FOOD DELIVERY SYSTEM

CS5200 Project report

Dheekksha Rajesh Babu
Adithya Narayan B

Technical Specifications

MySQL Workbench Version: 8.0.19
Java Version: 11
JDBC Connector Version: 8.0.22
MySQL Version: 8.0.19
Additional Libraries used: None

This project is a CLI based Food Delivery Database Management System built using Java and MySQL. We use the JDBC connector in order to store, retrieve and update data belonging to the different users of the application.

README

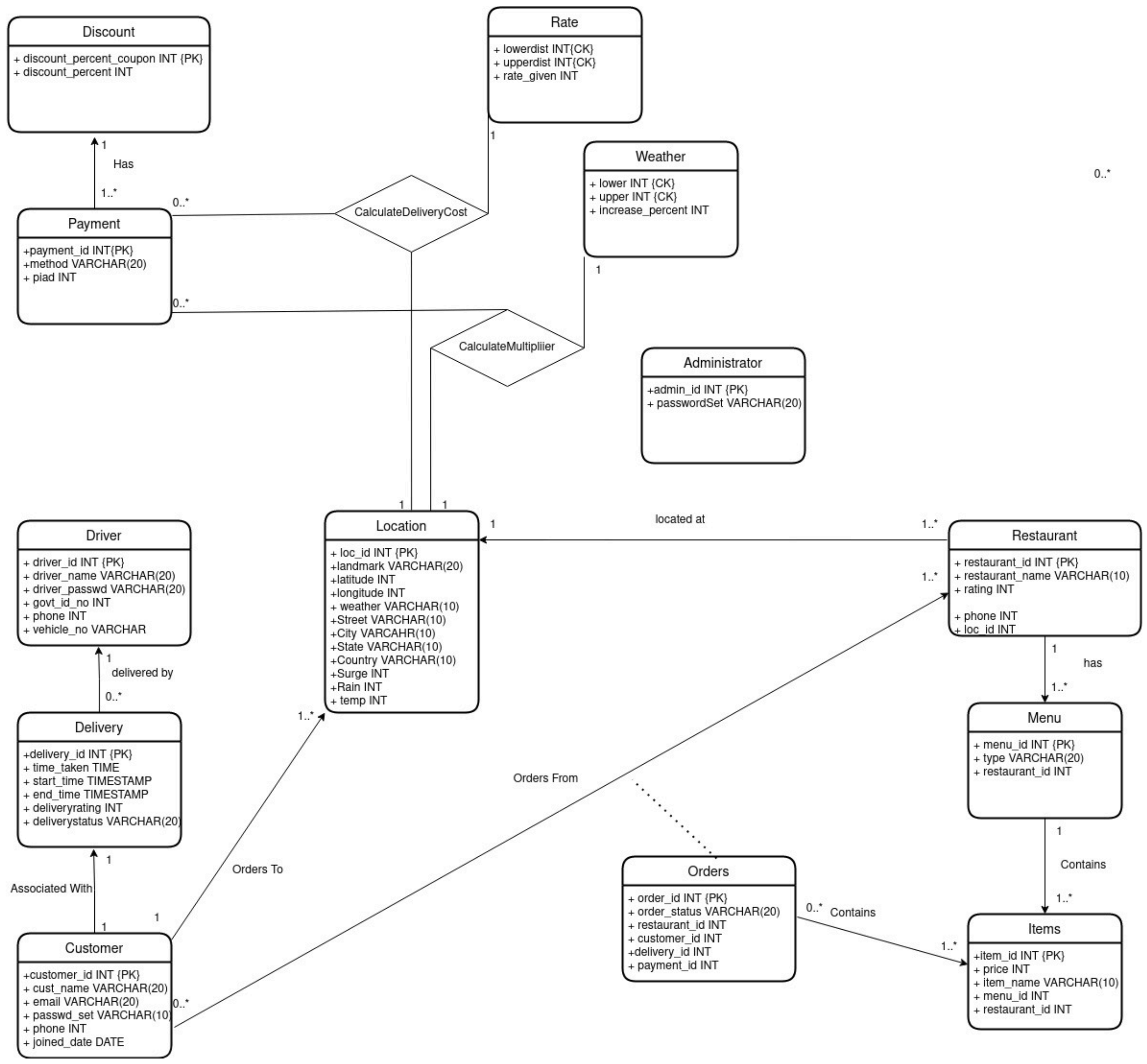
Steps to run the project:
The source code provided in the directory can be run as is, given that the target systems JDK is at version 11 and the JDBC driver is installed and configured correctly.

