**INTRODUCTION TO DATABASE DESIGN & DEVELOPMENT**

Table of Contents

[Task 1 3](#_Toc201715141)

[1a. Logical Model 3](#_Toc201715142)

[1b. Physical Model 4](#_Toc201715143)

[Task 2 7](#_Toc201715144)

[2a Creating Database 7](#_Toc201715145)

[2b Insertion 8](#_Toc201715146)

[2c Data insertion 10](#_Toc201715147)

[Task 3 12](#_Toc201715148)

[Case 1: Better Rental details 12](#_Toc201715149)

[Case 2: Search 13](#_Toc201715150)

[a Contact based search 13](#_Toc201715151)

[b Return on specific date 13](#_Toc201715152)

[Case 3: Data Manipulation 14](#_Toc201715153)

[b Update Car 15](#_Toc201715154)

[c Delete from car 15](#_Toc201715155)

[Case 4: Statistics 15](#_Toc201715156)

[a Customer Rented Most 16](#_Toc201715157)

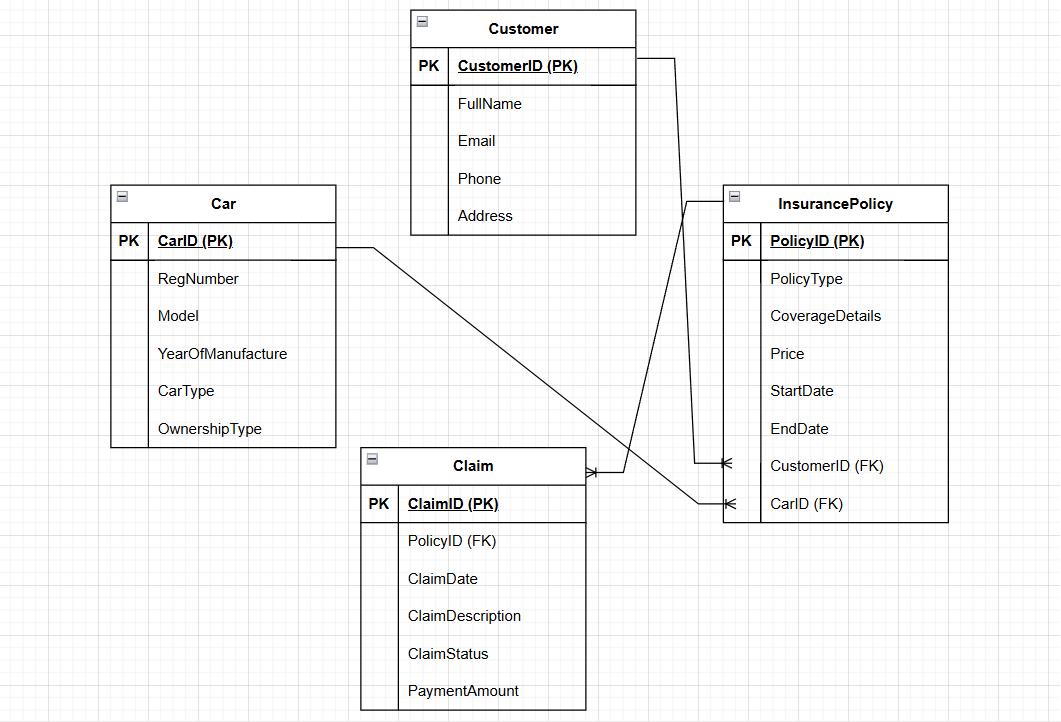
[b Frequent Rented cars 16](#_Toc201715158)

[c Transaction received 16](#_Toc201715159)

[d Car Type 16](#_Toc201715160)

