Data Management System - YELP Dataset

Karan Manghi
kxm9436@rit.edu

Aamir Jamal
aj1243@rit.edu

Tejas Arya
ta2763@rit.edu

Akshay Kshirsagar
ak1865@rit.edu

1. INTRODUCTION

This paper illustrates how we built our data management system and the challenges we faced. Our system uses the https://www.yelp.com/dataset YELP Dataset. We have used SQL database and Microsoft SQL Server for connecting to database. The flow for the paper is as follows: Initial phase which includes the reasoning for choosing the dataset and the end application which we propose. The next phase involves how we conceptualized the data and the Entity Relationship Model, ER diagram to table conversion and so forth. The next phase includes The actual implementation details. The last part includes the screenshots of queries run on both the front end as well as that run on back end.

2. INITIAL PHASE

The YELP dataset is structured , useful and relevant information is provided. The tables are well connected which makes it possible to join tables and make inferences. The dataset contains some tables namely, Business, user, Tip, Review tables.Overall, the structure seems tightly bound. Like for example, even the users are connected to each other as friends and hence there might be a lot of useful data which can be extracted from the relationships between the tables. And because of this we will have to make sure about a lot of integrity constraints and also that there is no redundancy or inconsistency. Since all these points are the heart of a relational database, we feel that with this database, we will be able to apply what we learn in class. The data consists of multiple tables which are described in below sections.

3. DATA SET

3.0.1 Business Table

Business table includes values such as described below:
businessid:id unique id for business
name: Name for business
categories: Contains string separated category values
address: Contains address for a business
city: Contains city for a business
state: Contains state of a business
postalcode: Contains postal code for a business and other attributes like sample example given below:
"business$_id$" : "$1SWheh84yJXfytovILXOAQ$","$name$" : "$ArizonaBiltmoreGolfClub$",
"$address$" : "$2818ECaminoAcequiaDrive$","$city$" : "$Phoenix$","$state$" : "AZ","$postal_code$" : "85016",
"$latitude$" : 33.5221425,"$longitude$" : -112.0184807,
"$stars$" : 3.0,"$review_count$" : 5,"is_open" : 0,"$attributes$" : "$GoodForKids$" : "$False$",
"$categories$" : "$Golf, ActiveLife$","$hours$" : $null$

3.0.2 Review table

reviewid: Contains unique review id
userid: Contains userid
businessid: Contains businessid for business
stars: Contains int value
date: Contains date
text: Contains description

3.0.3 User Table

userid: Contains userid of user
name: Contains name of user
friends: Contains a list of friends id [which is a subset of userid]

3.0.4 Checkin Table

businessid: Contains business id of business
date: Contains a list of dates

3.1 Initial Phase

After having looked at the YELP dataset, we decided to normalize the tables. Initially we had kept only one business table. But that was a design flaw which we rectified in later design. We decided to separate the tables based on the categories. 4 tables we created were Restaurants, Shopping, HomeServices, Beauty and Spa. All other categories will be present in the main table itself. All the attributes were not required. so only attributes which were common for most of the business was kept in the main business table. If any attribute value was mostly null, it was removed. Also, the attributes which were specific to a category was kept in their table. Like RestaurantTakeout is a attribute which is common to restaurant category, so it was used in the restaurants table. Similarly specific attributes were put in the individual tables. Also we did not consider tables which we did not use for our query management systems, like tip and photos.

