Using 18 Late hours CMPSC 431W Final Report Hannah Sugden & Manasi Das

Introduction:

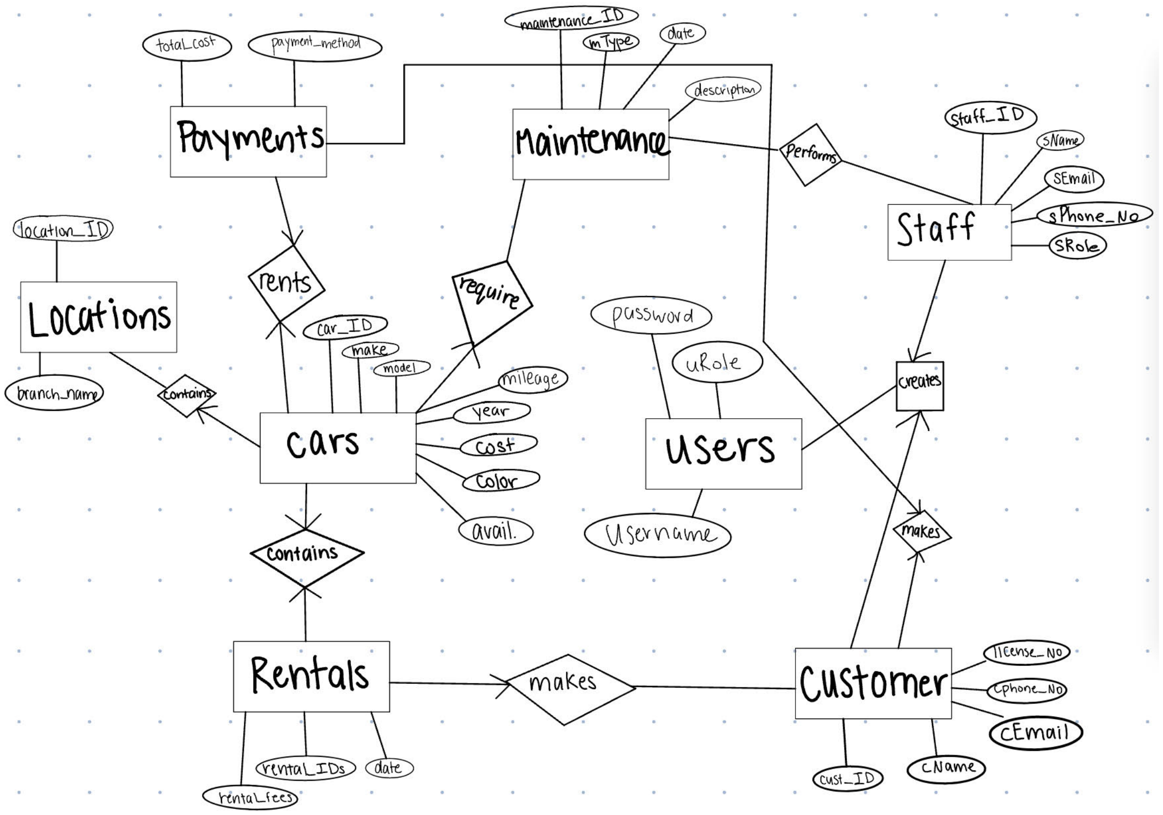
The primary objective of this project was to develop a comprehensive database system that could efficiently manage multiple aspects of a car rental service. This includes handling detailed records of cars, customers, rentals, payments, staff, and maintenance services. The purpose was to ensure that these diverse data sets interact seamlessly, providing a cohesive and user-friendly experience for both the car rental service staff and their customers.

To bring our vision to the full effect, we utilized MySQL Workbench for the database creation and Postman as the API server. MySQL Workbench was chosen for its robust features in managing complex databases, allowing us to design, model, and maintain our database effectively. Postman, on the other hand, was used to develop and test APIs, ensuring smooth communication between the front-end and our database. In addition, React, HTML, and Javascript were used to capture the data from the database and connect to the Postman API, and project to a desktop application webpage. Together, these technologies formed the backbone of our project, enabling us to create an efficient system.

The anticipated outcome of our project is a scalable, reliable, and intuitive car rental database system. It is expected to significantly improve the efficiency of data management, streamline the rental process, and enhance overall customer satisfaction. By automating key aspects of the car rental service, we aim to reduce manual workload and minimize errors, leading to a more efficient business operation. **The github link to our project is at the very bottom.**

Designing and Implementation of the Database:

The process began with a clear understanding of the requirements of our car rental service system. We identified the key entities and their relationships, which formed the foundation of our database schema. These entities included Cars, Customers, Users, Rentals, Payments, Staff, Maintenance, and Locations.



Designing the ER diagram:

Cars Table:
The Cars table has eight attributes. In relation to the other tables:
Various locations contain many cars. Many cars require Maintenance. A payment can rent one or more cars, and cars can have various payments (towards renting, or maintenance, or other types of payments). A customer can make many rentals on many cars, and cars can have many rentals (from various customers).