# Task 1

## 1a. Logical Model



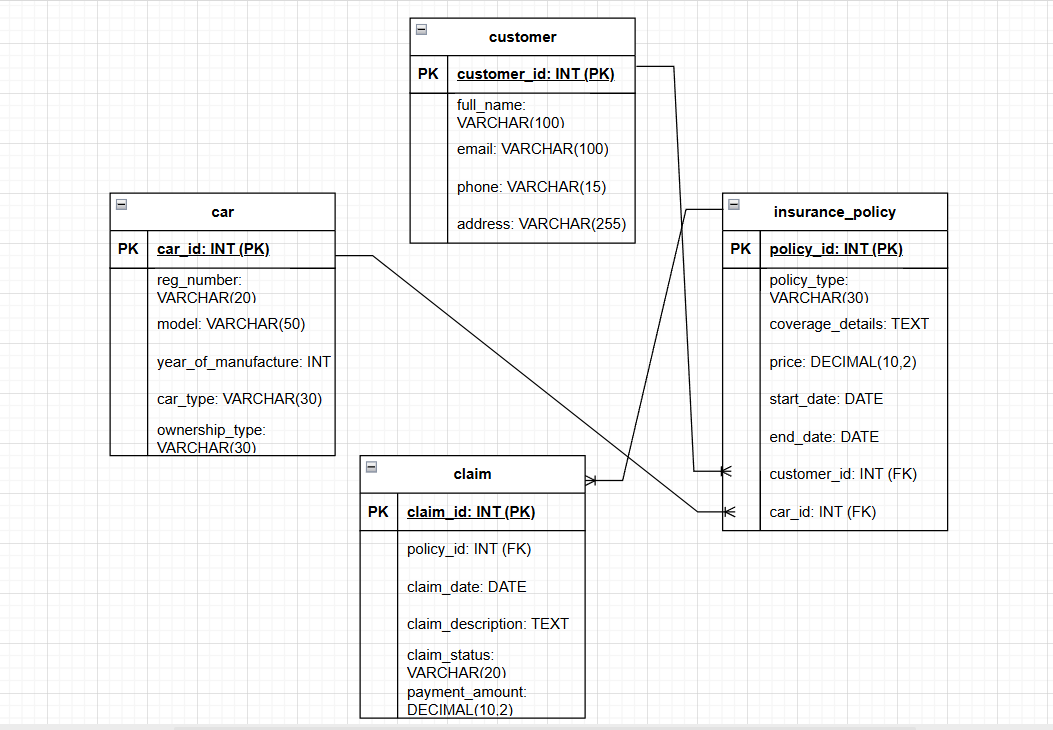
**Figure 1: Logical Model**

The data in this model was verbose up to the normal form (3NF) and made certain that it did not create awkwardness and inconsistency (normalization). The process of normalization we followed began with 1NF where we checked all the attributes in consideration are atomic. Here, one attribute field in a table would only have a single contained value without any repeating values group’s fields such as Customer Contact details like Full Name, Email, Phone, and address are kept in separate columns rather than grouped together. Similarly each claim has a single Date and Description field as opposed to list of events. Also each entity in model has a clearly defined primary key to uniquely identify records thereby ensuring 1NF requirements are met.

The second NF was achieved by ensuring that all non-key attributes are fully functional dependent on the whole of the primary key (2NF). Since each entity uses a single column primary key, such as CustomerID, CarID, PolicyID or ClaimID respectively, there are no partial dependencies such as In the Table InsurancePolicy, attributes like PolicyType, CoverageDetails, Price and StartDate are dependent on PolicyID only and not on a composite of fields. Similarly, attributes in the Claims table are dependent on ClaimID only thereby ensuring compliance with 2NF. Lastly come the 3NF practical, here non-prime attributes are not dependent on other non-prime attributes in table. In the Customer table, FullName, Email, Phone, Address depends on CustomerID and none of these attributes depend on each other. In the Car table, the details like RegNumber, Model, OwnerShip Type are dependent on CarID. That is why Transitive Dependency is redundant data avoidance that ensures every time an attribute is updated it doesn't cause changes in multiple places.

During the design few assumptions were made. For example, using CarID as a surrogate primary key instead of the Registration Number, even though the registration numbers are usually unique, to handle duplicate or historic registration number possibilities. As well it was assumed a customer can have multiple cars and policies, where one policy would refer to one and only car (Car to Policy). A number of cars may be used by different customers over different durations (such as rental cars) hence multiple insurance policies might refer to 1 car. Each policy will have multiple claims linked to it as the real world scenario could have accidents or issues during the course of the policy. These assumptions have been aligned with the practical approach to the scalable and real life scenario based relational design supporting business goals and normalization principle.

## 1b. Physical Model

****

**Figure 2: Physical Model**

With the use of suitable data types, naming standards and normalization up to the third normal form (3NF), the physical model of the auto insurance database has been organized to faithfully depict the main entities and relationships mentioned in the scenario. Cardinality and integrity restrictions between the tables are explicitly defined in the diagram using Crow's Foot notation. To adhere to database best practices and enhance readability and implementation accuracy, each entity has been renamed using snake case formatting. Essential user data, including full\_name, email, phone number, and address, is stored in the customer table. Customer\_id, an integer that is presumed to be auto-incremented, is the main key. Proper space allocation is ensured by data types like VARCHAR (15) for phone numbers and VARCHAR (100) for names and emails. Although restrictions like NOT NULL and UNIQUE would be imposed during SQL implementation to maintain data integrity, it is assumed that each customer's email address and phone number are unique.

