

Team 104

Saumya Tiwari (2021351) & Yash Chauhan (2021368)

Saarthi - Cab Booking System

Abstract

This report describes the design and implementation of a cab booking database system for the course - Database Management Systems (Winter 2023). The system allows customers to book a cab for a desired destination and time and provides drivers with a platform to view and accept customer requests. The database is designed to support the efficient management of cab booking data, including user information, ride details, and driver records. The database schema is organised into several tables, including a customer table to store user information, a reservation table to store ride details such as pickup and drop-off locations and timestamps, and a driver table to store driver records. The system also includes several other tables to support operations such as billing, ride rating, payment, and route details. There is an admin table which manages all of the above-mentioned tables and acts as a supervisor. The database is implemented using MySQL, a widely used open-source relational database management system. The system is designed to support a large number of users and ride requests and to handle concurrent access to the database by multiple users and drivers. The front end is designed with the help of Python, and it is in the form of a command line interface which allows users to give input and retrieve data from the database.

Introduction

For the course project, we were given four choices - "Online Grocery Store", "Pharmaceutical Company", "Dairy Company", and "Online Cab Booking System". We decided to work on the lattermost. In recent years, the demand for ride-hailing services has grown rapidly, driven by the convenience and accessibility of app-based cab booking platforms. These platforms have revolutionised the way people travel, providing a more convenient and efficient alternative to traditional taxi services. However, managing the large volumes of data generated by these platforms can be a daunting task, requiring a robust and scalable database system.

We started out by first deciding the scope of the project as well as the tech stack. After that, we developed data schema, models, and E-R diagrams and implemented the relational database using MySQL and the command line interface using Python. Throughout this user guide, we will explain all the work that was done throughout the semester and finally explore the features and workings of our project.

Project Scope and Tech Stack

The application aims to allow potential riders to find the most suitable cab/driver near them along the lines of Uber/Ola.

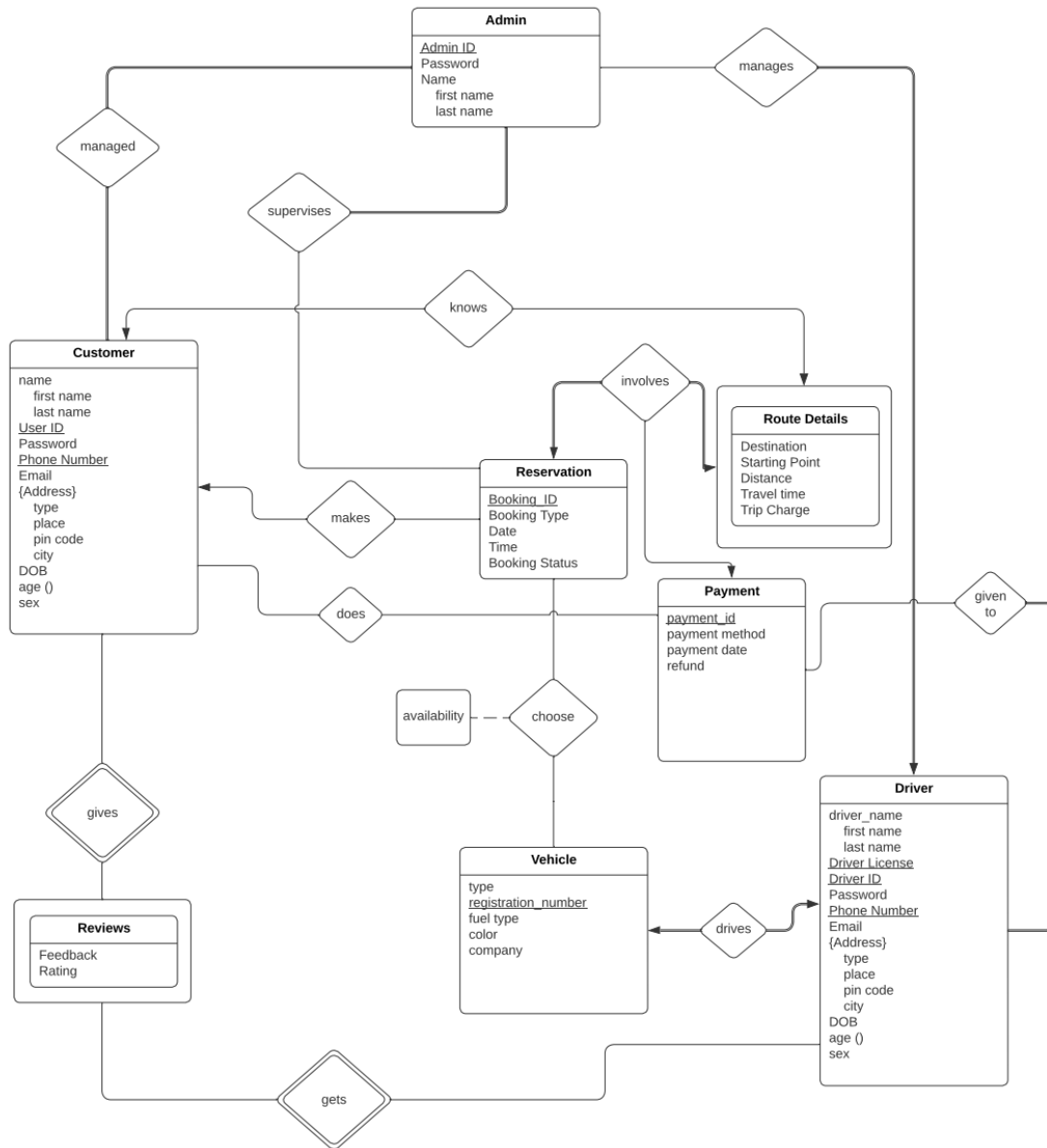
1. User registration and login: Allows users to create an account, log in, and manage their personal information and booking history.
2. Booking: Allows users to request a ride, view the estimated fare, and select a vehicle type.
3. Payment: Allows users to make payments for the ride, typically through a credit card or online payment system.
4. Dispatch and routing: Allows the transportation company or independent drivers to receive and accept ride requests and navigate to the pick-up location using a GPS-enabled device.
5. Tracking and monitoring: Allows users to track the location of their vehicle in real time and receive notifications on the status of their ride.
6. Rating and feedback: Allows users to rate their ride experience and provide feedback to the transportation company or independent driver.
7. Admin Panel: Allows the owner to manage the whole system, including drivers, rides, earnings, and other related things