Customers Table:
The Customers table has five attributes. Any customer can make many car rentals. Customers can also make many payments. Many customers can create many user accounts.

Rentals Table:
The Rentals table has three attributes. Any customer can make many car rentals. Each rental can consists of many cars (to be rented)

Users Table:
The Users table has three attributes. Staff members can create a user account, and customers can create a user account as well.

Payments Table:
The Payments table has two attributes. Many customers can make many payments. Payments make many cars available for renting, and many cars require payments to be rented.

Staff Table:
The Staff table has five attributes. A staff member can perform maintenance on a car. Many staff members can create user accounts.

Locations Table:
The locations table has three attributes. One location contains many cars.

Maintenance Table:
The maintenance table has four attributes. Many cars require various types of maintenance, but all maintenance on cars is on one maintenance application. Thus, many cars may request maintenance. Only one staff member may perform each maintenance.

Designing the Tables:

Cars Table

```
CREATE TABLE Cars(
CarID Int PRIMARY KEY,
Make Varchar(30),
Model Varchar(30),
MYear Int,
Color Varchar(15),
Availability_Status Varchar(15) NOT NULL,
Mileage Int,
Foreign Key (locationID),
References Maintenance(Mtype)
);
```

Customer Table

```
CREATE TABLE Customer(
CustomerID Int PRIMARY KEY,
Cname varchar(50),
Email varchar(50),
CPhone_number Varchar(15),
License_Number Int,
Foreign Key (carID),
References Car(carID)
);
```

Users Table

```
CREATE TABLE Users(
Username varchar(25) PRIMARY KEY,
Password varchar(50),
uRole Varchar(15) NOT NULL,
);
```

Rentals Table

```
CREATE TABLE Rentals(
rentalID int PRIMARY KEY,
Date datetime,
Rental_fees DECIMAL(10,12) NOT NULL,
Foreign Key (carID, customerID),
References Customer(customerID), Car(carID)
);
```

Payments

```
CREATE TABLE Payments(
ReceiptID int PRIMARY KEY,
Total_Cost DECIMAL(10,12) NOT NULL,
Payment_method Varchar(15) NOT NULL,
Foreign Key (carID, customerID),
References Car(carID), Customer(customerID)
);
```

Staff

```
CREATE TABLE Staff(
staffID int PRIMARY KEY,
Sname Varchar(50) NOT NULL,
Semail Varchar(50) NOT NULL,
SPhone_Number Varchar(15),
sRole Varchar(20)
);
```

Locations

```
CREATE TABLE Location(
locationID int PRIMARY KEY,
Branch_name varchar(35) NOT NULL,
Foreign Key (staffID,carID),
References Staff(staffID), Car(carID)
);
```

Maintenance

```
CREATE TABLE Maintenance(
maintenanceID int PRIMARY KEY,
Mtype int,
Date datetime,
Description Varchar(100) NOT NULL,
Foreign Key (staffID,carID),
References Staff(staffID), Car(carID)
);
```

Challenges and Solutions:

As we were creating these when implementing the tables in MySQL, we ran into some issues.

- We struggled with understanding how to connect MySQL to be able to work from it. We did some research and realized we needed a server connection and postman was a free option. We used localhost as the host, ports 4200, 8081, 5000, and 8080, and username as root which we connected to the API.
- Error saying “locationID did not exist”. We had already created the Locations table, so at first we were very unsure on why this was an error. We figured out that when creating the Cars table, we had to include the foreign keys as a column along with the attributes of that table.
- The order of which we needed to add all of the tables. We ended up creating all the tables without any foreign keys, that way we had no issues. We then used the ALTER TABLE statement for each foreign key and made sure the constraint was unique for that foreign key. This was ensuring we had no problems when creating the tables.

    - ALTER TABLE Cars ADD CONSTRAINT fk_locationID FOREIGN KEY (locationID) REFERENCES Location(locationID)
    - TRUNCATE TABLE Cars

- After inserting a good amount of data, we realized we hadn’t been tracking all of it and that it was inconsistent. When we tried our join statement that joined the cars table, rentals, customer, payments, staff, and customer, it just returned the columns but no records of any data. That led us to believe something was wrong and so we decided to delete all of the data and restart. We were able to highlight all rows of some tables and delete the rows, then apply it. However, three tables - staff, cars, and customers - had some issues. They all were dependent on each other so there was no order in which we could delete the data without a foreign key constraint error. We ended up truncating the table which deleted the records and could start again. We made sure to track each record and make sure the data was consistent. For example, in the Cars table, it is either available, not available (because someone had already rented it), or in maintenance.

- Connecting the API files to the react proved to be difficult. It was easy to connect the localhost endpoint links to the react files, and run it and simply display data. But when it came to formatting the application, choosing what to view, which tables and data to hide until toggled on, etc. was a challenge as we had no prior experience in this type of development. There were some faulty statements in the code when writing certain functions to display various tables.