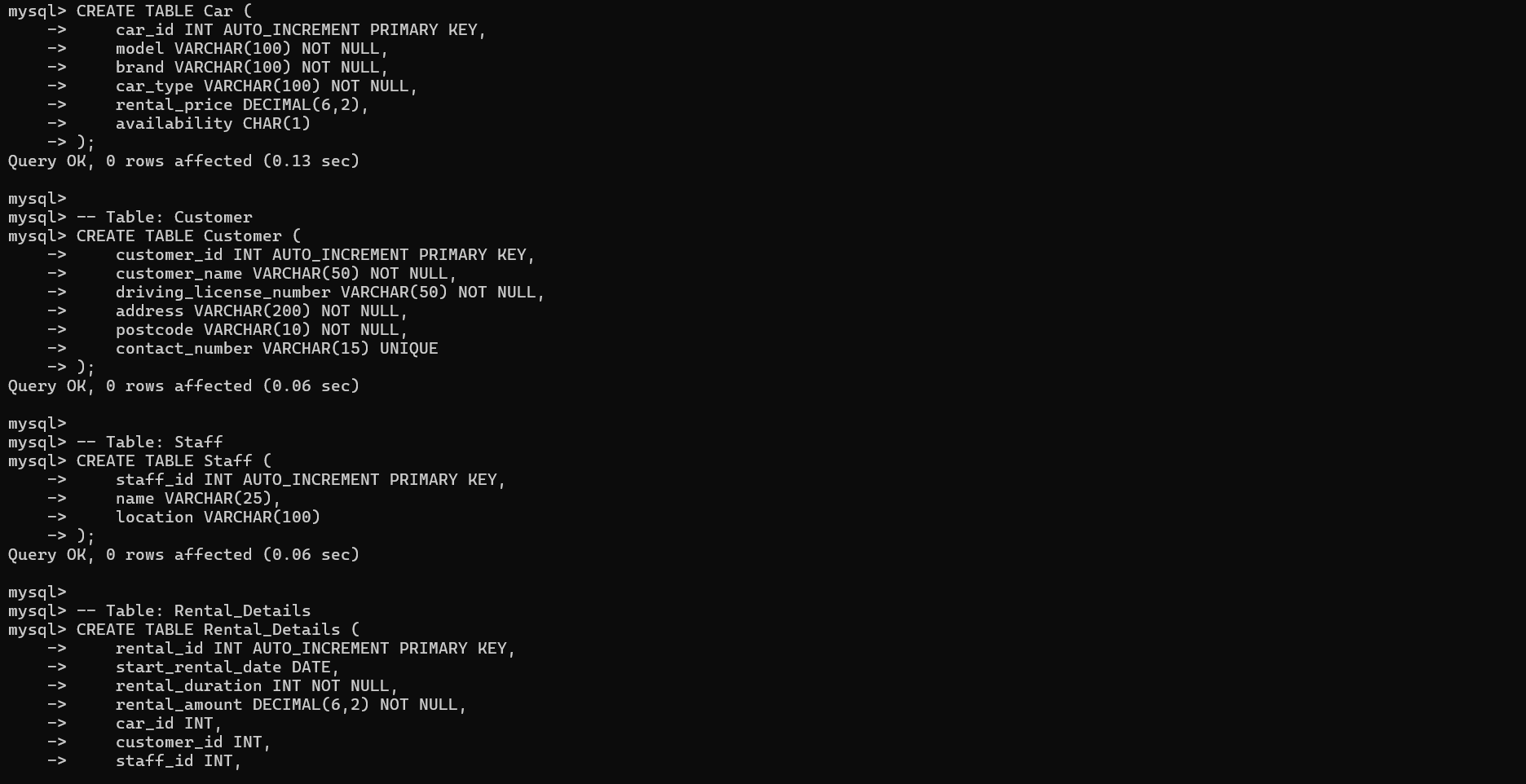
Details such as registration number, model, and year of manufacturing, car type, and ownership type are recorded in the car table. Car\_id is the primary key. It is presumed that each car's reg\_number is distinct. INT for the year and VARCHAR (20) to VARCHAR (50) for text fields are among the field types. Since cars can be either owned or rented, ownership\_type is required to differentiate between the two. Several clients may insure the same vehicle at various periods, particularly while renting it.

The primary connection between consumers and automobiles is the insurance\_policy table. Policy-related information like policy\_type, coverage\_details, price, start\_date, and end\_date are included. In order to link a customer and a vehicle to a particular policy, this database contains the foreign keys customer\_id and car\_id. Appropriate data types are selected; for example, DATE is used for the policy duration fields and DECIMAL (10,2) is used for price to accommodate monetary values. Every car may have many policies throughout the course of its lifetime, and each consumer may purchase multiple policies for various vehicles. Referential integrity across tables is guaranteed by all foreign keys.

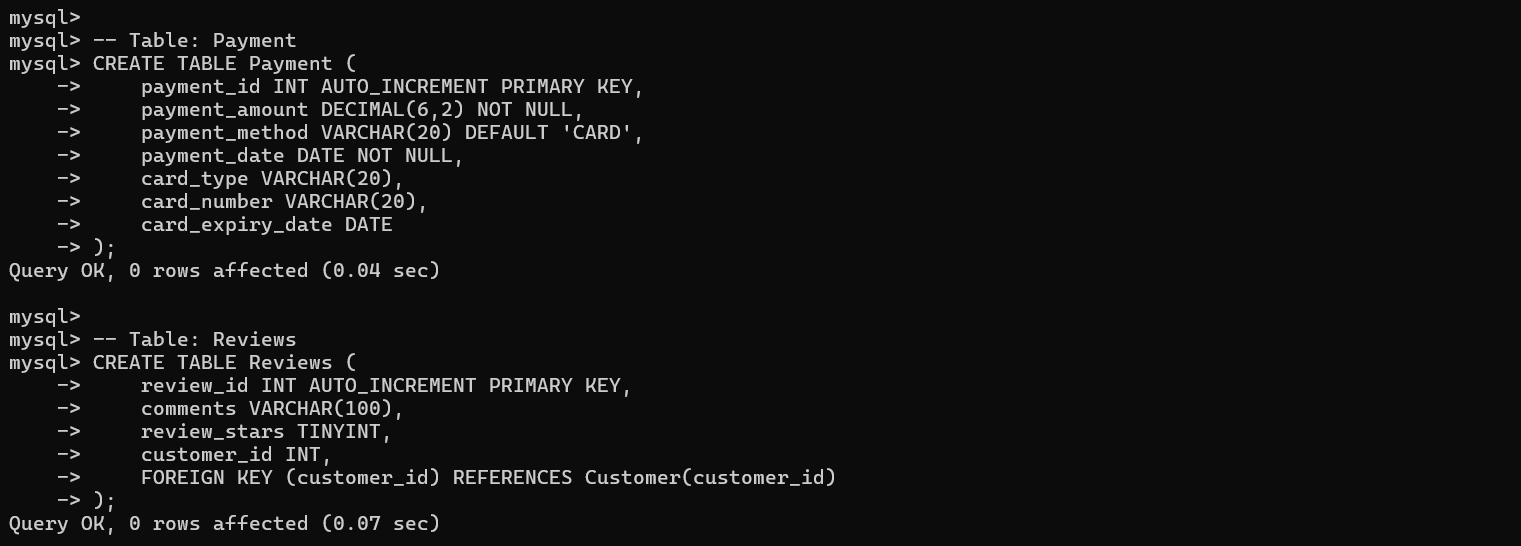
Information on insurance claims is kept in the claim table. Claim\_id is its main key. Using the proper data types, such as TEXT, DATE, and DECIMAL, fields like claim\_date, claim\_description, claim\_status, and payment\_amount are provided (10,2). Each claim is associated with a particular policy using the foreign key policy\_id. Whether a policy has claims or not is presumed. Validation rules are implemented at the application level, and claim\_status will accept limited values such as "Pending," "Approved," or "Rejected." Throughout the design, foreign keys will employ ON UPDATE CASCADE for secure record updates and ON DELETE RESTRICT to prevent unintentional deletions. To prevent redundancy, enhance efficiency, and guarantee long-term scalability and consistency, the model is completely normalized.