Tech Stack

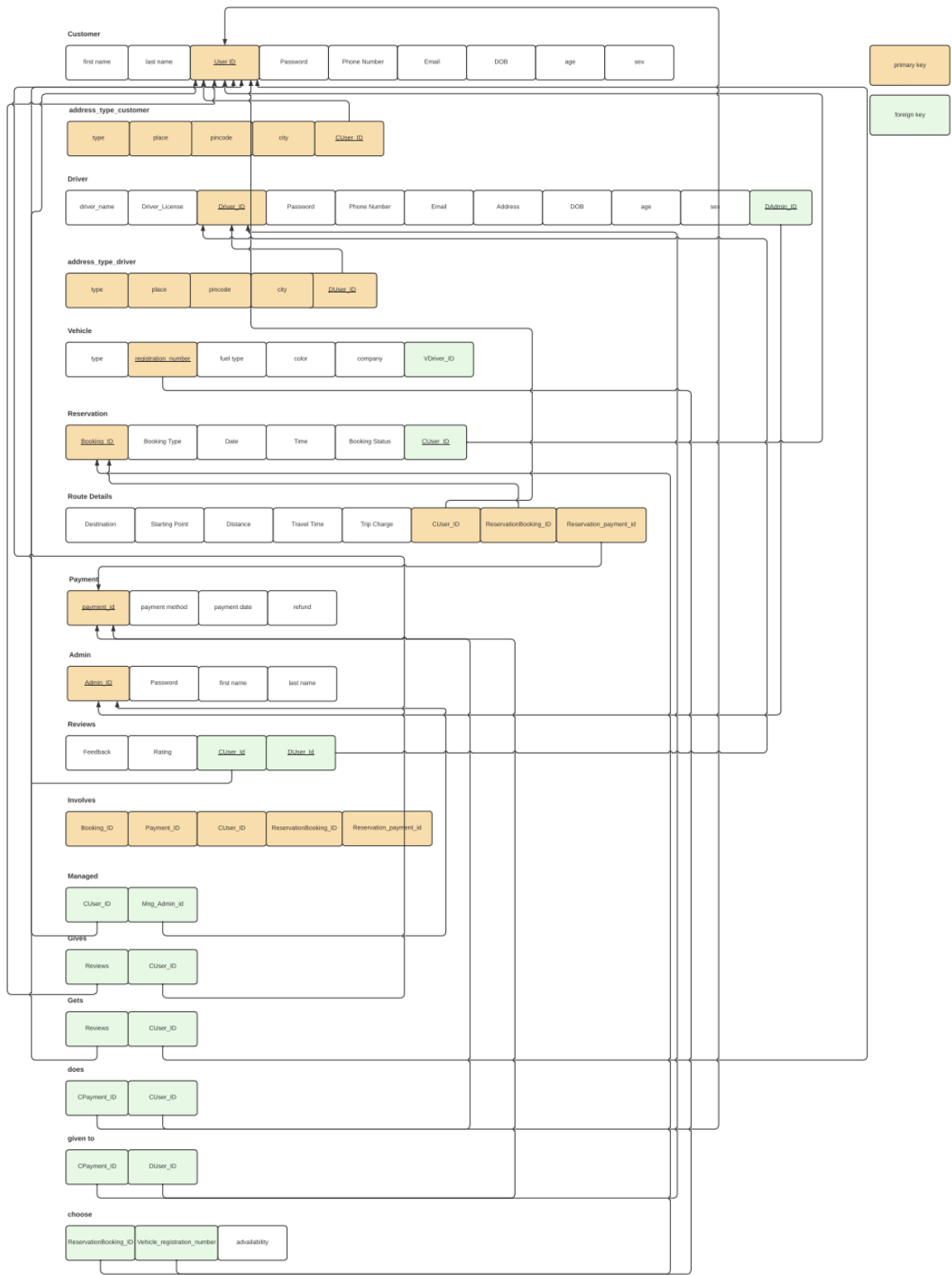
Front end: Command Line Interface using Python

