CSCI620.1 Group2 Phase 1

Shuo Yao
Rochester Institute of Technology
sy9663@rit.edu

Yi Lan
Rochester Institute of Technology
yl8217@rit.edu

Haoran Sheng
Rochester Institute of Technology
hs1911@rit.edu

Jack Smith
Rochester Institute of Technology
jss7268@rit.edu

1. DATA MODELING

This part describes the information of the dataset, justifies the selection of data from the Yelp dataset and presents the design of the ER model.

1.1 Dataset Information

The Yelp Dataset contains following information:

1. `Business` data includes `business ID`, `name`, `address`, `city`, `state`, `postal code`, `latitude`, `longitude`, `stars`, `review count`, `open or closed`, `attribute (including business parking, restaurant take out, etc.)`, `business categories`, `hours`.

2. `Review` data includes `review ID`, `user ID`, `business ID`, `star rating`, `date`, `review text`, `number of useful votes it received`, `number of funny votes it received`, `number of cool votes it received`.

3. `User` data includes `user ID`, `name`, `review count`, `date joined Yelp`, `array of friends' user ID`, `number of useful votes sent`, `number of funny votes sent`, `number of cool votes sent`, `number of fans, the years the user was elite`, `average review star rating`, `number of hot compliments received`, `number of more compliments received`, `number of profile compliments received`, number of cute compliments received, number of list compliments received, `number of note compliments received`, `number of plain compliments received`, `number of cool compliments received`, `number of funny compliments received`, `number of writer compliments received`, `number of photos compliments received`.

4. `Check in` data on a business includes `business ID`, `list of timestamps for each check in`.

5. `Tip` (written by user on a business. Shorter than review, convey quick suggestions) data includes `user ID`, `business ID`, `tip text`, `written date`, `number of compliments received`.

6. `Photo` data includes `photo ID`, `business ID`, `photo caption text`, `photo category label`.

1.2 Data Selection

Our original Data set has information including `business`, `Review`, `User`, `Check in`, `Tip` and `Photo`. After reading this Data Description of data set, we found that the more important among the information is `business`, `user` and `review`. After that, we made some modifications. For example: In `User` Data, because the `review count` can be generated from `review` table, so it is not required. Some attributes, such as `number of votes` and `number of fans`, was not considered to be included in the database since they are not related to our design. In `business` Data, `stars` will cause duplicate information because it can be obtained from the `star rating` attribute from `reviews`. In addition, some information like `latitude` and `longitude`, we think they are redundant information since we have already know their accurate location.

1.3 E-R Model Design

Initially, we only designed three entities, consisting only of: `business`, `review` and `user`. Later, we found that `review` table does not need to have its own unique ID, because the Primary Key of this entity can be `User ID`, `Business ID` and `date`. So, we deleted the unique ID of the `review`. Regarding the business part, we started out as a whole and did not classify it in detail. Later, after many discussions with the professor, combined with the advice given by the professor, we divided the entity into different categories. After researching the data, we specified three business categories: `food`, `shopping` and `beauty`. At this point, these three categories each had their own attribute `food type provided`, `shopping service type` and `Beauty service type`. But their attributes were considered to be too flat. To solve this problem, we use Python to run through all of the business data and then filter out the common and unique attributes among those category:

Unique attributes for `food` category:
RestaurantsReservations, number of occurrences: 66,671
RestaurantsDelivery, number of occurrences: 51,833
OutdoorSeating, number of occurrences: 51,632
RestaurantsGoodForGroups, number of occurrences: 51,385
Alcohol, number of occurrences: 45,349
GoodForMeal, number of occurrences: 29,913
RestaurantsTableService, number of occurrences: 17,168

Unique attributes for `shopping` category:
WheelchairAccessible, number of occurrences: 4,508
DogsAllowed, number of occurrences: 1,892

Unique attributes for `beauty` category:
AcceptsInsurance, number of occurrences: 1,670
HairSpecializesIn, number of occurrences: 1,006

Common attributes among `food`, `shopping` and `beauty`:

