Prakhar Bhandari, Nathaniel Chan
Emmanuel Genene, Vincent Gubitosi

## Introduction & Project Goals

When getting a meal with a few friends of colleagues, the distance and travel time to a restaurant is often a main concern, whether it is a quick lunch break from work, or a dash for food between classes. There are many cases where individuals in a group are in different initial locations, and often times they try to find a common ground where a restaurant is closest to all of them.

It is clear that this is an opportunity for an improvement in efficiency. With the Yelp dataset, an effective database application would provide the functionality that solves this exact problem.

Our application aims to find the optimal distance and travel time to a restaurant for a group of people. Optimal distance and travel time includes the absolute distance from the restaurant, and the modes of transport taken by individuals. For example, if two people are trying to find a restaurant to eat that will them both roughly the same amount of time to reach, and one has a bicycle and the other is walking, the restaurant chosen should be closer to the person that is walking.

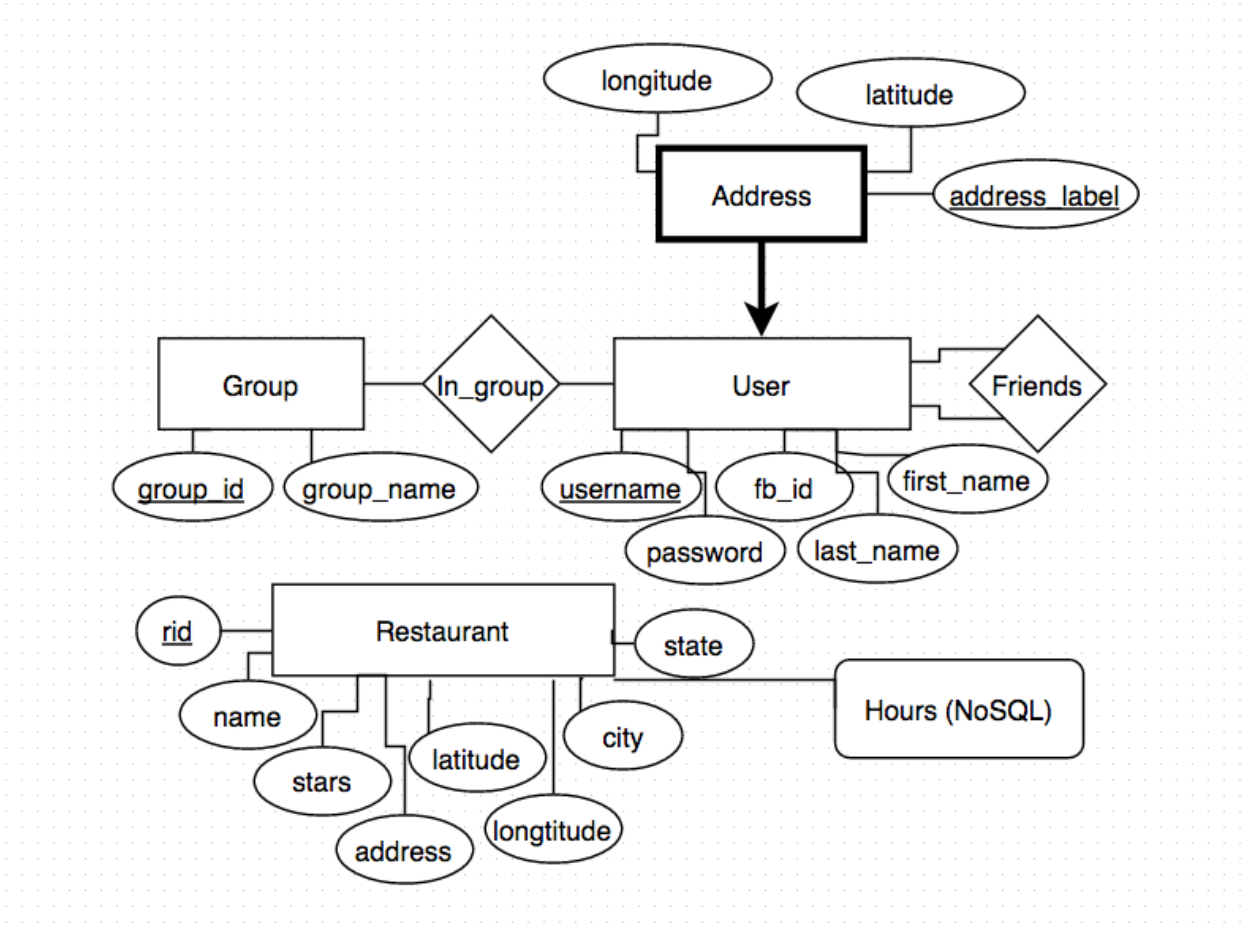
## Modules and Architecture

With much discussion, we realized that there were many possible schemas that would be appropriate for the application. After consideration, we decided to focus on:

1. Efficiency for commonly used queries
2. Avoiding update, insertion and deletion anomalies
3. Readability, for development and testing

With the above in mind, the ER diagram we resulted with is seen on the following page.

Note that the hours of each restaurant were stored in a NoSQL database, more specifically the AWS DyanmoDB. We thought about how the hours could be stored in a relational schema, and thus included properly in the ER diagram above, and this was definitely possible. However, it would use up space unnecessarily, and since queries will not need to be made to retrieve specific hours of a restaurant, but rather all the hours will be retrieved every time the information was required, a NoSQL database would be appropriate. The dataset given to us would also allow a very easy data cleaning and import into the NoSQL database.




As we starting developing the app, however, we realized that there were opportunities for schema refinement. We identified some of the functional dependencies of the schema, including the following:

- username -> fb_id
- username -> password
- address -> name (Restaurant)
- address -> latitude
- address -> longitude

We understood the importance of normalization, so that update, insertion and deletion anomalies could be minimized. However, we were also aware that fully normalizing the database into 3NF or BCNF would present practical inefficiencies when implementing queries, since many joins would need to be made even for simple queries. For example, to get all the information for a user, one would only need to join the address and user table on the username currently. With normalization, many joins with tables that were decomposed would need to be made, e.g. latitude longitude, address, fb_id, password, first_name last_name. Therefore, the schema represented in the ER diagram above was implemented and used.