# Task 2

## 2a Creating Database



**Figure 3: Created car, customer, staff and rental details table**

****

**Figure 4: Created payment and reviews table**

The Car Rental Services database has been implemented using the provided physical model in Figure 1 to come up with relational database by using Structured Query Language (SQL). All steps followed during this project were executed within our official database account on Arden University to comply with assessment rules and to ensure data integrity. First this had to create or select a database by using the CREATE DATABASE and USE statements. A database called car\_rental\_db was created if it didn’t exist already and here we house all the given tables. Subsequently, the six core tables were built: Car, Customer, Staff, Rental\_Details, Payment, and Reviews structured based on what is shown in the ER diagram with the corresponding fields, data types, and relationships. Using suitable SQL CREATE TABLE statements each table was made to incorporate constraints such as PRIMARY KEY, AUTO\_INCREMENT, NOT NULL, and FOREIGN KEY to enforce entity integrity and maintain connections between tables.

The Car table contains data about every car such as model, brand, and type of car, rental price per day and whether or not it’s available in the inventory. The Customer table holds information regarding customers like their full name, address, driving license number and a unique contact number. Staff that will be necessary to carry out rentals was introduced to us via the Staff table with their name and location where they are. The Rental\_Details table is a transactional log that ties cars, customers and staff together in each rental transaction. This table holds information regarding rental duration, start date and total amount and has foreign relations with the Car, Customer and Staff tables to make sure we are normalizing our database. A separate table was created for Payments to keep track of transactions such as the amount paid, method of payment (i.e., card), card type, expiry date making sure every financial transaction was recorded with the rental they pertain to. Lastly, a Reviews table was added to get customer reviews and satisfactions scores related to the customer by using a foreign key.

Data was fed into each table with the need to INSERT INTO SQL data for each table for testing purposes after the schemas were in place. We had to ingest a minimum of 12 rental records, 6 customers, 5 staff, and 4 reviews as per the requirements outlined in the assignment. Validation work was carried out post population using SELECT \* FROM queries in every table. We went on to take screenshots when each of these was done from executing the SQL commands right through the output results to show evidence that this was a success. The company’s database servers to revolutionize how customers rent and return cars by maintaining accurate records, storing transaction data without redundancy, and allowing for easy access to all rental, payment, customer and staff information.

