Restaurant Databasing

Project Proposal

Team Foogle

Daniel Yan
Pratit Vithalani
Tarcisius Hartanto
Will Thomas

Table of Contents

Introduction

This document is an overview for the Foogle database created by and implemented by Team Foogle. The database is meant to help a user determine what restaurant they want to eat at with their specific requirements set. Moreover, the database allows users to review for specific restaurants. With these, the user will be able to find a restaurant they want to eat at easier.

Design Documentation

Entity-Relationship Model

The Entity-Relationship Model of the the database is shown in Figure 1. In this design, the database is laid out to show every main player within the system. The USER is the main client of this product; they will reply heavily on the features given to them to decide where their next meal will take place. The USER will be identified by their email address, which will be given when they register. Along with that, their name will be stored, which will be used when displaying reviews.

Since this database is holding information about food, there is an entity for RESTAURANTs. Each RESTAURANT has a unique integer ID that identifies it (as there can be multiple same-named restaurants in the same city). Along with the name of the RESTAURANT, the CITY, cuisine that is served, and street address is stored.

The RESTAURANT is an entity that can be reviewed by the USERS. These REVIEWs hold the USER that wrote the review and the RESTAURANT that it was intended for. Each REVIEW has an integer value, ranging from 0 to 5, depending on how the USER felt the experience went at the RESTAURANT. There is also a section where the USER can write about their experience.

Each USER and RESTAURANT are located in a CITY. Each CITY has its name and the name of the state that it resides in.

Every RESTAURANT has at least one MENU, which outlines what type of cuisine the RESTAURANT has to offer to its customers. Each MENU consists of its name and the language that each MENU is offered in. While each RESTAURANT has at least one MENU, they can have multiple MENUs depending on the season or time of day (i.e. lunch menu and dinner menu).

Each FOOD ENTRY has to be from a MENU. Every FOOD ENTRY contains the name of the dish that is served, which is unique, and the price of that dish.

Each RESTAURANT has to have at least one HOURS. Each HOURS consists of its unique name, the time that the duration starts, and the end time of the duration. This can be the hours of operation, happy hour timings, or other special hours that a RESTAURANT would like to have.

Both USER and RESTAURANTS have their own version of

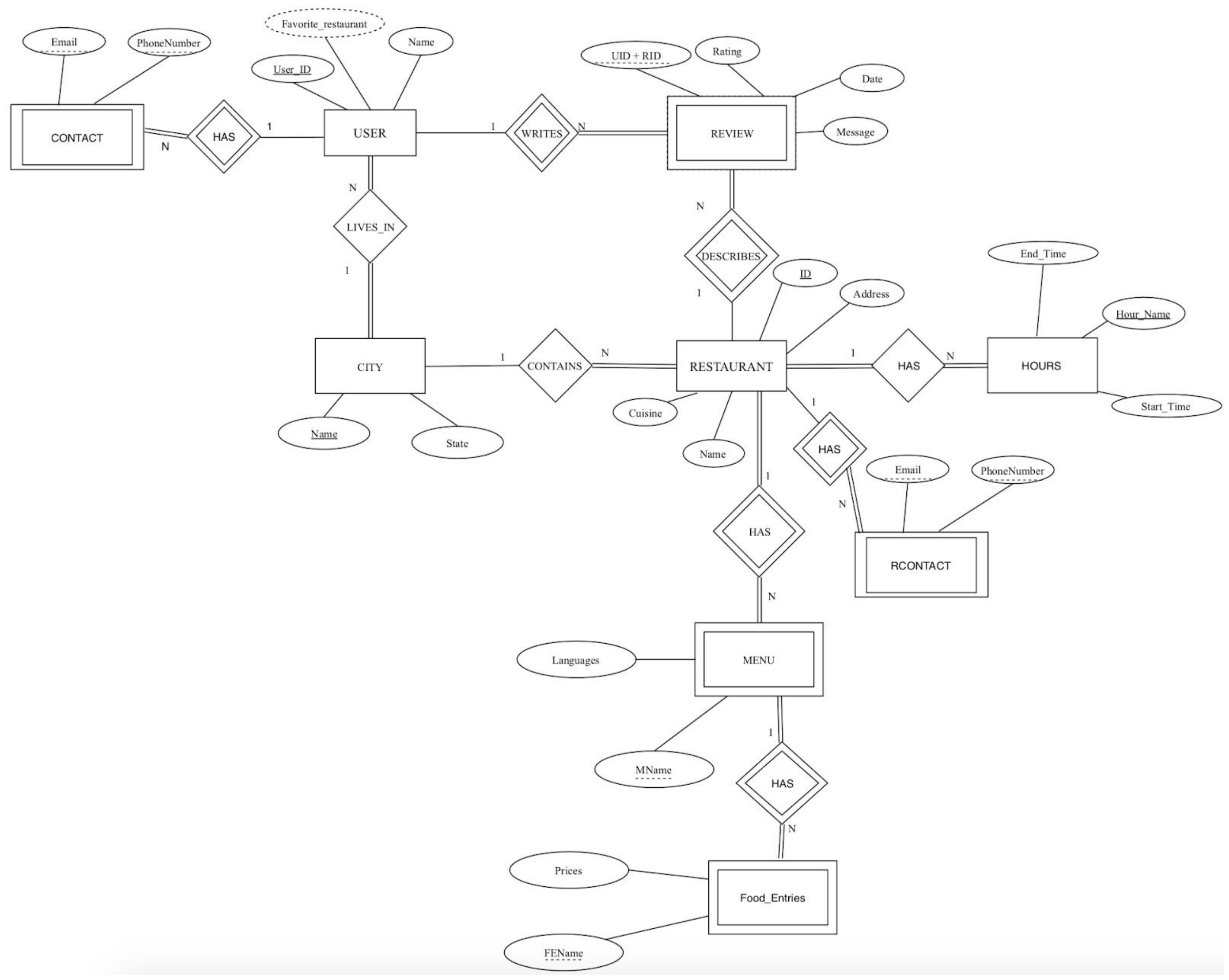


Figure 1: Entity-relationship model of Foogle database

As part of the design, there were ten assumptions that were made, as listed:

1. 1. A REVIEW must be written by a USER. A USER may write 0 or more REVIEWs.
2. 2. A RESTAURANT must contain at least 1 MENU, and likewise a MENU must be associated with a RESTAURANT.3. REVIEWs must have a RESTAURANT, while RESTAURANTS may have 0 or more REVIEWs.
3. 4. Multiple or no USERs may desire one or more RESTAURANTs.
4. 5. A MENU and REVIEWs must depend on RESTAURANT to be uniquely identifiable.
5. 6. REVIEWs additionally depends on USER for unique identification.
6. 7. A USER can have one or no Favorite_Restaurant.
7. 8. Every USER must live in a CITY, and every CITY must contain at least 1 USER or RESTAURANT.
8. 9. A CONTACT or RCONTACT can have an Email and/or a PhoneNumber.
9. 10. A RESTAURANT has at least one HOURS.

These assumptions better outline how each entity can be in any state and how they interact with one another.

Relational Model

In the Relational Model (RM), as shown in Figure 2, all entities and their attributes are listed as described in the Entity-Relationship Model.

