

Cab Booking System

DBMS Project Group-35

Parveen (2021079)

Shubham Sharma (2021099)

Project Scope

To create an application that will allow cab booking with functionalities that will automate the user data and application data on records using databases. The main aim is to create a fast and reliable application for providing and managing real-time data regarding cabs, drivers, bookings, etc. The application will be able to handle extensive data manipulation requests with proper constraints and minimal redundancy. Data organization will be closely related to real-world entities.

The cab booking system aims to automate key functionalities, such as cab assignment and payment processing, to improve overall operational efficiency. This results in faster booking confirmations, reduced manual effort, and increased productivity.

Stakeholders Identified

I. Users

The application allows users to create an account and login securely using authentication mechanisms. Users can manage their profile information, including name, contact details, and payment preferences. Also, Users can book a cab by specifying the pickup and drop-off locations, preferred cab type, and payment method. The system assigns the nearest available cab and sends a confirmation to the user. Users can also view and manage their booking history, including upcoming and completed rides.

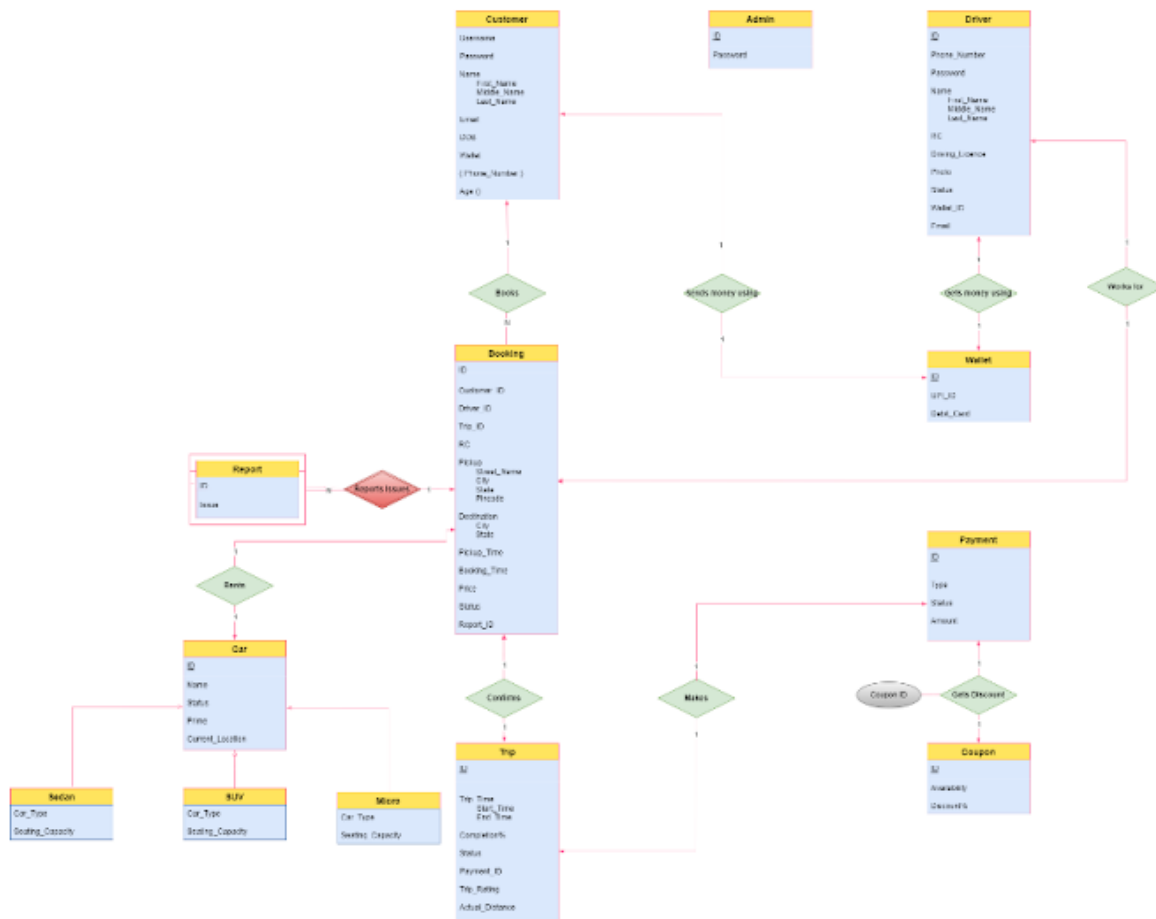
II. Driver

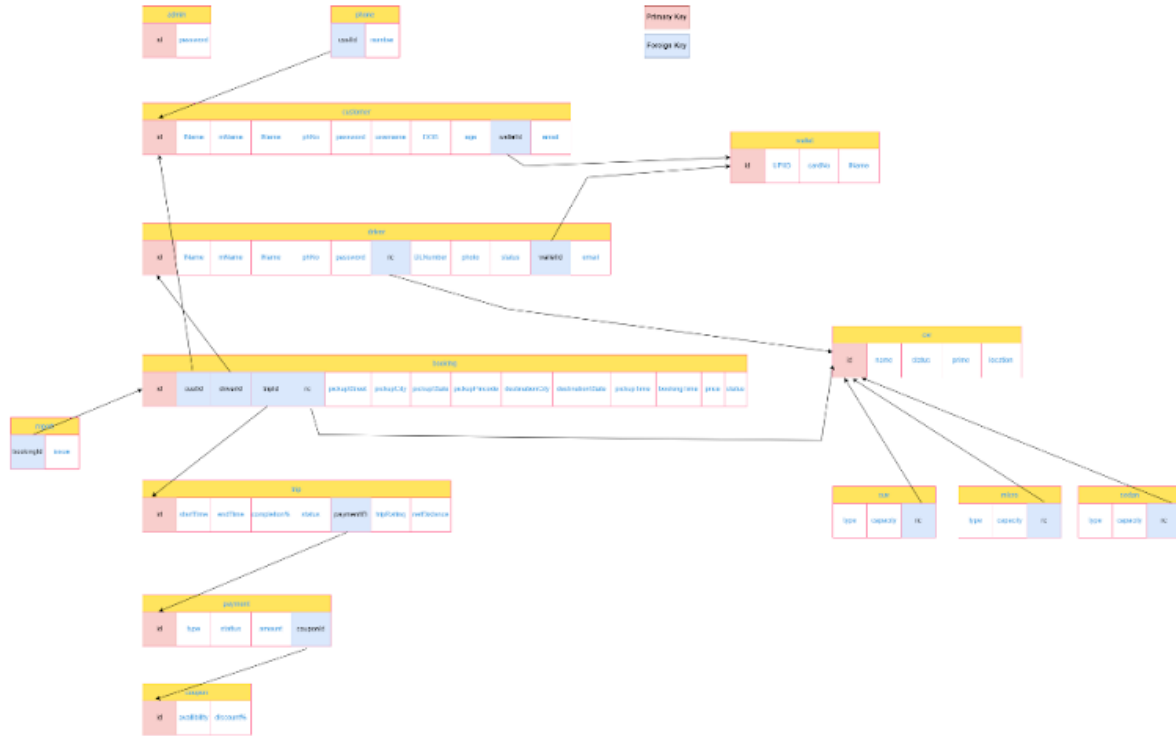
When a ride request is made by a user, the system assigns the nearest available cab to the driver. The driver receives the ride details, including the pickup and drop-off locations, passenger information, and payment method.

III. Admin

The admin panel allows the system administrator to efficiently manage user accounts, cab types, and ride requests. This helps in maintaining system integrity, generating reports, and performing necessary maintenance tasks.