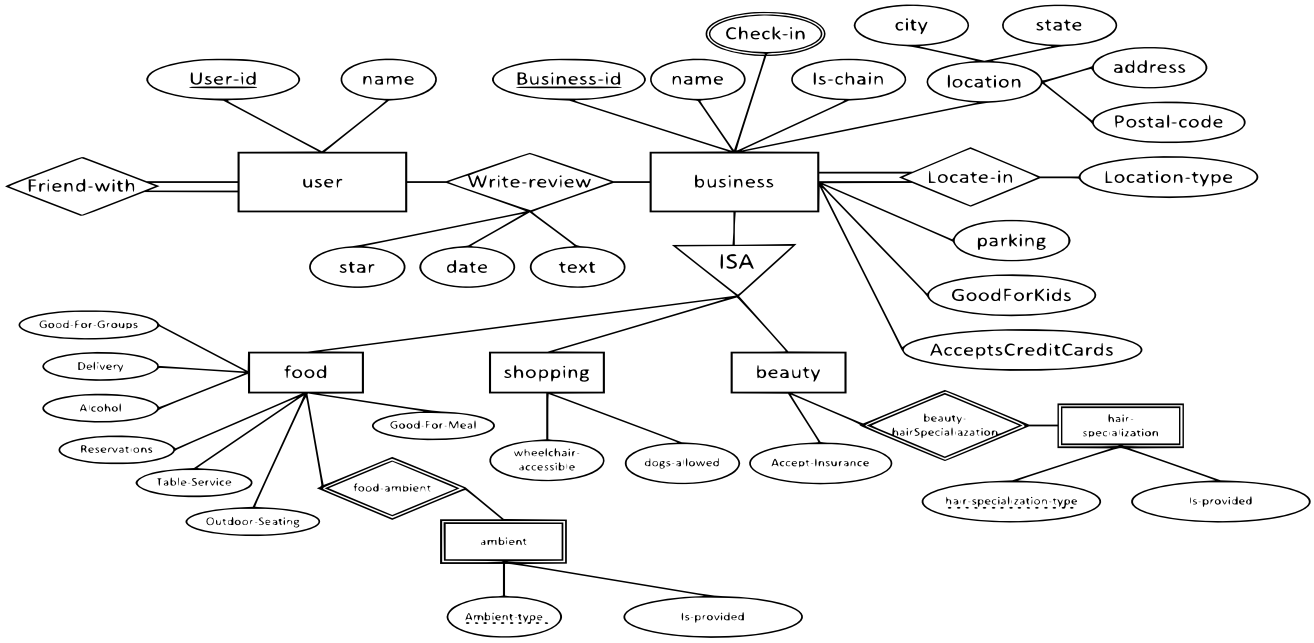

Figure 1: ER Diagram

Parking, number of occurrences: 110,337
GoodForKids, number of occurrences: 51,123
BusinessAcceptsCreditCards, number of occurrences: 43,636

The other thing to be specifically pointed out is the `locate-in` relationship between businesses. This relationship considered the case that an airport or train station can have food and shopping, a shopping mall can have food, shopping and beauty, etc. To be mentioned, special location such as `airport`, `train station` and `shopping mall` are also in the business data. We queried out those businesses and queried other businesses having the same address with them and connect them together. The attribute location type was recorded and saved as an attribute. By doing this, user can find certain type of business in certain type of special locations more efficiently.
Figure 1 shows our final E-R model.

2. DATABASE ESTABLISHMENT

This part describes the creation and population of the database.

2.1 Data Import

We use JavaScript in SQL shell8.0 to parse the original data materials. The files are then stored in SQL server as native supported JSON document. We utilize the SQL server's ability to search within a JSON object to directly query its content and convert the query result into tables that we desire. As for the composite JSON object, we unwrap the data into JSON objects and normal columns for further process and analyses. After we have a satisfying table design, the redundant data are dropped from the table, whereas useful data are moved into our final working tables by `insert...select` queries. In the light of 'noise' in the data set, we first identify them by trying to create candidate keys in our table. The rows showing complete duplication are considered noise and are removed. Moreover, some extremely weird rows that display abnormal behaviors under our model are considered as noise. For example, User data with a valid id but null in the name field is considered noise.

2.2 Partitioning

For some tables with a large amount of data (such as review table), we split it horizontally into multiple tables. Take the Primary Key to do the hash. For example, there are `Business ID`, `User ID`, `Date`, serve as primary key in the review table. Concatenate those fields together into a long string and use the hash function to find the table where the corresponding data is located. Because the hash function that is known to the SQL server, thus when a query is send to the partitioned table the server automatically knows which partition or partitions it should work on, if the given query searches among the primary index in some manner.

2.3 Normalization

The first step in normalization is to get our relationships into 1st Normal Form. To get the schema into 1st Normal Form, we also had to ensure that only single values were present in each row. This required splitting the categories of businesses into several 'is-a' relationships. These include `beauty`, `food`, and `shopping`, to start. The business attributes also needed to be turned into single-value attributes to get into 1NF. This required creating columns for different attributes in either the general business table, or in the specific 'is-a' tables. Many of these were simple boolean values, but some required the addition of weak entity sets. These were the `Ambience`, and `HairSpecializesIn` because they contained json defining whether a business had any of many attributes (such as a classy, romantic, casual, and/or etc. ambience).