The RM follows the same structure displayed in the ER. No new tables were added in addition to the ones shown in the ER.

Along with the map, Figure 2 also shows the primary and foreign keys for each table.

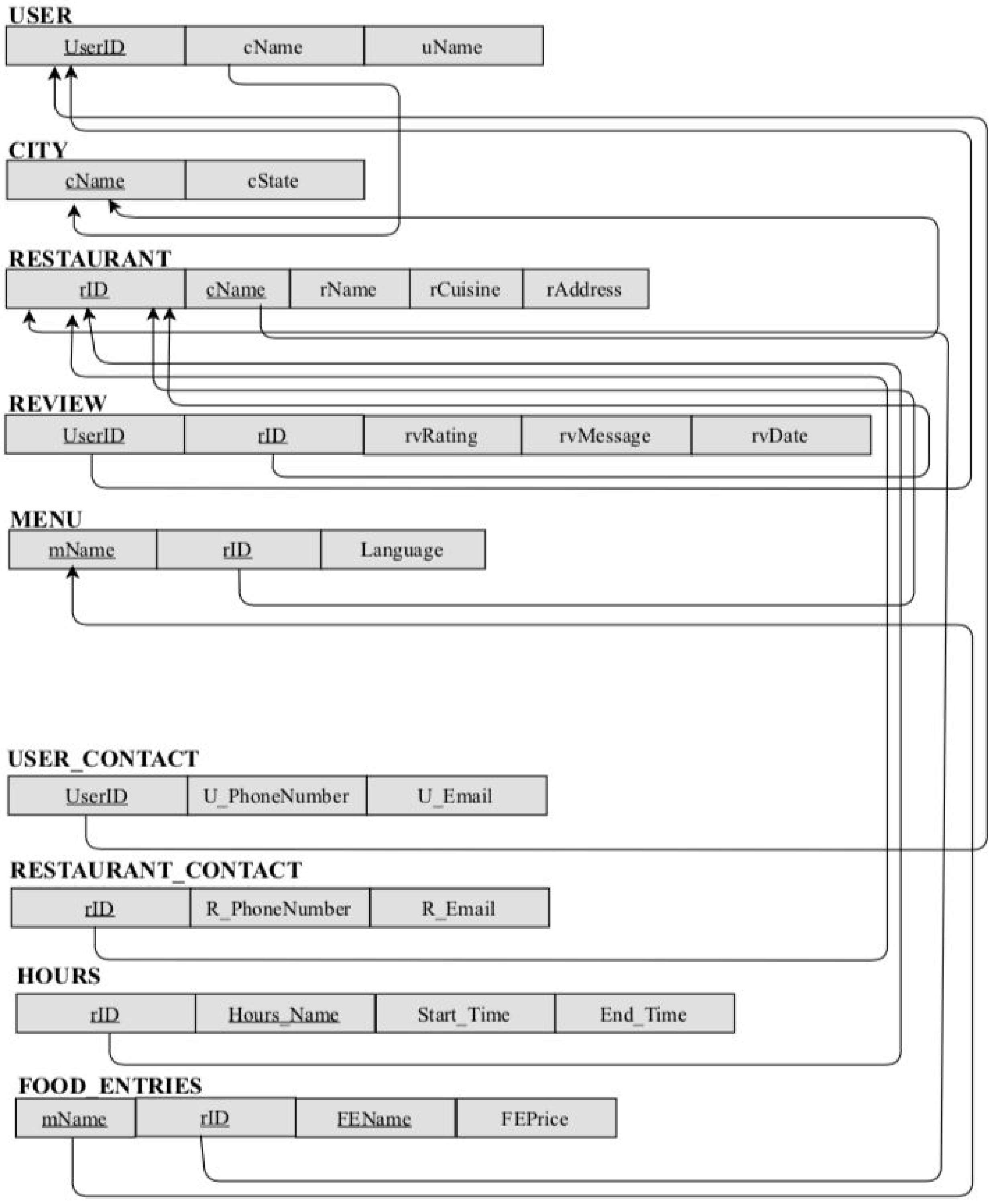


Figure 2: Relational Model of Foogle database

Domain Constraints:
1. All emails are alphanumeric strings up to 20 characters ending with @ domain
2. rAddress is a alphanumeric string up to 100 characters with the format consisting of: street address, state, and zip code. 3. uName, rName, cState, cName, rCuisine, FEName, and mName are alphabetic strings up to 50 characters long.
4. rID is a series of 4digit single integers from 09.
5. Start_Time and End_Time are type DATE.
6. rvRatings are single integers from 15 formatted in R/5 (where R = defined integer)
7. rMessage is an alphanumeric string up to 150 characters.
8. mLanguages is a string that must be an officially recognized language.
9. FEPrice is a double following US Pricing conventions ($dollars.cents).
10. Start_Time has to always be less than End_Time.

Referential Constraints:
1. Every USER must be from a CITY.
2. Every RESTAURANT must be in a CITY.
3. Every REVIEW must have a USER (who writes it) and RESTAURANT (who is being reviewed).
4. Every USER can have RESTAURANTS that they desire to visit, but have not visited yet.
5. Every RESTAURANT must have at least one MENU.
6. Every MENU can only be associated with one RESTAURANT.
7. A USER_CONTACT must be associated with only one USER.
8. A RESTAURANT_CONTACT must be associated with only one RESTAURANT.

Key Constraints:
1. All primary keys must be unique from one another.

Assumptions:
1. Address attributes only include street address and ZIP code. The CITY and state come from the CITY entity. 2. All emails are assumed to be valid.

Design Discussion

In designing our ER, there are several decisions that we have made in the process:
- The database allows a USER to write REVIEWs to a RESTAURANT, so that every USER can help others in deciding on which RESTAURANT are good.
- The database knows which CITY each USER and RESTAURANT are located in so that recommendations of RESTAURANTs can be provided to the USER based on the location of both the USER and RESTAURANT.

- The database records every available MENU of a RESTAURANT so that the database can also give recommendation of RESTAURANTS to the USER based on specific dishes or prices from different RESTAURANTs.

The following decisions are every aspects that has driven us to make the relationships between all the entities in our database.

In designing our RM, we made several decisions to further specify how we plan to use attributes and entity relationships from the ER diagram. In addition to adding constraints on the mentioned attributes, we ensure that the data that is inputted can be read and queried in a realistic manner.
Here are the constraints we have come up with:

- All strings except U_Email, R_Email, mLanguage, and rvMessage contain the same domain constraint of 50 alphabetic characters. Given all strings except those specified in the list, don't need a large character count and should only contain alphabetic letters.
- The excepted strings all have some sort of special identifier or shouldn't be constrained by just a 50 character limit (such as rvMessage which has a 150 alphanumeric character limit). Emails must contain a @domain at the end of a 20 character alphanumeric string, and rAddress must follow a specific format of: Street address, State, and Zip Code. mLangauge is to be specified by the list of officially recognized languages in the US.
- For non-string domain constraints, we primarily limited the integers to a range of either 0-9 for all phone numbers or as 1-5 for rvRating, as they are common integer ranges for those types of numbers.
- rID is specified as a 4-digit integer that is unique from 0-9 for easier database querying of restaurants rather than a potentially non-unique name.
- For referential constraints, we thought that every USER must be from a CITY, and likewise every CITY must have a USER. Otherwise it wouldn't make sense that we would have a database for this particular CITY with no USERs. Following the same logical conclusion, we decided that every REVIEW must be from a RESTAURANT and every MENU should also be from a RESTAURANT. However, MENUs should only refer to a single RESTAURANT, as they're unique to the RESTAURANT.

