foodieCat

Connect with Best Food and Friends

Group #42

Presented by Yiwen Tang, Chenyuan Li, Yi-Nung Huang and Jialin Lou

Table of Content

Table of Content **2**

Introduction **3**

Datasets **3**

Database **4**

Technology **5**

Description of System Architecture **6**

Queries **7**

Performance evaluation **10**

Technical Challenges **11**

Extra Credit **12**

Appendix **13**

I. Introduction

In recent years, people's quality of life is gradually increasing and many of them are setting a higher standard in their lives in the pursuit of satisfaction. Instead of simply filling the stomach, many are willing to devote time to find great restaurants and make every meal an enjoyable experience. Therefore, many crowdsourced review apps emerged, such as Yelp and TripAdvisor.

Our Web Application has a similar purpose. We aim to provide multiple services that connect the users to great food in the United States, including restaurant recommendations, trip planning, and friend suggestions. For instance, our website provides guidance for the users to select restaurants of their interest based on their current physical location. Furthermore, a user can learn the quality of a certain restaurant by viewing its most useful comments and reviews. Besides the functionality contained in most of the crowdsourced review apps, our website will also introduce the idea of social media by connecting the users and recommending restaurants based on their mutual preferences and locations.

II. Datasets

1. Data source

Our datasets are provided by Yelp and offered by Kaggle. We selected 2 datasets that contain information about Yelp's businesses and the reviews on businesses *(Appendix B for web links)*. In a nutshell, our datasets include 5,200,000 user reviews and information on 174,000 businesses from 11 metropolitan areas. More details are introduced in Section 2.2.

2. Summary statistics

For the **Business** dataset, the size of the original dataset is 30.29 MB. It includes 174,567 rows and 13 attributes. The attributes are business_id, name, address, city, state, latitude, longitude, stars, categories, review_count, neighborhood, postal_code and is_open. The businesses in the dataset span 654 different cities. 128,302 of the businesses are located in the US. Among businesses located in the US, 108,332 businesses are still open. The average rating of these businesses is approximately 3.67 stars and each business has about 35 reviews.

For the **Review** dataset, the size of the original dataset is 3.53 GB. It contains 5,261,668 rows and 9 attributes. The attributes are review_id, business_id, user_id, stars, review_date, text, useful, funny and cool. Among all the reviews, only 2.5 million of the reviews are voted useful at least once. By calculating the mean, the average rating each review has is about 3.7 stars and each review has about 113 characters in average.

3. *Preprocessing*

For the **Business** data, we firstly removed unused columns such as "neighborhood" and "postal_code". Since our app is for users in the US, we keep the businesses only if they are located in the US and are still open. We also removed double quotes on name and address for better interface. Here is the example instance after processing. We successfully reduced data to 108,332 businesses with 10 attributes.

For the **Review** dataset, we removed unused columns such as "funny", "cool" and "user_id" since we construct our own user data. We also only kept the reviews on open business. Furthermore, we filtered reviews that are not very useful, in our case, reviews are not voted useful 5 times or more. To save storage, we removed reviews that are too long and got rid of the urls in the review text. By doing this, we reduced our review data by 95% and left with 264,479 reviews.

4. *Usage of the datasets*

One of our goals is to recommend restaurants based on the user's location. To achieve this, we need two pairs of coordinates, from a business and a user, to compute a distance. The **Business** dataset contains the geographical information of the restaurants and many valuable indicators of the success of them, such as star ratings. We also extract categories information of the businesses so that our App can return businesses that are restaurants and belong to the user preferred categories. The **Review** dataset is used as a further reference when our users want to know a bit more about a certain restaurant. We will return the reviews that most people labeled as "useful".