Back end: MySQL

E-R Diagram and Relational Model



Relational Model - Saarthi



Implementation using MySQL

We used the following queries to create the database for our cab booking service Saarthi on the MySQL 8.0 workbench :

```
create database SAARTHI;  
USE SAARTHI;
```

We then used our ER Diagram and Relational Model submitted in the second deadline to create all the necessary entities and keys required to create relationships. **The commands used were as follows:**

```
create table admin(  
admin_id varchar(50) not null,  
password varchar(100),  
first_name varchar(50),  
last_name varchar(50),  
primary key (admin_id) );
```

```
create table customer(  
user_id int not null,  
first_name varchar(100) not null,  
last_name varchar(45),  
password_user varchar(50),  
phone_number int(10),  
email varchar(200),  
dob date not null,  
age int, sex varchar(20),  
Primary key(user_id),  
unique key(email), unique key(phone_number) );
```

```
create table driver(  
driver_name varchar(100) not null,  
driver_license varchar(50) not null,  
driver_id int(50),  
password varchar(50),  
phone_number int(10),  
email varchar(50),  
dob date not null,  
age int, sex char(1),  
DAdmin_ID varchar(50) not null,  
Foreign Key(DAdmin_ID) references admin (admin_id),  
Primary key(driver_id), UNIQUE KEY driver_license_UNIQUE (driver_license) );
```

We created separate entities for multivalued attributes required by a couple entities as follows:

```
create table address_type_driver(  
driver_id int,  
address_type varchar(15),  
place varchar(50),  
pincode int,  
city varchar(50),  
Primary key(driver_id),  
foreign key(driver_id) REFERENCES driver(driver_id) );
```

```
create table address_type_customer(  
user_id int,  
address_type varchar(15),  
place varchar(50),  
pincode int(10),  
city varchar(50),  
Primary key(user_id),  
foreign key(user_id) REFERENCES customer(user_id) );
```

```
create table vehicle(  
vehicle_type varchar(50) not null,  
registration_number varchar(40) not null,  
fuel_type varchar(50),  
color varchar(20),  
company varchar(50),  
VDriver_ID int,  
Primary key(registration_number), foreign key(VDriver_ID) REFERENCES driver(driver_id) );
```

```
create table reservation(  
booking_id varchar(30),  
booking_type varchar(30),  
date_of_booking date, time_of_booking time,  
booking_status varchar(20),  
CUser_id int, foreign key (CUser_id) references customer(user_id),  
primary key(booking_id) );
```

```
create table payment(  
payment_method varchar(30),  
payment_date date, refund char(1),  
payment_id int not null, primary key(payment_id) );
```

```

create table route_Details(
destination varchar(150),
starting_point varchar(150),
distance float,
travel_time varchar(30),
trip_charge float,
CUser_id int,
reservation_booking_id varchar(30),
reservation_payment_id int,
foreign key (reservation_payment_id) references payment(payment_id),
foreign key (CUser_id) references customer(user_id),
primary key(reservation_booking_id),
foreign key (reservation_booking_id) references reservation(booking_id) );

```

```

create table reviews(
feedback varchar(200),
rating int,
CUser_id int,
DUser_id int,
foreign key (CUser_id) references customer(user_id),
foreign key (DUser_id) references driver(driver_id) );

```

Link to the folder with data:

[DBMS Data](#)

All the data produced to populate the relational tables was generated by online software Mockaroo.

SQL queries demonstrating the functionality of our application:

Selection

BOOKINGS MADE BY A SPECIFIC USER:

```

SELECT *
FROM reservation
WHERE CUser_id = 78 ;

```

Result Grid						
	booking_id	booking_type	date_of_booking	time_of_booking	booking_status	CUser_id
▶	12	reservation	2022-10-09	15:32:22	cancelled	78
	66	reservation	2022-05-24	05:19:44	successful	78
*	NULL	NULL	NULL	NULL	NULL	NULL

RESERVATIONS MADE FOR A SPECIFIC CAB:

SELECT *

FROM vehicle natural join reservation




where registration_number = "0BnYKp7" ;