Data Generation and Testing

In order to create a sufficient amount of restaurants, we will spoof restaurants. To do so, we will either use an outside website to create data to our specifications, or we will write a program to spoof data for ourselves. Fake data will include a name (more than one of each is allowed), an ID number unique to the restaurant, an address, a phone number, hours, a menu (5 items max), and the type of food. Each restaurant will be in a specific format and include the same names for foods to allow for easier searching (Hamburger vs Burger). These spoofed restaurants will be based off real restaurant data that we gathered for accuracy, and should cover all edge cases for valid restaurants and invalid restaurants.

An example of a valid Restaurant's data would be the following:

Name: Picasso's Dreamy Dishes
1ID: 0001
Address: 12252 Picasso Drive, Lynnwood WA 92922
Cuisine: American
Menu:

Steak	$15.00
Hamburger	$6.50
French Fries	$3.30
Chicken Strips	$4.25
Peach Pie	$3.30

We will also generate each user's data once they have registered to the system. We will use these following attributes to describe the user in our database: name, email, address (city, state, and zip code).

An example of a valid User's data would be the following:
Email Address: greatRestaurantCritic@rest.com
City Name: Lynnwood
User Name: Maxwell Broth

Data Methodology

Due to the large amounts of data our team needed for our restaurant database, we forwent our previous idea of creating a program to produce data. We determined it would have been to difficult to come up with a reasonable amount of data opposed to using Mockaroo. With Mockaroo, we could create thousands of data points with ease.

Our data is based off of real life food items and locations. Where we first found restaurants manually to their corresponding fields and adjusted our attributes as needed. These manual restaurants were found through online sources.

Processing our data required us to run scripts to insert our data into their corresponding locations which was deemed too inconsistent and unsustainable comparative to Mockaroo generation.

To create our test data we utilized Mockaroo data generation. Essentially we used built in features, and custom features of our own to create data.

Each table's data was generated independently, and where ID's were necessary we utilized Mockaroo's row number counting function which allowed us to have unique identifiers. The first ID would be 1, the second would be 2, and so on.

Using Mockaroo we were able to come up with 500 users, 250 restaurants, accompanying reviews, user contacts, restaurant contacts, Menus, hours, and food entries. All together we have around 2750 inserts proving Mockaroo to be the best choice for our data creation.

Application Area

The application of this project shall be centered around Restaurants and all the variety of cuisines that pertain to each respective restaurant. Through the application of this project, the user shall be able to determine a restaurant's review and rating through aggregate site results, relative location of the restaurant, the cuisine and menu catalog of the restaurant.

Data Storage

Reviews and Ratings

The database will store information that users enter regarding to their experience at a restaurant. The users' rating - provided in a scale of 5 - will be stored along with their comments for the restaurant they want to review.

Menu Catalog

The database system will store the complete menu for every restaurant in our made up world. Each will have the restaurants' dishes and their me us that they utilize, such as breakfast, lunch, and dinner (although this depends on the restaurant). Moreover, the menu will save the price for each item.

Location

The database system will save the location of each restaurant in a form of traditional street address, which includes the city and state.

Contact Information

The phone numbers and email addresses will be stored for every restaurant.

Hours

For each restaurant, the hours of operations will be stored on a 12 hour clock. Also, if a restaurant has special timings for any events or promotions, the name of the timing, start time, and end time will be stored in HOURS.

Type of Cuisine

The database system will store each restaurants' type of cuisine. It can use the restaurant's country of origin (e.g. Chinese, Mexican, etc.) or its specific type of food (e.g. pizza, burgers, etc.).

Querying

Users will be able to query these followings restaurant specifiers: the highest rated restaurant in the area, cuisine type, menu items, availability times, distance from current location, approximate range of price, and potential experiences to see. In addition to that, users will be able to combine any of the following listed restaurant specifiers by applying 'custom filters' features. From this, restaurants could be found by using two or more specifiers (e.g. the highest rated restaurant by a specific cuisine, the closest restaurant with potential experiences to see)

Each of the restaurant specifiers that are able to be queried are listed in their own separate tables. Examples would be a 'Ratings' table that has the aggregate ratings corresponded directly to the restaurant through a numerical form, such as 3.4/5.

Through each restaurant specifier, the user is able to filter and access each aspect of the intended applicable areas to the restaurants are supported. An example would be people that utilize the restaurant specifier for the highest rated restaurant in their specified area. They will be able to find a list of several restaurants with the highest rate and review. From this, they would be able to determine which specific restaurant from the list they want to go to. Another example would be utilizing the menu items specifier from our database system. User that uses this specifier will be able to view the menu catalog of each restaurant that are listed from the specifier. This will help the user so that they can pick which restaurant to go by choosing certain meal they want to eat.

User Scenarios

Restaurant Sightseer

A sightseer may be a local or a tourist looking for a new restaurant to try out. To appeal to their curiosity, the database will query a top 5 or top X (specified by user) places to eat at. The top X places to eat shall be judged based on the reviews and ratings of restaurants in a user specified area.

Restaurant Food Critic

This type of user is an experienced food traveller who critics restaurants. The critics can search for certain restaurants using the ‘Search’ feature from the database system. Then, the food critics can give their feedbacks on certain restaurants by using our ‘Write comments’ and ‘Rate the restaurant’ features. From these, the critics can be recorded for the restaurant that has been reviewed by the food critics.

Casual Diner

This type of user is just looking to find somewhere to eat. A ‘local eateries’ feature shall be considered to just display the top 5 restaurants recommended by a specific cities local users.

Experienced Local

This user may need help choosing from one of the many restaurants local or not local to them. A past ‘search history’ dropdown menu shall be considered displaying the last 5 entries to allow users to quickly find their past searches. With this, the database system can help locals to pick a restaurant to go to.

Whimsical Moodies

This user is in search of not just food, but a show. An example of a whimsical moody is a restaurant that engages the audience to take part in a murder mystery.

Tooling Assessment

Database Management System

The Database Management System (DBMS) that we intend to use is MySQL as it's widely documented, easy to procure and utilize, and overall a solid DBMS that suits all of our needs to implement the project functions.

User Interface

For our interface we will be using Java, specifically Eclipse WindowBuilder, in setting up a front end and to connection to AWS RDS DB instances.

Database Hosting

We are hosting our database on Amazon Web Services servers because it has support for MySQL, it's free, and easily accessible with a reputable uptime and usage. It also is flexible in terms of use and easy to setup after a small baseline of understanding.

Data Generation

For creating random data entries to be stored in our database, we will be using the terms generated by Mockaroo for all of our entities' attributes. This tool allows us to produce hundred or thousands of entries in a short amount of time while giving us the ability to check if the data that we are receiving is valid or not.