## 2b Insertion



**Figure 5: Inserting data in car, staff and customer table**

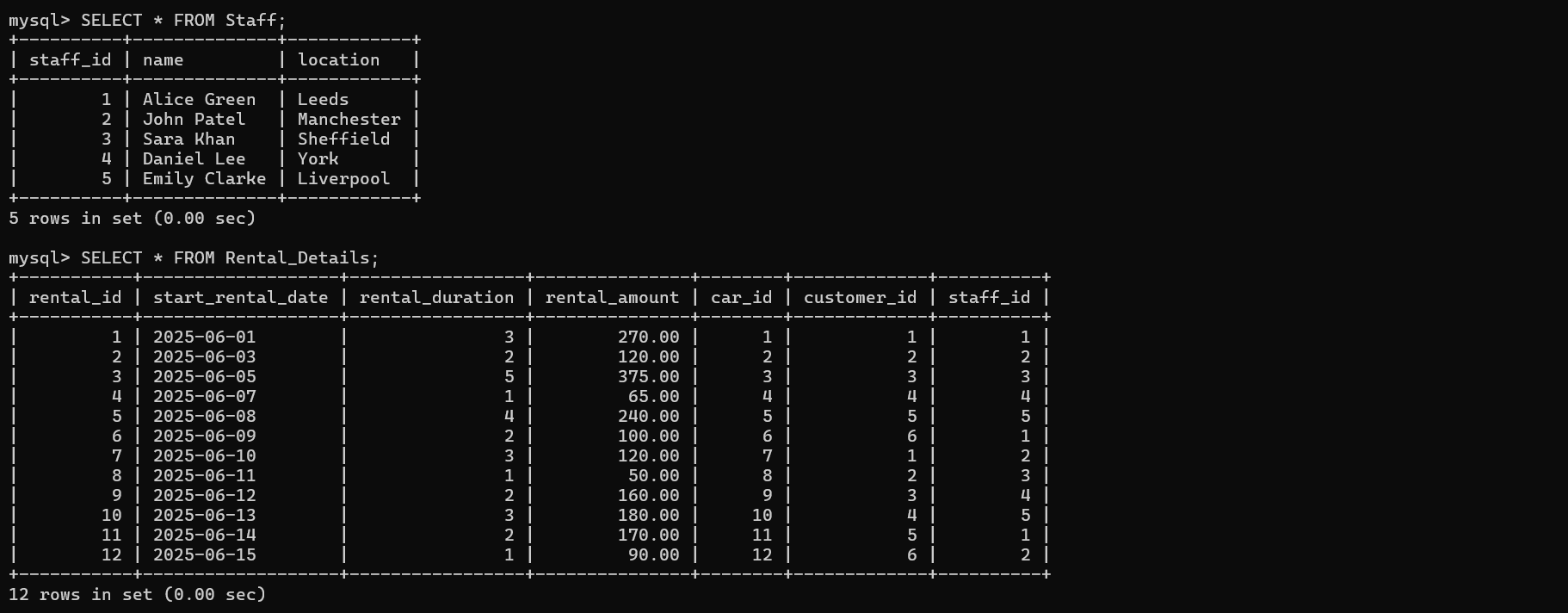


**Figure 6: Inserting data in rental details, payments and reviews**

Post generation of relational database schema successfully, the subsequent immediate phase was to fill defined Car table with data. It was inserted into Car table created by using structured SQL INSERT INTO statements On Car Details Table 1 which lists various cars available for rental along with their respective models, types, and daily rental rates so-on in line New-Car Database populated on Arden University SQL Environment. These information were achieved by translating each row off Table 1 to a structured data format, mapping every attribute properly into its columns Car table is structured with columns as car\_id (auto-incremented primary key), model, brand, car\_type, rental\_price and availability. Every record had its Car ID auto-generated by the system using the AUTO\_INCREMENT field while other fields were filled out manually for each row. The default for the availability field in all rows was 'Y', meaning all cars are currently available for rent - could be dynamically changed based on real-time booking and returns process.

The data was entered for the cars such as BMW 520 (Sedan) have been given a rental price of £90.00;, and the Toyota Corolla/ Chevrolet Spark, each, had £60.00; specific models like Kid Picanto and Hyundai i10 (both sedans) were priced at £40.00 and £50.00 respectively as well. For SUVs like Honda CR-V, Ford Puma, and Citroen C3, the price ranged between £60.00 and £85.00 due to brand and market value basis. A total of 12 cars were inserted into the database with individual INSERT INTO statements for added correctness and traceability After all the car entries were completed S Verified each row was saved in the table by applying the SQl command SELECT \* FROM Car; The command showed all records noting details on model, brand, type, price and availability. This page has screenshots of both INSERT and SELECT commands and their output placing the presentation - proof of successful execution in this report. At this stage of Database was well prepared with the updated and precise list of cars that could be rented right away making bookings and easy pricing adverts to the Car rental service.

## 2c Data insertion



**Figure 7: Staff and Rental details table**

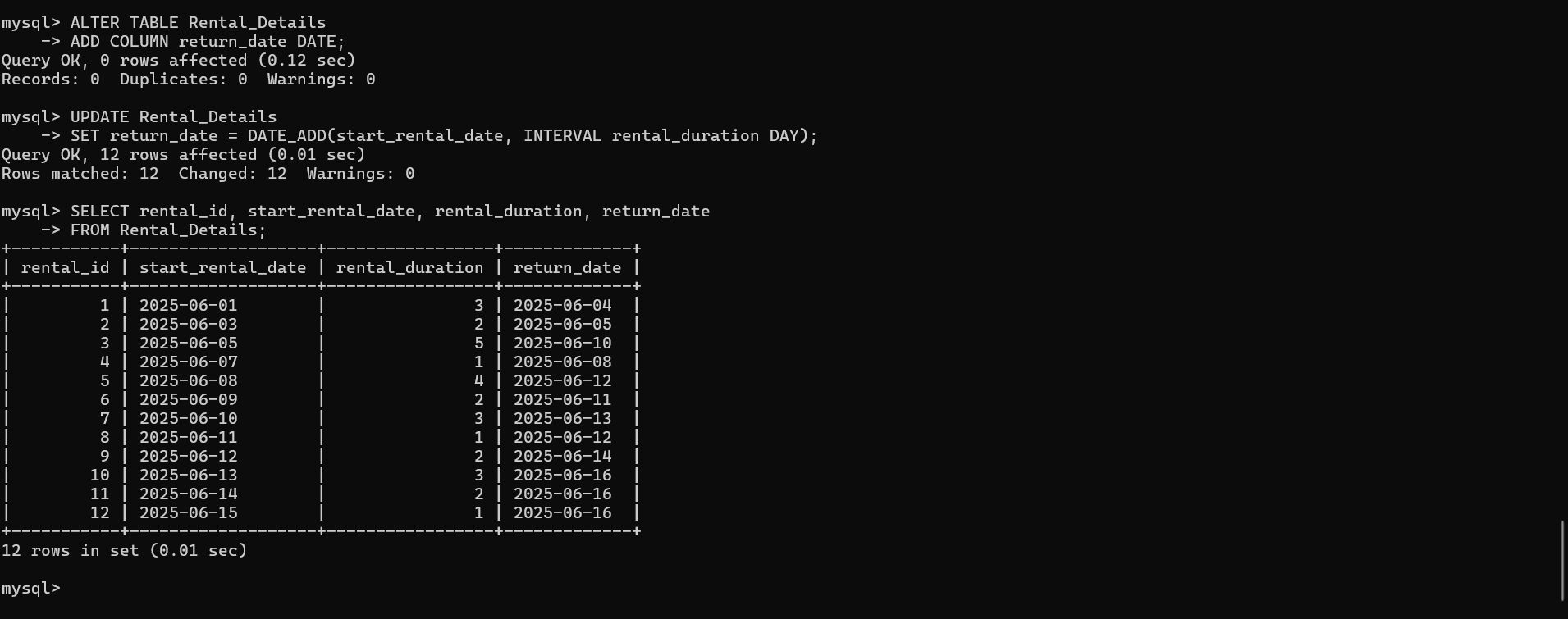
Database schema once done we moved ahead with inserting the car details successfully. And hence, loading database with transactional and operational data was next step. And as par the requirement of the assignment this included made up details of five staff and same for the twelve car rental transactions and twelve payments that well through SQL queries. This stage was mimic’d real world usage by create a sample data environment that represents how the car rental service operates on daily basis. Step one was to insert staff records into the Staff table. Staff location and name, where given to indicate disparate branches or service point of the business. INSERT INTO statement was used to add five staff members like Alice Green (Leeds), John Patel (Manchester) etc. These entries were assigned unique staff\_id values via the autoincrement function present in the table.

