Project 1 - Restaurant Finder

Akshay Kr Pansari, 2015ME20713
Pulkit Srivastava, 2015CH10122
Vishesh Goyal, 2015CH10145

Due date: February 26, 2019, 11:55pm IST

1 Project Description

We aim to make a restaurant finder website using Python Flask, and using Postgresql server. There are many apps available where we can order food from but we are trying to find a website where we will be able to search for restaurants in any part of the world and filter according to various attributes. There is a button for the users to register and login as well if they wish to look into their previous searches and view previous table bookings. User can also update their information if they wish to.
Moreover there is a search feature based on city, ratings of restaurant, and it can also be sorted based on price, restaurant name and rating, etc.
If user want to book a table, he can also do that. He can also specify the time and the number of people for whom he has booked the table.
After booking the table, he can give the rating to the restaurant which is stored in the database and the rating of the restaurant is also updated.

1.1 ER Diagram

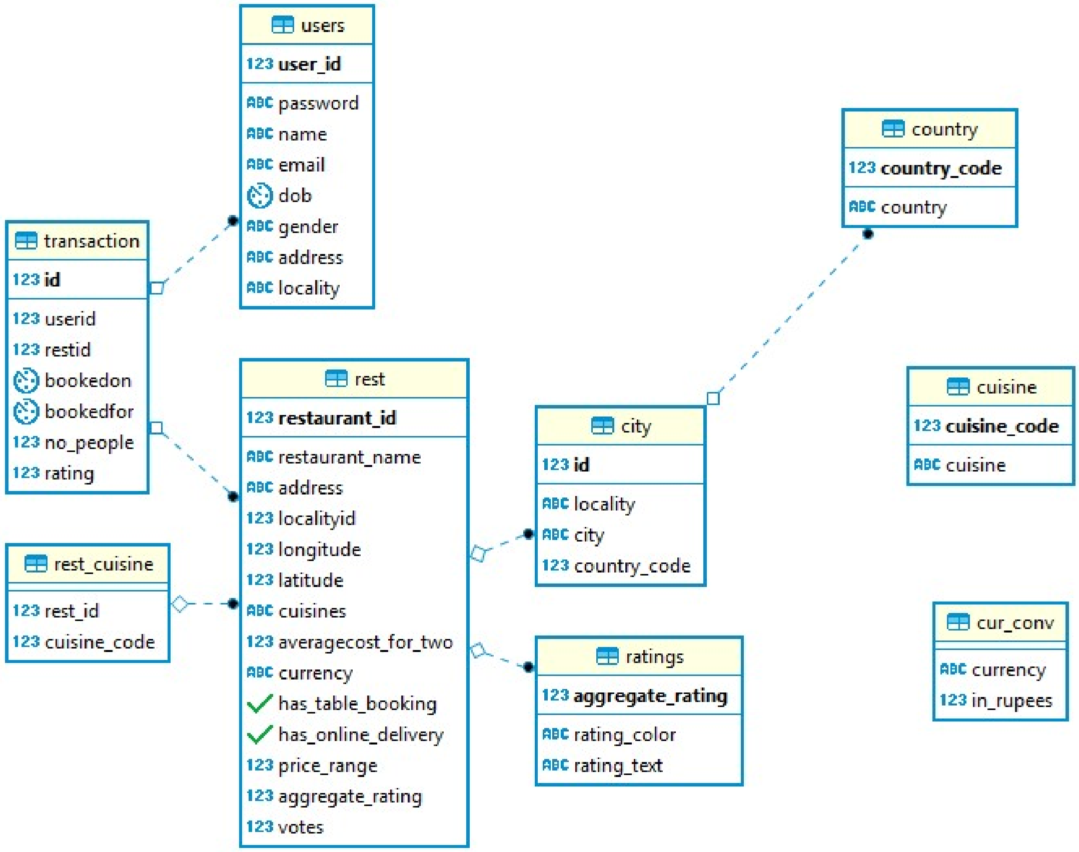

Figure 1: ER Diagram

2 Data Sources and Statistics

The data source for our project was taken from
https://github.com/MehtaShruti/Zomato-Restaurants-Recommendations/tree/master/csv
where the file could be found under zomato.csv which was cleaned to remove tuples with characters not encodable in UTF-8 (8 tuples) and normalized using SQL commands and python dictionary to insert data into multiple tables from the master table. From the master table, relation locality was formed by taking out columns: locality, city and country code. A Id for locality was generated for relation locality and used in the table storing restaurant information. A relation cuisine was made which stores the cuisines available in all restaurants and a code given to it using python dictionary. This table is used to relate restaurants to the cuisines they serve in the relation rest_cuisine which is used to search a restaurant based on the cuisines available. We also made a new table for user registrations and table bookings the data for which was manually entered by us. The table cur_conv was created to convert the currency into another. The data for this was taken from Google's currency

convertor as per data on 25th Feb. 2019.

Table	Attribute
rest	Restaurant_Id, Restaurant_Name, address, localityid, longitude, latitude, cuisines, averagecost_for_two, currency, has_table_booking, has_online_delivery, price_range, aggregate_rating, votes
users	user_id, password, name, email, dob, gender, address, locality
rest-cuisine	rest_id, cuisine_code
transaction	User_id, Restid, bookedon(datetime), bookedfor(datetime), no_people, rating
cuisine	cuisine_name, cuisine_code
city	id, locality, city, country_code
country	country_code, country
ratings	aggregate_rating, rating_color, rating_text