CreatingDB.sql is a sql file that needs to be executed in the MySQL workbench tool. This creates the database and the tables necessary for the code to work.
createFunctionalities.sql is a sql file that creates all the functions, procedures and triggers necessary to provide all the functionalities that the user wants. This script also creates and populates a set of tables with starter data, that can be used by the user to test and run functionalities independently. This also needs to be run in the MySQL workbench tool.

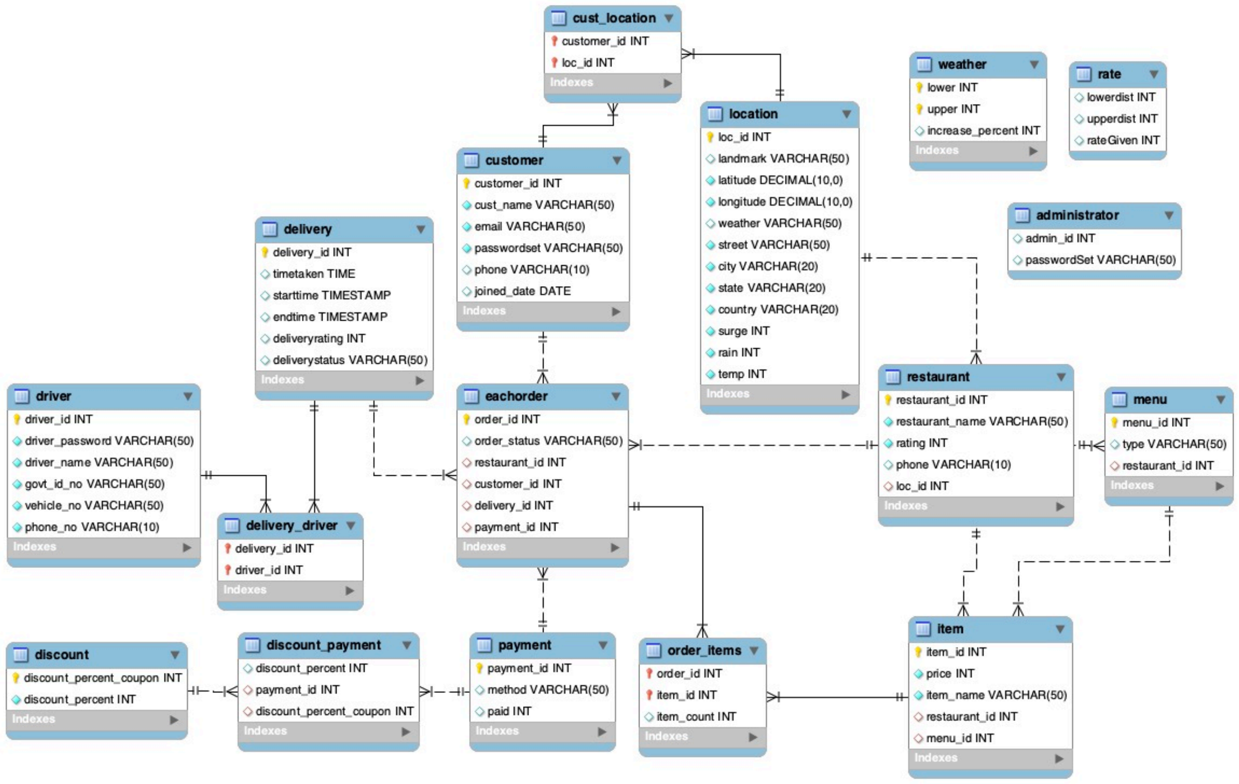
The CLI application can be launched on the target system by opening deliverysystem.java in an IDE, and running it like any other java program(Press the green Arrow mark).

- If the database dump is used then just running the dump is sufficient for creation of all tables, procedures, functions, triggers and the data. Just running this script and then the java file is sufficient.

