Local Eats

Celeste Nobrega & Pierce Gillim

Databases Final Project Report

Introduction

Our application we modeled a food delivery service called LocalEats. We chose this application because of the current state of the world in the COVID-19 pandemic, we have seen many small businesses and restaurants going out of business because people don't feel safe dining in or leaving their homes to support these businesses. Therefore, we have created an application that helps these small "mom & pop shop" restaurants stay afloat and fill orders for people who cannot physically come to the restaurant. This application has restaurants, users, and employees (drivers) and allows for users to order from multiple restaurants, order together, and get their order delivered to their house by the drivers. This application only services small restaurants and no large chains because the large chains often already have their own delivery services set up or have deals set up with the other food delivery apps that makes them show up more often in searches. Our app gives every restaurant a fair and fighting chance to make it through the pandemic financially.

Our app is different from other food delivery apps because we only service small businesses and we allow for collaborative ordering. The importance of only servicing small businesses was discussed above. The unique functionality of collaborative orders involves allowing multiple users to place an order together allows restaurants to fulfill bigger orders and make more money while also giving users an easier way to split the bill. Collaborative orders also lets multiple restaurants fulfill one order, allowing customers to not have to limit their choices when ordering.

Pierce and Celeste split up the work as evenly as possible. In the beginning, we collaborated in person as much as possible on the proposal, ER diagram, and initial schema. From there, we split off into working on different aspects of the project. Celeste wrote the rest of the database schema and changed it as was necessary as we discovered different design changes that needed to be made. Celeste also worked with Mockaroo to generate, import, and perform manual quality control of the data. Pierce did the writing portion of the progress report as well as the implementation of the Python front end. We then collaborated on the powerpoint slides as well as the final report. Celeste created the GitHub at the very end to bring everything together in a neat and organized way (link in Implementation section).

Database Details

We designed the database to have 6 tables: users, driver, restaurant, orders, alert, and menu. The users table contains all user information such as an ID, their name, and their address. The driver table details their ID, their name, their car's make, model, and license plate. The

restaurant table has the restaurant's ID, address, name, and category. A restaurant's category is a descriptor of what kind of food they serve, such as Fast Food, Chicken, Burgers, Desserts, Sandwiches, Mexican, Breakfast, Pizza, Healthy, Italian, Japanese, Asian, Chinese, Vegan, Steak, Sushi.

Orders is a table that keeps track of every order made on the app and lots of information about the order. Each order has an ID, the user who ordered it, the time it was ordered, the time it was delivered, the item ID that was ordered, the quantity of that item, the unit price of that item, the extended price (which is price * quantity), and the restaurant that filled that order. It should be noted that orderID is not unique in this table because an orderID can be repeated for each restaurant that is fulfilling a part of the order or each user that is a part of the order. Alert is a table that tracks when drivers are alerted of a new order that needs to be picked up. All drivers are alerted, but only one driver can pick the order up. The alert table contains the ID of the order that was placed, the driver that picked the order up, the time the driver was alerted and the time the order was picked up.

Lastly, the menu table is the different menus of the restaurants. Each menu has a unique ID from 1-200, each item in the table has a unique ID that is 4 digits, and each restaurant can have multiple menus, but each menu ID can only be assigned to one restaurant. Each menu has multiple items so menuID is not unique in the table, but itemID is unique. In summary of the menu table, each menu has one restaurant but multiple items and each restaurant has multiple menus. The menu table also has a type attribute that describes the menu such as Breakfast, Lunch, Dinner, or All Day. Lastly, each item in the menu table has a price.

Our E-R diagram can be seen in Figure 1 and our relational model can be seen in Figure 2. The main notes about the ER diagram are that everything has a many to many relationship except for the relation between menu and restaurant which is a many to one. Only one menu can be assigned to a restaurant, but a restaurant can have multiple menus. The relational model is as described above in terms of the meanings of attributes. The primary keys are as such due to the notes about uniqueness made above as well.