cur_conv	currency, in_rupees

2.1 Statistics

Table	No. of tuples	Time to load	Raw dataset size	Size After Clean-up
temp	9531	249.700 ms	2205 kB	2199 kB
rest	9531	641.817 ms	-	1944 kB
users	10	(user input)	-	(variable)
rest-cuisine	19686	408.047 ms	-	704 kB
transaction	6	(user input)	-	(variable)
cuisine	143	3.421 ms	-	8192 Bytes
city	1252	150.877 ms	-	88 kB
country	15	109.983 ms	-	(too small)
ratings	50	48.200 ms	-	1944 kB
cur_conv	12	(manual input)	-	(too small)

3 Functionality and Working

1. User view of the System

   (a) **Home**: In the home page, there is a option to search for the restaurant based on the city one would like to search for. On the top left corner of the website there are options to do detailed search for the restaurant and one about the contact page. On the right hand side, there is a Previous Order, Login and Registration page. You can go to the home location by clicking on the name of the website.

   (b) **Find Restaurant** : We can reach this page by either searching on the home page or by directly entering by using Find Restaurant. It helps users to search to different types of filters which is required to search for a restaurant. The various types of search queries are based on city, restaurant name, locality, rating, cuisine, etc and are sort-able by rating, price and restaurant name. Clicking on the name of a restaurant gives more details about it.

   (c) **Book Table**: After searching for a restaurant, the user can click on it to see it more detail. From this page he/she can also book a table giving the details like no. of people and the date to be booked for. All bookings are stored along with the time, user and restaurant booked for.

   (d) **Login** : In the login page, the user can login using email-id and password. The password has been hashed in the database so that it will be protected.

   (e) **Register**: Just like any other app, we have added the functionality to register for an user. In this the column that needs to be added are: Name, Email, Password, Date-of-Birth, Gender, Address, Locality.

   (f) **Update**: If someone by mistake has put his/her information wrong then he/she can update the information through the website only.

   (g) **Previous Bookings**: After a user is logged in he/she can see the previous bookings he/she has done from the 'Bookings' tab in the navigation bar. The user can also **rate** the visit from here.

   (h) The **About** the section page show information about the author of the page.

2. Special Functionality

   (a) **Constraints**: Apart from primary keys and foreign keys in the tables we have the following constraints:

      i. Table Booking Constraint: A table can't be booked before the current date and we can't book a table if booked for after 7 days.

      ii. Uniqueness Constraints: Each user has to have a unique email-id through which he/she has registered.

      iii. Constraint on Gender: The field can accept only 'M' or 'F' as a value.

