BIDMYRIDE

CAB BOOKING APP

Prof. Vikram Goel

Prepared by: Ayush Sachan (2021031)
Chaitanya Arora(2021033)

CSE202 - Fundamentals of Database Management Systems

PROJECT DESCRIPTION

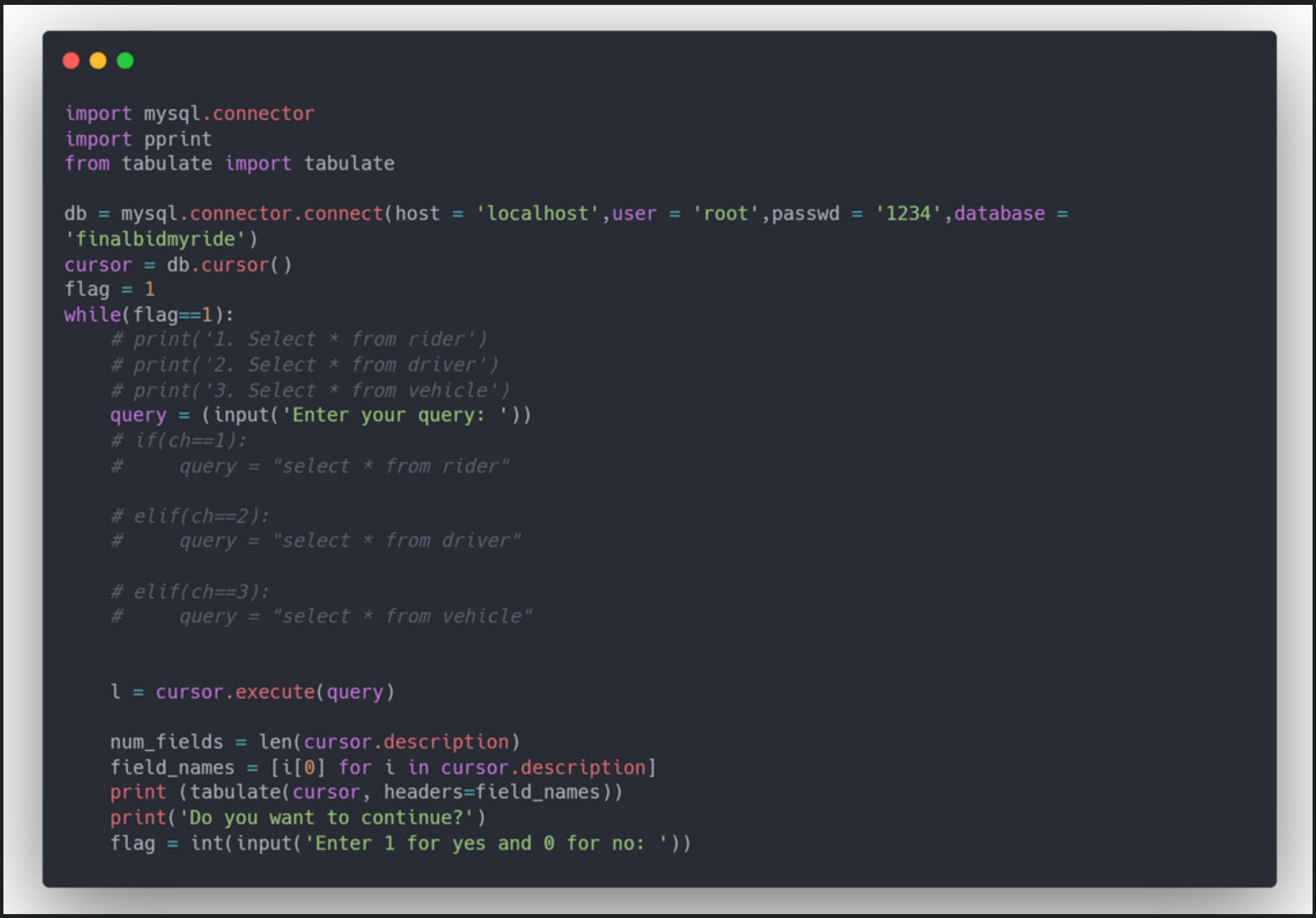
BidMyRide is a revolutionary cab booking app that allows riders to choose from a variety of cab driver prices for their ride. Our app offers a unique bidding feature that allows cab drivers to offer a price for a ride and compete for the rider's business. This feature empowers riders to have more control over the cost of their ride, while also giving cab drivers a chance to earn more business by offering competitive prices.

