERE Cuisine LIKE
'%${inputCuisine}%'
  ),
  joined AS (
  SELECT rests.*,
cuisines.cuisine
  FROM rests
  JOIN cuisines ON
rests.restaurant_id =
cuisines.restaurant_id
  ),
  rest_ids AS (
  SELECT restaurant_id FROM
joined),
  num_dishes AS (
  SELECT COUNT(name) AS
num_dishes, FoodItem.rest_id
  FROM FoodItem
  JOIN rest_ids ON
FoodItem.rest_id =
rest_ids.restaurant_id
  GROUP BY FoodItem.rest_id
  )
  SELECT joined.*, num_dishes
FROM num_dishes
  JOIN joined ON
num_dishes.rest_id =
joined.restaurant_id
  ORDER BY num_dishes DESC
LIMIT 1;`
```

The biggest difference between these two queries is that the first version applies selection on price range, cuisine, city, and website after joining the Cuisine and Restaurant tables whereas the second applies this projection before the joins. The second version also only retains necessary columns for each intermediate table. This optimization of limiting table sizes as early as possible by pushing down projections and selections reduced the time this query took to execute substantially. Throughout our optimization, we noticed that the biggest gains in performance came from restructuring queries to reduce the sizes of intermediate results we generated.

Indexes

In addition to the indexes already placed on each table by MySQL, we added several indexes to improve query execution. MySQL places indexes on primary key and foreign key fields which resulted in the system creating the following indexes: restaurant_name in Restaurant table, (restaurnt_id, cuisine) and restaurant_id in Cuisine table, (rest_id, name, description) and rest_id in FoodItem table. In addition to these indexes we added the following indexes: restaurant_name in Restaurant table (and cached result), cuisine in Cuisine table, name in FoodItem table (and cached result). For each index added, we noticed run time improvements ranging from several milliseconds to several seconds. The index on restaurant_name sped up the second complex query, the index on cuisine sped up the first, third, and fourth complex queries, and the index on name sped up the first complex query. Still, we believe the biggest benefits from indexes likely came from the indexes the database automatically imposes as these indexes capture the fields of our most frequent joins.

Caching

In addition to our other optimization efforts, we cached several results that took a long time to compute. First, to speed up our first complex query, we cached results for the average price of a given food item to allow us to quickly determine where dishes are cheaper than average. Additionally, we cached results for the cheapest restaurant of a given chain to speed up our second complex query. It was important for us to cache results for these two queries specifically since their results are displayed often. Further, for the first complex query, we wanted the output of this query to be shown on the Menu Page, meaning that we would need to run it for each item on a menu. Due to the large number of dishes on a restaurant's menu, it was particularly important that this query execute quickly. For the third and fourth complex queries, the large amount of user input hindered us from being able to effectively cache results. Also, to optimize searching for restaurants either serving a given dish or offering a given cuisine directly, which was not a complex query, we cached a result for the join of the FoodItem, Restaurant, and Cuisine table. This result enables us to search for a dish or cuisine without having to join all three tables each table we want to search for a new item.

Computing the tf-idf score of each food item to link it to a picture required us to iterate through the entire FoodItems table (2+ million entries) and count the number of times each word in a dish's title and description was included in another title or description. Just this part of the query took far too long to run.

Luckily, our dataset is fixed. The number of times a word is used is a constant number we were able to store in another table and then query for. This query then took milliseconds to run instead.

Final Results

After our various optimization efforts, we achieved the following run times for our complex queries. These queries, before and after optimization, are included in the appendix. The rest of the queries implemented in our web application are included in the code deliverable.

Query	Original Time	Optimized Time	Speedup
1	122.923 s	0.082 s	99.9%
2	1.125 s	0.031 s	97.2%
3	0.177 s	0.036	80.0%
4	9.976 s	0.317 s	96.8%

Technical Challenges

We encountered several technical challenges over the course of the project. First, the shear size of our data set imposed challenges as we struggled to get queries to run without optimizing them. Since the FoodItem data contains over 2 million records even after limiting the data to restaurants in NYC, we had to be highly strategic in terms of our query implementations and were limited in our ability to efficiently try out various ideas. Additionally, this led to our query optimization steps requiring careful thought, as if otherwise left alone, many queries that

formed the basis for pages on our web application would load very slowly.

Another technical challenge we faced was integrating an interactive map into our web application. Our team had no previous experience with using the google maps api and integrating this with the rest of the React framework. Thus, there was a learning curve to climb once we decided our web application would benefit greatly from an interactive map. In the end, we were able to integrate the map both into the results page of our website, as well as embed a map in the advanced search section of our home page.