Result Grid		Filter Rows:		Export:		Wrap Cell Content:						
	vehicle_type	registration_number	fuel_type	color	company	VDriver_ID	booking_id	booking_type	date_of_booking	time_of_booking	booking_status	CUser
▶	sedan	0BnYKp7	petrol	Violet	Sable	90	1	reservation	2022-02-22	09:30:23	cancelled	100
	sedan	0BnYKp7	petrol	Violet	Sable	90	10	reservation	2022-12-28	07:08:22	cancelled	93
	sedan	0BnYKp7	petrol	Violet	Sable	90	100	now	2022-04-09	12:47:05	delayed	6
	sedan	0BnYKp7	petrol	Violet	Sable	90	11	reservation	2023-01-17	23:18:01	cancelled	96
	sedan	0BnYKp7	petrol	Violet	Sable	90	12	reservation	2022-10-09	15:32:22	cancelled	78
	sedan	0BnYKp7	petrol	Violet	Sable	90	13	reservation	2022-02-13	15:55:46	cancelled	99
	sedan	0BnYKp7	petrol	Violet	Sable	90	14	reservation	2022-10-28	14:55:29	cancelled	32
	sedan	0BnYKp7	petrol	Violet	Sable	90	15	reservation	2022-04-08	06:14:24	cancelled	49
	sedan	0BnYKp7	petrol	Violet	Sable	90	16	reservation	2022-11-10	22:41:41	cancelled	60
	sedan	0BnYKp7	petrol	Violet	Sable	90	17	reservation	2022-11-21	06:37:57	cancelled	67
	sedan	0BnYKp7	petrol	Violet	Sable	90	18	reservation	2022-06-28	01:50:10	cancelled	91
	sedan	0BnYKp7	petrol	Violet	Sable	90	19	reservation	2022-04-28	15:51:05	cancelled	36
	sedan	0BnYKp7	petrol	Violet	Sable	90	2	reservation	2022-06-15	06:10:46	cancelled	37
	sedan	0BnYKp7	petrol	Violet	Sable	90	20	reservation	2022-05-25	00:18:56	cancelled	53

PROJECTIONS OF BOOKINGS MADE IN A SPECIFIC TIME RANGE:

SELECT booking_id, date_of_booking, CUser_id, time_of_booking

FROM RESERVATION WHERE time_of_booking >= '15:00:00' AND time_of_booking <= '20:00:00' ;

Result Grid			 Filter Rows:	<input type="text"/>	Edit: 
	booking_id	date_of_booking	CUser_id	time_of_booking	
▶	12	2022-10-09	78	15:32:22	
	13	2022-02-13	99	15:55:46	
	19	2022-04-28	36	15:51:05	
	25	2022-08-31	11	17:43:29	
	27	2023-01-13	10	16:31:30	
	3	2022-09-11	84	15:26:57	
	34	2022-07-19	60	16:21:42	
	35	2022-05-31	80	16:34:22	
	39	2023-01-20	39	16:55:21	
	41	2022-06-23	7	19:12:29	
	43	2022-09-27	38	19:48:05	
	44	2023-01-21	5	15:48:03	
	46	2022-10-17	40	15:22:13	
	50	2022-05-28	8	17:18:04	
	73	2022-09-16	24	17:21:11	

MOST POPULAR PICK-UPS AND DROP-OFF LOCATIONS (BY USING AGGREGATE FUNCTIONS):

SELECT destination,

COUNT(*) as num_pickups

FROM route_details

GROUP BY destination

ORDER BY num_pickups DESC LIMIT 5;

Result Grid			Filter Rows:
	destination	num_pickups	
▶	Azul	1	
	Gibato	1	
	Krapivna	1	
	Mantingantengah	1	
	Yazykovo	1	

```

SELECT starting_point,
COUNT(*) as num_pickups
FROM route_details
GROUP BY starting_point
ORDER BY num_pickups DESC LIMIT 5;

```

Result Grid			Filter Rows:
	starting_point	num_pickups	
▶	Xinhua	2	
	Zouma	1	
	Gunungmenang	1	
	Brušperk	1	
	Ushibuka	1	

RETRIEVING AVERAGE RATING OF EACH CAB (AGGREGATE FUNCTIONS):

```

SELECT registration_number,
AVG(rating) as avg_rating
FROM reviews natural join vehicle
GROUP BY registration_number;

```

Result Grid			Filter Rows:
	registration_number	avg_rating	
▶	5tccDMfpOt	2.9300	
	jMwwdma7nmoU	2.9300	
	T3lGUfj	2.9300	
	HPBXfy	2.9300	
	Z7xVh8s	2.9300	
	H1dM43f	2.9300	
	sQGQcVvXa	2.9300	
	do5KZGZ	2.9300	
	c8qSxbF	2.9300	
	2Ob10ssKg	2.9300	
	0BnYKp7	2.9300	
	cox8TWpVshw	2.9300	
	hLUsvC	2.9300	
	gZRTSuyo	2.9300	
	MwomMNj	2.9300	

RETRIEVING TOP RATED CAB (AGGREGATE FUNCTIONS):

```
SELECT VDriver_ID, registration_number,
AVG(rating) as avg_rating
FROM reviews natural join vehicle
GROUP BY registration_number
ORDER BY avg_rating DESC LIMIT 1;
```

Result Grid			
	VDriver_ID	registration_number	avg_rating
▶	100	5tccDMmfpOt	2.9300

DISTANCE TRAVELED BY A USER:

Select CUser_id,SUM(distance) from route_details group by CUser_id;