In addition to the bidding feature, BidMyRide also offers the standard features you would expect from a cab booking app, such as view driver ratings and reviews, and track the driver's location in real-time. Our app also utilizes a user-friendly interface.

Overall, BidMyRide is the perfect solution for riders looking for a more cost-effective and efficient way to book rides, and for cab drivers looking to increase their business. With BidMyRide, everyone wins.

EMBEDDED QUERIES

CODE SNIPPET



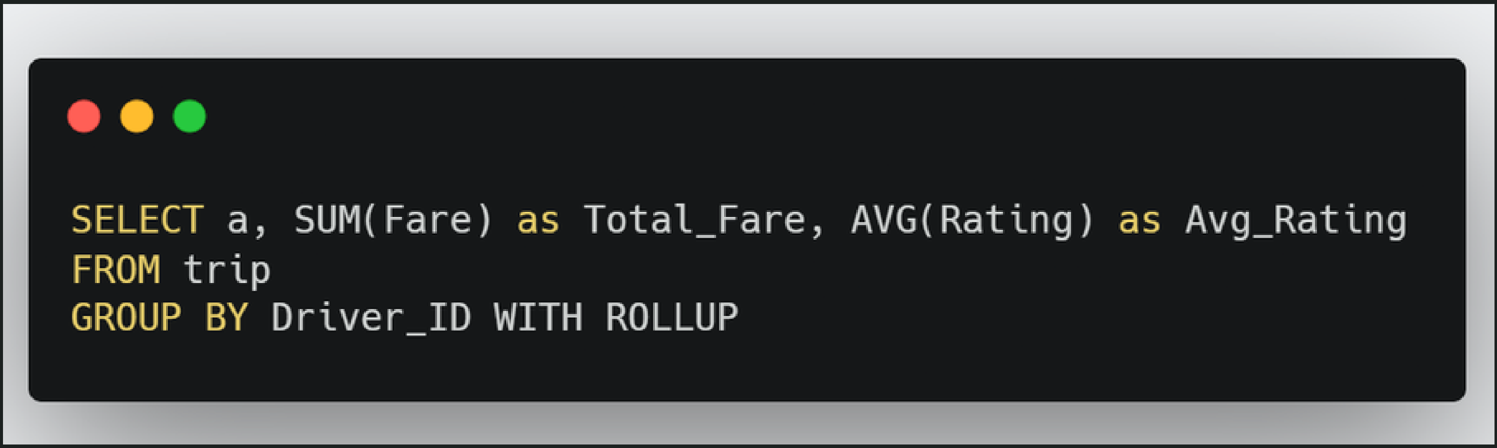
```python
import mysql.connector
import pprint
from tabulate import tabulate

db = mysql.connector.connect(host = 'localhost',user = 'root',passwd = '1234',database =
'finalbidmyride')
cursor = db.cursor()
flag = 1
while(flag==1):
    # print('1. Select * from rider')
    # print('2. Select * from driver')
    # print('3. Select * from vehicle')
    query = (input('Enter your query: '))
    # if(ch==1):
    #     query = "select * from rider"

    # elif(ch==2):
    #     query = "select * from driver"

    # elif(ch==3):
    #     query = "select * from vehicle"


    l = cursor.execute(query)

    num_fields = len(cursor.description)
    field_names = [i[0] for i in cursor.description]
    print (tabulate(cursor, headers=field_names))
    print('Do you want to continue?')
    flag = int(input('Enter 1 for yes and 0 for no: '))
```

The given Python code imports the mysql.connector library to establish a connection with a MySQL database and tabulate library to display query results in a tabular format. It then creates a connection to a database by specifying the host, username, password, and database name.