III. Database

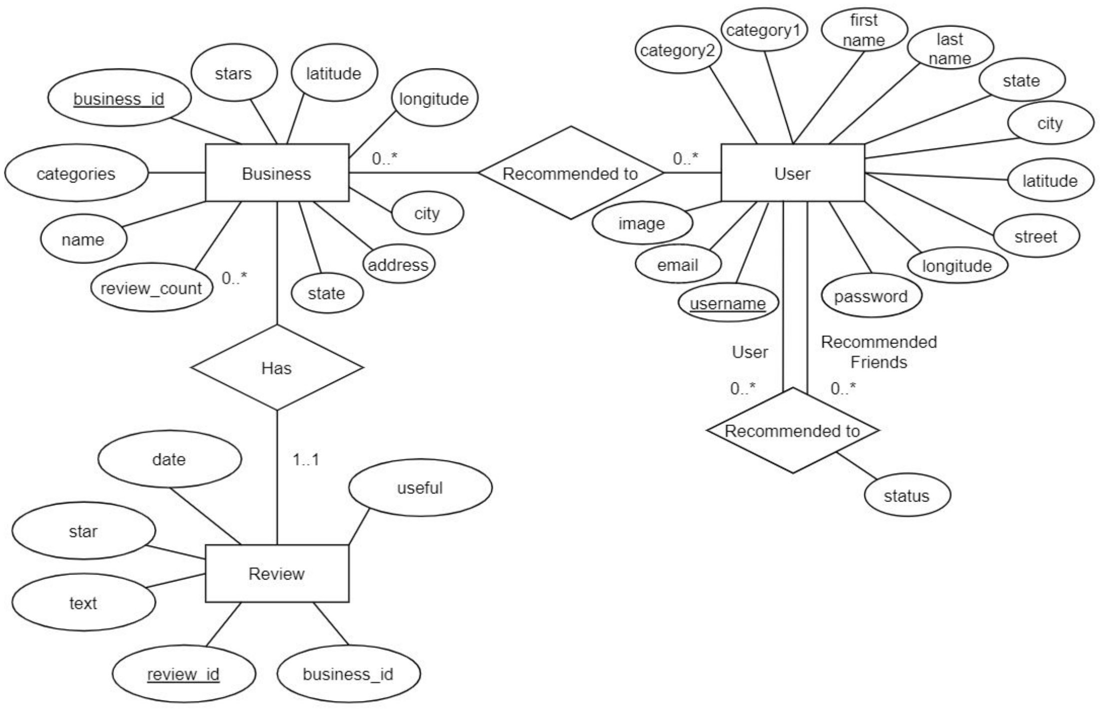
1. *MySQL*

For ingesting review and business data, we first used SQL DDL to generate two tables for our database (refer to Appendix B), which are business and reviews. Then we use the command load data infile to load the csv files into our tables. The ingestion process for business data is about 2-3 seconds and ingestion process for review data is about 5-6 seconds.

2. *MongoDB*

To store and manage the data in MongoDB, we used the package mongoose and created a User Schema with it. Since we have a small **User** dataset, retrieving information from MongoDB only takes less than 1s. We note the importance of privacy, and therefore we use bcrypt to hash the password before sending it to MongoDB.

3. *ER Diagram*



4. *Relational Schema*

Business(<u>business_id</u>, name, address, city, state, latitude, longitude, stars, categories, review_count)

Reviews(<u>review_id</u>, business_id, stars, review_date, text, useful) business_id refer to Business(business_id)

Users(<u>username</u>, email, firstName, lastName, password, street, city, state, imagte, category1, category2, longitude, latitude, status)

All of our schemas are in BCNF, since it is true for all three schemes, every functional dependency that holds over each scheme X->A, either A∈X (it is trivial), or X is the superkey of the scheme.

IV. Technology

Framework: React, NodeJS

Database:

AWS MySQL store/manipulate reviews and business data

MongoDB Atlas to store/manipulate user data

User Interface: UIKit

Log-In: Passport

Privacy: Bcrypt

APIs: Google Map Geocoding API

We built our frontend with the React framework and backend with Node.js. Our css styles mainly come from UIkit. Passport is the tool we implemented to validate log-ins. The package Bcrypt helps us hashed the password of users. Last but not least, the Google Map Geocoding API returns a coordinates for us when we feed it an address.

V. Description of System Architecture

1. *Login page*

   Users can login with an email and password if they have already registered.

2. *Registration page*

   To create an account, a user needs to fill in the first name, last name, email, username and password. A user can choose to upload a profile picture. For the security purpose, we also encrypted the password by storing a hashed version of it. We also validate the email format by checking if '@' or dot is contained in the email field.

3. *Homepage*

   Users may select a city and food category with buttons. The results page will return restaurants matching the criteria ordered by average star rating and review count.

4. *Profile page*

   The profile page displays the user's username, first name, last name, address, favorite food categories, coordinates, and account privacy status.

5. *Edit Profile page*

   The edit profile page allows users to change their address, favorite food categories, and account privacy settings. By clicking "YES" for the account privacy, the user agrees to display his/her information by other users for restaurant recommendation and connection.