After that, the rental transactions were appended to the Rental\_Details table. Each transaction was inserted by providing the start date, duration (in days), total amount & the FKs (foreign keys) referencing the correct car\_id, customer\_id & staff\_id. INSERT INTOs made sure that each transaction uses cars, customers and staff from your Car, Customer and Staff tables. Twelve rental transactions were entered. Once the rentals were done, matching payment records were added into the Payment table. Every payment had values like amount paid, method (all cards by default), paid date, card type (either Visa or MasterCard), random card num and expiry. This was to make sure each rental has an underlying financial record assigned to it. Though Payment table is not directly linked with by foreign keys with the Rental\_Details, but each entry was inserted in ordered manner so as to show one-to-one mapping with the rental transactions.

All this completed, those SELECT \* FROM Staff;, SELECT \* FROM Rental\_Details;, SELECT \* FROM Payment; queries were executed to make sure data was successfully inserted. The screenshots were then taken of those queries output and appeared in the report for verification. This phase made sure that the database has real business operations; it showed customer transactions for payments too in a normalized tamed way.

# Task 3

## Case 1: Better Rental details

****

**Figure 8: Altering and Updating Rental Details**

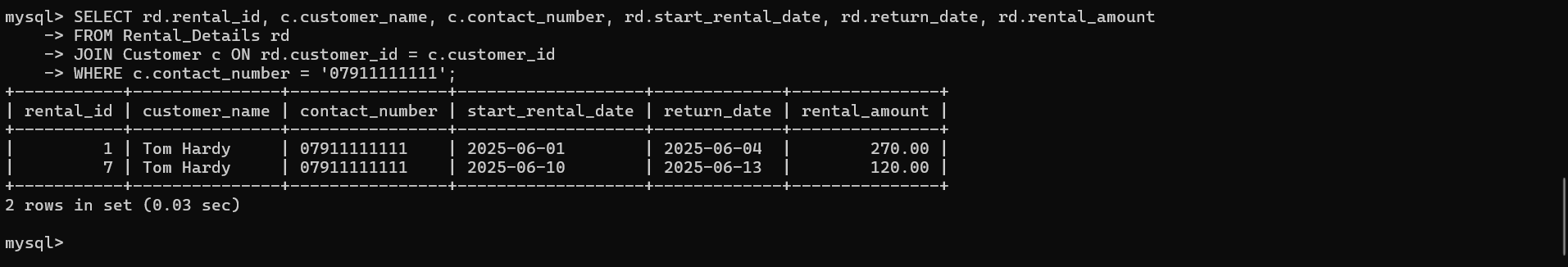
In the upkeep of the car rental database, a request from the management is made to insert return date of the car into each rental record for better functional use and information Added functionality As well as tracking vehicle availability more accurately, scheduling can be done more precisely and this will ultimately lead to better decision-making. When the Rental\_Details table was initially created it had only the start\_rental\_date and rental\_duration which meant you had to calculate the return date manually for each transaction In case if you need to update that date as per transaction so for easiness of use and making the data complete a new column added

To fulfill this requirement one of the initial steps was to make changes in the existing Rental\_Details table by introducing a new column with the name return\_date. This was done with SQL's ALTER TABLE command that aides in modifying existing structure of table without hurting the ongoing data. After we created the column, now its time to populate it as well with the right values and this was done through the UPDATE statement with the help of DATE\_ADD() that gives you return\_date by adding rental duration (in days) from rental's start date. Make sure that you have a new computed column called return date for each rental creating the column stopped updating the return date manually, which means risks of errors are covered, data can now be used for additional reports, operational decisions and so on.

To check the programmed codes, the it listed some already rented data, which should include rental ID, start date, duration and return date where insertion took place and it goes without a saying that new column would have been updated successfully. This modification enhances the database with more accurate data and simplifies rental booking. As confirmation that the process worked we queried using SELECT statement to get the rental ID, start date, duration and computed return date for all rows.