Entity Relationship Model





Entities, Attributes & Schema

I. Admin

Schema: admin(ID, password)

ID	INT PRIMARY KEY NOT_NULL AUTO_INCREMENT
password	VARCHAR(50) NOT NULL

II. Customer

Schema: customer(ID, firstName, midName, lastName, password, username, age, email, phoneNumber, altPhoneNumber)

ID	INT PRIMARY KEY NOT_NULL AUTO_INCREMENT
firstName	VARCHAR(25) NOT NULL
midName	VARCHAR(25) DEFAULT NULL
lastName	VARCHAR(25) DEFAULT NULL
password	VARCHAR(25) NOT NULL
username	VARCHAR(25) NOT NULL
age	INT DEFAULT NULL
email	VARCHAR(25) NOT NULL

phoneNumber	INT NOT NULL
altPhoneNumber	INT DEFAULT NULL

III. Driver

Schema: driver(ID, firstName, lastName, phoneNumber, password, RC, DLNumber, status, email, balance)

ID	INT PRIMARY KEY NOT_NULL AUTO_INCREMENT
firstName	VARCHAR(25) NOT NULL
lastName	VARCHAR(25) DEFAULT NULL
phoneNumber	INT NOT NULL
password	VARCHAR(25) NOT NULL
RC	INT NOT NULL FOREIGN KEY REFERENCES car
DLNumber	INT NOT NULL
status	VARCHAR(10) NOT NULL
email	VARCHAR(50) NOT NULL

IV. Booking

Schema: booking(ID, custId, driverId, RC, pickupStreet, pickupCity, pickupState, pickupPincode, destinationCity, destinationState, pickupTime, bookingTime, price, status)

ID	INT PRIMARY KEY NOT_NULL AUTO_INCREMENT
custId	INT KEY NOT NULL FOREIGN KEY REFERENCES customer
driverId	INT KEY DEFAULT NULL FOREIGN KEY REFERENCES driver
RC	INT KEY DEFAULT NULL FOREIGN KEY REFERENCES car
pickupStreet	VARCHAR(50) NOT NULL
destinationCity	VARCHAR(50) NOT NULL
bookingTime	DATETIME NOT NULL
price	INT NOT NULL
status	VARCHAR(20) NOT NULL
carType	VARCHAR(30) DEFAULT NULL

V. Car

Schema: car(ID, name, status, capacity, type, location)

ID	INT PRIMARY KEY NOT_NULL AUTO_INCREMENT
name	VARCHAR(30) NOT NULL
capacity	INT NOT NULL
type	VARCHAR(30) NOT NULL
location	VARCHAR(70) NOT NULL

VI. Coupon

Schema: coupon(ID, discount)

ID	INT PRIMARY KEY NOT_NULL
discount	INT NOT NULL

VII. Payment

Schema: payment(ID, type, status, amount, couponId)

ID	INT PRIMARY KEY NOT_NULL
type	VARCHAR(15) NOT NULL

status	VARCHAR(10) NOT NULL
amount	INT NOT NULL
couponId	VARCHAR(7) KEY DEFAULT NULL FOREIGN KEY REFERENCES coupon

VIII. Report

Schema: report(bookingId, issue)

bookingId	INT KEY NOT_NULL FOREIGN KEY REFERENCES booking
issue	VARCHAR(200) NOT NULL

IX. Trip

Schema: trip(ID, bookingId, startTime, endTime, completion, status, paymentId, tripRating, netDistance)

ID	INT PRIMARY KEY NOT_NULL AUTO_INCREMENT
bookingId	INT KEY NOT NULL FOREIGN KEY REFERENCES booking
endTime	DATETIME DEFAULT NULL

completion	INT DEFAULT NULL
status	VARCHAR(20) NOT NULL
paymentId	VARCHAR(11) DEFAULT NULL FOREIGN KEY REFERENCES payment
tripRating	INT DEFAULT NULL
netDistance	INT DEFAULT NULL