A third challenge we faced in the development of our website was utilizing a template for the front-end design of the application. This meant both learning how to integrate our components with a React template, as well as customizing various areas of the template to fit the needs of our application. One of the most challenging components of this step was filling in the image placeholders in the template with meaningful images for our purposes. Specifically, in the menu page of our application, each dish on the restaurant menu is displayed as a card. The main space on the card is taken up by stock images by default. Our dataset did not include images for each of the dishes in the food items table. In order to replace the stock images from the template with pictures that corresponded to the dishes each restaurant serves, we web scraped images of food from the internet. We then used natural language processing techniques to match the titles of dishes in the food items table with the names of dishes that we web scraped. We were then able to map each dish to its most similar image, and visual inspection of many restaurant menus showed successful results.

Extra Credit Features

Google Maps API
Integrating an interactive map into our website required learning how to use the Google Maps API, set up an API key, and create features on top of the API for our specific use-cases, such as pop-up markers and info windows, and panning and zooming to center on clicked markers. Therefore, we believe this feature was a significant extension to the web development part of our project, enhancing the utility of our application.

TF-IDF Picture Linking
Linking words to pictures is a notoriously hard task researchers are building complex machine learning models to solve.

Given this was a database class and not an ML class, we elected to not build our own convolutional neural net. Instead, we searched for and scraped images from the web. These images were then labeled by the terms we searched for which we then matched with the titles of other dishes.

Linking the titles of dishes together still wasn't a trivial task. Using an embedding might have worked, but because every dish was a unique title we weren't sure that the distance between the embeddings would properly represent the difference between

the words. For instance, a steak with the title *steak 16oz with fries* was actually very similar to a *new york strip* entity.

Our solution to this was to store a tf-idf dictionary of how often each word was used in a title and description of a food item. Words that weren't used often like *new york* would carry more weight than words like *with* or *fries*.

From the dictionary, we then looped through the key for each image and gave a score of 1/occurances(word) for every word in the key that matched a word in the menu item's title or description. For every word that didn't fit we subtracted a score of 1/(occurances(word) *1000).

This was intended to be a miniscule penalty which was much weaker than the reward for matching any word that would break ties on examples where we want to match *breaded chicken* and had to decide between *chicken* and *chicken with snow peas*. The first term is clearly better because it doesn't include extraneous words that don't match.

This system worked extremely well in practice and we are very happy with how well the pictures matched the description. There are a few examples where the labeled pictures we scraped aren't images of food like for *octopus,* we have an image of an actual octopus, but overall the images do a fantastic job of giving more life and context to our menu page.

Appendix 1- Complex SQL Queries Optimized in Performance Evaluation Section. The first version of query (a) is not optimized, and the second version of query (b) is optimized.

1 . For a given cuisine and dish, get places where the dish is priced below average.

a.
```
WITH choose_food_cuisine
AS(SELECT restaurant_name,
name, description, price,
cuisine
FROM FoodItem F
JOIN Cuisine C ON F.rest_id =
C.restaurant_id
JOIN Restaurant R ON
R.restaurant_id = F.rest_id)
SELECT restaurant_name, name,
description, price
FROM choose_food_cuisine
WHERE price < (SELECT
AVG(price) FROM
choose_food_cuisine) AND
cuisine = 'Italian' AND name =
'Pizza' AND price <> 0;
```

b.
```
SELECT name, AVG(price) as
price, section_name as category
FROM
average_price_by_name_cuisine
WHERE name = 'Pizza'
GROUP BY name;
```
*price comparison portion moved to JavaScript

2. Find the restaurant with cheapest dishes on average with regards to the equivalent dishes at the other restaurant locations.

a.
```
WITH avg_prices_rest AS
((SELECT r1.restaurant_id,
AVG(f1.price) as avg_price
      FROM Restaurant r1
    INNER JOIN FoodItem f1 ON
r1.restaurant_id = f1.rest_id
     WHERE restaurant_name =
     "McDonald's"
     GROUP BY r1.restaurant_id
    HAVING avg_price != 0))
SELECT DISTINCT
r.restaurant_id,
r.restaurant_name
FROM Restaurant r
INNER JOIN FoodItem f ON
r.restaurant_id = f.rest_id
WHERE r.restaurant_name LIKE
"%McDonald's%"
AND r.restaurant_id = (SELECT
av.restaurant_id
                FROM
avg_prices_rest av
                WHERE
av.avg_price <= ALL(SELECT
avg_price