To become 2NF, aggregate and duplicate values from the data were removed. This included values such as the number of stars that businesses have and the number of reviews that

a user has. Storing these attributes, that can be calculated from other data, leads to inconsistencies that shouldn't be in the database. Since most candidate keys were unique id values, nothing needed to be done to ensure that there were no partial dependencies.

To become 3NF the transitive dependency of `business_name`, determining whether a business was a chain, had to be removed. This was achieved by creating a table that stored whether a business name was a chain, as chain status was calculated based on whether there were multiple distinct business with the same name.

The only overlapping candidate key was for the `business` table, wherein knowing the `business_name`, `address`, `state`, `city`, and `postal-code` would also be a super key that determined anything that the `id` itself would determine. To convert into BCNF, a new relation called `business-location` was created that separated location attributes from the rest of the business attributes. This made the `business` table smaller, and also converted it into BCNF at the same time. The `business_location` table uses `name`, `address`, `city`, `state`, and `postal_code` as the primary key, and has `id` as a surrogate key to facilitate joins. The `name` in this table is also foreign key to the `business_chain` table.

By making `name`, `address`, `city`, `state`, and `postal_code` the primary key, we ensure that a name, location uniquely determines a business so there are no duplicate businesses in the system. This concept was applied to combine or remove duplicate rows from the system.

3. QUERY SYSTEM DEVELOPMENT

We implemented the basic query scenarios that the professor provided for the Yelp data set. In addition, we also developed 4 additional, complex query scenarios that aim to demonstrate the quality of our relational schema design.

Those 9 query scenarios are as follows:

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

2.List the names of a given type of businesses (such as "restaurant") located in a given zip code (such as "89109") and over 50% of the received ratings are higher than 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.

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.

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).

6.List the user that gives higher average rating on certain type of business (food, shopping) than a different type of business (food, shopping, not the same). On a given postal code(area).

7.List the name of users that have recently (from time range) reviews given type of business in certain type of special location (airport, shopping center, train station).

8.List the business in certain city has rating greater than certain value.

9.Return the difference in ratings between the chain business and not-chain business on certain business type and certain location (specify city).

3.1 System Overview

Our overall logic is: User select through a series of dynamic drop-down list to a certain point that refers to a certain query scenario. If the user determines to search at this point, he can click on the Query button to execute the query on this certain query scenario. Of course, the actual number of selections in the dynamic drop-down list will be more than our actual number of queries, because there is a permutation combination among the options. (For example, I chose an option on the first level, there will still be several options on the next level...) The structure of selection tree avoids the user's choice to go beyond the optional range, which means that the user can only select the query we set in the GUI interface. So there is no exception handling. For each query, we first came up with the selection options in the query, and then generate a selection tree based on the options and the order of options, which is a special point of our design. The advantage of this is that we never let the user pick something we can't query. This maintains the system stability. On the presentation layer, because we are the tree generated this way, this gives each level of the tree a name. So we only need to display the corresponding name on the corresponding option.

Our preliminary design for the user interface is to use a series of hard-coded structure with `else-if` statement for query selection. However, if we use such design, the maintenance of the code can be very difficult and it's not flexible. Adding a new query to this system in the future will be very difficult.

Based on this situation, we conducted some researches and came up with a tree structure. This tree structure can automatically generate a search tree each time when it is initialized, which improves the stability of the system and makes maintenance very easy. Adding a new query scenario will be really easy since the drop-down list options are updated to the original tree. This improvement also avoids using a large number of `else-if` statements. One drawback of this structure is that it doesn't support `step-back` operations. If the user chooses the wrong query option in the selection process, in this case, the root node of the tree cannot be returned. User can only click the reset button to come back rather than go directly back to the previous level.

3.2 Implementation

The overall process is: Each node will present a method called `acceptUserInput()`, which is used to receive a user input on the GUI. The statement is executed after the corresponding node is found. After the execution ends, it is presented to Java in the form of a resultSet (the result of the query in the database is presented in bytecode) on the back end. Next, the `processResult()` method is called to convert the result to a string again. Finally, the `display()` function is called to display the result on the GUI.