We used Node.js for the backend of the application, and set up the architecture of the application mainly as follows, below each component is a brief explanation, and the main files that are relevant:

- Server application
    - Sets up server environment and server, initializes Express and view engine EJS, sets up paths for routing
    - *files:* app.js
- Database connection
    - Provides methods to access both the relational and NoSQL database. There are a variety of methods for each table, in anticipation of different features that may have different queries for the database. This method of connecting to the database was chosen so that it would be very convenient to develop the frontend of the application, since the developer simply has to instantiate the db object, and call the methods on that object from the frontend.
    - *files:* db.js
- Routing
    - The paths of each route are set up in the server application, but the functionality for each route is implemented in the routing component of the application. The methods handle requests from the frontend, usually initiated by users, and performs the relevant queries to the database, and sends a response at the end.
    - *files**:*** index.js
- Views
    - The HTML pages that will be displayed to the user are compiled by the EJS engine, and the .ejs files are templates that the engine will compile the page with, given the relevant information from the routing.
    - *files:* views/*.ejs

With how the application is setup, it is very easy to add features and components, and this was experienced during the development of the application.

## Data instance use

The main problem that we are solving in our project is in planning restaurant outings with groups. For this reason, locations of the restaurants in the yelp dataset are integral in the application. By storing these addresses appropriately in our database and also indexing to retrieve this information quickly, our application allows for efficient lookup of top restaurants based on a group's location. Because we hoped to make our application convenient to use, we made sure to also regard information like stars to allow for filtering of results by the user. Finally, in the setup of our application we kept track of the hours for each restaurant in DynamoDB. While this isn't directly used currently, it can be easily and quickly integrated into the application in the future if we decide to move further with our venture.

## Data cleaning and import mechanism

To clean and import the data into our database, we utilized JDBC. We populated the database in a similar manner to the techniques used in Homework 3 (Extra Credit). The main hurdle here was making sure that the restrictions and checks in the database were being respected when uploading the restaurant data. These restrictions included foreign key constraints, primary key constraints, and other integrity constraints. Some of these constraints were not perfect and needed to be changed with trial and error. Also due to the size of the yelp dataset, it would take a good amount of time before it would be clear that the database's constraints needed to be changed.