FROM avg_prices_rest));
```

b.
```
SELECT * FROM cheapest_chain
WHERE restaurant_name LIKE
"%McDonald's%";
```

3. Find all restaurants of a cuisine type serving food items of a certain price range within a close distance from ourselves.

a.
```
WITH dists AS
(SELECT r.*, 3959 * acos( cos(
radians(40.7) ) * cos( radians(
lat ) ) * cos( radians(lon) -
radians(-74)) +
sin(radians(40.7)) * sin(
radians(lat))) AS distance
FROM Restaurant r),
cr AS
(SELECT r.*, cuisine
FROM Cuisine c JOIN Restaurant
r ON r.restaurant_id =
c.restaurant_id),
```

```
joined AS
(SELECT cr.*, distance FROM
dists JOIN cr ON
dists.restaurant_id =
cr.restaurant_id)
SELECT *
FROM joined
WHERE cuisine = "Italian" AND
price_range = "$$" AND borough
= "Manhattan" AND distance < 1
ORDER BY distance;
```

b.
```
SELECT r.*,( 3959 * acos( cos(
radians(40.7) ) * cos( radians(
lat ) ) * cos( radians(lon) -
radians(-74)) +
sin(radians(40.7)) * sin(
radians(lat)))) AS distance
FROM Cuisine c JOIN Restaurant
r ON r.restaurant_id =
c.restaurant_id
WHERE c.cuisine = "Italian" AND
r.price_range = "$$" AND
borough = "Manhattan"
HAVING distance < 1
ORDER BY distance;
```

4. Find the restaurant (less than or equal to a certain price range) name, number of sides, and website that serves a certain cuisine in a certain city and has the most number of dishes.

a.
```
WITH joined AS (
SELECT r.*, cuisine
FROM Restaurant r
JOIN Cuisine c ON
r.restaurant_id =
c.restaurant_id
),
rest_ids AS (
SELECT restaurant_id FROM
joined),
num_dishes AS (
SELECT COUNT(name) AS
num_dishes, FoodItem.rest_id
FROM FoodItem
JOIN rest_ids ON
FoodItem.rest_id =
rest_ids.restaurant_id
GROUP BY FoodItem.rest_id
)
SELECT num_dishes,
restaurant_name,
restaurant_website,cuisine FROM
num_dishes
JOIN joined ON
num_dishes.rest_id =
joined.restaurant_id
WHERE cuisine LIKE '%Chinese%'
AND price_range <= '$$' AND
city = 'STATEN ISLAND' AND
LENGTH(restaurant_website) >0
ORDER BY num_dishes DESC LIMIT
1;
```

b.
```
WITH rests AS (
  SELECT *
  FROM Restaurant
  WHERE price_range<=
'${inputPrice}' AND borough
LIKE '%${inputBorough}%' AND
LENGTH(restaurant_website) >0
),
  cuisines AS(
  SELECT * FROM Cuisine
  WHERE Cuisine LIKE
'%${inputCuisine}%'
  ),
  joined AS (
  SELECT rests.*,
cuisines.cuisine
  FROM rests
  JOIN cuisines ON
rests.restaurant_id =
cuisines.restaurant_id
  ),
  rest_ids AS (
```

```
  SELECT restaurant_id FROM
joined),
  num_dishes AS (
  SELECT COUNT(name) AS
num_dishes, FoodItem.rest_id
  FROM FoodItem
  JOIN rest_ids ON
FoodItem.rest_id =
rest_ids.restaurant_id
  GROUP BY FoodItem.rest_id
  )
  SELECT joined.*, num_dishes
FROM num_dishes
  JOIN joined ON
num_dishes.rest_id =
joined.restaurant_id
  ORDER BY num_dishes DESC
LIMIT 1;`
```

Appendix 2 - All SQL Queries

1. Select a random restaurant

```
SELECT * FROM Restaurant r
ORDER BY RAND()
LIMIT 1;
```

2. Select a restaurant based off its id

```
SELECT restaurant_name
FROM Restaurant r
WHERE r.restaurant_id =
"${inputId}";
```

3. Select a restaurant based off its name

```
SELECT *
FROM Restaurant r
WHERE r.restaurant_name LIKE
"%${inputName}%";
```

4. Get all the dishes a given restaurant serves

```
SELECT *
FROM FoodItem f
WHERE f.rest_id =
"${inputId}";
```

5. Get all of the food pictures

```
SELECT * FROM FoodPictures;
```

6. Get all of the tf-idf scores matching dishes to picture

```
SELECT * FROM foodword_tf;
```

7. Get the average price of a given dish by its category