X. Wallet

Schema: wallet(ID, money)

ID	INT PRIMARY KEY NOT_NULL FOREIGN KEY REFERENCES customer
money	VARCHAR(30) DEFAULT NULL

Indexing Attributes

```
CREATE UNIQUE INDEX username ON customer(username);
```

The above index is useful for queries involving searching directly by username.

```
CREATE INDEX index1 ON booking(status);
```

The above index is useful for queries involving searching directly by status type. It helps to process some analytical queries very fast.

```
CREATE INDEX index2 ON booking(driverId);
```

The above index is useful for queries involving searching directly by driver id. It helps to process queries related to driver net revenue generated very efficiently.

```
CREATE INDEX index1 ON payment(status);
```

The above index is useful for queries involving searching directly by status type. It helps to process some analytical queries very fast.

```
CREATE INDEX index2 ON payment(type);
```

The above index is useful for queries involving searching directly by status type. It helps to process queries very fast belonging to mode of payment chosen. For eg finding revenue generated using online payment only.

Auto-Created Indexes

Indexes on unique keys & foreign keys, as defined in tables, are automatically created by MySQL. For ex:- Index on the payment id in the trip table is created automatically.

Triggers Implemented

I. Data Validation Trigger

This trigger maintains the consistency of data and used to automate some task also it ensures the correctness of data according to real world constraints

```

DELIMITER //
Create trigger TripChecker2
before insert on trip for each row
BEGIN
    DECLARE tripstatus varchar(20);
    IF (NEW.netDistance) < 0 OR (NEW.completion<0) THEN
        SET NEW.status= 'aborted';
    ELSEIF (NEW.completion>=95) THEN
        SET NEW.status = 'success';
    ELSE
        SET NEW.status = 'started';
    END IF;
    IF NEW.completion >100 THEN SET NEW.completion =100;
    END IF;
    IF NEW.netDistance <0 THEN SET NEW.netDistance =0;
    END IF;
    IF NEW.completion <0 THEN SET NEW.completion =0;
    END IF;
    IF NEW.endTime <= (Select bookingTime from booking where id = NEW.bookingId)THEN
        SIGNAL SQLSTATE '50001' SET MESSAGE_TEXT = 'Please enter valid Trip End Time';
    END IF;
END//
DELIMITER ;

```

II. Automated Driver Assignment Trigger

This trigger is used to automate the driver assigning job in the system, as soon as the request is made, a temporary driver is assigned to it based on the request and table entries are updated accordingly.

```
DELIMITER //
Create trigger driverAssigner
before insert on booking for each row
BEGIN
    DECLARE driverId int;
    DECLARE Rc int;
    DECLARE carType varchar(20);
    Select id into driverId from driver where status = 'active' limit 1;
    Select driver.RC into Rc from driver where id = driverId;
    Select car.type into cartype from car where id = Rc;
    IF carType = 'Sedan' THEN
        SET NEW.price = 1000 ;
    ELSEIF carType = 'SUV' THEN
        SET NEW.price = 1200 ;
    ELSEIF carType = 'Micro' THEN
        SET NEW.price = 500 ;
    ELSE
        SET NEW.price = 800 ;
    END IF;
    Update driver set status ='busy' where id = driverId;
    SET NEW.driverId=driverId;
    SET NEW.RC=Rc;
    SET NEW.status='waiting';
END//
DELIMITER ;
```

III. Audit Trail Trigger

This trigger is used to store the old details of the driver for any future reference, as it prevents some confidential data from being overwritten.

```
CREATE TABLE driver_history (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    DriverDL varchar(100) NOT NULL,  
    lastname VARCHAR(50) NOT NULL,  
    oldRC int NOT NULL,  
    changedate DATETIME DEFAULT NULL,  
    action VARCHAR(50) DEFAULT NULL  
);
```

```
DELIMITER //  
Create trigger driverUpdateHistory  
BEFORE UPDATE ON driver  
for each row  
Insert into driver_history  
SET DriverDL = OLD.DLNumber,  
name = OLD.lastname,  
oldRC = OLD.RC, changedate =NOW();  
DELIMITER ;
```