The code enters into a while loop where the user can input a query string to be executed on the connected database. The input query is passed to the cursor.execute() method to execute the query, and the result is displayed using the tabulate function in a formatted table with column headers.

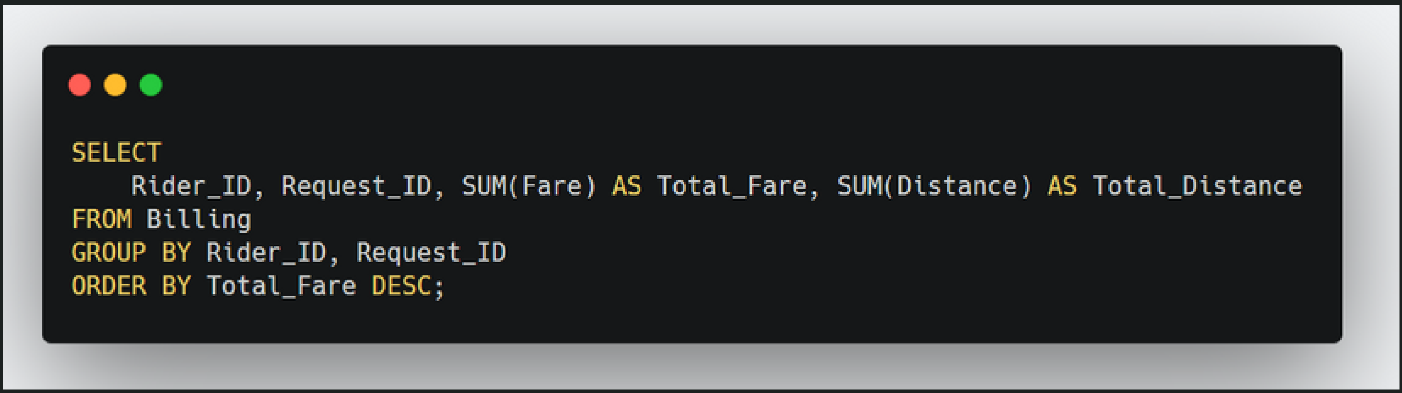
The user is prompted to continue or exit the loop based on their input. Overall, this code serves as a basic framework for establishing a connection with a MySQL database and fetching data based on user input queries.

OLAP QUERIES



```
SELECT a, SUM(Fare) as Total_Fare, AVG(Rating) as Avg_Rating
FROM trip
GROUP BY Driver_ID WITH ROLLUP
```

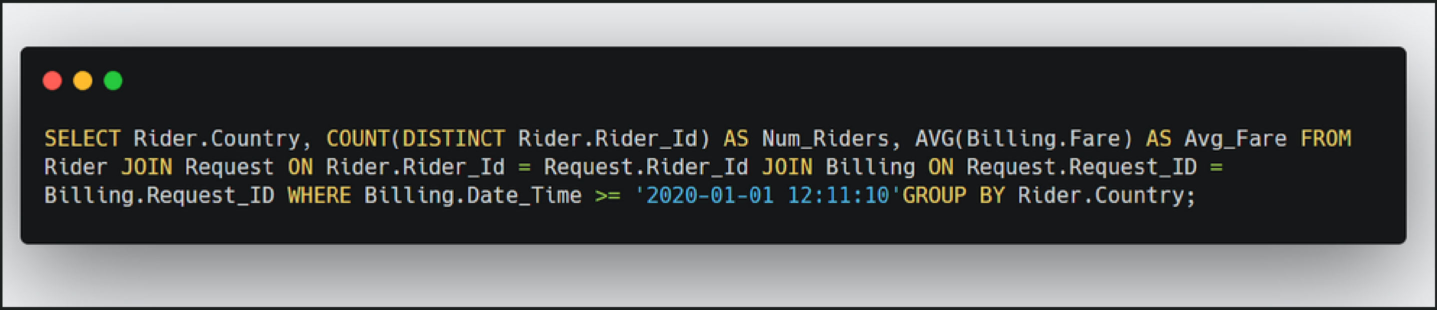
- The OLAP query is retrieving data from the "trip" table in the database. It selects the Driver_ID column and calculates the SUM of the Fare column as Total_Fare, and calculates the AVG of the Rating column as Avg_Rating. The query then groups the data by Driver_ID, and includes a ROLLUP operator to generate a subtotal row for each distinct value in the Driver_ID column, as well as a grand total row.
- The database schema consists of several tables related to ride-sharing services, including credentials, wallet, vehicle, rider, driver, request, sends, approve, trip, and billing. The tables contain information on users, vehicles, ride requests, rides, and billing. The OLAP query is likely used to analyze the revenue and rating performance of drivers in the ride-sharing service.



```
SELECT
    Rider_ID, Request_ID, SUM(Fare) AS Total_Fare, SUM(Distance) AS Total_Distance
FROM Billing
GROUP BY Rider_ID, Request_ID
ORDER BY Total_Fare DESC;
```