Embedded SQL Language

For the Embedded SQL Language we are currently Java Database Connection (JDBC) due to its wide support and ease of use to connect to AWS RDS and use querying logic.

IDEs

The only IDE that we used for this was Eclipse, as it allowed for each development of the GUI outline. It also was easy to connect it to mySQL server that we have set up.

Version Control

Github/Git is a free to use powerful version control software that our group members are familiar with to utilize.

Communication

We will be using Discord as our primary medium to host meetings and communicate with one another.

Diagram Creation

We will be using Draw.IO as our primary generator for diagrams regarding ER and RM.

Credentials to connect to AWS RDS

AWS RDS Service
Endpoint: mysql-uwb-css475-db.civewz6zb6nl.us-west-2.rds.amazonaws.com
Port: 3306

Credentials:
User: guest
Password: {blank}

Note: There may be some odd SSL certifications issues present with AWS RDS and MySQL connection. In the case that you do get the issue, follow the steps below.

Helpful steps to run:

From Workbench C.E.

1. Add a Database Connection
2. In port number put: 3306
3. In host name: mysql-uwb-css475-db.civewz6zb6nl.us-west-2.rds.amazonaws.com
4. In username, put your respective credentials
5. Go to SSL tab
6. In 'Use SSL' set it to **No**
7. Test connection

For Terminal Connection:

1. Navigate to wherever MySQL has been installed:
2. Enter the below command

```
mysql -h mysql-uwb-css475-db.civewz6zb6nl.us-west-2.rds.amazonaws.com -P 3306 -u
<your_user_name> -p --ssl-mode=DISABLED
```

Please contact Daniel Yan at daniely@uw.edu if there are issues with user credentials.

In the case that you are unable to connect to the hosted AWS RDS DB, please use the attached SQL files for data population and grading.

Updated Iteration

We have made several changes to our RM and ER design from our previous iteration through discussion. Following are the changes that we had made.

As part of our original proposal, we intended on having a system where USERs could, in a sense, bookmark RESTAURANTs that they would like to visit in the future. We decided not to have this feature because this is not what we envisioned this database to be used for. We want this database to be used for people who want a quick decision to be made on their behalf regarding where they will eat their next meal. Also, we already had a feature for each USER to have a favorite RESTAURANT, so it did not make sense to have a running list of places for them to eat.

Another change that was made to the original proposal was to omit the STATE entity in the ER model and in the RM. We thought that we should be able to just use CITY as a way to get the STATE, instead of having a separate table with 50 states in them.

A third change that was made was the creation of CONTACT and RCONTACT. While the idea behind the two remained the same, having the email and phone number for every USER and RESTAURANT, respectively, we had not considered it to be a multivalued attribute. In other words, we wanted to allow USERs and RESTAURANTs to have multiple emails and phone numbers, as this may be the case in real life. To fix this issue, we decided to create a table for each, one that stores emails and phone numbers for USERs and one for RESTAURANTs. The reason why these are different is in the off chance that the userID for a USER is the same as rID for a RESTAURANT. Separating the two will eliminate that possibility.

In addition to adding a contact table for RESTAURANT, we also added an HOURS table. We did this because we ran into the same issue as we did with contact information; hours are also multivalued attributes. Each RESTAURANT may have different specialty hours and a different number of them. In the previous proposal, we only gave each RESTAURANT two options: operation and happy hours. We did not account for other hours, such as lunch and dinner timings or various other timings. Adding this table fixed that for us.

We had also envisioned using real life restaurants' MENUs by collecting all the data into an Excel file and importing that into the database. Doing this for hundreds of RESTAURANTs would have taken up most of our time, so, instead, we decided to spoof all of the information through Mockaroo.

The last major change that we ran into was storing each food item from the MENUs. Earlier, we would store all the items in the MENU table, but we quickly realized that we could not effectively search that table. So, what we did, was we made a FOOD ENTRIES table, which stores all of the food items from the MENUs. Doing this enables us to search by food type, MENU, and price.

We have also reduced our scope in terms of project from having maps and actual distance calculations to guide users to our current implementation (with just relative locations) as we realized some issues with tooling and overall complexity. Even though adding Google Maps (and distance by extension) was a stretch goal to begin with, we were unable to implement it.

Discussion

Database Design and Results

It is no secret that we made many changes to our design throughout the quarter, but we firmly believe that the design in no way deteriorated with each iteration. Over time our design transformed from a 4 entity table schema to a 9 table schema. Over the course of the quarter we learned how to better implement our database (i.e Normalize it). We originally started out with an attribute-crammed layout, but now we have spread things out and added more functionality. Our design includes: FOOD_ENTRIES, CONTACT, REVIEW, RUSER, HOURS, RCONTACT, MENU, RESTAURANT, CITY tables. The way our database work is: Every RUSER (Restaurant User) has RCONTACT (Contact(s)) and these users can REVIEW restaurants and give them a score (1-5) and a little message. Each CITY has RESTAURANT which is defined with a RCONTACT (can have many) and HOURS (many different types of hours). Each RESTAURANT has a MENU and every MENU has FOOD_ENTRIES. Essentially we have created a design that allows for a lot of modularity. The resultant of having a modular design gives us the ability to contain more complex data.

Evaluation

Overall, we have spent *many* hours / days on this project. We spent a considerable amount of effort trying to perfect our schema and create a UI to interact with our database, which I believe shows in our schema and our UI. We had a lot go right for our project, we managed to learn MySQL quite easily which helped us in the creation of our database. We managed to get our database hosted on AWS with ease. And finally, we were able to decide on things quite easily, which in turn made the project a bit easy. Although, we had our successes we also had a number of failures. As a group we were quite dysfunctional, with all of our other projects and meetings going on. We also had a very rough time with our UI and connecting to our database with our UI. No one in our group had much experience with any form of interface, so we had to learn it all as we go.

Feedback Incorporation

After receiving feedback on our Project Iteration 2 we deciding upon scraping PHP and moving forward with a Java based UI. We decided that because none of us really had much experience with PHP that we should pick a language we all collectively knew (Java). As for the us having experience with AWS we had already started researching and determined that hosting with AWS would be fairly straightforward. No one in our group had previous experience with Mockaroo, but we were able to learn it and utilize its vast capabilities.

How’s

Data Generation

To generate our data, we utilized Mockaroo. Mockaroo is a highly versatile and easy to use data generator (after learning how it works). In order to generate our data, we had to generate data individually for each table. Which became quite a chore because Mockaroo can’t communicate (that we found) between tables (ie utilize the same id across different tables), so to cope with this we utilized the count by row function in Mockaroo. In using the count by row utility we managed to create data that makes sense (kind of) across tables.

Testing

Throughout the entirety of our project we utilized many tests. In the beginning we hand drew our database in tables and than attempted to insert information. In doing so, we determined whether or not the methodology made sense. Using this method we managed to normalize our tables before having the lecture on normalization. We noticed that having to make a new user every time we wanted to change around the users email made no sense, so we created more tables to store information. To test our database, we utilized specific insertions to verify answers were correct. After inserting our test cases, we ran our test queries and verified that answers were what they should be. Following the verification of our database, we pushed 2850 inserts into our database. Than we ran our queries and made sure that the answers made sense. Essentially we would test after each “phase” and move forward depending on results.