IV. Automated Wallet Creation Trigger

This trigger is used to automatically create a new wallet for a newly registered user.

```
DELIMITER //
Create trigger walletMaker
AFTER insert ON customer
for each row
Insert into wallet set money = '500', id = NEW.id;
DELIMITER ;
```

V. Transaction History Trigger

This trigger is used to store the old values of the user's wallet after any changes/payments are being made from the wallet, it ensures consistency in case of payment failures.

```
CREATE TABLE wallet_history (
    id INT AUTO_INCREMENT PRIMARY KEY,
    customerId int NOT NULL,
    oldMoney int NOT NULL,
    changedate DATETIME DEFAULT NULL,
    action VARCHAR(50) DEFAULT NULL
);

Create trigger walletHistory
BEFORE UPDATE ON wallet for each row
Insert into wallet_history set action = 'UPDATED',
customerId = OLD.id, oldMoney = OLD.money, changedate =NOW();
```

VI. Customer Data Retention Trigger

This trigger is used to store the old details of the customer for any future reference as it prevents some confidential data from being lost after the deletion of the customer account.

```
CREATE TABLE user_history (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    customerId int NOT NULL,  
    name varchar(50) NOT NULL,  
    username varchar(50) NOT NULL,  
    changedate DATETIME DEFAULT NULL,  
    action VARCHAR(50) DEFAULT NULL  
);
```

```
Create trigger userHistorty  
AFTER DELETE ON customer for each row  
Insert into user_history set action = 'DELETED',  
customerId = OLD.id, name = OLD.name,  
username = OLD.username, changedate =NOW();
```


OLAP Queries

- I. Analyze payment details based on grouping between payment status and mode used with customer count to study customer habits

```
SELECT type,status, COUNT(*) AS payment_count,
SUM(amount) AS payment_total, ( SELECT COUNT(DISTINCT id)
FROM payment p2 WHERE p2.type = payment.type AND p2.status = payment.status )
AS unique_customers FROM payment GROUP BY type, status WITH ROLLUP;
```

- II. Analyze users with maximum fare paid by them to understand prime users

```
Select t.custId, t.maxPrice, customer.firstName
from customer inner join (select custId, max(price) as maxPrice
from booking where status = 'success'
group by (custId) with rollup) as t on customer.id = t.custId;
```

- III. Analyze trip rating feedback at all levels from 0 to 5 based on users to understand user satisfaction with rides

```
select count(*) as Count ,tripRating,c.id from customer as c
inner join (select b.*,t.tripRating from booking as b
inner join trip as t on b.id = t.bookingId ) as t1 on t1.custId=c.id
group by tripRating,id with rollup;
```

- IV. Analyze user Complaints

```
select count(*) as NumberOfReports, custId from booking as b
inner join report as t on b.id = t.bookingId
group by custId with rollup order by count(*) desc;
```

- V. Analyze the overall Financial Status of different grouping, such as status with net revenue generated and loss faced due to ride canceling

```
Select type,SUM(Total) as ActualTotal ,
SUM(PriceT) as Potential, (SUM(PriceT) - SUM(Total)) as NetPotentialLoss
from (SELECT coalesce(type,'Sub Total 1') as type,
SUM(amount) as Total, SUM(price) as PriceT
FROM payment p1 inner join booking
where booking.id =(Select bookingId from trip where paymentId = p1.id )
GROUP BY type with rollup
UNION ALL
SELECT coalesce(p2.status,'Sub Total 2') as type,
SUM(amount) as Total , SUM(price) as PriceT
FROM payment p2 inner join booking
where booking.id =(Select bookingId from trip where paymentId = p2.id )
GROUP BY p2.status with rollup)
as t group by t.type with rollup;
```