In the users table, the only functional dependency that exists is ID → (address, name) and in this way is it in BCNF. The driver table has functional dependency ID → (name, carMake, carModel, license) and is in BCNF. Restaurant has has one FD: ID → (address, category, name) and in this way, the restaurant table is in BCNF. The menu table has the following FDs: (menuID, itemID) → (restaurantID, price, type) and itemID → price, thus the menu table overall is in 3NF. The alert table has one FD: (orderID, deliveredBy) → (alertTime, pickupTime) and is in BCNF. Lastly, the orders table has 3 FDs: (orderID, itemID, orderedBy) → quantity, orderTime, delieveryTime), itemID → (fulfilledBy, price), (quantity, price) → extendedPrice and is in 3NF. There is additionally a NOT NULL constraint on all data, since there are no attributes that make sense to be NULL in our database.

Another important feature of our database is how we generated our data. We used an online tool called Mockaroo. Mockaroo is a very powerful tool that allows a user to create several schemas and upload datasets to make mock data. There are 145 different types of data

mockaroo can create, on top of being fully customizable using the Ruby language, SQL Expressions, formulas, and more. Figure 3 shows a table that explains how we generated this data using Mockaroo.

Figure 1

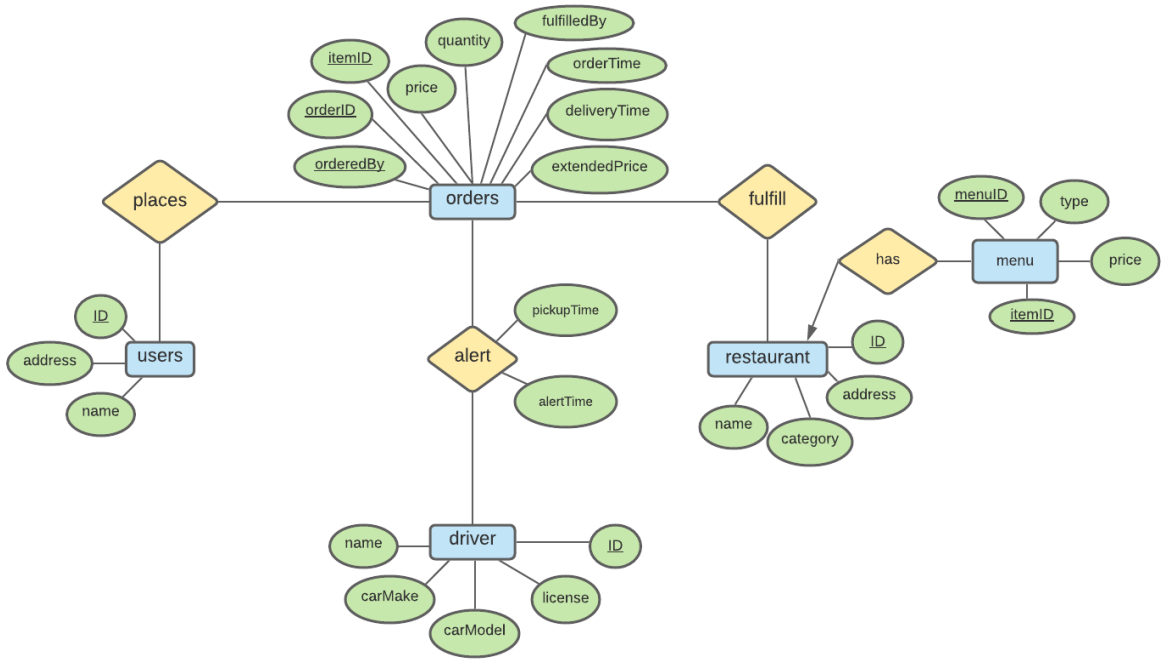

Figure 2

```
CREATE TABLE "users" (
	"ID" INTEGER NOT NULL UNIQUE,
	"address" TEXT NOT NULL,
	"name" TEXT NOT NULL,
	PRIMARY KEY("ID")
);
CREATE TABLE "driver" (
	"ID" INTEGER NOT NULL UNIQUE,
	"name" TEXT NOT NULL,
	"carMake" TEXT NOT NULL,
	"carModel" TEXT NOT NULL,
	"license" TEXT NOT NULL,
	PRIMARY KEY("ID")
);
```

```
CREATE TABLE "restaurant" (
	"ID" INTEGER NOT NULL UNIQUE,
	"address" TEXT,
	"category" TEXT,
	"name" TEXT,
	PRIMARY KEY("ID")
);
CREATE TABLE "menu" (
	"menuID" INTEGER NOT NULL,
	"type" TEXT NOT NULL,
	"price" REAL NOT NULL,
	"itemID" INTEGER NOT NULL UNIQUE,
	"restaurantID" INTEGER NOT NULL,
	FOREIGN KEY("restaurantID") REFERENCES "restaurant"("ID"),
	PRIMARY KEY("itemID","menuID")
);
CREATE TABLE "orders" (
	"orderID" INTEGER NOT NULL,
	"orderedBy" INTEGER NOT NULL,
	"delieveryTime" TEXT NOT NULL,
	"orderTime" TEXT NOT NULL,
	"itemID" INTEGER NOT NULL,
	"price" REAL NOT NULL,
	"quantity" INTEGER NOT NULL,
	"extendedPrice" REAL NOT NULL,
	"fulfilledBy" INTEGER NOT NULL,
	FOREIGN KEY("fulfilledBy") REFERENCES "restaurant"("ID"),
	FOREIGN KEY("orderedBy") REFERENCES "users"("ID"),
	FOREIGN KEY("itemID") REFERENCES "menu"("itemID"),
	FOREIGN KEY("price") REFERENCES "menu"("price"),
	PRIMARY KEY("orderedBy","orderID","itemID")
);
CREATE TABLE "alert" (
	"orderID" INTEGER NOT NULL UNIQUE,
	"deliveredBy" INTEGER NOT NULL,
	"alertTime" TEXT NOT NULL,
	"pickupTime" TEXT NOT NULL,