UML


Discount
+ discount_percent_coupon INT {PK}
+ discount_percent INT
Rate
+ lowerdist INT{CK}
+ upperdist INT{CK}
+ rate_given INT
Weather
+ lower INT {CK}
+ upper INT {CK}
+ increase_percent INT
Payment
+payment_id INT{PK}
+method VARCHAR(20)
+ piad INT
CalculateDeliveryCost
CalculateMultipliier
Administrator
+admin_id INT {PK}
+ passwordSet VARCHAR(20)
Has
1
1..*
0..*
0..*
1
1
0..*
Driver
+ driver_id INT {PK}
+ driver_name VARCHAR(20)
+ driver_passwd VARCHAR(20)
+ govt_id_no INT
+ phone INT
+ vehicle_no VARCHAR
Location
+ loc_id INT {PK}
+landmark VARCHAR(20)
+latitude INT
+longitude INT
+ weather VARCHAR(10)
+Street VARCHAR(10)
+City VARCAHR(10)
+State VARCHAR(10)
+Country VARCHAR(10)
+Surge INT
+Rain INT
+ temp INT
located at
1
1
1
1..*
1..*
Restaurant
+ restaurant_id INT {PK}
+ restaurant_name VARCHAR(10)
+ rating INT
+ phone INT
+ loc_id INT
has
1
1..*
Menu
+ menu_id INT {PK}
+ type VARCHAR(20)
+ restaurant_id INT
delivered by
1
0..*
Delivery
+delivery_id INT {PK}
+ time_taken TIME
+ start_time TIMESTAMP
+ end_time TIMESTAMP
+ deliveryrating INT
+ deliverystatus VARCHAR(20)
1..*
Orders From
Orders To
Associated With
1
1
1
Contains
1
1..*
Orders
+ order_id INT {PK}
+ order_status VARCHAR(20)
+ restaurant_id INT
+ customer_id INT
+delivery_id INT
+ payment_id INT
0..*
Contains
1..*
Items
+item_id INT {PK}
+ price INT
+ item_name VARCHAR(10)
+ menu_id INT
+ restaurant_id INT
Customer
+customer_id INT {PK}
+ cust_name VARCHAR(20)
+ email VARCHAR(20)
+ passwd_set VARCHAR(10)
+ phone INT
+ joined_date DATE
0..*

Logical Database Design



Application Flow and Stakeholders Involved:

The application can be operated by 3 different types of users; An end user(customer), An admin(A restaurant Manager), or a driver(the delivery driver who gets you your food!).

Each of these users are able to make a login, or create a new account for themselves.
The User is provided with the following functionalities:

1. User can order food from a restaurant
2. Customers work with a data object called a Cart, through which they can either: CREATE the cart(By adding a single item from one menu, from one restaurant, add to the cart(By adding more items to the cart)
3. Remove items from the cart(By deleting items from the cart)
4. View a total when the user is ready to checkout.

   This total is calculated using a rate calculation engine, that retrieves discounts applicable, and uses it against a multiplier that factors into account the cartesian distance between the user and the restaurant, as well the temperatures of both places at a time. These are termed as conditions and each condition impacts a total in some way, that may change at the discretion of the administrator.
   Payments are modelled using 2 variables, a payment method, and payment id, both of which are created and attached to a delivery and a order.

There are some trend calculation queries that the customer has access to, such as:

1. Displaying Order History, both as a collective, and at some specified restaurant.
2. Frequency based Item Ordering(Show the customer how many times some item has been ordered, and let the customer use that as an insight.)
3. Frequency based Order Ordering(Show the customer how frequently some order of theirs has been ordered).

These functionalities employed the extensive use 1 or more of the below:

1. Procedures
2. Joins(One Table to Another, One table with 2 other tables).
3. Group By's for frequency calculations
4. Order By's for frequency ranking
5. Triggers
6. Application Level Exception Handling

Separating all of the above queries, and the CRUD operations into single modular pieces of code required the use of procedures that would be executed using a callableStatement in Java. This minimises coupling between the Java application code, and the SQL storage/retrieval/processing code. We have adhered to these principles for all the functionalities associated with every type of user in the application.

The administrator is the second type of user, and as the name suggests has capabilities to perform CRUD operations on top of data objects like Restaurants and Drivers, that the Customer does NOT have update privileges for.