## Case 2: Search

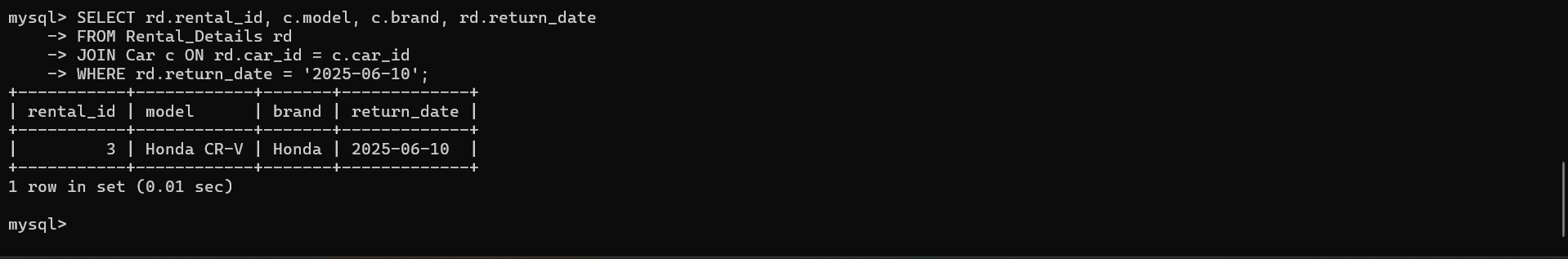
### a Contact based search



**Figure 9: Customer Number Based rental records**

Two critical searches defined to prove the worth of efficient data search and retrieval in car rental database. The objective was to find the rental transactions specific to a contact number. As a contact number is the unique identifier for Customers Table, hence enabled fast and precise search. Because contact numbers are a unique identifier in Customer table, this approach allows for an effective and accurate search.

### b Return on specific date

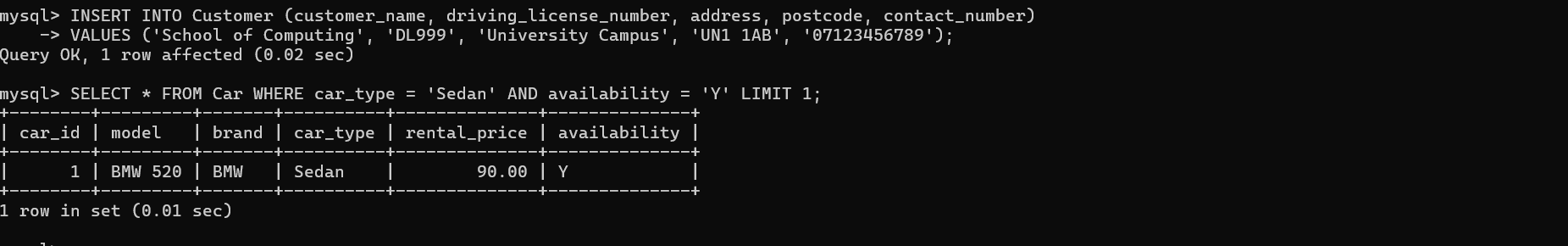
****

**Figure 10: Return on specific date**

The second search addresses a particular operational requirement such as finding vehicles due for return on a specific date. It additionally helps in tracking - keeping fleet availability in view, and proper scheduling of vehicles inspection. The query pulls data from the Car and Rental\_Details tables to return any cars whose return date matches the inputted date. This search helps operations staff by giving them notice of cars likely to be returned soon. Both searches showcase how a well designed Relational Database can guarantee certain requested information is available when needed. This further means that core businesses services including customer support, booking availability and maintenance scheduling are all enabled Screenshots of these queries accompanied by the usernames of the students were taken to ensure documentation and verification of the same.