	FOREIGN KEY("orderID") REFERENCES "orders"("orderID"),
```

```
    FOREIGN KEY("deliveredBy") REFERENCES "driver"("ID"),
    PRIMARY KEY("orderID","deliveredBy")
);
```

Figure 3

User — 200 rows		
Attribute	***Type***	***Notes***
ID	Row Number	
address	Street Address	
name	Full Name	
Driver — 200 rows		
ID	Row Number	
name	Full Name	
carMake	Car Make	
carModel	Car Model	
license	Car VIN	`this[0,6] #only uses the first 6 digits of the VIN`
Restaurant — 100 rows		
ID	Row Number	
address	Street Address	
category	Custom List [Fast Food, Chicken, Burgers, Desserts, Sandwiches, Mexican, Breakfast, Pizza, Healthy, Italian, Japanese, Asian, Chinese, Vegan, Steak, Sushi]	Random allocation, but then manually re-allocated based on restaurant name so that the categories were meaningful.

name	Dataset Column	Pulls from a list Celeste uploaded as a dataset that contains 100 restaurant names. Then manually edited to include actual local restaurants as well as removing large chains.
Menu — 500 rows		
menuID	Number	1 to 200, no decimals. There are 500 rows and only 200 menu IDs so that menus can have more than one item.
type	Custom List [Breakfast, Lunch, Dinner, All Day]	Random allocation, and then manually re-allocated based on the restaurant category to ensure meaningful data.
price	Number	1 to 20 with 2 decimal places
itemID	Number	1000 to 9000, itemID is always a 4 digit number
Orders — 627 rows		
orderID	Row Number, it should also be noted that there are 300 order IDs but 627 rows because orders are placed by 1 user, 2 users, or 3 users where each user has a new row but same order ID.	`this + 100000 #orderID will always be a 6 digit number`
orderedBy	Dataset Column	pulls from “user” dataset, at random
deliveryTime	Time	from 12:00 AM to 11:59 PM, 24 hour format
orderTime	Formula	`delieveryTime-minutes(random(10,30)) #Guarantees that the order time will be a random amount of minutes (10 to 30) before the delivery time`
itemID	Dataset Column	pulls from “menu” dataset, at random.

price	Dataset Column	pulls from “menu” dataset. mockaroo already knows that itemID and price are from the same table, so the correct price will be associated with the appropriate itemID (I checked to guarantee as well)
quantity	Number	1 to 20 no decimals
extendedPrice	Formula	`[price * quantity]`
fulfilledBy	Dataset Column	pulls from “menu” dataset, the restaurant ID associated with the appropriate item ID.
Alert — 300 rows		
orderID	Dataset Column	pulls from “order” dataset, sequentially so that all orders get alerted.
deliveredBy	Dataset Column	pulls from “driver” using “driverID”
alertTime	Dataset Column	pulls from “orders” using “orderTime”
pickupTime	Formula	`time(date(from_dataset("order", "orderTime", orderTime: alertTime)) + minutes(random(2,10))) #ensures that the pick up time is 2 to 10 minutes after the alert time`