3.2 Entity Relationship Diagram

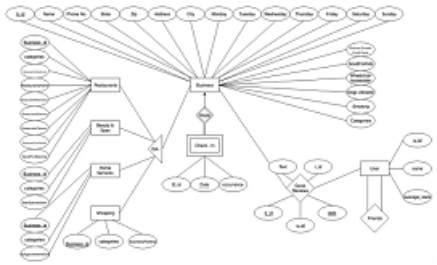

Figure 1: ER Diagram

3.3 Data conceptualization

ER to table conversion:
business ($b_id, categories, name, phone, mon, tues, wed, thurs, fri, sat, sunday, state, zip, address, city, BusinessAcceptsCreditCards, GoodForKids, WheelchairAccessible, DogsAllowed, Smoking$)
$PK : b_id$
$restaurant(b_id, categories, RestaurantsAttire, RestaurantsTableService, RestaurantsReservation, RestaurantsTakeout, RestaurantsPriceRange2, GoodForDancing)$
$PK : b_id$
$FK : b_id$
$beautyspas(b_id, hairsalons, nailsalons, skincare, massage)$
$PK : b_id$
$FK : b_id(b_id, realestate, movers, heatandairconditioning, plumbers)$
$PK : b_id$
$FK : b_id$
$shopping(b_id, fashion, women'sclothing, jewellery, men'sclothing, shoestore)$
$PK : b_id$
$FK : b_id$
$chekin(b_id, date)$
$PK : b_id, date$
$FK : b_id$
$Descriminator : date$
$user(u_id, name, stars)$
$PK : u_id$
$friends(u_id, friends_id)$
$PK : (u_id, friends_id)$
$FK1 : u_id$
$FK2 : friends_id$
$givesReview(b_id, u_id, text, r_id, date)$
$PK : (b_id, u_id, date)$
$FK1 : b_id$
$FK2 : u_id$

4. SECOND PHASE

In this section we made changes to our original ER diagram. Initially we had decided to use the category names as attributes. But we corrected it and took all the categories in a single column which consists of comma separated string values. Next we made sure that our tables were in 3NF and BCNF normalized forms as observed in above section. The next section elaborates the implementation details and challenges.

5. IMPLEMENTATION PHASE

To insert data into SQL database: Python To check for functional dependencies: SQL script To clean data: Pyhton , Pandas SQL Server used: MSSQL Middle Layer: Flask Front end: HTML, Javascript.
First we read the file which was in json format using Python. For each table corresponding script which read the file and stored in a pickle object, such as

```
def readFile():
    records = []
    for line in open("D:\\YELP\\business.json", encoding="utf8", mode='r'):
        records.append(json.loads(line))
    pickle_out = open("business.pickle", "wb")
    pickle.dump(records, pickle_out)
    pickle_out.close()
    print("over")
    return records
```

Figure 2: Read File

The pickle object ensures that once we have read the large file in a pickle object, we wont have to read it again and again. The object stores the entire file contents for easy access.
The script to pre process the file contents is as follows:

```
def checkBusiness_table(records):
    #if "postal_code" in x.keys():
    print(records)
    if records["attributes"]["NoiseLevel"][0] == "u":
        temp =eval(records["attributes"]["NoiseLevel"])
        print("done!",temp)
    if records["postal_code"][0] == "u":
        temp2= eval(records["postal_code"])
        print("done!!!!",temp2)
    if records["attributes"]["Alcohol"][0] == "u":
        temp3 = eval(records["attributes"]["Alcohol"])
        print("done!!!!", temp3)
    if records["name"][0] == "u":
        temp4 = eval(records["name"])
        print("done!!!!!!!!", temp4)
```

Figure 3: Pre Process Data

Similar script was run to check if all the data exists, is not null, or does not consist of invalid literals. If invalid literals exist then remove it and check if some attribute consists of all null values for all records.
Scripts to populate the data is as below:

6. DATA REDUNDANCIES, FUNCTIONAL DEPENDENCIES

We ran Sql scripts to check for various functional dependencies. Basically we checked if more than one attribute could uniquely identify the records and if it is not a part of the primary key. In such a case, we checked for normalization again. This took care of that constraint. Checked if some record did not have primary key , i.e null key using python script. All user should have unique id, if that was not the case we ignored that record since that was faulty data and would not help with the data management. Furthermore we took care that while our implementation design, minimal to no redundencies would exist in our table design. Initially the categories was divided into separate columns.

This was rectified and the categories were stored as a single string value. Also, the user table was divided into two tables. One table consisted of userid,name and average stars and the other friends table consisted of userid and friends id. The friendsid is a list of user ids essentially. This gave us a normalized view. In checkin table, there was a situation from the tables, that many records had same timestamp. This was not ideal, so we separated the records based on the businessid and individual timestamps. We had to make the businessid and timestamp as primary key to ensure the primary key constraint and also to make the table normalized in BCNF. So we added another column to checkin table, namely the occurence. This lkept count of number of times a timestamp occured for a given businessid. This also gave us an idea of the popularity of the business. Also we could use this to find the restaurants or any business popular during a certain time period.

7. MANDATORY QUERIES IMPLEMENTATION

This section includes the screenshots of mandatory queries run on Flask UI as well as on SSMS.

8. QUERY 1

1) List the names of users whose name starts with a given keyword (such as"Phi")and wrote more than 10 reviews.