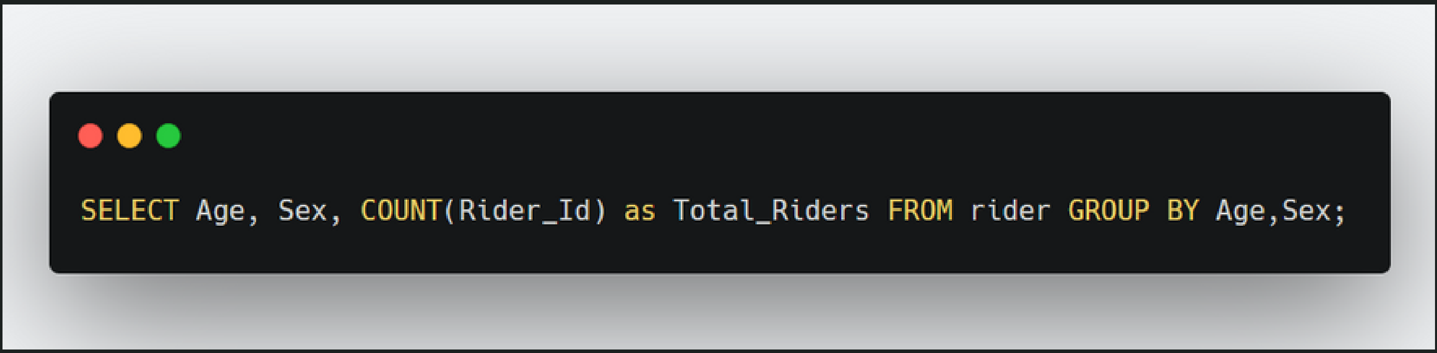
- The OLAP query selects data from the Billing table in the given database schema and groups the data by Rider_ID and Request_ID. It then calculates the sum of the fare and distance for each group and aliases the results as Total_Fare and Total_Distance, respectively. Finally, the query orders the result set in descending order by Total_Fare.

OLAP QUERIES



```
SELECT Rider.Country, COUNT(DISTINCT Rider.Rider_Id) AS Num_Riders, AVG(Billing.Fare) AS Avg_Fare FROM
Rider JOIN Request ON Rider.Rider_Id = Request.Rider_Id JOIN Billing ON Request.Request_ID =
Billing.Request_ID WHERE Billing.Date_Time >= '2020-01-01 12:11:10'GROUP BY Rider.Country;
```