SQL Queries

Some more SQL Queries to show Database Functionalities & Constraints

```
select * from booking where custId = 1004 order by bookingTime;
select * from booking where status = 'waiting';
select (issue) from report where bookingId =10005;
select * from booking where driverId = 104 order by bookingTime;
select * from customer where id=1001;

select * from car where status='active' AND location='SouthDelhi';
select * from wallet where id =1001;
select count(*) as TotalCars,type from car group by type;
select count(*) as TotalCars,status,type from car
where status = 'notActive' group by type;
select custId,count(*) as TotalBooking
from booking group by custId having count(*) >3;

select * from trip where bookingId in (select (id)
from booking where custId=1004 AND status='success');
select sum(amount) from payment
where id in ( select paymentId from trip where bookingId in
(select id from booking where driverId=104 AND status = 'success'));

select amount from payment
where id in (select paymentId from trip where bookingId in
(select id from booking where custId=1004));
```

```

select * from driver where id in (select distinct driverId from booking);
select firstName, DLNumber from driver where rc in (select id from car);
select * from car where id not in (select rc from driver);
select id,rc from driver
where rc in(select id from car where status='active' AND location='SouthDelhi');
select id,rc,status from driver
where rc in(select id from car where status='active' AND type='Sedan');
select issue from report
where bookingId in (select id from booking where custId =1004);

select * from customer left join wallet on customer.id = wallet.id;
select distinct * from customer
where id in(select custId from booking where id in (select bookingId from report));
select * from trip
where bookingId in(select id from booking where custId =1004 );
select * from trip left join payment on payment.id=trip.paymentId
where bookingId in(select id from booking where custId =1004 );

select count(*) from trip where bookingId in(select id from booking where custId =1004 );
select * from booking join trip on booking.id = trip.bookingId where booking.custId = 1004;
select * from trip left join payment on trip.paymentId = payment.id where trip.status = 'success';

select * from payment where amount between 6000 and 7000 order by amount desc;
SELECT * FROM car WHERE location LIKE '%SouthDelhi%' and status='active';
select CONCAT(firstName, ' ', lastName) AS NAME from customer;
select id from customer where id not in( select distinct custId from booking);
Update customer set firstName = 'Shubham' where id = 1001;
Update trip set startTime='2024-01-01 09:45:00',endTime='2024-01-01 11:45:00' where id%2!=0;
delete from customer where id = 1001;

alter table wallet add column debitCard int;
alter table wallet drop column debitCard;
alter table customer modify column altPhoneNumber int default null unique;

```

Some Embedded SQL Queries used in Application to Establish User, Driver & Admin Functionalities.

```
I. try{
    Connection con= DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/cab2","root","parveen");
    String sql = "select * from booking where id= "+bookingId;
    PreparedStatement pst = con.prepareStatement(sql);
    ResultSet rs= pst.executeQuery();
    String all="";
    while(rs.next()){
        String row = "Booking Id: "+rs.getInt(1) +"\n" + "Driver Id: "+rs.getInt(3)+"
            "+
            "Car RC No. : "+rs.getInt(4) +"\n"+
            "Pickup Location: "+rs.getString(5) +"\n"+
            "Destination Location: "+rs.getString(6) +"\n"+
            "Booking Time: "+rs.getString(7) +"\n"+
            "Trip Fare: "+rs.getString(8) +" "+
            "Status: "+rs.getString(9) +" ";
        ;
        System.out.println(row);
        all = all.concat(row+"\n");
    }
    editorPane1.setText(all);
}
```

```
II. try{
    getNewPaymentId();
    //Class.forName("com.mysql.cj.jdbc.Driver");
    Connection con= DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/cab2","root","parveen");
    String sql = "update trip set paymentId=? , status=? where id=?";
    PreparedStatement pst = con.prepareStatement(sql);
    pst.setInt(3,tripId);
    pst.setInt(1,paymentId);
    pst.setString(2,"success");
    int count1= pst.executeUpdate();
}
}
```




```
try{
    //Class.forName("com.mysql.cj.jdbc.Driver");
    Connection con= DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/cab2","root","parveen");
    String sql = "insert into
customer(firstName,midName,lastName,password,username,email,phoneN
umber,altPhoneNumber,age)values(?,?,?,?,?,?,?,?,?)";
    PreparedStatement pst = con.prepareStatement(sql);
    pst.setString(1,firstNameStr);
    pst.setString(2,midNameStr);
    pst.setString(3,lastNameStr);
    pst.setString(4, String.valueOf(passStr));
    pst.setString(5,usernameStr);
    pst.setString(6,emailStr);
    pst.setInt(7,phoneStr);
    pst.setInt(8,altphoneStr);
    pst.setInt(9,ageStr);
    int count = pst.executeUpdate();
}
```