Result Grid				
	user_id	first_name	last_name	SUM(distance)
▶	1	Verla	Mollene	4930.300007224083
	2	Erich	Branwhite	4930.300007224083
	3	Waverley	Schule	4930.300007224083
	4	Petey	Stiger	4930.300007224083
	5	Charita	McTurlough	4930.300007224083
	6	Emory	Samper	4930.300007224083
	7	Yancey	Sheffield	4930.300007224083
	8	Derek	Pauncefort	4930.300007224083
	9	Ruddie	Brockbank	4930.300007224083
	10	Carry	Kyd	4930.300007224083
	11	Johnna	Allibone	4930.300007224083
	12	Wallis	Albasiny	4930.300007224083
	13	Abdul	Lammert	4930.300007224083
	14	Shirleen	Ricci	4930.300007224083
	15	Shurwood	Ledbury	4930.300007224083

VIEWING CONSTRAINTS OF AN ENTITY:

desc customer;

Result Grid						
	Field	Type	Null	Key	Default	Extra
▶	user_id	int	NO	PRI	NULL	
	first_name	varchar(100)	NO		NULL	
	last_name	varchar(100)	YES		NULL	
	password_user	varchar(45)	NO		NULL	
	phone_number	varchar(10)	YES	UNI	NULL	
	email	varchar(200)	NO	UNI	NULL	
	dob	date	NO		NULL	
	age	int	NO		NULL	
	sex	varchar(20)	YES		NULL	

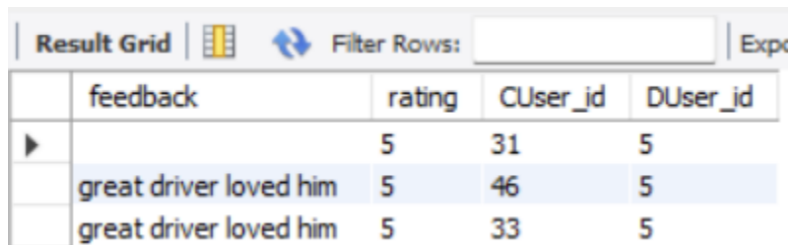
DATA MANIPULATION USING "UPDATE" COMMAND:

UPDATE reviews

SET rating = 5

WHERE DUser_id = 5;

Select * from reviews where DUser_id = 5;



	feedback	rating	CUser_id	DUser_id
▶		5	31	5
	great driver loved him	5	46	5
	great driver loved him	5	33	5

Triggers

Here are the two triggers we specified for our project.

TRIGGER ONE : It keeps a check on the ride fare and makes sure that the user does not enter any negative amount in the field entered, and if that does happen, it gets changed to 0 to prevent any errors.

```
Create trigger before_insert_tripcharge
Before Insert ON route_details for EACH ROW
BEGIN
IF NEW.trip_charge < 0 then set NEW.trip_charge = 0 ;
END IF;
END
```

TRIGGER TWO : It keeps a check on the user details (in particular, name) to see that if they did not fill their first name correctly, it is replaced with the customer user ID for no confusion and easy access.

```
Create trigger before_insert_firstname
Before Insert ON customer for EACH ROW
BEGIN
IF NEW.first_name = " " then set NEW.first_name = NEW.user_id ;
END IF;
END
```

OLAP Queries

OLAP ONE: This query retrieves data from multiple tables in the database and aggregates it using various functions like SUM, COUNT, and AVG to provide insights into the performance of the cab booking system by analyzing things like identifying popular routes, popular drivers, and popular vehicles. It can also help identify any issues with payment processing or customer satisfaction by analyzing the payment and review data.

```
SELECT driver.driver_name, route_details.distance, vehicle_details.vehicle_type,  
SUM(payment.fare) as total_payments, COUNT(reservation.booking_id) as total_reservations,  
AVG(reviews.rating) as avg_rating  
FROM vehicle_details  
JOIN driver ON vehicle_details.VDriver_ID = driver.driver_id  
JOIN route_details ON vehicle_details.vehicle_booking_id =  
route_details.reservation_booking_id  
JOIN reservation ON reservation.booking_id = vehicle_details.vehicle_booking_id  
JOIN payment ON payment.payment_id = reservation.booking_id  
LEFT JOIN reviews ON reviews.CUser_id = reservation.CUser_id  
GROUP BY vehicle_details.vehicle_type, driver.driver_name, route_details.distance  
ORDER BY total_payments DESC;
```

Result Grid

Filter Rows:

Export:

Wrap Cell Content:

	driver_name	distance	vehicle_type	total_payments	total_reservations	avg_rating
▶	Harper	26.9	hatchback	45570.20849609375	7	3.0000
	Leland	65.6	hatchback	45570.20849609375	7	3.0000
	Rosy	97.2	suv	39060.1787109375	6	2.3333
	Arleyne	43.8	sedan	39060.1787109375	6	2.3333
	Dun	14.3	hatchback	39060.1787109375	6	2.3333
	Georas	36.6	suv	32550.14892578125	5	3.4000
	Aviva	95	hatchback	32550.14892578125	5	3.0000
	Ramona	57	suv	32550.14892578125	5	3.4000
	Crystal	68	hatchback	26040.119140625	4	3.2500
	Collen	75.9	suv	26040.119140625	4	2.2500
	Araldo	31.7	sedan	26040.119140625	4	3.5000
	Grove	59.2	sedan	26040.119140625	4	3.5000
	Bearnard	87	suv	26040.119140625	4	2.2500
	Mariette	10	sedan	26040.119140625	4	3.2500

OLAP TWO: (with rollup)

This query uses GROUP BY ROLLUP to provide a summary of the number of reservations and total payments for each combination of vehicle model, driver name, and route distance, as well as for the total of all combinations. We have used “total” to fill in null values while calculating rollup.