6. *Friends page*

   Users who have chosen their privacy settings to be public can be viewed on this page. Their address, favorite food categories, and distance from the current user are shown. By clicking "Recommend for Us", the webapp uses both users' geographical locations and returns restaurants that match the first food category of each user and shows the distance of that restaurant.

7. *Recommend Restaurant page*

   Users can retrieve a list of recommended restaurants based on their locations by selecting a minimum star rating and choosing among one of the user's favorite categories. If the users' locations or their favorite categories are unset, the website will redirect users to set their profiles.

8. *Restaurant Search page*

   Users can get a list of recommended restaurants by selecting two categories and a minimum star rating. The difference between this page and *Recommend Restaurant page* is that this page will use input categories instead of using categories in users' profiles. If the users' locations are unset, the website will redirect users to set their locations.

9. *Local Restaurant Discover page*
   a. Users can select a city and category to view the top rated local restaurants in this category and their most useful 5-star review
   b. Users may also just select a city and explore the top rated local restaurants' information including average stars, review count, and address

VI. Queries

1. *Homepage:* returns top rated restaurants in category X ordered by stars and review counts (name, stars, review_count)

```
SELECT name, stars, review_count
FROM Business
WHERE city = '${city}' AND categories LIKE '%Restaurants%' AND
categories LIKE '${category}'
ORDER BY stars DESC, review_count DESC
LIMIT 10;
```

2. *Find restaurant page:* returns restaurant's information and top 5 most useful reviews that matches user-selected categories and ordered by distance and average stars (name, address, city, state, avg_stars, text, distance)

```
WITH distance AS (
SELECT business_id, name, address, city, state, stars, categories,
ROUND((6371*     acos(cos(radians(latitude))     *     cos(
radians(${latitude})) * cos(radians(${longitude}) -
radians(longitude)) + sin(radians(latitude))
* sin(radians(${latitude})))),2) AS distance
FROM Business
WHERE stars >= ${stars} AND categories LIKE
'%Restaurants%' AND categories LIKE '${category1}' AND categories
LIKE '${category2}'
ORDER BY distance, stars DESC
LIMIT 10),
rnk AS (
SELECT business_id, text, stars, useful, RANK() OVER
(PARTITION BY
business_id ORDER BY useful) AS rnk, review_date
FROM Reviews
WHERE business_id IN(SELECT business_id FROM
distance))
SELECT name, address, city, state, d.stars AS
avg_stars, text, distance,
d.business_id, r.review_date, r.useful
FROM distance d
JOIN rnk r
ON d.business_id = r.business_id
WHERE rnk < 6
ORDER BY distance, d.stars DESC
```

3. *Friends page:* recommend restaurants based on two people's preference categories with minimum average rating of 3 and ordered by distance (name, stars, address, city, state, distance)

```
SELECT name, stars, address, city, state,
ROUND((6371 * acos(cos(radians(latitude))
* cos(radians(${latitude}))
* cos(radians(${longitude}) - radians(longitude)) +
sin(radians(latitude))* sin(radians(${latitude})))),2)
AS distance
FROM Business
WHERE categories LIKE '%Restaurants%' AND (categories
LIKE '${category1}' OR categories LIKE '${category2}') AND stars >=
3
ORDER BY distance, stars DESC
LIMIT 10;
```

4. *Local restaurant page:* find the most useful five star review at local restaurants given a specific city and food category (name, avg_stars, useful, stars, text)

```
WITH chain AS (SELECT NAME, COUNT(*) as cnt, ROUND(AVG(Stars),2) AS
avg_stars
FROM Business
WHERE city = '${city}' AND categories LIKE '%Restaurants%' AND
categories LIKE '%${category}%'
GROUP BY name
HAVING COUNT(*) =1
ORDER BY avg_stars DESC, cnt DESC),
category_cte AS (
SELECT business_id, name
FROM Business
WHERE city = '${city}' AND categories LIKE
'%Restaurants%' AND categories LIKE '%${category}%'),
max_useful AS (
SELECT r.business_id, b.name, r.text, r.stars, r.useful
FROM Reviews r
JOIN category_cte b
ON r.business_id = b.business_id
JOIN (
SELECT business_id, stars, MAX(useful) as max_use
FROM Reviews
GROUP BY business_id, stars
HAVING MAX(useful) > 1
ORDER BY max_use DESC, stars DESC) mm
ON r.business_id = mm.business_id AND r.stars =
mm.stars AND r.useful = mm.max_use
WHERE r.stars = 1 or r.stars = 5
ORDER BY business_id),
final AS (
SELECT c.name, c.avg_stars, m.useful, m.stars, m.text,
ROW_NUMBER() OVER (PARTITION BY c.name ORDER BY m.useful DESC) rn
FROM chain c
JOIN max_useful m
ON c.name = m.name
ORDER BY c.avg_stars DESC, c.name ASC, m.stars DESC
LIMIT 20)
SELECT name, avg_stars, useful, stars, text
FROM final
WHERE rn = 1
```