Transactions

I. Refund Transaction

TRANSACTION T1

START TRANSACTION;

```
UPDATE trip SET status = 'Cancelled' WHERE id = 789;
UPDATE payment SET status = 'Refunded' WHERE id = 456;
UPDATE wallet SET money = money + 500 WHERE id = 123;
IF ROW_COUNT() = 0 THEN
    ROLLBACK;
ELSE
    COMMIT;
END IF;
```

TRANSACTION T2

START TRANSACTION;

```
UPDATE trip SET status = 'Cancelled' WHERE id = 100;
UPDATE payment SET status = 'Refunded' WHERE id = 200;
UPDATE wallet SET money = money + 500 WHERE id = 300;
IF ROW_COUNT() = 0 THEN
    ROLLBACK;
ELSE
    COMMIT;
END IF;
```

SCHEDULE

T1	T2
Read(Trip)	
Write(Trip)	
	Read(Trip)
	Write(Trip)
Read(Payment)	
Write(Payment)	
	Read(Payment)
	Write(Payment)
Read(Wallet)	
Write(Wallet)	
COMMIT	
	Read(Wallet)
	Write(Wallet)
	COMMIT

This transaction executes whenever a trip is cancelled, so it first updates the status of a trip to "Cancelled", then updates the payment status to "Refunded", and updates the customer's wallet with the amount he pays for the trip, then there is a consistency check condition to maintain consistency across database so it uses ROW_COUNT() function for this if it returns 0 then it means money is not refunded back to the customer hence it rolls back the entire transaction; otherwise, it commits the transaction.

II. Book a Ride

```
START TRANSACTION;
```

```
INSERT INTO booking (custId, pickupStreet,  
destinationCity,bookingTime, price)  
VALUES (1002, 'Govindpuri', 'Harkesh Nagar', NOW(), 500);  
UPDATE wallet SET money = money - 500 WHERE id = 1002 AND  
money >= 500;  
IF ROW_COUNT() = 0 THEN  
    ROLLBACK;  
ELSE  
    COMMIT;  
END IF;
```

```
TRANSACTION T2
```

```
START TRANSACTION;
```

```
INSERT INTO booking (custId, pickupStreet,  
destinationCity,bookingTime, price)  
VALUES (1003, 'Kashmere Gate', 'Raj Bagh', NOW(), 600);  
UPDATE wallet SET money = money - 600 WHERE id = 1003 AND  
money >= 600;  
IF ROW_COUNT() = 0 THEN  
    ROLLBACK;  
ELSE  
    COMMIT;  
END IF;
```

SCHEDULE

T1	T2
Write(Booking)	
	Write(Booking)
Read(Wallet)	
Write(Wallet)	
COMMIT	
	Read(Wallet)
	Write(Wallet)
	COMMIT

This transaction executes whenever a new booking request is generated by the customer, so it first inserts a new booking record in the booking table, then it tries to deduct the booking amount from the customer's wallet, but if the current available amount in customer's wallet is less than the booking amount then whole transaction is rolled back; otherwise, it commits the transaction.