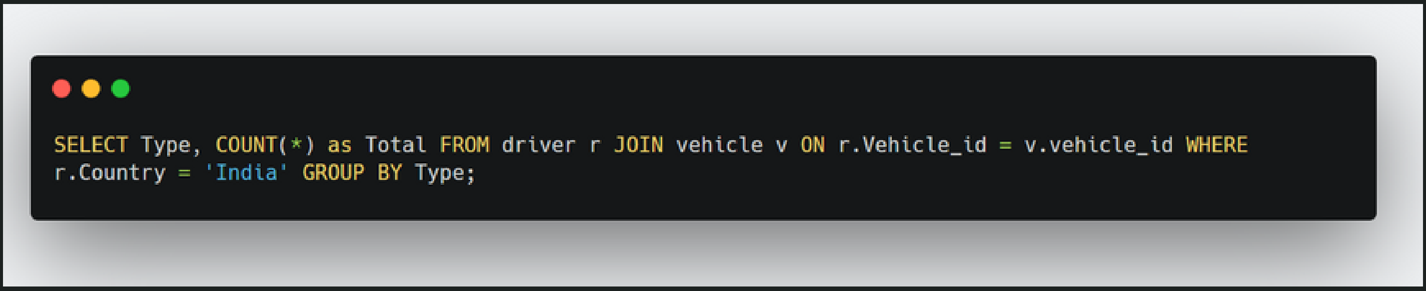
- The query performs the following operations on the tables in the database schema:
- It joins the Rider, Request, and Billing tables using their primary and foreign keys.
- It selects the Rider's country, counts the number of distinct Rider IDs for each country, and calculates the average fare for each group.
- It applies a filter on the Billing table to select only those records that have a date and time greater than or equal to January 1, 2020, at 12:11:10 pm.
- It groups the results by the Rider's country.



```
SELECT Age, Sex, COUNT(Rider_Id) as Total_Riders FROM rider GROUP BY Age,Sex;
```

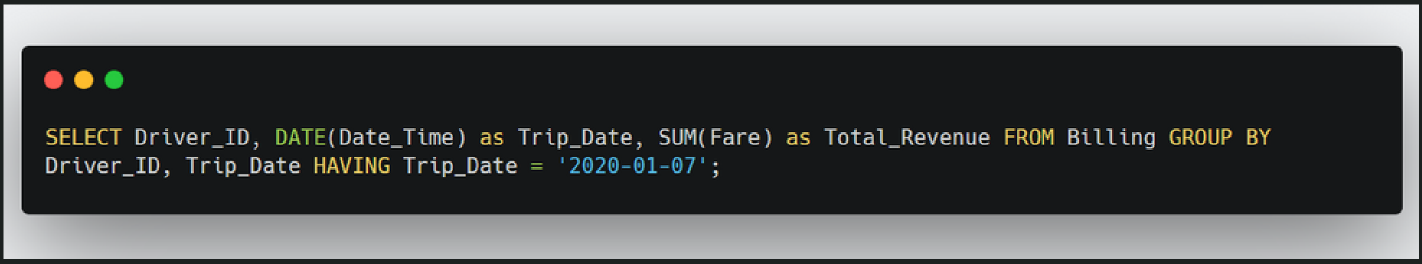
- The OLAP query selects data from the "rider" table and groups it by the "Age" and "Sex" columns. It also counts the number of unique "Rider_Id" values for each combination of "Age" and "Sex" and gives it an alias of "Total_Riders". The result will show the total number of riders for each unique combination of age and sex.
- The remaining code defines several tables for a ride-hailing service, including tables for riders, drivers, requests, trips, and billing. These tables are used to store data related to ride requests, driver information, trip details, and billing information for completed trips. The foreign key constraints in the tables ensure that data is correctly linked between the different tables.

OLAP QUERIES



```sql
SELECT Type, COUNT(*) as Total FROM driver r JOIN vehicle v ON r.Vehicle_id = v.vehicle_id WHERE
r.Country = 'India' GROUP BY Type;
```

- The OLAP query is selecting the "Type" column and counting the number of occurrences of each distinct value in that column in the "driver" and "vehicle" tables joined on the "vehicle_id" column. The query is filtering the results to only include records where the "Country" column in the "driver" table equals 'India'. The results are then grouped by the "Type" column.