To ensure some degree of security, the administrator is expected to login with a valid set of credentials, after which they are given the options to:

1. Add restaurants (create operations in restaurant table)
2. Add menus for a restaurant
3. Add items to a menu belonging to a restaurant
4. Create and Discount coupons that map to some payment id's.
5. Recruit a driver, or in simpler terms add a driver to the database.
6. Delete or Remove a driver from the database.
7. Edit rate multiplier info (to change rates based on distances according to season or economic inflation etc)
8. Delete some restaurant from the database.
9. View how many customers each restaurant has to rank by popularity.
10. View how many orders each restaurant has.
11. View number of deliveries each driver has completed.

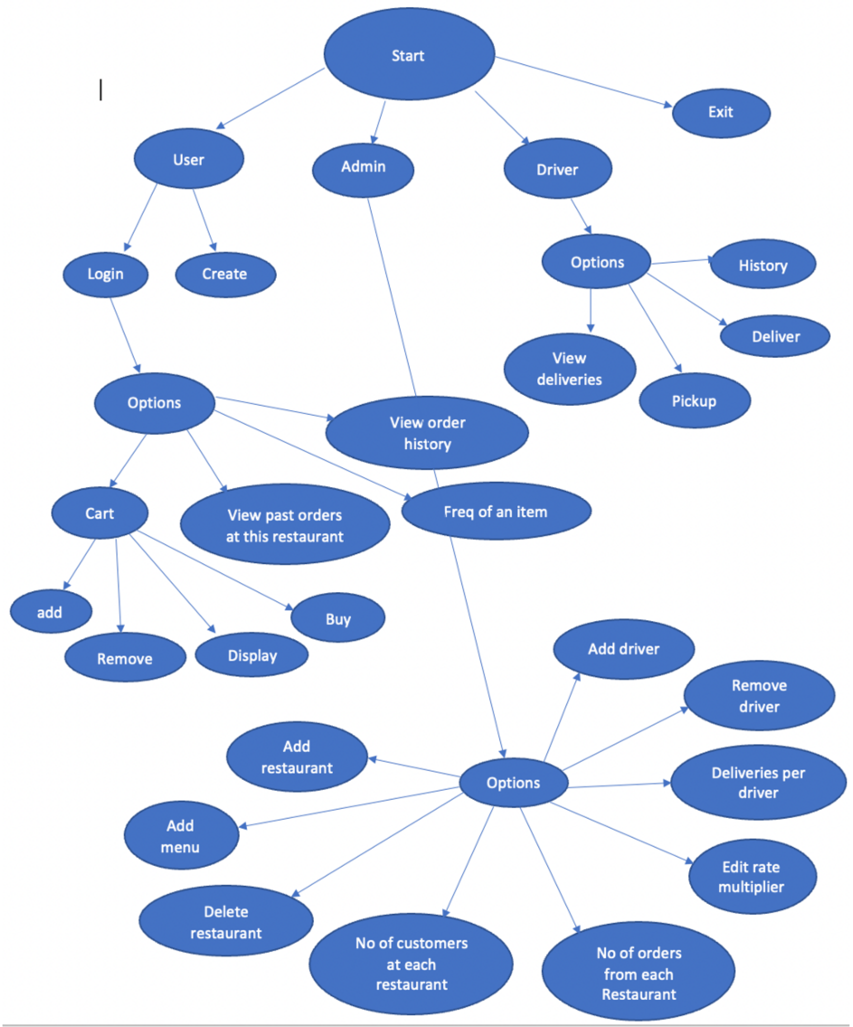
Given the above, it is apparent that the administrator manipulates data objects that are tightly coupled with the data objects that the customer manipulates. That is, deleting an administrator associated data object, will lead to mitigating cascading effects with a customer associated data object . In this application, deleting a restaurant would mean deleting every order associated with that restaurant as well.
We have handled these sort of scenarios within our database.

The remaining user type, being the Driver, also requires a valid set of credentials to login to the application, after which he or she is shown the following options:

1. View all the unassigned deliveries.
2. Pickup a delivery by accepting it. (create a mapping)
3. Declare the delivery as delivered. (update status of delivery, using a trigger this also updates the order status in the cart/order table for the customer to see)
4. See all past deliveries (history).