Normalization

For normalization, all of our tables satisfy the First Normal Form(1NF), where each relation does not have any multi-valued or complex attributes. Beyond that, most of the relations in our Foogle database have satisfied Second Normal Form(2NF), Third Normal Form (3NF), and Boyce–Codd Normal Form (BCNF). Here are the specific details on normalization for each of the relations we have in our Foogle database:

- USER
    - This relation only has one Full Dependency (FD) where all the non-prime attributes (cName and uName) are dependent to the primary key attribute (UserID). This relation is within 2NF because there are no partial dependencies in the relation. It also within 3NF, because the primary key is not determined by any of the non-prime attributes.

- CITY
    - This relation only has primary keys (cName, cState), where there are non-prime attributes that is dependent to those primary keys. This relation is within 2NF since there is no partial dependency in the relation. It also is within 3NF, because the primary key is not determined by any of the non-prime attributes.
- RESTAURANT
    - This relation only has one FD where all the non-prime attributes (rName, rCuisine, rAddress) are dependent to the primary key attributes (rID and cName). This relation is within 2NF since there is no partial dependency within the relation. It also is within 3NF because none of the primary key is determined by any of the non-prime attributes.
- REVIEW
    - This relation only has one FD where all the non-prime attributes (rvRating, rvMessage, rvDate) are dependent to the primary key attributes (UserID and rID). This relation is within 2NF since there is no partial dependency within the relation. It also is within 3NF because none of the primary key is determined by any of the non-prime attributes.
- MENU
    - This relation only has one FD where the non-prime attribute (Language) is dependent to the primary key attributes (rID and mName). This relation is within 2NF since there is no partial dependency within the relation. It also is within 3NF because none of the primary key is determined by any of the non-prime attributes.
- USER_CONTACT
    - This relation only has one FD where all the non-prime attributes (U_PhoneNumber, U_Email) are dependent to the primary key attributes (UserID). This relation is within 2NF since there is no partial dependency within the relation. It also is within 3NF because none of the primary key is determined by any of the non-prime attributes.
- RESTAURANT_CONTACT
    - This relation only has one FD where all the non-prime attributes (R_PhoneNumber, R_Email) are dependent to the primary key attributes (rID). This relation approves 2NF since there are no partial dependency within the relation. It also approve 3NF because none of the primary key is determined by any of the non-prime attributes.
- HOURS
    - This relation only has one FD where all the non-prime attribute (Start_Time and End_Time) are dependent to the primary key attributes (rID and Hours_Name). This relation is within 2NF since there is no partial dependency within the relation. It also is within 3NF because none of the primary key is determined by any of the non-prime attributes.
- FOOD_ENTRIES

- This relation only has one FD where all the non-prime attributes (FEName, FEPrice) is dependent to the primary key attributes (mName, rID). This relation is within 2NF since there are no partial dependency within the relation. It also is within 3NF because none of the primary key is determined by any of the non-prime attributes.

Database Creation

The following database is used:
Restaurant_DB

The following tables are used:
CITY
RESTAURANT
RUSER
CONTACT
REVIEW
HOURS
RCONTACT
MENU
FOOD_ENTRIES

Below is the syntax utilized in MySQL to create the DB and the associated tables. This is provided in an attached SQL file and already in the hosted AWS RDS DB.

```
CREATE DATABASE IF NOT EXISTS
Restaurant_DB;
USE Restaurant_DB;
-- Drop the tables if it exists

DROP TABLE IF EXISTS FOOD_ENTRIES;
DROP TABLE IF EXISTS CONTACT;
DROP TABLE IF EXISTS REVIEW;
DROP TABLE IF EXISTS RUSER;
DROP TABLE IF EXISTS HOURS;
DROP TABLE IF EXISTS RCONTACT;
DROP TABLE IF EXISTS MENU;
DROP TABLE IF EXISTS RESTAURANT;
DROP TABLE IF EXISTS CITY;

 -- new tables
 -- When creating data we only need 5
cities and their corresponding states
```

```
CREATE TABLE IF NOT EXISTS REVIEW
(
    rating INT NOT NULL
    CHECK (rating >= 1 AND rating <=
5),
    uID INT NOT NULL CHECK (uID >= 0),
    rID INT NOT NULL CHECK (rID >= 0),
    review_day DATE NOT NULL,
    message TEXT,
    FOREIGN KEY(uID) REFERENCES
RUSER(userID),
    FOREIGN KEY(rID) REFERENCES
RESTAURANT(rID)
);

CREATE TABLE IF NOT EXISTS HOURS
(
    start_time TIME NOT NULL,
CREATE TABLE IF NOT EXISTS CITY
(
    name VARCHAR(30) NOT NULL,
    state VARCHAR(30) NOT NULL,
    PRIMARY KEY (name, state)
);

CREATE TABLE IF NOT EXISTS RESTAURANT
(
    rID INT UNIQUE NOT NULL
    CHECK (rID >= 0),
    address VARCHAR(50) NOT NULL,
    rCity VARCHAR(30) NOT NULL,
    cuisine VARCHAR(20) NOT NULL,
    name VARCHAR(50) NOT NULL,
    PRIMARY KEY (rID),
    FOREIGN KEY (rCity) REFERENCES
CITY(name)
);

CREATE TABLE IF NOT EXISTS RUSER
(
    userID int UNIQUE NOT NULL
    CHECK (userID >= 0),
    name VARCHAR(30) NOT NULL,
    favorite_restaurant INT,
    PRIMARY KEY (userID),
    FOREIGN KEY (favorite_restaurant)
REFERENCES RESTAURANT (rID)
);

CREATE TABLE IF NOT EXISTS CONTACT
(
    email VARCHAR(30) UNIQUE NOT NULL,
    phone_number VARCHAR(10) UNIQUE NOT
NULL,
    uID INT NOT NULL CHECK (uID >= 0),
    counter INT NOT NULL
    CHECK (counter > 0),
    FOREIGN KEY (uID) REFERENCES
RUSER(userID),
    PRIMARY KEY (counter, uID)
);
```

```
    end_time TIME NOT NULL,
    time_name VARCHAR(20) NOT NULL,
    rest_id INT UNIQUE NOT NULL
    CHECK (rest_id >= 0),
    PRIMARY KEY (time_name, rest_id),
    FOREIGN KEY (rest_id) REFERENCES
RESTAURANT(rID)
);

CREATE TABLE IF NOT EXISTS RCONTACT
(
    email VARCHAR (50) UNIQUE NOT NULL,
    phone_number VARCHAR(10) UNIQUE NOT
NULL,
    rest_id INT UNIQUE NOT NULL
    CHECK (rest_id >= 0),
    PRIMARY KEY (email, phone_number),
    FOREIGN KEY (rest_id) REFERENCES
RESTAURANT(rID)
);

CREATE TABLE IF NOT EXISTS MENU
(
    m_name VARCHAR(30) NOT NULL,
    r_id INT UNIQUE NOT NULL
     CHECK (r_id >= 0),
    LANGUAGE VARCHAR(30) DEFAULT
'English',
    PRIMARY KEY (m_name, r_id),
    FOREIGN KEY (r_id) REFERENCES
RESTAURANT(rID)
);
CREATE TABLE IF NOT EXISTS FOOD_ENTRIES
(
    price DECIMAL(13,2) NOT NULL
    CHECK (price > 0.00),
    r_menu_id INT NOT NULL
    CHECK (r_menu_id = 0),
    name VARCHAR(60) NOT NULL,
    PRIMARY KEY (name, r_menu_id),
    FOREIGN KEY (r_menu_id) REFERENCES
MENU(r_id)
);
```