Our index is added after the first version of query was written based on the actual needs. The index improves the performance of the system by greatly improving the retrieval speed. However, adding an index takes up more memory space and the index also needs dynamic maintenance when adding, deleting, and modifying data in the table, which increases the time required for maintenance. So we didn't

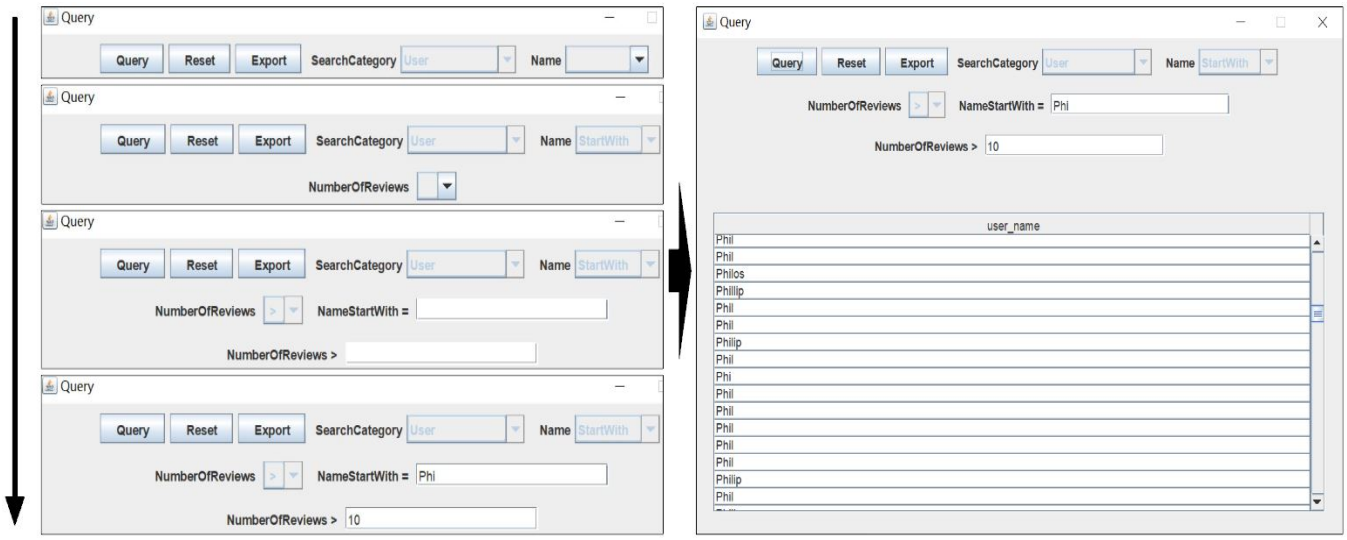

Figure 2: UI Operation Demo

add an index to all the tables.

For the ease of query, we divided the review table into ten tables. Because it's difficult to build a cross-table index and foreign key, we tried to put all the data in one table at first, but we found that the query performance is poor and the index will exceed the maximum acceptable size. This will greatly improve the efficiency of the query but the disadvantage is that the review table cannot create foreign keys (other tables have foreign keys).

3.3 Query System Demo

Figure 2 presents a operation demo for our user interface. User can specify the query they want to select by selecting from the dynamic drop-down list. If the query scenario requests input variables, user can input into input box. During this process, if the user make mistakes, by clicking the `Reset` button they can initialize the dynamic drop-down list and input box. When the query selection and variable input is done, by clicking the `Query` button will start running the query. When the query is done, the result will be presented below (the result window for query scenario 1 is used here). User will be able to export the result by clicking `Export` button. The result window for the rest 8 query scenarios is showing in Figure 3.

4. LIST OF DELIVERABLE

This section describes the list of deliverable in the submission. The README.md file under main folder provides overview and operation guides for this project. Please also see README.md in each folder for more detailed information.

1. Folder name: MySQLScript

MySQL script for creating the schema and tables, and importing data.

2. Folder name: Python

Python files populating the normalized tables with data in order to complete the normalization process. The MySQL scripts in this directory provide a view of create table commands that were used to generate certain tables that these python scripts operate on.

3. Folder name: ERmodelingScript

Python file which reads through all of the business data and then filter out the common and unique attributes among category 'food', 'shopping' and 'beauty'.

4. Folder name: Queries

MySQL script for the 9 query scenarios.

5. Folder name: src

The implementation package for the query system.

5. WORKLOAD DISTRIBUTION

In the workload distribution part, the 'Main in-charge' people will take the lead, other group members will assist them and provide feedback.