SELECT

IFNULL(vehicle_details.vehicle_type, 'Total') as type_or_model,

IFNULL(driver.driver_name, 'Total') as DriverName,

IFNULL(route_details.distance, 'Total') as distance,

COUNT(DISTINCT reservation.booking_id) as total_reservations,

SUM(payment.fare) as total_payments

FROM reservation

JOIN vehicle_details ON reservation.booking_id = vehicle_details.vehicle_booking_id

JOIN driver ON vehicle_details.VDriver_ID = driver.driver_id JOIN route_details ON

vehicle_details.vehicle_booking_id = route_details.reservation_booking_id

JOIN payment ON payment.payment_id = reservation.booking_id

GROUP BY vehicle_details.vehicle_type, driver.driver_name, route_details.distance

WITH ROLLUP ;

Result Grid	  Filter Rows:	Export:	 Wrap Cell Content:		
	type_or_model	DriverName	distance	total_reservations	total_payments
	hatchback	Wernher	63	1	6510.02978515625
	hatchback	Wernher	Total	1	6510.02978515625
	hatchback	Total	Total	35	227851.04248046...
	sedan	Aimil	12.4	1	6510.02978515625
	sedan	Aimil	Total	1	6510.02978515625
	sedan	Amabelle	69.9	1	6510.02978515625
	sedan	Amabelle	Total	1	6510.02978515625
	sedan	Amitie	57.9	1	6510.02978515625
	sedan	Amitie	Total	1	6510.02978515625
	sedan	Andrei	74.5	1	6510.02978515625
	sedan	Andrei	Total	1	6510.02978515625
	sedan	Araldo	31.7	1	6510.02978515625
	sedan	Araldo	Total	1	6510.02978515625
	sedan	Arleyne	43.8	1	6510.02978515625
	sedan	Arleyne	Total	1	6510.02978515625
	sedan	Benjamin	28.7	1	6510.02978515625
	sedan	Benjamin	Total	1	6510.02978515625
	sedan	Bernv	3.1	1	6510.02978515625

OLAP THREE:

This query provides data that can help us find the most popular vehicle for a particular location by directly looking at the maximum payment made to reach a particular destination. For explanation, the result has been displayed at the end of the OLAP query as well.

SELECT

IFNULL(vehicle_details.vehicle_type, 'Total') as vehicle_type,

IFNULL(driver.driver_name, 'Total') as driver,

IFNULL(route_details.destination, 'Total') as destination,

SUM(payment.fare) as total_payments

FROM reservation

JOIN vehicle_details ON reservation.booking_id = vehicle_details.vehicle_booking_id

JOIN driver ON reservation.CUser_id = driver.driver_id

JOIN route_details ON reservation.booking_id = route_details.reservation_booking_id

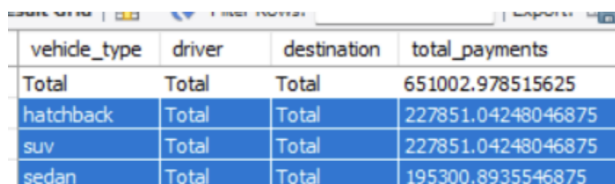
JOIN payment ON payment.payment_id = reservation.booking_id

GROUP BY vehicle_details.vehicle_type, driver.driver_name, route_details.destination with
rollup

ORDER BY driver;

WHEN WE MAKE THE ABOVE INTO A VIEW AND RUN THE BELOW QUERY

select * from find_most_popular_destination order by total_payments desc;



vehicle_type	driver	destination	total_payments
Total	Total	Total	651002.978515625
hatchback	Total	Total	227851.04248046875
suv	Total	Total	227851.04248046875
sedan	Total	Total	195300.8935546875

THE RESULTS OF THE ORIGINAL OLAP QUERY ARE:

vehicle_type	driver	destination	total_payments
suv	Cordelia	Total	6510.02978515625
suv	Cordelia	Senduro	6510.02978515625
sedan	Cordelia	Total	6510.02978515625
sedan	Cordelia	Polzela	6510.02978515625
sedan	Crystal	Total	13020.0595703125
sedan	Crystal	Cincinnati	6510.02978515625
sedan	Crystal	Kumanovo	6510.02978515625
suv	Dannel	Total	6510.02978515625
suv	Dannel	Bulaevo	6510.02978515625
hatchback	Darrin	Gibato	6510.02978515625
hatchback	Darrin	Total	6510.02978515625
sedan	Darrin	Total	6510.02978515625
sedan	Darrin	Hitoyoshi	6510.02978515625
suv	Devina	Total	6510.02978515625
suv	Devina	Jaru	6510.02978515625
hatchback	Dex	Il'skiy	6510.02978515625
hatchback	Dex	Jiuhe	6510.02978515625
hatchback	Dex	Total	13020.0595703125

OLAP FOUR:

This query uses the GROUP BY rollup clause to generate a result set that includes subtotals and totals for all combinations of the customer and driver names. This query can help you analyze the revenue generated by each customer and driver and identify any patterns or trends in their booking behavior.