Database Population

```
-- The following insert snapshots shows 5 items being inserted
per entity
```

```
insert into CITY (name, state) values ('Seattle', 'Washington');
insert into CITY (name, state) values ('Bellevue', 'Washington');
insert into CITY (name, state) values ('Lynnwood', 'Washington');
insert into CITY (name, state) values ('Los Angeles', 'California');
insert into CITY (name, state) values ('San Francisco', 'California');
```

```
insert into RESTAURANT (rID, address, rCity, cuisine, name) values (1, '3519
Gateway Center', 'Bellevue', 'Mexican', 'Lubowitz, Franecki and Little');
insert into RESTAURANT (rID, address, rCity, cuisine, name) values (2, '73 Marcy
Drive', 'San Francisco', 'Mexican', 'Ullrich, Wintheiser and Gleason');
insert into RESTAURANT (rID, address, rCity, cuisine, name) values (3, '8
Florence Point', 'Los Angeles ', 'Mexican', 'Lang LLC');
insert into RESTAURANT (rID, address, rCity, cuisine, name) values (4, '771
Dayton Way', 'Bellevue', 'Mexican', 'Wolff, Gibson and Conroy');
insert into RESTAURANT (rID, address, rCity, cuisine, name) values (5, '7 Sloan
Alley', 'San Francisco', 'Mexican', 'Gleason and Sons');
```

```
insert into RUSER (userID, name, favorite_restaurant) values (1, 'Jammie', 5);
insert into RUSER (userID, name, favorite_restaurant) values (2, 'Rafa', 4);
insert into RUSER (userID, name, favorite_restaurant) values (3, 'Akim', 3);
insert into RUSER (userID, name, favorite_restaurant) values (4, 'Arvin', 2);
insert into RUSER (userID, name, favorite_restaurant) values (5, 'Thebault', 1);
```

```
insert into CONTACT (email, phone_number, uID, counter) values
('dseymark0@webmd.com', '9916199853', 1, 1);
insert into CONTACT (email, phone_number, uID, counter) values
('rkubach1@diigo.com', '9083609679', 2, 1);
insert into CONTACT (email, phone_number, uID, counter) values
('wmcgilroy2@amazon.de', '3198948485', 3, 1);
insert into CONTACT (email, phone_number, uID, counter) values
('kbackshell3@xrea.com', '9824342042', 4, 1);
```

```
insert into CONTACT (email, phone_number, uID, counter) values
('emedway4@goodreads.com', '9372937077', 5, 1);
```

```
insert into HOURS (start_time, end_time, time_name, rest_id) values ('8:21
AM', '8:35 AM', 'Operating Hours', 1);
insert into HOURS (start_time, end_time, time_name, rest_id) values ('9:05
AM', '12:52 PM', 'Work Hours', 2);
insert into HOURS (start_time, end_time, time_name, rest_id) values ('8:13
AM', '4:24 AM', 'Work Hours', 3);
insert into HOURS (start_time, end_time, time_name, rest_id) values ('8:13
AM', '12:38 PM', 'Food Hours', 4);
insert into HOURS (start_time, end_time, time_name, rest_id) values
('10:13 AM', '10:58 AM', 'Food Hours', 5);
```

```
insert into RCONTACT (email, phone_number, rest_id) values
('ddoul0@businessweek.com', '5243587137', 1);
insert into RCONTACT (email, phone_number, rest_id) values
('sschettini1@economist.com', '2505828719', 2);
insert into RCONTACT (email, phone_number, rest_id) values
('atommis2@java.com', '5091976141', 3);
insert into RCONTACT (email, phone_number, rest_id) values
('cdoers3@sphinn.com', '5072299875', 4);
insert into RCONTACT (email, phone_number, rest_id) values
('rverdun4@cyberchimps.com', '8552730060', 5);
```

```
insert into MENU (m_name, r_id, LANGUAGE) values ('Food', 1, 'English');
insert into MENU (m_name, r_id, LANGUAGE) values ('TastesForTastes', 2,
'English');
insert into MENU (m_name, r_id, LANGUAGE) values ('TheFood', 3,
'English');
insert into MENU (m_name, r_id, LANGUAGE) values ('Food', 4, 'English');
insert into MENU (m_name, r_id, LANGUAGE) values ('IPromiseItIsFresh', 5,
'English');
```

```
insert into FOOD_ENTRIES (price, r_menu_id, name) values (4.7, 1,
'Tamales');
insert into FOOD_ENTRIES (price, r_menu_id, name) values (19.1, 2,
'Tamales');
insert into FOOD_ENTRIES (price, r_menu_id, name) values (5.09, 3,
'Tamales');
insert into FOOD_ENTRIES (price, r_menu_id, name) values (16.02, 4,
'Tamales');
```

insert into FOOD_ENTRIES (price, r_menu_id, name) values (13.9, 5, 'Tamales');

insert into FOOD_ENTRIES (price, r_menu_id, name) values (15.52, 1, 'Tacos');
insert into FOOD_ENTRIES (price, r_menu_id, name) values (17.1, 2, 'Tacos');
insert into FOOD_ENTRIES (price, r_menu_id, name) values (5.33, 3, 'Tacos');
insert into FOOD_ENTRIES (price, r_menu_id, name) values (13.53, 4, 'Tacos');
insert into FOOD_ENTRIES (price, r_menu_id, name) values (18.71, 5, 'Tacos');

Database Query

SQL Query Statements

SQL STATEMENT	PURPOSE
SELECT restaurant.NAME FROM restaurant, review WHERE 3 < (SELECT Avg(review.rating)) from review AS r, restaurant AS rst WHERE r.rid = rst.rid) GROUP BY restaurant.NAME;	This query determines the name of all the restaurants with an average rating higher than 3 stars! We select the restaurant names, then we filter them out by selecting the average review rating from each one.
SELECT food_entries.price, food_entries.NAME, restaurant.cuisine FROM food_entries, restaurant WHERE food_entries.r_menu_id = restaurant.rid AND food_entries.price IN (SELECT Max(price) FROM food_entries, menu, restaurant WHERE r_menu_id = r_id AND r_id = rid GROUP BY cuisine) ORDER BY price ASC;	The following query determines the most expensive foods in each ethnic food category. It prints out the price, food name, and the restaurants cuisine type. It queries the menu id from each restaurant (which is unique), than it finds the MAX priced item.
SELECT food_entries.price, food_entries.NAME, restaurant.NAME FROM food_entries,	This query selects the lowest price amongst all the restaurants in each ethnic

restaurant
WHERE food_entries.r_menu_id =
restaurant.rid
AND food_entries.price IN
(SELECT Min(price)
FROM food_entries, menu, restaurant
WHERE r_menu_id = r_id AND r_id = rid
GROUP BY cuisine)
ORDER BY price ASC;
	category. It functions the same as the previous sql

SELECT *
FROM food_entries
WHERE food_entries.price IN
(SELECT Min(price)
FROM food_entries, menu, restaurant,
city
WHERE r_menu_id = r_id AND r_id = rid
AND rcity = city.NAME
GROUP BY city.NAME)
ORDER BY price ASC;
	This query displays the least expensive food in each city. Essentially it lists all the restaurants in a city, than selects the menus, than selects the minimum price from the menu.