Functionality Details

One of the most basic functions that we implemented into our project was the usage of INSERT, UPDATE, and DELETE into our python front end. To create this, we create a main menu output that the user can interact with. If the user input “2” then a new prompt will appear asking if the user wants to make an insert, update or deletion. For explanation purposes, we will explain how an insertion is made. When the user inputs “1” the program knows an insertion is coming. The program will then prompt the user for what table they would like to insert to. The program will then check the table names to make sure the user inputted a valid name. Then, the

program will prompt the user to input each attribute of the entity they are interested in inserting. The program will then concatenate the proper SQL query, execute it, and then let the user know the insert was done. Please see Figure 4 below for a diagram of this entire process. In the future, we would like to have a confirmation output of the last inserted ID into the database. This will make it more user friendly. On top of this, only the insert function has implemented the asking for each attribute line by line. Delete and Update work in a non-user-friendly way as exemplified in Figure 5.

Figure 4

```
+----------------------------------------+
Welcome to the LocalEats Database!
What would you like to do today?
+----------------------------------------+

+----------------------------------------+
Main Menu:
Enter 0 to exit
Enter 1 to print out a table
Enter 2 to insert/delete/update into a table
Enter 3 to enter a custom SQL command
Enter 4 to go to the statistics menu
+----------------------------------------+
Your input: 2
```

```
Enter 1 for insert, 2 for delete, and 3 for update: 1

Enter the table you want to insert into: driver
```

```
Please enter a name: Pierce Gillim
Enter a car make: Toyoda
Enter a car model: RAV4
Enter a license plate number (405ZX2): H169GQ

Insertion committed
```

Figure 5

```
Enter 1 for insert, 2 for delete, and 3 for update: 3

Enter a table you want to update: driver
Enter a SET clause argument(s): carMake = 'Honda'
Enter a WHERE clause argument(s): name = 'Pierce Gillim'

Update committed
```

```
Enter 1 for insert, 2 for delete, and 3 for update: 2

Enter the table you want to delete from: driver
Enter a WHERE clause argument(s): name = 'Pierce Gillim'

Deletion committed
```

One of our most advanced features implemented was the statistics menu. When entering 4 at the main menu, the statistics menu will then pop up on screen. There are another 4 options in the statistics menu, show the top 10 most popular restaurants, show the top 10 most popular restaurant categories, show all the users who have made an account but have not ordered anything, and then enter a custom statistic. One of the most time consuming parts of this section was creating meaningful statistics that people most users will want to know for a given area. It also gives restaurants ideas on who to advertise their menu to, as they can see any user who has not ordered before. We specifically chose these statistics because it allows the user to meaningfully interact with the data. One note about the 3rd option is that the SQL statement used to create the report originally had “ANTI JOIN” in it. However, in SQLite, this is not possible so NOT IN was used instead. Please see Figure 6 below to see the statistics menu.

Figure 6

```
+---------------------------------------+
This is the statistics menu!
Enter 1 for the top 10 restaurants that fulfilled the most orders
Enter 2 to look at all the users that have an account, but have not ordered anything
Enter 3 to look at the top 10 most popular restaurant categories
Enter 4 to enter a custom SQL query
+---------------------------------------+
Your Choice:
```

Here are the top 10 most popular restaurants

orderCount	restaurantName
29	Garden Grille
20	Mooyah
20	Puebla By Night
17	AJ's Stone Oven
17	Jersey Mike's Subs
16	Muskie Lounge
15	Grady's Restaurant
15	Blue Moon Thai
15	Syracuse Housing Authority Snack Bar
14	Plant City

Here are all the users who have an account, but have not ordered anything

customerID	customerName
22	Kendra Sybry
30	Maurice Canepe
32	Hurley Aymeric
58	Maribelle McKeran
64	Vikki Yair
85	Tess Burdfield
94	Damara Chave
115	Linda Spivie
148	Gabriella Rozanski
286	Pierce Gillim