   (b) **Sequences** : To give unique id to the new user and transaction, we used the command of sequences (SERIAL). This helped to automatically give id to a new user in a sequential manner. It was also used to give the localities an id.

   (c) **Trigger** : We have implemented a trigger in our database which updates the rating of a restaurant and increments the votes it has received whenever an user gives a rating to the restaurant and also recalculates the rating.

```
CREATE TRIGGER trig_updateratings AFTER UPDATE of rating on Transaction FOR
EACH ROW EXECUTE PROCEDURE update_ratings();
CREATE OR REPLACE FUNCTION update_ratings() RETURNS trigger as
$update_ratings$
DECLARE
    temprate FLOAT;
tempvote INT;
BEGIN
    select aggregate_rating, votes into temprate, tempvote from rest where
    rest.restaurant_id = new.restid;
    UPDATE rest SET aggregate_rating = ROUND((((temprate*tempvote)+new.rating)/
    (tempvote+1))::numeric,1) where rest.restaurant_id = new.restid;
```

```
        UPDATE rest SET votes = votes+1 where rest.restaurant_id = new.restid;
        RETURN NULL;
    END;
    $update_ratings$
    language plpgsql;
```

(d) **Indexes** : We didn't have to put indices on the primary key because we found that it was already present in the keys and the attributes we wanted to put it in.

3. SQL Queries for different parts along with timings at the end.

(a) **Login**
Select email
FROM users
where email = email-id;

0.253ms

(b) **Register**
INSERT into users (password, name, email, dob, gender, address, locality)
values (hashedPassword,name, email, date, gender, address, locality))

0.756ms

(c) **USERLOGIN**
select password, userid
FROM users
where email=email;"

0.303 ms

(d) **Table-Booking**
SELECT restaurantname,address, cuisines, averagecostfortwo, currency, aggregaterating
FROM rest
WHERE restaurantid= str(ID);

0.541ms

(e) **Book Table**
INSERT into transaction (userid, restid, bookedon, bookedfor, nopeople)
values (userid, restid, bookedon, bookedfor, nopeople);

0.656ms

(f) **Search Restaurant**
SELECT restaurantname,address, cuisines, averagecostfortwo, currency, aggregaterating, restaurantid
FROM rest LIMIT 10;

0.486ms

(g) **Search Restaurant**

```
SELECT distinct restaurantname,address, cuisines, averagecostfortwo, currency, aggregaterating, restaurantid, ratings.ratingcolor
FROM rest, city, ratings
WHERE rest.localityid=city.id and lower(city.city)=lower(city) and ratings.aggregaterating = rest.aggregaterating '' and lower(rest.restaurantname) like lower(restaurantname) and lower(locality) like lower(locality) and rest.aggregaterating>= rating and
ORDER BY rest.aggregaterating desc;
```

5.745ms

(h) **Update user inforamtion**

```
UPDATE users
SET name= name, email= email, dob= dob, gender=gender, address= address,locality= locality
WHERE userid= userid;
```

0.432ms

(i) **Projection detail of User using userid**

```
SELECT name, email, dob, gender, address, locality
FROM users
WHERE userid=userid;
```

0.345ms

(j) **Booking a Table**

```
SELECT rest.restaurantname, rest.restaurantid, rest.address, city.locality, transaction.bookedon, transaction.bookedfor, transaction.nopeople, transaction.rating , transaction.id
from transaction, rest, city
WHERE userid=userid and rest.restaurantid= transaction.restid and city.id= rest.localityid
ORDER BY transaction.bookedon desc;
```

1.345ms

(k) **User Rating Update**

```
UPDATE transaction
SET rating= rating
WHERE id = id;
```

0.273ms

(l) **OPEN restaurant page**

```
SELECT restaurantname,address, cuisines, averagecostfortwo, currency, aggregaterating
FROM rest
WHERE restaurantid= restid;
```

0.453ms

4 References

https://github.com/miguelgrinberg/microblog
https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-ii-templates