SELECT RC.phone_number
FROM restaurant AS R,
rcontact AS RC
WHERE R.rid = RC.rest_id
AND R.rcity = 'Bellevue'
AND R.cuisine = 'Chinese';
	Lists all of the restaurant's phone numbers in Bellevue that server chinese food. Essentially this is just a pilot to what a user could query. IE the contact information for a restaurant.

SELECT *
FROM food_entries
WHERE (food_entries.r_menu_id =
(SELECT restaurant.rid
FROM restaurant
WHERE restaurant.NAME =
'Corwin Inc'));
	Lists all of the food entries for a certain restaurant named Corwin Inc. This query acts as a gateway to more queries listing out the entries. All that needs to be changed is the name of the restaurant.

User Query Statements From GUI

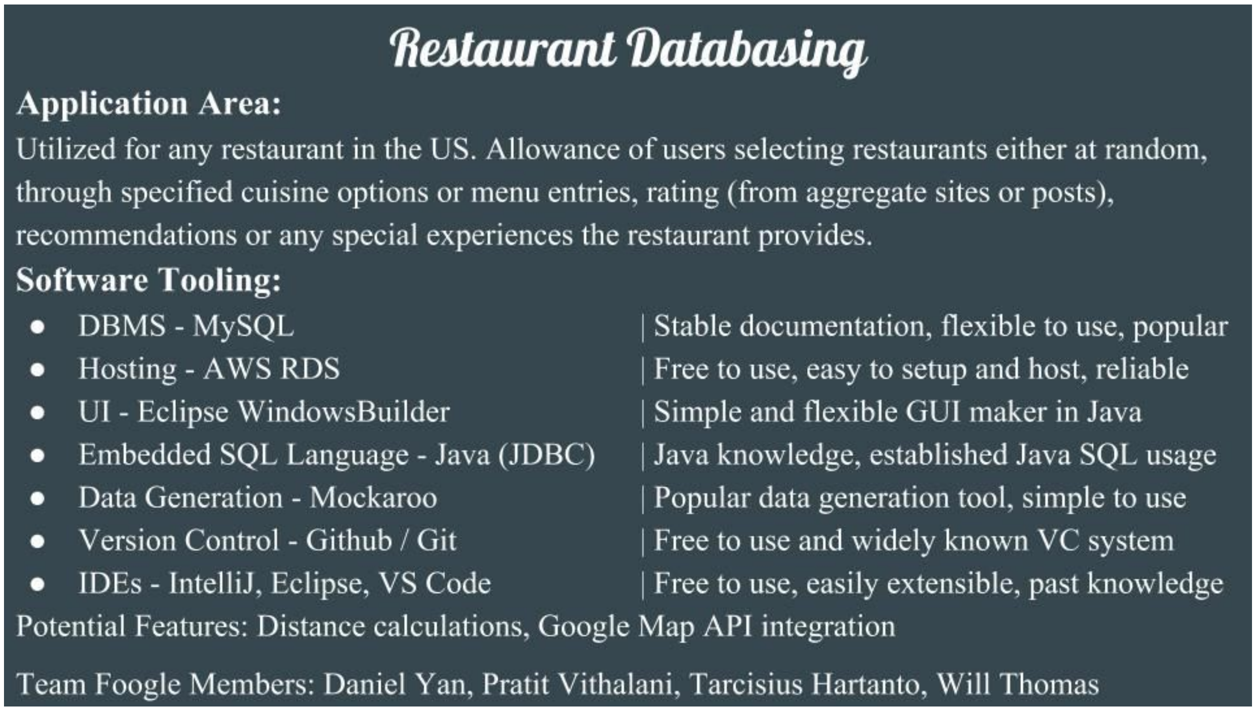
STATEMENT	PURPOSE
"INSERT INTO RUSER (userID, name) VALUES (uID, userName)";	This is used to add a new user to the database. All they need to provide is the userName variable as the uID is a randomly generated primary key.
"INSERT INTO CONTACT (email,	This adds a user's contact

phone_number, uID, counter) VALUES (email, telephone, userID, count)";	information to the CONTACT database. All the user needs to provide is their email and phone number, which is stored. The userID will be figured out by their name, which is also given. Count is simply incremented depending on how many contacts they have.
"SELECT name, price FROM FOOD_ENTRIES WHERE FOOD_ENTRIES.price = (SELECT MAX(price) FROM FOOD_ENTRIES);"	This finds the name and price of the most expensive food item in the database.
"SELECT FOOD_ENTRIES.price , FOOD_ENTRIES.name, RESTAURANT.cuisine FROM FOOD_ENTRIES, RESTAURANT WHERE FOOD_ENTRIES.r_menu_id = RESTAURANT.rID AND FOOD_ENTRIES.price IN(SELECT MAX(price)FROM FOOD_ENTRIES AS FE, MENU AS M, RESTAURANT AS R WHERE FE.r_menu_id = M.r_id AND M.r_id = R.rID GROUP BY cuisine) ORDER BY price ASC;"	This looks for the most expensive food in the database and sorts it by the category/cuisine it belongs to.
"SELECT FOOD_ENTRIES.price , FOOD_ENTRIES.name, RESTAURANT.cuisine FROM FOOD_ENTRIES, RESTAURANT WHERE FOOD_ENTRIES.r_menu_id = RESTAURANT.rID AND FOOD_ENTRIES.price IN(SELECT MIN(price)FROM FOOD_ENTRIES AS FE, MENU AS M, RESTAURANT AS R WHERE FE.r_menu_id = M.r_id AND M.r_id = R.rID GROUP BY cuisine) ORDER BY price ASC;"	This looks for the least expensive food in the database and sorts it by the category/cuisine it belongs to.
"SELECT * FROM FOOD_ENTRIES, CITY	This finds the cheapest food