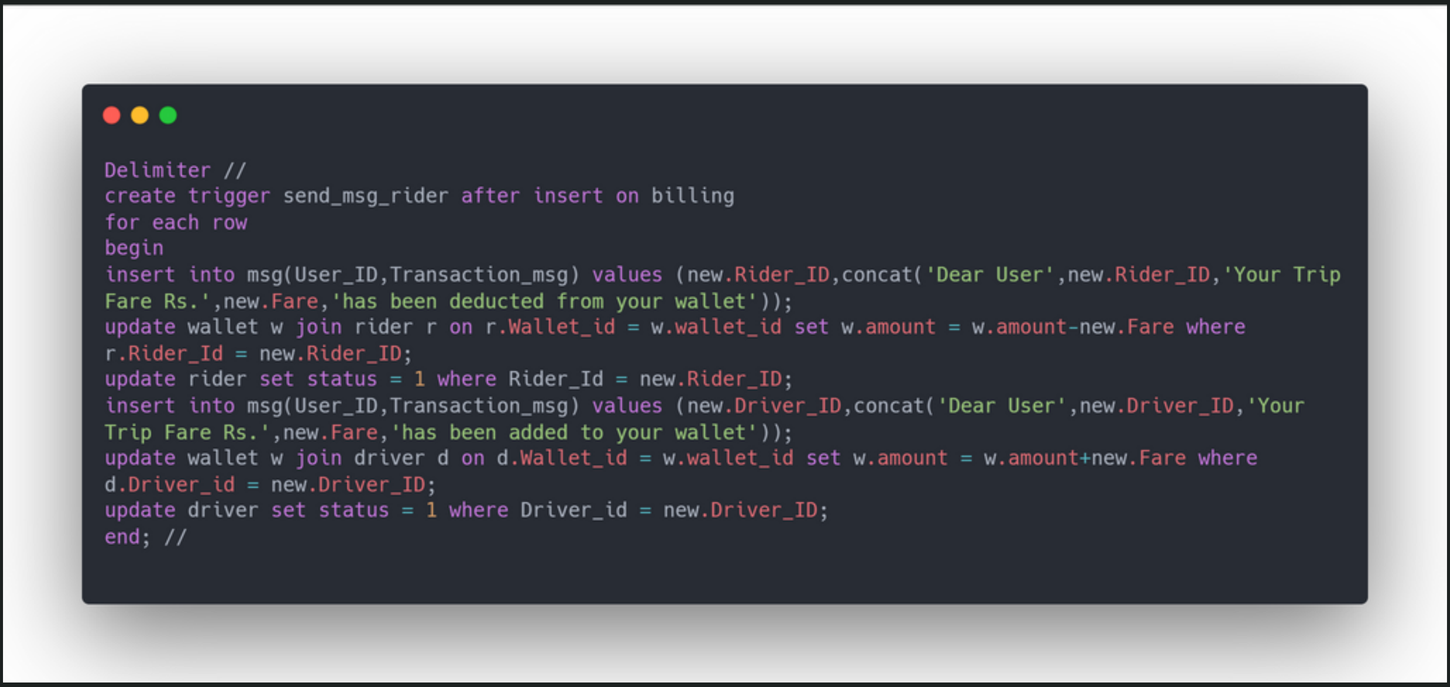


```sql
SELECT Driver_ID, DATE(Date_Time) as Trip_Date, SUM(Fare) as Total_Revenue FROM Billing GROUP BY
Driver_ID, Trip_Date HAVING Trip_Date = '2020-01-07';
```

- The OLAP query provided is retrieving data from the "Billing" table. The query selects three columns: "Driver_ID", "Trip_Date" and "Total_Revenue". The "Trip_Date" column is created using the DATE function to extract only the date part from the "Date_Time" column. The "Total_Revenue" column is created by summing up the "Fare" column. The query groups the data by "Driver_ID" and "Trip_Date" columns and filters the results by only including data where "Trip_Date" is equal to '2020-01-07'.