```
SELECT name, AVG(price) as
price, "${cat}" as category
FROM
average_price_by_name_cuisine
WHERE name = "${foodName}"
GROUP BY name;
```

8. Get all restaurants either offering a certain cuisine or serving a certain dish

```
SELECT distinct
restaurant_name,
restaurant_website,
price_range, price_range_num,
restaurant_id, formatted,
lat, lon, borough
FROM rest_cuisine_food r
WHERE r.cuisine =
"${inputCuisine}" OR r.name =
"${inputCuisine};
```

9. Get all restaurants within a one mile radius from a given location

```
SELECT r.*,( 3959 * acos(
cos( radians(${inputLat}) ) *
cos( radians( lat ) ) * cos(
radians(lon) -
radians(${inputLong})) +
sin(radians(${inputLat})) *
sin( radians(lat)))) AS
distance
FROM Restaurant r
HAVING distance < 1
ORDER BY distance;
```

10. Get all restaurants in a given borough

```
SELECT *
FROM Restaurant r
WHERE r.borough =
"${inputBorough}";
```

11. Get all restaurants of a given price range

```
SELECT *
FROM Restaurant r
WHERE r.price_range =
"${inputPrice}";
```

12. Get all restaurants of a given price range in a given borough

```
SELECT *
FROM Restaurant r
WHERE r.price_range =
"${inputPrice}" AND r.borough
= "${inputBorough}";
```

13. Get all restaurants within a one mile radius from a given location in a given borough

```
SELECT r.*,( 3959 * acos(
cos( radians(${inputLat}) ) *
cos( radians( lat ) ) * cos(
radians(lon) -
radians(${inputLong})) +
sin(radians(${inputLat})) *
sin( radians(lat)))) AS
distance
FROM Restaurant r
WHERE borough =
"${inputBorough}"
HAVING distance < 1
ORDER BY distance;
```

14. Get all restaurants within a one mile radius from a given location of a given price range

```
SELECT r.*,( 3959 * acos(
cos( radians(${inputLat}) ) *
cos( radians( lat ) ) * cos(
radians(lon) -
radians(${inputLong})) +
sin(radians(${inputLat})) *
sin( radians(lat)))) AS
distance
FROM Restaurant r
WHERE price_range =
"${inputPrice}"
HAVING distance < 1
ORDER BY distance;
```

15. Get all restaurants of a given cuisine and price range

```
SELECT *
FROM Cuisine c JOIN
Restaurant r ON
r.restaurant_id =
c.restaurant_id
WHERE c.cuisine =
"${inputCuisine}" AND
r.price_range =
"${inputPrice}"
```

16. Get all restaurants of a given cuisine in a given borough

```
SELECT *
FROM Cuisine c JOIN
Restaurant r ON
r.restaurant_id =
c.restaurant_id
WHERE c.cuisine =
"${inputCuisine}" AND
r.borough = "${inputBorough}"
```

17. Get all restaurants within a one mile radius from a given location offering a given cuisine

```
SELECT r.*,( 3959 * acos(
cos( radians(${inputLat}) ) *
cos( radians( lat ) ) * cos(
radians(lon) -
radians(${inputLong})) +
sin(radians(${inputLat})) *
sin( radians(lat)))) AS
distance
FROM Cuisine c JOIN
Restaurant r ON
r.restaurant_id =
c.restaurant_id
WHERE cuisine =
"${inputCuisine}"
HAVING distance < 1
```

```
ORDER BY distance;
```

18. Get all restaurants within a one mile radius from a given location offering a given cuisine and in a given borough

```
SELECT r.*,( 3959 * acos(
cos( radians(${inputLat}) ) *
cos( radians( lat ) ) * cos(
radians(lon) -
radians(${inputLong})) +
sin(radians(${inputLat})) *
sin( radians(lat)))) AS
distance
FROM Cuisine c JOIN
Restaurant r ON
r.restaurant_id =
c.restaurant_id
WHERE cuisine =
"${inputCuisine}" AND
r.borough = "${inputBorough}"
HAVING distance < 1
ORDER BY distance;
```

19. Get all restaurants within a one mile radius from a given location of a given cuisine and price range

```
SELECT r.*,( 3959 * acos(
cos( radians(${inputLat}) ) *
cos( radians( lat ) ) * cos(
radians(lon) -
radians(${inputLong})) +
sin(radians(${inputLat})) *
sin( radians(lat)))) AS
distance
FROM Cuisine c JOIN
Restaurant r ON
r.restaurant_id =
c.restaurant_id
WHERE cuisine =
"${inputCuisine}" AND
r.price_range =
"${inputPrice}"
HAVING distance < 1
ORDER BY distance;
```

20. Get all restaurants of a given cuisine and price range within a given borough