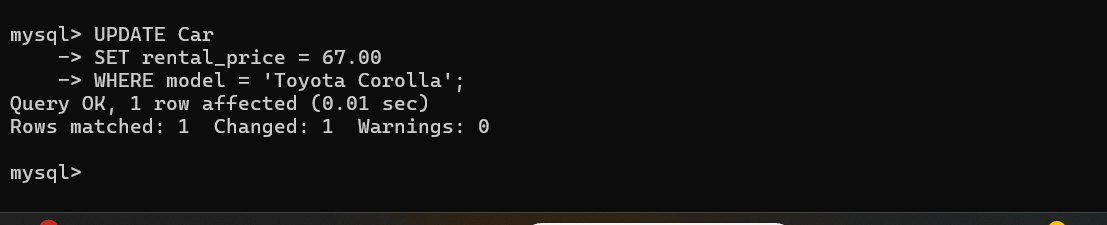
## Case 3: Data Manipulation

**a Booking sedan record**



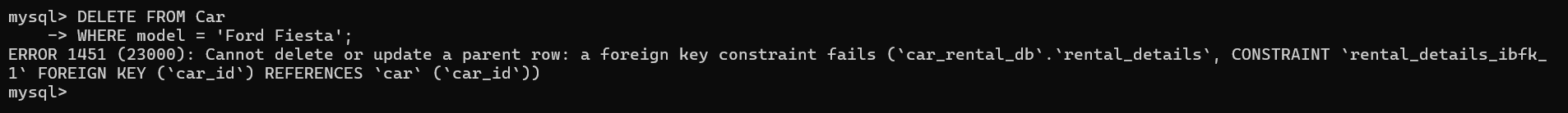
**Figure 11: Sedan availability**

### b Update Car



**Figure 12: Update Car**

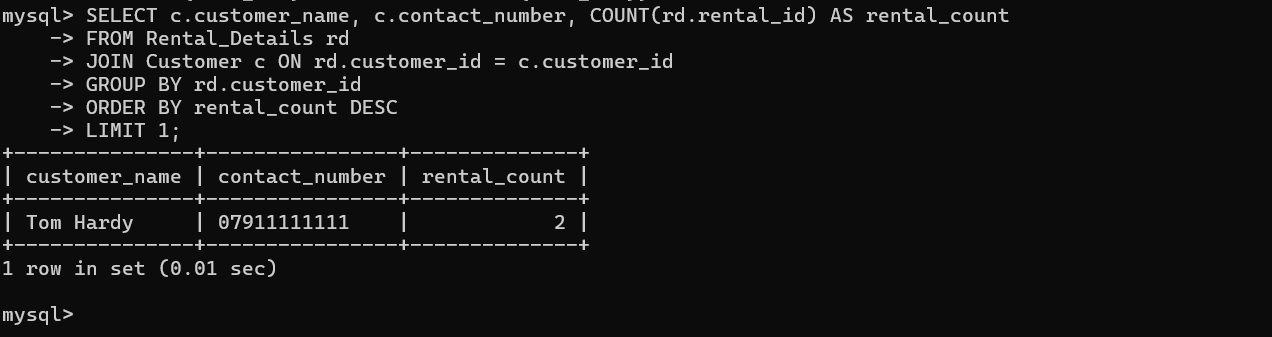
### c Delete from car



**Figure 13:** **Delete from car**

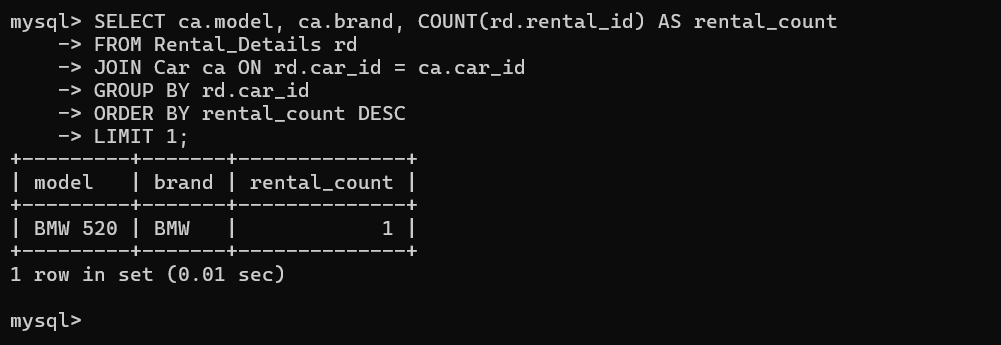
## Case 4: Statistics

### a Customer Rented Most



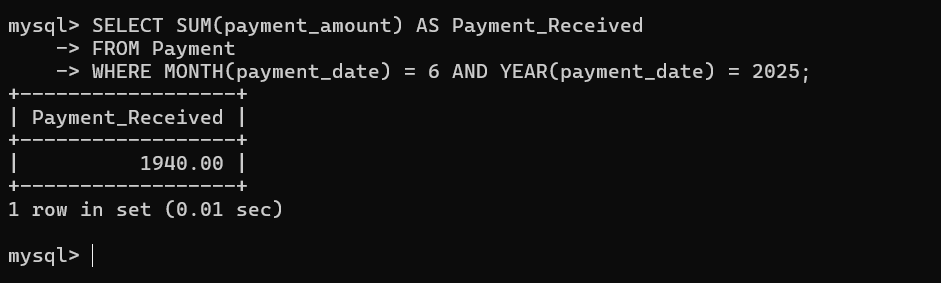
**Figure 14:** **Customer Rented Most**

### b Frequent Rented cars



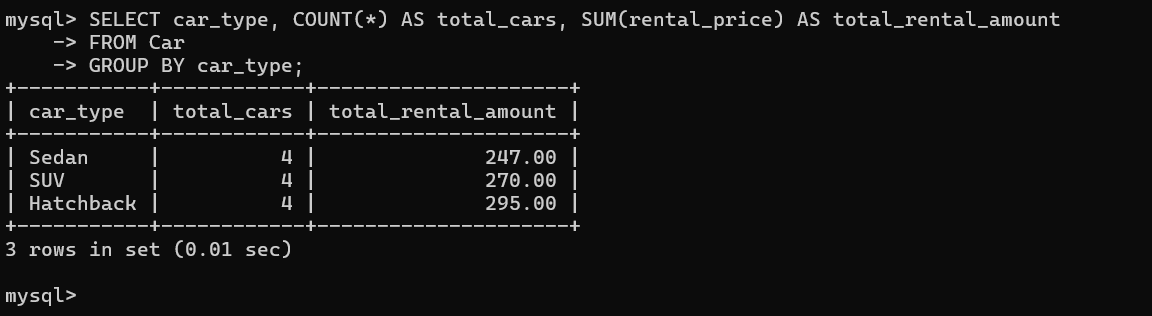
**Figure 15: Frequently Rented cars**

### c Transaction received

****

**Figure 16: Transaction received**

### d Car Type

****

**Figure 17: Car Type**

The purpose of Case 3 was to test the system’s capability to execute the common data manipulation tasks by mimicking the real world abstractions in a car rental business. A new customer was inserted into the table as “School of Computing” and a car rental record was justified by adding/mutating an entry into the Rental\_Details table for the Sedan car starting on the 25th May 2025. The return\_date was calculated using the SQL DATE\_ADD () function. So, we were trying to simulate the process of booking a car and satisfying all foreign key constraints while doing that. Subsequently, we did a price update of “Toyota Corolla” by setting its rental\_price field from £60 to £67 which reflects a pricing policy alteration that was effective starting the next month Finally, we deleted the “Ford Fiesta” from the Car table due to mechanical fault. This was to confirm that the system was able to handle the updates and deletions while maintaining the data integrity.

Case 4 on the other hand, focused on gaining some statistical insights from the data. The first query identified the key renting customer through grouping an counting the records of each rental by customer. The second one extracted the most rented car. A monthly revenue report was generated by using the SUM () function to total the payments received in a month. Lastly, grouping of the cars by their price and head to head pricing tests has helped in business analysis. These operations highlighted the database's robustness in responding to operational alterations and its utility in generating concrete insights aiding business decisions.