Query solution:

select name from userTable where name like 'ak%' and u_id in (select u_id from reviewTable group by u_id having count(u_id) > 3);

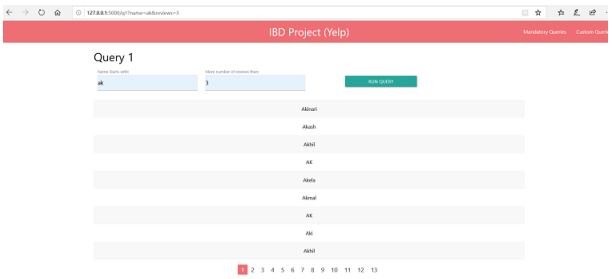

Figure 4: Query 1 UI

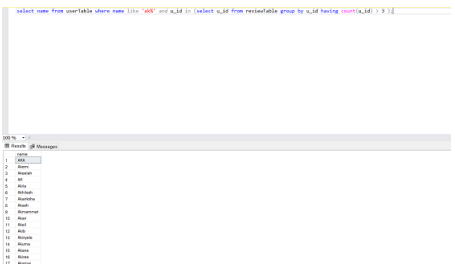

Figure 5: Query 1 SSMS

8.1 Explanation Query 1

Essentially we want to retrieve the names of users from where name is like some text taken input from user and the user should have given more than 3 reviews, as found from the review table.

9. QUERY 2

2) List the names of a given type of businesses(such as "restaurant")located in a given zipcode (such as "89109") and over 50 percent of the received ratings are higher than 3

Query solution:

SELECT tab1.name from (SELECT b1.name, b1.business_id ,count(rev1.r_id) as count1 from businessTable as b1 join restaurantTable as r1 on b1.business$_id$ = $r1.business\_id joinreviewTableasrev1onb1.business\_id = rev1.b\_idwhereb1.postal\_code =' 89121' groupbyb1.business\_id, b1.name)astab1$
$join$
$(SELECTb2.name, b2.business\_id, count(rev2.r\_id)$
$ascount2frombusinessTableasb2$
$joinrestaurantTableasr2onb2.business\_id =$
$r2.business\_idjoinreviewTableasrev2onb2.business\_id =$
$rev2.b\_idwhereb2.postal\_code =' 89121'$
$andrev2.stars > 3groupbyb2.business\_id, b2.name)astab2$

ON tab1.business_id=tab2.business_id WHERE count2/count1 >0.5;

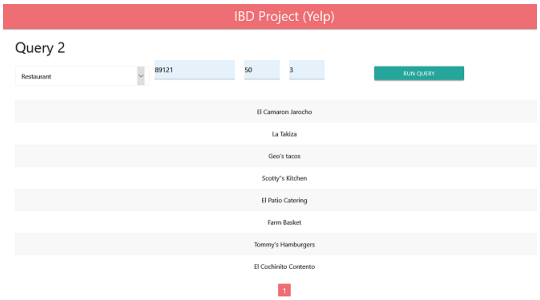

Figure 6: Query 2 UI

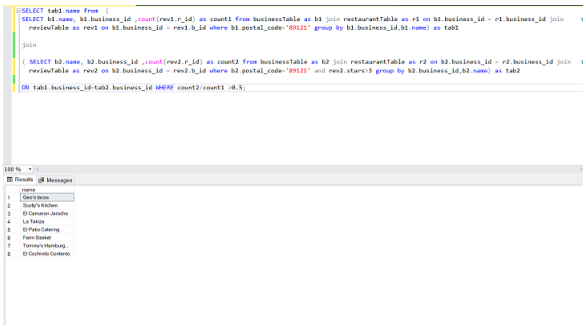

Figure 7: Query 2 SSMS

9.1 Explanation Query 2

Need to access the names of a business based on some category and given the postal code as input based on the condition that ratings is higher than 50 percent. This query requires use of business table, and reviewtable.

10. QUERY 3

3) List the names of a given type of business (such as "restaurant") located in a given zip code (such as "89109"), ordered by their popularity (check-in counts) during a given time period.