```
SELECT *
FROM Cuisine c JOIN
Restaurant r ON
r.restaurant_id =
c.restaurant_id WHERE
c.cuisine = "${inputCuisine}"
AND r.borough =
"${inputBorough}" AND
r.price_range =
"${inputPrice}"
```

21. Get all restaurants within a one mile radius from a given location of a given price range in a given borough

```
SELECT r.*,( 3959 * acos(
cos( radians(${inputLat}) ) *
cos( radians( lat ) ) * cos(
radians(lon) -
radians(${inputLong})) +
sin(radians(${inputLat})) *
sin( radians(lat)))) AS
distance
FROM Restaurant r
WHERE r.price_range =
"${inputPrice}" AND borough =
"${inputBorough}"
HAVING distance < 1
ORDER BY distance;
```

22. Get all restaurants within a one mile radius from a given location of a given cuisine and price range in a given borough

```
SELECT r.*,( 3959 * acos(
cos( radians(${inputLat}) ) *
```

```
cos( radians( lat ) ) * cos(
radians(lon) -
radians(${inputLong})) +
sin(radians(${inputLat})) *
sin( radians(lat)))) AS
distance
FROM Cuisine c JOIN
Restaurant r ON
r.restaurant_id =
c.restaurant_id
WHERE c.cuisine =
"${inputCuisine}" AND
r.price_range =
"${inputPrice}" AND borough =
"${inputBorough}"
HAVING distance < 1
ORDER BY distance;
```

23. Get the restaurant of a given price range and cuisine that has a website in a given borough that offers the most dishes

```
WITH rests AS (
  SELECT *
  FROM Restaurant
  WHERE price_range<=
'${inputPrice}' AND borough
LIKE '%${inputBorough}%' AND
LENGTH(restaurant_website) >0
),
  cuisines AS(
  SELECT * FROM Cuisine
  WHERE Cuisine LIKE
'%${inputCuisine}%'
  ),
  joined AS (
  SELECT rests.*,
cuisines.cuisine
  FROM rests
  JOIN cuisines ON
rests.restaurant_id =
cuisines.restaurant_id
  ),
  rest_ids AS (
  SELECT restaurant_id FROM
joined),
  num_dishes AS (
  SELECT COUNT(name) AS
num_dishes, FoodItem.rest_id
  FROM FoodItem
  JOIN rest_ids ON
FoodItem.rest_id =
rest_ids.restaurant_id
  GROUP BY FoodItem.rest_id
  )
  SELECT joined.*, num_dishes
FROM num_dishes
  JOIN joined ON
num_dishes.rest_id =
joined.restaurant_id
  ORDER BY num_dishes DESC
LIMIT 1;`
```

24. Get the restaurant with the lowest average price compared to restaurants with the same name i.e. best value chain

```
SELECT * FROM cheapest_chain
WHERE restaurant_name LIKE
"%${inputRest}%";
```

Appendix 3 - SQL Queries for Generating Cached Results

1. Cached result generated for query #7

```
INSERT INTO
average_price_by_name_cuisine
WITH choose_food_cuisine
AS(SELECT cuisine, name, price
FROM FoodItem F
JOIN Cuisine C ON F.rest_id =
C.restaurant_id
JOIN Restaurant R ON
R.restaurant_id = F.rest_id
WHERE price <> 0)
SELECT cuisine, name,
AVG(price) as average
FROM choose_food_cuisine
GROUP by cuisine, name;
```

2. Cached result generated for query #24

```
INSERT INTO cheapest_chain
WITH avg_prices_rest AS (
     SELECT restaurant_id,
restaurant_name, AVG(f1.price)
as avg_price
     FROM Restaurant r1
     JOIN FoodItem f1 ON
r1.restaurant_id = f1.rest_id
    GROUP BY r1.restaurant_id,
restaurant_name
    HAVING avg_price != 0)
SELECT restaurant_id,
restaurant_name
FROM avg_prices_rest av
WHERE avg_price <= ALL(SELECT
avg_price
FROM avg_prices_rest
WHERE av.restaurant_name =
restaurant_name);
```

3. Cached result generated for query #8

```
INSERT INTO rest_cuisine_food
SELECT r.*, cuisine, name,
description
FROM Restaurant r
JOIN Cuisine c ON
r.restaurant_id =
c.restaurant_id
JOIN FoodItem f ON
r.restaurant_id = f.rest_id;
```

Appendix 4 - Code Repository

https://github.com/arstern/CIS550_FinalProject