TRIGGERS

TRIGGER 1

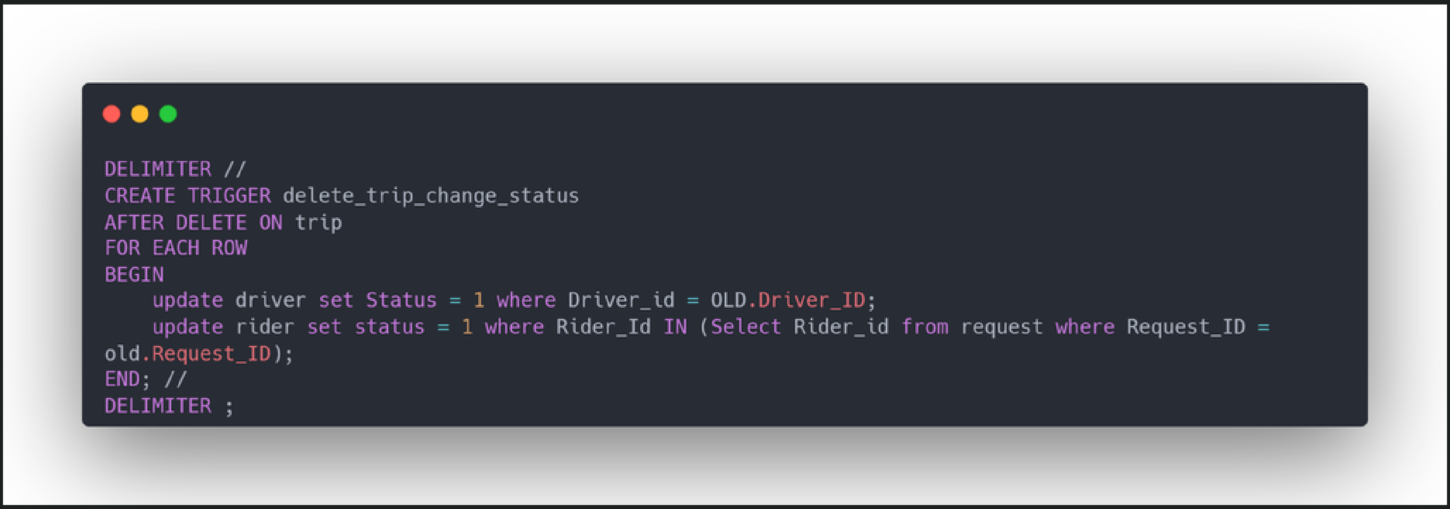


```
Delimiter //
create trigger send_msg_rider after insert on billing
for each row
begin
insert into msg(User_ID,Transaction_msg) values (new.Rider_ID,concat('Dear User',new.Rider_ID,'Your Trip
Fare Rs.',new.Fare,'has been deducted from your wallet'));
update wallet w join rider r on r.Wallet_id = w.wallet_id set w.amount = w.amount-new.Fare where
r.Rider_Id = new.Rider_ID;
update rider set status = 1 where Rider_Id = new.Rider_ID;
insert into msg(User_ID,Transaction_msg) values (new.Driver_ID,concat('Dear User',new.Driver_ID,'Your
Trip Fare Rs.',new.Fare,'has been added to your wallet'));
update wallet w join driver d on d.Wallet_id = w.wallet_id set w.amount = w.amount+new.Fare where
d.Driver_id = new.Driver_ID;
update driver set status = 1 where Driver_id = new.Driver_ID;
end; //
```

The "send_msg_rider" performs performs several actions after a row is inserted into the "billing" table. The trigger inserts messages into the "msg" table for the rider and driver to inform them of the fare transaction and also updates the "wallet" amount for both rider and driver accordingly. It also sets the rider and driver's status to 1 to indicate they are valid for another trip. The delimiter is temporarily changed to "//" to define the trigger and set back to ";" after the trigger definition.