Query solution: SELECT businessTable.name from businessTable join restaurantTable on businessTable.business_id = restaurantTable.business_id
join checkinTable on businessTable.business_id = checkinTable.b_id where businessTable.postal_code='89121' and CONVERT(datetime,checkinTable.date) between CAST('2016-01-01' as date) and CAST('2017-01-01' as date) group by businessTable.business_id,businessTable.name order by sum(checkinTable.occurence) DESC;

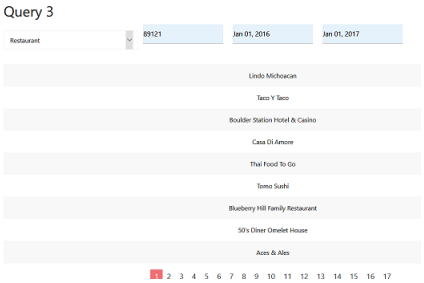

Figure 8: Query 3 UI

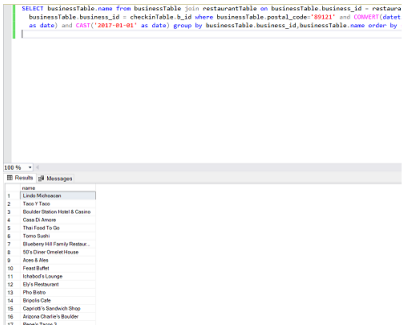

Figure 9: Query 3 SSMS

10.1 Explanation Query 3

The user selects the type of the business, its postal code and a start date and an end date, and then we are displaying the business name from a joint table of business table, selected business type table and checkin table where the postal code is as entered by the user and checkin is between the selected dates and then we are printing the the results ordering it on the basis of the occurrences present in one day.

11. QUERY 4

4) Given a user(using the primary key or choosing from a drop-down list to identify the user), list the restaurants receiving more high ratings (4 or above) than low ratings from his/her friends (3 or below). For example, suppose John has 5 friends and 3 of them rated a restaurant as 4, 5, 2, respectively. The restaurant should then be returned

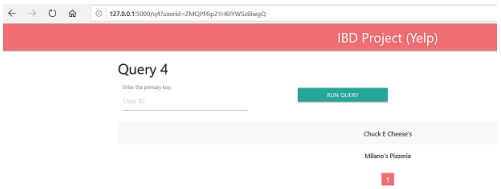

Figure 10: Query 4 UI

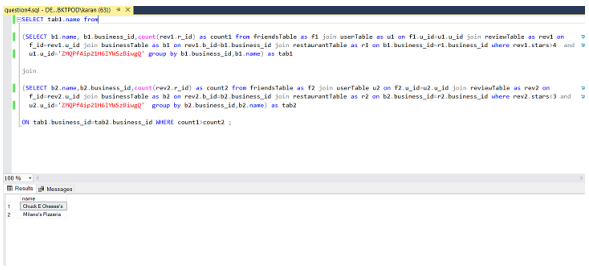

Figure 11: Query 4 SSMS

11.1 Explanation Query 4

First of all we are creating a table where we are selecting the business name, business id and count of review ids as count1 from a joined table of review table, business table and the restaurant table, where the star given by the user is greater than 4 and the user id is as input and the result is grouped by the business name.Then we create another table where we are selecting the name, id and count from the joint table of friends table as count2, user table, review table, business table and the restaurant table and where reviews are less than 3 and the user id is as per the given input and we have grouped it by business id and business name.Then finally we join the above two tables on the basis of business id and select the business names where count1 is greater than the count2.

12. QUERY 5

5) List the unique name pairs of users that had rated the same to a restaurant more than three times. (They are likely to have the same taste)

SELECT tab1.u$_id, tab2.u_id, count(tab1.b_id) as countOfSimilar FROM (SELECT b$_id, userTable.u_id, reviewTable.stars FROM businessTable join restaurantTable on businessTable.business$_id = restaurantTable.business$_id join reviewTable on businessTable.business$_id = reviewTable.b_id join userTable on reviewTable.u_id = userTable.u_id) as tab1 join (SELECT b_id, userTable.u_id, reviewTable.stars FROM businessTable join restaurantTable on businessTable.business_id = restaurantTable.business_id join reviewTable on businessTable.business_id = reviewTable.b_id join userTable on reviewTable.u_id = userTable.u_id) as tab2 on tab1.b_id = tab2.b_id where tab1.stars = tab2.stars and tab1.u_id <> tab2.u_id group by tab1.u_id, tab2.u_id having count(tab1.b_id) > 3;