SELECT

IFNULL(customer.first_name, 'TOTAL') AS customerNAME,

IFNULL(vehicle_details.VDriver_ID, 'TOTAL') as DriverID,

SUM(payment.fare) AS total_payment



FROM reservation

JOIN customer ON reservation.CUser_id = customer.user_id

JOIN vehicle_details ON reservation.booking_id = vehicle_details.vehicle_booking_id

JOIN payment ON reservation.booking_id = payment.payment_id

GROUP BY customer.first_name, vehicle_details.VDriver_ID with rollup ;

Result Grid   Filter Rows: <input type="text"/>			
	customerNAME	DriverID	total_payment
▶	Aggi	58	6510.02978515625
	Aggi	TOTAL	6510.02978515625
	Arabele	4	6510.02978515625
	Arabele	33	6510.02978515625
	Arabele	TOTAL	13020.0595703125
	Avie	18	6510.02978515625
	Avie	91	6510.02978515625
	Avie	TOTAL	13020.0595703125
	Avigdor	86	6510.02978515625
	Avigdor	TOTAL	6510.02978515625
	Bethanne	95	6510.02978515625
	Bethanne	TOTAL	6510.02978515625
	Biron	45	6510.02978515625
	Biron	49	6510.02978515625

Embedded SQL Queries in Python

```
import sqlite3
import sys
import mysql.connector

connection1 = mysql.connector.connect(host = 'localhost', username =
'root', password = 'root', database = 'saarthi' )

my_cursor = connection1.cursor()
connection1.commit()

# BOOKINGS MADE BY A SPECIFIC USER:
print ( 'this is a query to check BOOKINGS MADE BY A SPECIFIC USER ' )
input1 = input("WHICH CUSTOMER ID DETAILS DO YOU WANT TO KNOW?")
query1 = " SELECT * FROM reservation WHERE CUser_id = %s " % input1
my_cursor.execute(query1)
result1 = my_cursor.fetchall()
for i in result1:
    print(i)

# PROJECTIONS OF BOOKINGS MADE IN A SPECIFIC TIME RANGE:

print ( 'this is a query to check PROJECTIONS OF BOOKINGS MADE IN A
SPECIFIC TIME RANGE ' )
t1 = "" + input(' enter starting time of booking (HH:MM:SS)') + ""
t2 = "" + input(' enter ending time of booking (HH:MM:SS)') + ""
query2 = " SELECT booking_id, date_of_booking, CUser_id FROM RESERVATION
WHERE time_of_booking >= %s AND time_of_booking <= %s " % (t1,t2)
my_cursor.execute(query2)
result1 = my_cursor.fetchall()
for i in result1:
    print(i)

connection1.close()

# print("connection successful !")
```


Command Line Interface and Implementation

We created our own self help library of functions so that the main run file is not too code heavy and complicated to understand. (self_library.py)

We establish a connection with the sql server and constantly use read and write operations to update and manipulate the database to our needs.

On running the CLI_Saarthi.py file on a command line you are asked to enter your name in order to recognise you as a first time or older user.

```
WELCOME TO SAARTHI, INDIA'S NUMBER ONE ONLINE CAB SERVICE
Please enter your first name : Saumya
Please enter your last name : Tiwari
error : We can't seem to find you here!
Let's create a new profile for you!
```

Incase you are already a registered user, this step is completely omitted. Then the user goes through a profile creation process which asks them the following details :

```
please enter a strong password within 45 characters *****
phone number : 9205662360
email : saumyatiwari510@gmail.com
date (YYYY-MM-DD): 2003-10-10
age : 20
sex (F/M): F
address type (work/home/other): work
place : okhla
pincode : 110020
city : delhi
```

As soon as the above is done, the program saves all customer details into the database. They are then asked their cab booking details as follows:

```
Press 1 to book a cab or 0 to exit 1
Press 1 to reserve or 0 to book now 0
please enter your booking details :
final destination : noida sector 82
current location : okhla phase 3
do you wish to leave now or reserve it? (now/reservation) : now
choose method of payment(cash/card/UPI) : cash
```

```
enter date of payment(yyyy-mm-dd) : 24-04-23
```