5. *Local restaurant page (2):* returns local restaurants' information that are in a specific food category

```
SELECT name, address, stars as avg_stars, categories, review_count
FROM Business
WHERE city = '${city}' AND categories LIKE '%Restaurants%'
GROUP BY name
HAVING COUNT(business_id) = 1
ORDER BY avg_stars DESC, review_count DESC
LIMIT 10;
```

These queries are written in Node.js as routes. They take input parameters from requests sent by the frontend. After these queries are sent to the MySQL host, the results are returned in the form of json and sent to the client side via APIs created by ExpressJS. The client side simply needs to fetch these results by calling these APIs.

Query 4 is the most complex query we used in this project. It contains four CTEs, uses subqueries, multiple joins, and window functions to achieve its goal.

VII. Performance evaluation

Two queries were optimized for performance in this project.

1. In query #2, our task involves joining the Business and Reviews dataset. The original relational algebra is the following:

$$\Pi_{name,\ address,\ city,\ state,\ avg\_stars,\ text,\ distance}$$
$$\sigma_{Business.stars\ >=\ \$\{stars\}\ AND\ .categories\ LIKE\ "\%\$\{category1\}\%"\ AND\ categories\ LIKE\ "\%\$\{category2\}\%"\ AND\ categories\ LIKE\ "\%Restaurants\%"}$$
$$(Business \bowtie_{Business.business\ id\ =\ Reviews.business\ id} Reviews)$$

This query will take more than 19 seconds to run. By pushing the selections into the base queries of Business and Reviews before they join, the query speed was improved to 0.2 seconds. The reasoning behind the massive improvement is that, instead of joining two datasets on all the rows (which will contain over 200,000 rows), we now only need to join 10 rows from Business with their corresponding rows in Reviews. The updated relation algebra is the following:

$$\Pi_{name,\ address,\ city,\ state,\ avg\_stars,\ text,\ distance}(\Pi_{business\ id,\ name,\ address,\ city,\ state,\ avg\_stars,\ distance}$$
$$\sigma_{Business.stars\ >=\ \$\{stars\}\ AND\ .categories\ LIKE\ "\%\$\{category1\}\%"\ AND\ categories\ LIKE\ "\%\$\{category2\}\%"\ AND\ categories\ LIKE\ "\%Restaurants\%"}$$
$$Business) \bowtie_{Business.business\ id\ =\ Reviews.business\ id}(\Pi_{business\ id,\ text,\ stars,\ useful}$$
$$\sigma_{Business.stars\ >=\ \$\{stars\}\ AND\ .categories\ LIKE\ "\%\$\{category1\}\%"\ AND\ categories\ LIKE\ "\%\$\{category2\}\%"\ AND\ categories\ LIKE\ "\%Restaurants\%"}$$
$$Reviews)$$

2. In query #4, our task involves grouping restaurants by name to find their counts. Then we need to join this dataset back to the Business dataset. The original relational algebra of the first two CTE is the following:

$$\Pi_{name,\ count,\ avg\ stars}$$
$$\sigma_{Business.count=1\ AND\ categories\ LIKE\ "\%\$\{category\}\%"\ AND\ categories\ LIKE\ "\%Restaurants\%"\ AND\ city\ =\ "\$\{city\}"}$$
$$(Business \bowtie_{Business.business\ id\ =\ Business.business\ id} Business)$$