WHERE FOOD_ENTRIES.price IN(SELECT MIN(price) FROM FOOD_ENTRIES AS FE, MENU, RESTAURANT, CITY WHERE FE.r_menu_id = r_id AND r_id = rID AND rCity = CITY.name) ORDER BY price ASC;";	items in each city. This is helpful because it is easy to see where the cheapest foods are in the cities within the database.
"INSERT INTO REVIEW (rating, uID, rID, review_day, message) VALUES (?, (SELECT userID FROM RUSER WHERE name = ?), (SELECT rID FROM RESTAURANT WHERE name = ? AND rCity = ?), curdate(), ?);"	This adds a review to the database. It takes in the restaurant name and city, it identify the restaurant, the user name, to identify the writer of the review, the rating that they give their experience, and an optional comment section to better explain their experience.
"SELECT DISTINCT RC.email, RC.phone_number FROM RCONTACT AS RC, RESTAURANT AS R WHERE rest_ID IN (SELECT rID FROM RESTAURANT WHERE name = '" + restName + "' AND rCity = '" + cityName + "');"	This searches for the contact information of a restaurant. The user inputs the restaurant name and the city it is in (to identify which restaurant), and they are given the emails and phone numbers.
"SELECT DISTINCT H.time_name, H.start_time, H.end_time FROM HOURS AS H, RESTAURANT AS R WHERE rest_ID IN (SELECT rID FROM RESTAURANT WHERE name = '" + restName + "' AND rCity = '" + cityName + "');";	This finds the hours that a restaurant has, whether it is simple operation hours or special hours that a restaurant has. All that the user needs to do is provide the restaurant name and city to identify where to get the hours information.
"SELECT DISTINCT RV.rating, RV.message, U.name FROM REVIEW AS RV, RESTAURANT AS R, RUSER AS U WHERE RV.uID = U.userID AND RV.rID IN (SELECT rID FROM RESTAURANT WHERE name = '" + restName + "' AND rCity = '" + cityName + "');”	This finds the reviews of the restaurant that the user wants to see. The user simply inputs the restaurant name and city and they will get all of the reviews, associated ratings, and the writers of those reviews.

"SELECT name, address, rCity FROM RESTAURANT WHERE rID rand;”	Gives a random restaurant when the user clicks on that button.
"SELECT name, address FROM RESTAURANT WHERE rCity = city AND cuisine = cuisine;"	Gives the user a list of restaurants based on their input for city and cuisine type.

Overall



Restaurant Databasing

Application Area:

Utilized for any restaurant in the US. Allowance of users selecting restaurants either at random, through specified cuisine options or menu entries, rating (from aggregate sites or posts), recommendations or any special experiences the restaurant provides.

Software Tooling:

● DBMS - MySQL	Stable documentation, flexible to use, popular
● Hosting - AWS RDS	Free to use, easy to setup and host, reliable
● UI - Eclipse WindowsBuilder	Simple and flexible GUI maker in Java
● Embedded SQL Language - Java (JDBC)	Java knowledge, established Java SQL usage
● Data Generation - Mockaroo	Popular data generation tool, simple to use
● Version Control - Github / Git	Free to use and widely known VC system
● IDEs - IntelliJ, Eclipse, VS Code	Free to use, easily extensible, past knowledge

Potential Features: Distance calculations, Google Map API integration

Team Foogle Members: Daniel Yan, Pratit Vithalani, Tarcisius Hartanto, Will Thomas

References

Brocato, Mark. "Random Data Generator and API Mocking Tool | JSON / CSV / SQL / Excel." Mockaroo, mockaroo.com/

Deliverable Schedule

Week 1

Proposal reviewed and polished. **Delivered**.
Tooling started and actively being worked on.
Design Documentation started and actively being worked on.

Week 2

Tooling polished. **Delivered**.
Design Documentation polished. **Delivered**.
Populated Database started and actively worked upon.
SQL query statements started and actively being worked upon.

Week 3

Populated Database reviewed and actively worked upon.
SQL query statements reviewed and actively worked upon.
Normalization started and actively worked upon.

Week 4

Populated Database polished. **Delivered**.
SQL query statements polished. **Delivered**.
Normalization actively worked upon and reviewed.
User Interface Strategy started and actively worked upon.
Documentation and Poster presentation started and actively worked upon.

Week 5

SQL query statements polished. **Delivered**.
Normalization polished. **Delivered**.

User Interface Strategy polished. **Delivered**.
Documentation and Poster presentation reviewed and actively worked upon

Week 6

Documentation and Poster presentation polished. **Delivered**.

Compliance Assurance:
In order to ensure that this deliverable schedule is met, we will be holding weekly meetings as specified in the Team Logistics document or otherwise notified through an online medium to work and discuss out the workload partitioning and progress on any part of the deliverables for the schedule.

In the case where our deliverable has not been completed by the assigned date, there must be a compelling reason documenting why the deliverable hasn't been finished by that date. A compelling reason can be comprised of: the work was more complicated than originally believed, an extenuating circumstance, there were severe issues in the original design needed to be retooled. In any case, a write-up by the team member(s) working on the deliverable is to be expected and the schedule will need to be adjusted to to ensure other parts of the deliverables are not impacted.

Work Partitioning

All Foogle Team members must abide by 3 distinct responsibilities

The first responsibility shall be intended to focus on gathering data, by using the Internet as a primary resource to determine most or all of the queryable data for the applicable restaurants. When the Internet is not sufficient, members shall use direct communication or direct confrontation with the restaurant in question to fill in the missing information.

The second responsibility shall be intended to solely focus on converting the data into a queryable form such as formations which SQL and other Relational Languages that are applicable. Duplicate data must be avoided and noted.

The third responsibility shall be intended to focus on utilizing the data placed in the database and formulating queries through direct SQL or an interfacing language to utilize SQL. Queries that are crafted must be able to finish in reasonable time and be of sufficient aid to users.

References

"Adding an Amazon RDS DB Instance to Your Java Application Environment." *Amazon AWS*, docs.aws.amazon.com/elasticbeanstalk/latest/dg/java-rds.html. Accessed 12 Mar. 2019.

Alexander, Alvin. "A Java MySQL INSERT example (using PreparedStatement)." *Alvin Alexander*, 19 May 2018, alvinalexander.com/java/java-mysql-insert -example-preparedstatement. Accessed 17 Mar. 2019.

"Amazon EC2 Instance Types." *Amazon AWS*, aws.amazon.com/ec2/instance-types/. Accessed 8 Mar. 2019.

"Amazon RDS Instance Types." *Amazon AWS*, aws.amazon.com/rds/instance-types/. Accessed 8 Mar. 2019.

"Creating a MySQL DB Instance and Connecting to a Database on a MySQL DB Instance." *Amazon*, docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_ GettingStarted.CreatingConnecting.MySQL.html. Accessed 8 Mar. 2019.

"Java #E3: Creating a GUI with Eclipse." *YouTube*, uploaded by Martin Carlisle, 13 May 2013, www.youtube.com/watch?v=OHbSx2iVujE. Accessed 18 Mar. 2019.

"MySQL Documentation." *MySQL*, dev.mysql.com/doc/. Accessed 18 Mar. 2019.

"SQL as Understood by SQLite." *SQLite*, www.sqlite.org/lang.html. Accessed 18 Mar. 2019.

"Step 1: Create an RDS DB Instance." *Amazon*, docs.aws.amazon.com/AmazonRDS/ latest/UserGuide/CHAP_Tutorials.WebServerDB.CreateDBInstance.html. Accessed 8 Mar. 2019.

Whelan, Michael. "Creating Test Data with Mockaroo." *Michael-Whalen.net*, 25 Sept. 2014, www.michael-whelan.net/creating-test-data-with-mockaroo/. Accessed 18 Mar. 2019.

Yong, MK. "Java - Generate Random Integers in a Ranger."*MKYong.com*, 19 Aug. 2015, www.mkyong.com/java/java-generate-random-integers-in-a-range/. Accessed 18 Mar. 2019.