As you can see, you can also choose to reserve a cab for a future time, below is an example:

```
WELCOME TO SAARTHI, INDIA'S NUMBER ONE ONLINE CAB SERVICE
```

```
Please enter your first name : yash
```

```
Please enter your last name : chauhan
```

```
Press 1 to book a cab or 0 to exit1
```

```
Press 1 to reserve or 0 to book now1
```

```
enter date of booking(yyyy-mm-dd) : 2023-10-10
```

```
enter time of booking (hh:mm:ss): 16:00:00
```

```
please enter your booking details :
```

```
final destination : goa
```

```
current location : delhi
```

```
do you wish to leave now or reserve it? (now/reservation) :
```

```
reservation
```

```
reservation
```

```
choose method of payment(cash/card/UPI) : cash
```

```
enter date of payment(yyyy-mm-dd) : 2023-10-10
```

```
your trip costs : 291.378831319472
```

```
would you like to leave a review ? y/n?n
```

```
Thank you for using Saarthi!
```

Then, for simplicity's sake we used the Python random library to generate distance values between the starting point and destination and a random trip fare to display to the customer.

```
your trip costs : 317.772962790316
```

```
would you like to leave a review ? y/n?n
```

```
Thank you for using Saarthi!
```

Transactions

Here are the non-conflicting queries we ran on our database.

One (and proof):

```
1 • START TRANSACTION;
2
3 • SELECT * FROM driver WHERE driver_id = 1 LIMIT 1 FOR UPDATE;
4 • UPDATE driver SET sex='F' WHERE driver_id = 1;
5 • COMMIT;
```

Result Grid

	driver_name	driver_license	driver_id	password_driver	phone_number	email	dob	age	sex	DAdmin_ID
▶	Manfred	RX668890	1	q3uvxzyDLqGX	8034266598	mmoxson0@spotify.com	1980-06-14	44	F	1
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Two:

```
1 • START TRANSACTION;
2
3 • SELECT * FROM payment WHERE payment_id=1 LIMIT 1 FOR UPDATE;
4 • UPDATE payment SET payment_date='2023-04-20' WHERE payment_id = 1;
5 • COMMIT;
```

Proof for the second transaction:

```
1
2
3 • SELECT * FROM payment WHERE payment_id=1;
4
5
6
7
```

Result Grid

	payment_method	payment_date	refund	payment_id	fare
▶	Credit Card	2023-04-20	0	1	100
*	NULL	NULL	NULL	NULL	NULL

Third :

[illegible]

Proof for the third transaction:

```
1 • START TRANSACTION;
2
3 • SELECT * FROM customer WHERE user_id=1 LIMIT 1 FOR UPDATE;
4 • COMMIT;
```

	user_id	first_name	last_name	password_user	phone_number	email	dob	age	sex
▶ 1	HULL	Ashish	Borit	3Q2E2sXSIB8E	6965450107	pborit0@msu.edu	1966-11-15	31	F

Four (and proof):

[illegible]

Conflicting Transactions

For our project, the following are the 2 cases where conflicting transactions could occur:

- 1) 2 users try to book a single cab

T1 -> User A books a cab for the time 10-11 AM

T2 -> User B books a cab for the time 11-12 PM

The following are the read and write sets for the above transactions

T1: Read the customer data (A) and the driver availability and writes to its data set that the driver is unavailable for the time

T2: Reads the customer data (B) and driver availability and writes to its data set that the driver is unavailable for the time

These 2 transactions are conflicting as T1 writes to the data set of the driver and T2 reads from it.

Conflict Graph:

T1->T2

As you can see the graph is acyclic, hence it is conflict serialisable. The schedule for the above conflicting transactions will be:

Transaction T1	Transaction T2
Read User A data	
Read Driver data	
Write to User A data	
Write to Driver's availability	
	Read User B data
	Read Driver data
	Write to User B data
	Write to driver's availability

We give T1 an exclusive lock and unlock it before T2 starts executing and give it an exclusive lock and unlock after it has been committed.

T1 acquires an exclusive lock on the driver's data item and then writes a booking with user A. After the trip, the driver's data item gets unlocked. Similarly for T2.

This schedule will keep the database consistent and make the schedule not conflict serialisable.

- 2) A customer tries to cancel a cab booking even though a driver has been assigned to them.

T1: User A books a cab and books a driver and writes to the driver's data item as booked

T2: User A tries to cancel the booking and if the driver has not been assigned then write ride cancellation to the user's data item.

Conflict Graph

T1->T2

These 2 transactions are conflicting as T1 writes to the data set of the driver and T2 reads from it.

Transaction T1	Transaction T2
Read User A data	
Read Driver data	
Write to User A data - cab booking	
Write to Driver's assigned	
	Read User A data
	Read Driver data
	Write to User B data - cab canceled
	Write to driver's availability - available

We give T1 an exclusive lock and unlock it before T2 starts executing and give it an exclusive lock and unlock after it has been committed.

T1 acquires an exclusive lock on the driver's data item and then writes a booking with user A. After releasing the lock, we can give T2 an exclusive on user A's data item and read it to check whether the driver is assigned to it or not. After writing to the user A' data item, release the exclusive lock.

This schedule will keep the database consistent and make the schedule not conflict serialisable.