## Algorithms & Communication Protocols

The key component of our application was figuring out what restaurants to display for a group of people. The first challenge to be faced was figuring out how to narrow down the number of restaurants to suggest to the users. The most important information we had on each user was their address (and through that, their latitude and longitude coordinates).

Since we wanted to find restaurants that would be at a convenient distance for each user, we first started by constructing a bounding box using each user's coordinates. In other words, we iterated through each of the users in the group and figured out the maximum latitude, minimum latitude, maximum longitude and minimum longitude. We next use these values to query the database for restaurants that are in our bounding box. One thing to note is that while we do this, we also order the restaurants in decreasing order of number of stars.

The next step is to rank these restaurants based on minimizing distance for each user in the group. Using the latitude and longitude values for the users and the restaurants, we calculate the taxicab distance between each user and each restaurant. The taxicab distance is a distance measurement in which the distance between two points is the sum of the absolute differences of their Cartesian coordinates instead of simply being the linear distance between the two points. This is designed to simulate distances in a city (eg: blocks and straight lines). Although this is only an approximation, this would work for most of our planned user base, which would largely consist of urban users. For the people that are biking or using a car, we multiply the distance/time taken for that user by 0.4 and 0.3, respectively. These numbers were calculated by testing out a large number of real world use cases and comparing the time taken between walking, biking and driving. In this method, we compute a value for each restaurant that corresponds to the total time it would take each user in the group to get there (lower is better).

Another key aspect that we had to consider was that the user might prefer the quality of the restaurant rather than minimizing distance that the group collectively has to travel. We judged this based on the Yelp star rating of the restaurant. So, we created three different ranking schemes that the user can choose from - one priorities minimizing time, one focuses on quality of restaurant, and the other is a combination of the two.

## Use Cases

This app is geared towards groups of people that already know each other. To add people to a group, you must know their username. This requires some familiarity with the people you are adding. This system is similar to how Venmo works, in that you have to know a person's username/phone

number to do a transaction with them. Once you have created a group of users, you are taken to a page where you can see each person's address and mode of transportation (and adjust that if necessary) and press the "Get Results" button to get the list of restaurants for the app.

On the results page, we show many metrics and options for the users. The user can choose the number of results to display, filter by a certain number of stars, and choose the ranking algorithm used for the results. The user also sees a list of restaurants (with name and address) and also can see directions to each restaurant on a Google Map.

## Optimization Techniques Employed

Even before developing the application, it was clear that some queries would be performed much more frequently than others, and some queries would require more significant processing from the Oracle DBMS. There were 2 main methods of optimization that were used, indexing and query optimization.

### Indexing

Initially, we thought that there were many indexes that could be created for all the tables. After creating the indexes and running the execution plans and timer, we realized that there were many indexes created that did not contribute to the efficiency of the queries. Therefore it would be costly since it simply took up space without making any improvements. We deleted some indexes, like the index on restaurant names, since our application did not perform searches on restaurant names. For other indexes we kept, we found that a speedup was seen, like the index on the latitude and longitude for restaurants, since the cost of looking up a particular set of coordinates were very costly without indexing.

### Query Optimization

One of the more CPU heavy queries was to select restaurants that were within a specific radius to a set of coordinates, originally seen as follows:

```
select rid, name, distance
from (
    select rid, name, ( 3959 * acos( cos( 40.440625 ) * cos( R.lat )
                        * cos( R.lon + 79.995886 ) + sin( 40.440625 )
                        * sin(R.lat) ) ) AS distance
    from restaurant R
)
where distance < 50
order by distance
```

Here the restaurants within 50 miles to the coordinates (40.440625, -79.995886). These coordinates are replaced by whatever required coordinates in the actual application, these are just test values. The distance is calculated from the Haversine formula.