TRIGGERS

TRIGGER 2

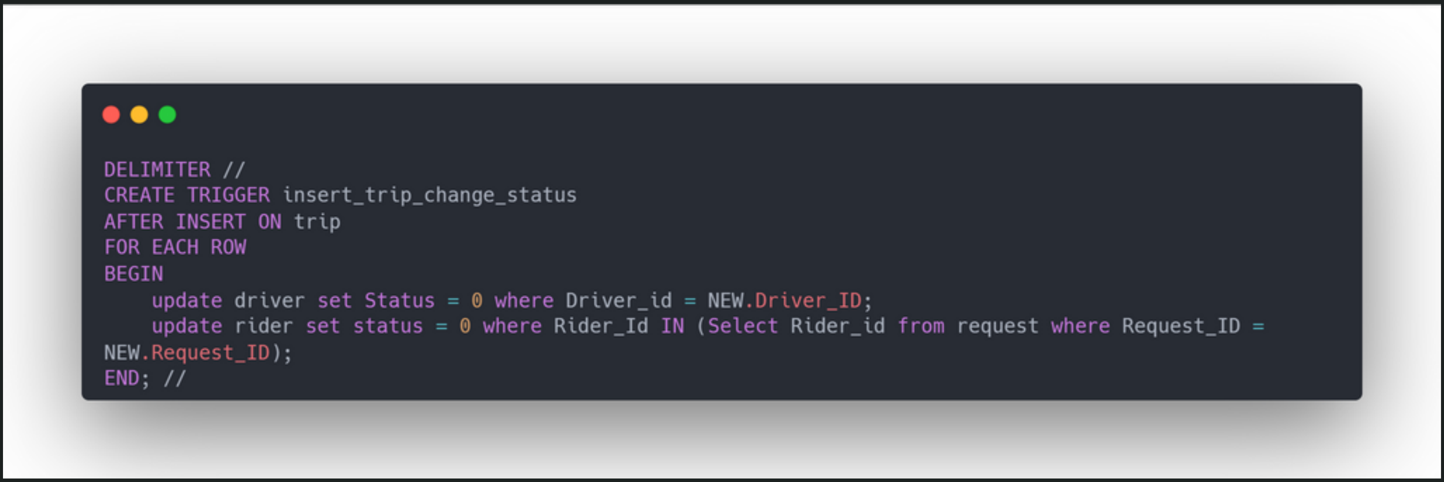


```
DELIMITER //
CREATE TRIGGER delete_trip_change_status
AFTER DELETE ON trip
FOR EACH ROW
BEGIN
    update driver set Status = 1 where Driver_id = OLD.Driver_ID;
    update rider set status = 1 where Rider_Id IN (Select Rider_id from request where Request_ID =
old.Request_ID);
END; //
DELIMITER ;
```

The "delete_trip_change_status" trigger is defined to run after a row is deleted from the "trip" table. The trigger updates the "status" column in the "driver" table to set the driver's status to 1 where the "Driver_id" matches the "Driver_ID" in the deleted row. It also updates the "status" column in the "rider" table to set the rider's status to 1 where the "Rider_Id" matches the "Rider_id" in the "request" table that has the same "Request_ID" as the deleted row.

TRIGGERS

TRIGGER 3



```
DELIMITER //
CREATE TRIGGER insert_trip_change_status
AFTER INSERT ON trip
FOR EACH ROW
BEGIN
    update driver set Status = 0 where Driver_id = NEW.Driver_ID;
    update rider set status = 0 where Rider_Id IN (Select Rider_id from request where Request_ID =
NEW.Request_ID);
END; //
```

The "insert_trip_change_status" trigger updates the "Status" column in the "driver" table and the "status" column in the "rider" table after a row is inserted into the "trip" table. The trigger is defined to run for each row inserted and sets the "Status" column in the "driver" table and the "status" column in the "rider" table to 0. The driver's status is updated based on the "Driver_ID" column in the inserted row, and the rider's status is updated based on the "Rider_Id" column in the "request" table, where the "Request_ID" matches the inserted row's "Request_ID". The delimiter is temporarily changed to "//" to define the trigger and then set back to ";" after the trigger definition.