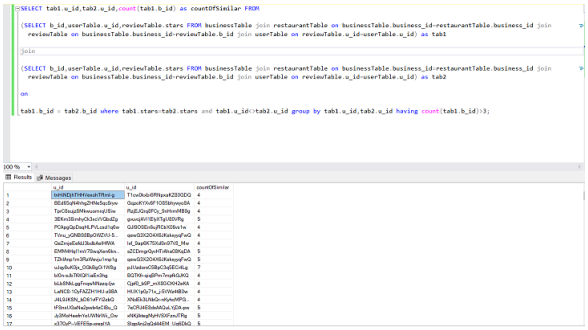

Figure 12: Query 5 SSMS

13. CUSTOM QUERY 1

: 1)

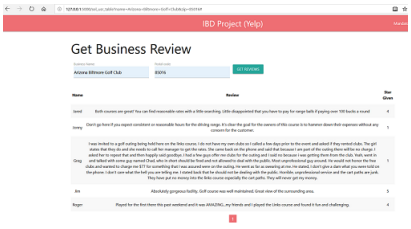

Figure 13: Custom Query 1 UI

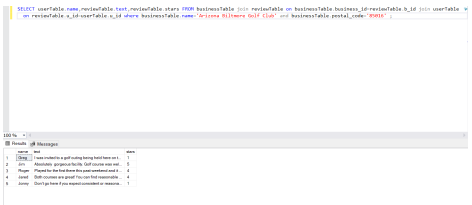

Figure 14: Custom Query 1 SSMS

13.1 Explanation Custom Query 1

We are selecting the reviews, star ratings and the name of the reviewer from joint table of business, review and user tables for a certain business name along with its zip code just to keep the name unique.

14. CUSTOM QUERY 2

2) Order records based on the postal code and categories. orderby desc

14.1 Explanation Custom Query 2

We are displaying all the businesses, its address and its rating on the basis of the inputs given by the user about the business type, zip code and minimum rating that he wants to look for. We are selecting our result from a joint table of business table, the selected type table and review table and we are ordering the result from top rated to least.

15. NOTE:

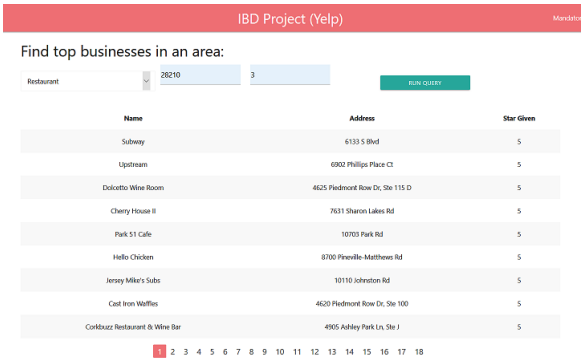

Figure 15: Custom Query 2 UI

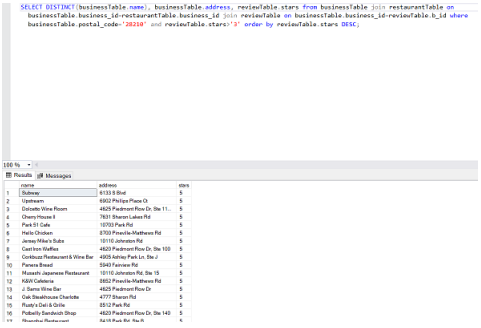

Figure 16: Custom Query 2 SSMS

We have attached some sample screenshots and images in the report, but some of them might be a bit blur. For the reader's convenience we have added the queries folder, screenshots of mandatory and custom queries and python snippets which can be viewed in full screen mode for more clarity. We have tried our best to type the queries by hand so it will be easier to view instead of image screenshot. Zooming in also helps to clear the image a bit. The screenshot folder is the best place to check all good quality and more representative image of input, output and query.

16. CONCLUSION

This was an interesting project to implement. We basically implemented a full fledged web application with pretty descent query support system. In doing so we learnt to a great deal what challenges we face when dealing with big data. Starting with obtaining data, extracting in readable format. Then cleaning data for inconsistencies, null values, invalid literals, duplicate records etc. Checking for Dependencies and inserting the data into database. Becoming familiar with MySQl, MSSQL, Python, Flask and most of all SQL joins! Figuring out how to get them working, conceptualizing and optimizing so that we get the results in less time was crucial. We observed that getting the query right and writing an optimized query is not the same thing when the data in question is this huge and it could have tremendous backlash. We realized this when we ran an unoptimized query, it would take over an hour for insertion or selection query. After modifying it, the same was manageable in less time. So overall we learnt quite a lot from this opportunity.