We ran the above query multiple times, and average time elapsed for each query is 15.55 seconds, with 504 rows selected. This is clearly a very slow query, so we looked at the execution plan of the query. For the inner query, it was seen that all 61184 rows were scanned for 2 operations, the select statement and the full table access. This is a very costly operation, and nested in the query for selecting the distance adds to the latency. We rewrote the query in relational algebra form, and tried to

use equivalent expressions that would give queries that are more optimized. However, this query could not be optimized further by using relational algebra techniques. We then thought about what the query was trying to retrieve from the database, and realized that the distance does not need to be calculated for all 61184 tuples in restaurant. A more narrow range of restaurants could be selected for distance calculation, based on the latitude and longitude values. Therefore a further nested query seen as follows would be more optimal:

```
select rid, name, distance
from (
    select rid, name, ( 3959 * acos( cos( 40.440625 ) * cos( R.lat )
                        * cos( R.lon + 79.995886 ) + sin( 40.440625 )
                        * sin(R.lat) ) ) AS distance
    from (
        select *
        from restaurant
        where POWER(lat - 40.440625, 2) + POWER(lon + 79.995886, 2) < 20
    ) R
)
where distance < 50
order by distance
```

With this query, distance values are calculated for 3059 tuples of restaurants instead. The overall query now takes an average of 0.86 seconds, which is significantly more efficient than the original.

## Technical Specifications

Here are some of the technologies and APIs we used:

- Node.js
- HTML/CSS/jQuery
- Google Maps API
- Facebook API
- AWS
    - Relational Database Service (SQL)
    - DynamoDB (NoSQL)
    - EC2 (to launch the Node application)

We also ended up using some modules within Node.js. They are:

Async, Aws-sdk, Body-parser, Cookie-parser, Ejs, Ejs-locals, Express, Express-session, Lru-cache, nib, Node-geocoder, Oracle, Stylus

## Special Features to Highlight

Some special features of our app include usage of the Facebook API and Google Maps API. We use the Facebook API to allow users to login through Facebook, which would be quicker and easier than logging in by using their username and password. We use the Google Maps API to give restaurant suggestions for groups within a certain distance from each group member (trying to minimize the distance for all/certain people). We also use the Google Maps API to show this information visually on

a map, and to provide directions for getting to the suggested restaurants. Also a minor but important feature is that all passwords are encrypted in the database.

## Performance Evaluation & Technical Challenges

After doing our optimization and indexing, the performance of the app was smooth and the user never feels any unreasonable or noticeable delay between any step in our application. The DB queries are the biggest bottleneck, and we've structured them in a way to make them quick and efficient.

One key technical challenge was implementing and working with the Google Maps API, since we never had any experience with it. In addition to that, the hardest part of this assignment was setting up the infrastructure - which includes interfacing with the DB, setting up the DB, setting up the app on EC2 and figuring out how to pass data between each of the views in node.js.

Another challenge we had was working with the throughput limits for the AWS services. This caused the most problems when uploading the data to DynamoDB. We wrote a short script (loadHoursJSON.js) to upload all the hours data to DynamoDB, and this script goes through the entire Yelp dataset at once, all 60000 restaurants. Given the capacity that is allowed by AWS, this method of importing data was not possible, and the hours data was only uploaded for 600 restaurants. An easy solution would be to create a cron job that uploads parts of the data at a time, so that the capacity would not be reached. However, we did not have much time left when this was implemented, and decided that there were other priorities.

## Division of Work Between Group Members

The initial setup was necessary to begin this project. This was covered by Nathaniel and Vincent. The database population was then covered by Prakhar and Emmanuel. Nathaniel then proceeded to work on the database methods in our node.js application. With this framework in place, the team was able to finish the networking aspect of the project, and implement our goals. Emmanuel and Vincent worked on the home page, login page, signup page, and the edit profile page. Nathaniel worked on the group page and Prakhar worked on the results page and the UI of the project. Prakhar has had previous experience with UI development and was perfect for this role. Emmanuel, Vincent, and Nathaniel had more experience with back-end development and were well-suited for covering that aspect of the project.

## Potential Future Extensions

Since the core of our application is to suggest restaurants that are of optimal distance to a group of users, other algorithms for computing the optimal results can be tested and used. The computation of these restaurants can also be further optimized and improved, perhaps with better indexing.

Another feature that could be implemented is for users to filter particular specifications for a restaurant before the results are retrieved. This would provide the user with more flexibility and customizability.