Designing the Desktop Application through API:

We connected the database through an API server called postman. We used javascript to be able to fetch the data from the API server and display it through html and react.

Two projects/folders were created in the process: the API files (backend) and the react.js files with HTML (frontend). The API files consisted of carlocation.js, report.js, people.js, and finances.js. The database is connected to the server through these API files in the Postman API express server. We used endpoints with the postman 'GET' functionality to retrieve our data from the SQL database. Using the get function, "get(/api/cars)" for example, pulled our data of "Select * from Cars" records.The react.js files consisted of html and css styling, connection to specific records/queries, and the implementation of the desktop application. This allowed us to be able to build part of an interactive interface but most importantly, we established the connection and showed that it works without any error trapping.

SQL Feature Queries:

Inserting a new car:

INSERT INTO Cars (CarID, Make, Model, MYear, Color, Availability_Status, Mileage, locationID)
VALUES (1, 'BMW', 'M6', 2023, 'Blue', 'Available', Available, 1);

Inserting a new customer:

INSERT INTO Customer (CustomerID, Cname, Email, CPhone_number, License_Number, carID)
VALUES (1, 'Frank Smith', 'SmithF@email.com', '412-451-7825', 1523685, 1);

Deleting a car:

DELETE FROM Cars WHERE CarID = 1;

Deleting a customer:

```
DELETE FROM Customer WHERE CustomerID = 1;
```

Updating car availability:

```
UPDATE Cars SET Availability_Status = 'Not Available' WHERE CarID = 1;
```

Updating customer information:

```
UPDATE Customer SET Email = FrankSmith@gmail.com' WHERE CustomerID = 1;
```

Analytical Report: Cars available by make:

```
SELECT * FROM Cars WHERE Availability_Status = 'Available' ORDER BY Make;
```

Analytical Report: Detailed Maintenance Records sorted by date:

```
SELECT M.maintenanceID, C.Make, C.Model, M.Mtype, M.Date, M.Description
FROM Maintenance M
JOIN Cars C ON M.carID = C.CarID
ORDER BY M.Date DESC;
```

Transaction: Rent a car with Rollback:

```
BEGIN TRANSACTION;
   -- Check if car is available
   DECLARE @carAvailable BIT;
   SELECT @carAvailable = CASE WHEN Availability_Status = 'Available' THEN 1 ELSE 0
END
   FROM Cars WHERE CarID = 1;

   IF @carAvailable = 1
   BEGIN
      -- Update car status
      UPDATE Cars SET Availability_Status = ‘Not Available’ WHERE CarID = 1;

      -- Insert rental record
      INSERT INTO Rentals (rentalID, Date, Rental_fees, carID, customerID)
      VALUES (1, GETDATE(), 100.00, 1, 1);
   END
   ELSE
   BEGIN
      -- Rollback if car is not available
```

```
        ROLLBACK TRANSACTION;
        PRINT 'Car not available';
    END
COMMIT TRANSACTION;
```

Joining multiple tables (Rental detail information):

```
SELECT R.rentalID, C.Make, C.Model, Cu.Cname, R.Date, R.Rental_fees
FROM Rentals R
JOIN Cars C ON R.carID = C.CarID
JOIN Customer Cu ON R.customerID = Cu.CustomerID
JOIN Payments P ON Cu.CustomerID = P.customerID
JOIN Staff S ON C.locationID = S.staffID
JOIN Maintenance M ON C.CarID = M.carID
WHERE R.rentalID = 1;
```

This query demonstrates how to join five tables in a useful way, providing detailed rental information including the car details, customer name, rental date, and fees. This would be useful if a customer wanted to know about a past rental, it would have all the necessary information and they would be satisfied.

Results:

Efficient Database System

The project successfully resulted in an efficient database system tailored for car rental service operations. Key tables like Cars, Customers, Rentals, Payments, Staff, Maintenance, and Locations were effectively integrated, providing a comprehensive data management solution.

Streamlined Rental Processes

The database significantly streamlined rental processes. Quick access to car details, customer information, and transaction histories improved operational efficiency, reducing the time needed for processing rentals and customer inquiries.

Enhanced Data Integrity and Consistency

The implementation of primary and foreign keys, and the resolution of dependency issues through careful table creation and alteration, led to improved data integrity and consistency. This ensured that the data stored in the database remained accurate and reliable.

Improved Financial Management

The database enabled better tracking and management of financial aspects, such as rental fees and payments. This facilitated more effective financial reporting and analysis, crucial for business decision-making.

Technical Challenges Overcome

The project team successfully overcame various technical challenges, including resolving foreign key constraint errors and ensuring data consistency through effective use of SQL commands like ALTER TABLE and TRUNCATE TABLE.

Impact on Business Operations

The implementation of the database had a positive impact on the overall business operations, enhancing the efficiency of car rental management, customer service, and financial tracking.

Github Link: https://github.com/mxnasi/431Project

Includes:

- User Manual
- This Final Report
- Frontend User Desktop Application Interface files (React.js)
- Backend API files (establishes connection to database)