Here are the top 10 most popular restaurant categories

orderCount	category
78	Vegan
73	Breakfast
70	Mexican
52	Chicken
48	Italian
48	Asian
45	Burgers
39	Chinese
36	Sandwiches
30	Japanese

Implementation Details

For our front end interaction with the database, we have chosen to use python. Specifically, python with an IDE of Wing 101 was used. This was found to be easiest for us as we both feel comfortable working in python, as well as python being one of the easiest ways to connect to the database. The main idea of our front end is to have a user interface that allows them to pick/create SQL statements to interact with the database. This is the main reason why our front end is structured in such a way that a user can enter a number or a name of a table, and does not have to think about the code behind it. The main menu is an example of this. Figure 4 shows what the main menu looks like.

If the user enters the number '1' into the main menu, the user will then be asked which table they would like to look at. The user will then input a table name and the program will check if that table name actually exists. If valid, then the program will print out the entire table the user wished to see. If the table name does not exist, then the user is kicked out of the print function and back into the main menu. This simple process does not make the assumption that the user will know how to use SQL. The python takes care of all the SQL while the user just has to know the table names. This process is shown in Figure 7.

Figure 7

```
+----------------------------------------+
Main Menu:
Enter 0 to exit
Enter 1 to print out a table
Enter 2 to insert/delete/update into a table
Enter 3 to enter a custom SQL command
Enter 4 to go to the statistics menu
+----------------------------------------+
Your input: 1


+----------------------------------------+
Which table would you like to see?
Please enter the name of the table you wish to look at.
Your input: alert
```

```
+----------------------------------------+
Which table would you like to see?
Please enter the name of the table you wish to look at.
Your input: alerts


Invalid table name. Exiting to main menu...
```

```
if (valid == True):
    query = "SELECT * FROM "
    query = query + choice

    cur.execute(query)

    names = cur.description

    for name in names:
        print("{: <30}".format(name[0]), end="")
    print()

    for row in cur.fetchall():
        for item in row:
            if item != None:
                print("{: <30}".format(item), end="")
            else:
                print("{: ,30}".format("None"), end="")
        print()
    print()

else:
    print()
    print("Invalid table name. Exiting to main menu...")
    print()
```

The other options of the main menu have more flexibility to them. The 3rd option in the main menu is to enter a custom SQL command. This option obviously makes the assumption that the user knows what SQL is and how to write a query. The program then tries the user inputted query in a try/except block just to catch any errors that the person might have. Then, the program prints out either the errors or the result of the user submitted SQL command. The goal for this section was to give the user full access to the database. Obviously this is quite an advanced feature and would normally be used for just administrative purposes.
The following is a link to our github repository where our code and other files can be found.
link: https://github.com/cgnobrega/LocalEats.

Experiences

We learned how to effectively go from an ER diagram to a relational model. In the past we thought you would just make a table for every entity in the diagram, but through working on it and getting feedback, we realized that this approach would lead to a fair amount of data redundancy as well as bad table names. We also learned how to use a new tool and how to import large amounts of data into DB browser. Through the process of managing very large tables and large amounts of data, we also learned valuable lessons about primary keys and meaningful data.

Through the development of a Python front end, we learned how to connect a database to Python and learned about what characteristics contribute to user friendly interactions. This application has evolved significantly over the course of the project. We learned that some

methods of data entry would limit our functionality (such as the menu) so we made new tables, which resulted in the loss of other tables (ratings) in order to uphold other functionality and keep the number of tables we were managing to a minimum.

In the future, we would like to add back more functionality that we didn't have time to include here as well as even more descriptive data to create a more overall interactive application. There are some details that are lacking in the database, but if we were to add those details, it would have complicated the whole application too far to be able to successfully complete the project. For example, the user would have to know order IDs, restaurant IDs, item IDs, etc to place an order in the order table; but in order to make the database more user friendly, that would be hundreds of rows of more descriptive text attributes that would have to be added such as actual item names and menu names etc.

Overall, it has been really enjoyable to work on this project and learn more about the scope and power of tools as simple and easy to use as DB Browser, SQLite, and Python.

References

Brocato, Mark. *Mockaroo*. Mockaroo, LLC., 7 February 2015. Software.
<https://www.mockaroo.com/>.