Step1: ER model design. Main in-charge: Shuo Yao, Yi Lan, Haoran Sheng.

Step2: Table implementation, data import script. Main in-charge: Yi Lan.

Step3: Data validation and normalization. Main in-charge: Jack Smith.

Step4: Use case design and corresponding database indexing. Main in-charge: Yi Lan, Shuo Yao.

Step5: Interface core logic development. Main in-charge: Yi Lan, Shuo Yao.

Step6: Interface GUI design. Main in-charge: Haoran Sheng.

Step7: Report and demo preparation. Main in-charge: Shuo Yao, Jack Smith, Haoran Sheng.

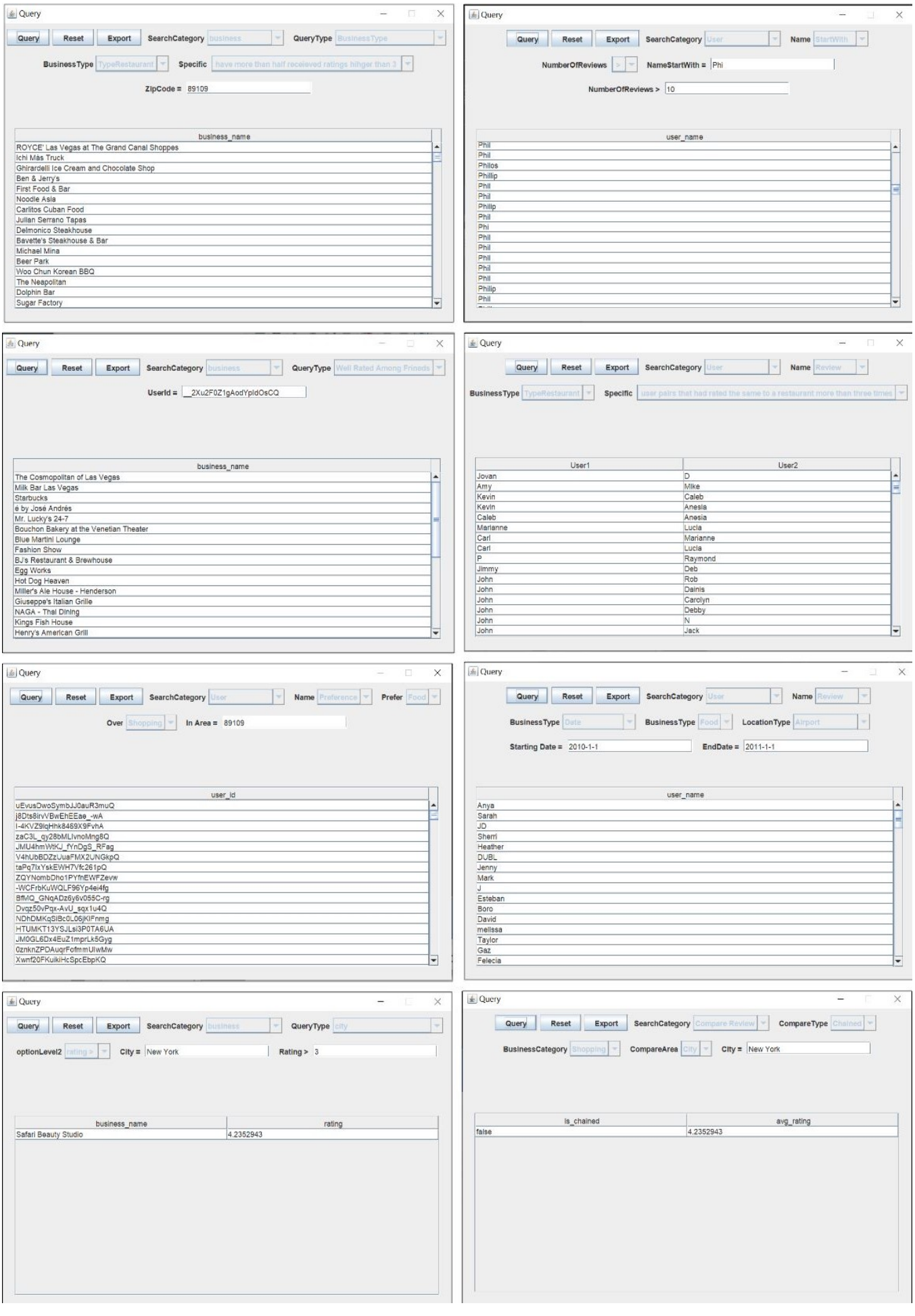

Figure 3: UI Operation Demo