Using this selection and projection relation, the eventual query would take over 6 seconds to run. To optimize this query, we created an index on the “city” column in Business and the “stars” column in Reviews. The indices would help because our query looks for 5-star reviews only in the Reviews dataset. In the meantime, we pushed the selection criteria in the query down to the Business dataset to return fewer rows in the before the join. The optimized query only takes 1 second on average to run. The updated relational algebra for the first two CTE is the following:

$$\Pi_{name,\ count,\ avg\ stars}$$
$$(\Pi_{name,\ count,\ avg\ stars}\sigma_{Business.count=1\ AND\ categories\ LIKE\ "\%\$\{category\}\%"\ AND\ categories\ LIKE\ "\%Restaurants\%"\ AND\ city\ =\ "\$\{city\}"}$$
$$Business) \bowtie_{Business.business\ id\ =\ Business.business\ id}$$
$$(\Pi_{business\ id,\ name}\sigma_{categories\ LIKE\ "\%\$\{category\}\%"\ AND\ categories\ LIKE\ "\%Restaurants\%"\ AND\ city\ =\ "\$\{city\}"}Business)$$

VIII. Technical Challenges

We have faced several technical challenges during the development of our web application. First, we had problems finding the best way to store the food category information of a restaurant. In the Business dataset, the category column contains all the food categories to which each restaurant belongs. For example, a burger place might have “Fast Food; Burgers; Restaurants” as the category value. We were thinking whether to separate the categories into different columns using one-hot encoding, so that it would be convenient to recommend users restaurants based on their preference in the food category. However, there were so many categories that it was not reasonable to create a separate column for each one of them. We therefore decided to first have the users set their preference in the food category, and use the SQL LIKE operator to check if the category column of each restaurant contains their preferred category. This way, we were able to recommend users restaurants according to their food preference without having a different column for each category.

While building the web application, we found the connection between the client and the databases rather challenging. At first, we used OracleDB to store the Business and Reviews data, but we had problems installing OracleDB clients. Since the operating systems and node versions on our computers were all different, the version of OracleDB client that worked on one computer did not work on the others. To solve the problem, we read the documentation and “Issues” section on Github, and finally were all able to install the correct version of OracleDB client for our computers. Yet, after we successfully connected to OracleDB, we decided to use MySQL instead because we found that MySQL in fact worked more efficiently for our rather small-scaled deployment. We therefore had to modify our queries so that they matched the MySQL syntax.

Finally, we encountered a problem regarding the display of reviews on our web application. In some of our pages where we included the restaurant reviews in the search result, we originally showed the full text to the user. However, some reviews were so long that it made the entire page too long and it became inconvenient for users to have to scroll down quite a few times to view all of the results. To solve this problem, we added a toggle button for each review text, so that the page only displays the first few lines of a review by default, but users can choose whether they want to “view more” of a review.

IX. Extra Credit

1. Using NoSQL(MongoDB) to store user information
2. Integrating with Google Map Geocoding API to get longitude and latitude from user addresses
3. User login experience with authentication using Passport
4. Password encryption with Bcrypt API; profile privacy status for users.

Appendix

Appendix A - *Acknowledgement of resources*

1. The icon is kindly offered by Iconka.com for non-commercial use. The license URL is http://www.iconarchive.com/icons/iconka/saint-whiskers/meow-me.txt
2. UIkit for better interface design: https://getuikit.com/
3. Calculating distance function adopted from: https://www.movable-type.co.uk/scripts/latlong.html
4. Calculating midpoint function adopted from: https://stackoverflow.com/questions/4656802/midpoint-between-two-latitude-and-longitude
5. Google map api for getting latitude and longitude: https://developers.google.com/maps/documentation/geocoding/start

Appendix B - *Data repositories*

1. The business and reviews dataset we used for are from: https://www.kaggle.com/yelp-dataset/yelp-dataset/version/6
2. SQL DDL to create tables

```
CREATE TABLE Business (
      business_id VARCHAR(255),
      name VARCHAR(255),
      address VARCHAR(255),
      city VARCHAR(50),
      state VARCHAR(10),
      latitude FLOAT,
      longitude FLOAT,
      categories VARCHAR(1000),
      stars FLOAT,
      review_count INT,
      PRIMARY KEY (business_id)
);
CREATE TABLE Reviews (
      review_id VARCHAR(255),
      business_id VARCHAR(255),
      stars FLOAT,
      review_date DATE,
      text  VARCHAR(4000),
      useful INT,
      username VARCHAR(255),
      PRIMARY KEY (review_id),
      FOREIGN KEY (business_id) REFERENCES Business (business_id)
);
```