III. Location Update Conflict

TRANSACTION T1

START TRANSACTION;

UPDATE car

SET location = 'Govind Puri'

WHERE id = (SELECT RC FROM driver WHERE status = 'Active');

COMMIT;

TRANSACTION T2

START TRANSACTION;

UPDATE car

SET location = 'Kashmere Gate'

WHERE id = (SELECT RC FROM driver WHERE status = 'Active');

COMMIT;

CONFLICT SERIALIZABLE SCHEDULE

T1	T2
Shared Lock(Driver)	
Read(Driver)	
	Shared Lock(Driver)
	Read(Driver)
Exclusive Lock(Car)	
Write(Car)	
COMMIT	
Release Shared Lock(Driver)	
Release Exclusive Lock(Car)	
	Exclusive Lock(Car)
	Write(Car)
	COMMIT
	Release Shared Lock(Driver)
	Release Exclusive Lock(Car)

NON-CONFLICT SERIALIZABLE SCHEDULE

T1	T2
Shared Lock(Driver)	
Read(Driver)	
	Shared Lock(Driver)
	Read(Driver)
	Exclusive Lock(Car)
	Write(Car)
	COMMIT
	Release Shared Lock(Driver)
	Release Exclusive Lock(Car)
Exclusive Lock(Car)	
Write(Car)	
COMMIT	
Release Shared Lock(Driver)	
Release Exclusive Lock(Car)	

When these transactions are executing concurrently and assume there is only 1 available car, then if T1 comes first and updates the car location, then before it commits changes, the transaction manager switches to the T2 and starts executing it, and T2 also updates the location of the same car and commits after that T1 commits then changes made by T2 will become irrecoverable hence this leads to a conflict between two transactions.

IV. Car Deletion and Booking Acceptance

TRANSACTION T1

START TRANSACTION;

```

UPDATE car
SET location = 'Govindpuri'
WHERE id = (SELECT RC FROM driver WHERE status = 'Active');

```

```

UPDATE booking
SET RC = (SELECT RC FROM driver WHERE status = 'Active' AND
RC = (SELECT RC FROM car WHERE location = 'Govindpuri')
LIMIT 1),status = 'Accepted' WHERE id = 456;

```

```

COMMIT;

```

TRANSACTION T2

```

START TRANSACTION;

```

```

UPDATE driver
SET status = 'Inactive'
WHERE id = (SELECT RC FROM booking WHERE id = 456);

```

```

DELETE FROM car WHERE id = (SELECT RC FROM driver WHERE id =
(SELECT RC FROM booking WHERE id = 456));

```

```

COMMIT;


```

SCHEDULE

T1	T2
Shared Lock(Driver)	
Read(Driver)	
Exclusive Lock(Car)	
Write(Car)	

	Shared Lock(Booking)
	Read(Booking)
	Exclusive Lock(Driver)
	Write(Driver)
Read(Car)	
Read(Driver)	
	Read(Booking)
	Read(Driver)
	Exclusive Lock(Car)
	Write(Car)
	COMMIT
	Release Shared Lock(Booking)
	Release Exclusive Lock(Driver)
	Release Exclusive Lock(Car)
Exclusive Lock(Booking)	
Write(Booking)	
COMMIT	
Release Shared Lock(Driver)	
Release Exclusive Lock(Car)	
Release Exclusive Lock(Booking)	

When these transactions are executing concurrently, then T1 updates the location of a car and also updates the booking's RC to the assigned car's RC and status to "Active". Then before T1 commits changes, the transaction manager switches to T2



and starts executing it, where it updates the driver's status to "Inactive" and deletes the car assigned in T1, then gets committed. This would lead to a conflict because T2 deletes the car that has been assigned to T1, and then T1 will commit, and the booking is completed based on a car that no longer exists because T2 already deleted that car.