What's important to note here, is that the driver manipulates intermediary data objects, like a delivery, where in the accepting of some order for delivery is modelled as inserting a mapping, or a tuple that relates the driver accepting this delivery, and the delivery id associated with the order id of the order he or she is supposed to deliver. The driver also can update the status of a delivery at different points, with some message. We also want to be able to propagate this status update to different tables, whenever it happens. This is possible with the help of a trigger, where in the trigger is activated whenever an update is made to a certain field in the delivery table.

In summary, this applications supports different stakeholders that maybe involved when some end user, like a customer wishes to order something, from some entity like a restaurant. To manage the adding, updating and removal of restaurants and drivers, an Administrator is necessary. To manage the adding and removing of items, as well as viewing some total given a subset of all the items in the database, the Customer is responsible. The driver has sole access to facilitate the delivery of some order to the customer by acting as a liaison between the order_id and the customer_id.



State diagram

The above graph exhaustively represents each path, that each end user can take without repeating, or going back within the same path of actions.

The options entities denoted in each diagram are loops which keep the user/admin/driver in state with their respective options displaying every time an operation is selected and completed, they are prompted to pick another option to try from the menu until they enter 0. On entering 0 they leave their loop and get back to main menu where they can log back in as a different type as user, on quitting here the application ends.

Lessons Learnt:

Technical Expertise Gained:

Procedural SQL, Error Handling, DB Administration, DB Troubleshooting, Application and Database Decoupling, Stakeholder Identification, User-State management and Building MultiState Applications.

Insights, time management insights, data domain insights etc.

Procedures help decouple application logic from DD and DM logic, and also help distribute workload to teammates in manageable units.
This data domain involved the use of location type data like latitudes and longitudes, as well temperature/rain quantification units, like temperature and rainfall/cm. Representing these objects are not trivial, and require additional validation strategies to ensure their correctness. These strategies will involve the use of real time API data, for weather and location validation. These strategies will be part of the next steps that we want to take to improve our existing application.

Realised or contemplated alternative design / approaches to the project

We were tasked with the following design choices:

1. NoSQL(MongoDB) vs Relational(MySQL)
   We chose to use a relational database model because the entire team had more experience working with MySQL than with MongoDB.
   Also, Working with JDBC made it easier to communicate with a Relational Database like MySQL, as compared to MongoDB.
   Transaction Management and Cascading Strategies were a lot more straightforward in MySQL as compared to MongoDB.
   We also didn't need the scalability that MongoDB promises.
2. GUI Based Vs CLI Based
   As this is a database class, putting more effort into ensuring our DD and DM logic worked, while ensuring that the application logic did not interfere with this logic, became the main aim of this project. We did not see the benefit in putting more investment in building a better frontend, though given more time its something we would like to have.
3. Java Vs Python
   Java’s static typing and its well supported database driver made it easy to work with, as compared to Python’s dynamic typing and equally well supported SQLAlchemy. The team had a better time communicating and understanding application logic not of their own, due to the statically typed nature of java.

Future work

Planned Use of the Database:

This application, and the logic it uses to communicate with the tables that we have created to simulate a food ordering system, can be used as a simulation engine, that allows end users to predict estimated cost given some set of items, bought on a day with some type of weather. End Users that benefit the most out of these predictions include the restaurant manager, who can now use these predictions to influence the prices of the products that they might sell in the future. Customers can use this application and database, to mine trends that can help them select what they want to order.

Potential Areas for Added Functionality:

Looking at a few things that we could have done better,

There are API's available that allow applications, such as ours to access real time trends regarding weather and climate conditions. These trends, when sampled frequently, can give us a dynamic rate calculation engine, instead of the static one we have right now. The tables we maintain that store these conditions would be frequently updated with these real time trends, thereby reducing a DBA's involvement whenever an Admin wants to update some metric or add a new range to the database.

Similarly, traffic data is something we have not considered using during ETA calculation. There are open access API's available that let us query real time traffic at some location, and estimate how much longer would it take on top of the best ETA for an item to reach a customer. This would allow us to manipulate the distance factor, used by the rate calculation engine, with the premise that a larger amount of traffic directly correlates with a larger distance.

LINK TO VIDEO : https://drive.google.com/file/d/1Jbo05wMZxLHdbIBR-a7DJUGQdDcBxDfE/view?usp=share_link