Design and Analysis of a Ride-Booking Service Database

Contents

Abstract

In this report, we detail the creation and analysis of a comprehensive database system for a ride-booking service, covering its inception, structure, and operational capabilities. The process began with the construction of pivotal tables such as Driver, Customer, RideBooking, Vehicle, Payment, Feedback, IncidentReport, and Scheduling, each tailored to capture specific aspects of the service. A UML diagram was developed to visually map the interrelations between these entities, providing a clear structural overview. The database underwent extensive optimization and indexing, focusing on efficient data retrieval and storage, with particular emphasis on indexing frequently queried columns. Functionality and integrity were rigorously tested through various queries. Advanced features like stored procedures and triggers were incorporated for streamlined operations and automated data updates, while transaction management was crucial in maintaining atomicity and consistency in critical processes like ride bookings and payments. Analytical queries were employed to extract key insights into driver performance, financial metrics, and customer usage patterns, offering valuable data for strategic decision-making. Despite its robustness, the database faced limitations in scalability, complexity of queries, and real-time processing, indicating areas for future enhancement. This comprehensive approach demonstrates the database's utility in managing and optimizing ride-booking services, while also highlighting potential avenues for further development in scalability and real-time analytics.

Introduction

In the rapidly evolving landscape of urban transportation, ride-booking services have emerged as a pivotal component, revolutionizing how people commute and interact with city infrastructure. The backbone of such services is a robust and efficient database system, capable of handling complex operations and large volumes of data. This report delves into the development and analysis of a comprehensive database system specifically designed for a ride-booking service. The project's inception involved the meticulous creation of a series of interrelated tables, each serving a distinct function within the broader service ecosystem. These tables included Driver, Customer, RideBooking, Vehicle, Payment, Feedback, IncidentReport, and Scheduling. The design was guided by a UML diagram, ensuring a coherent and interconnected structure.

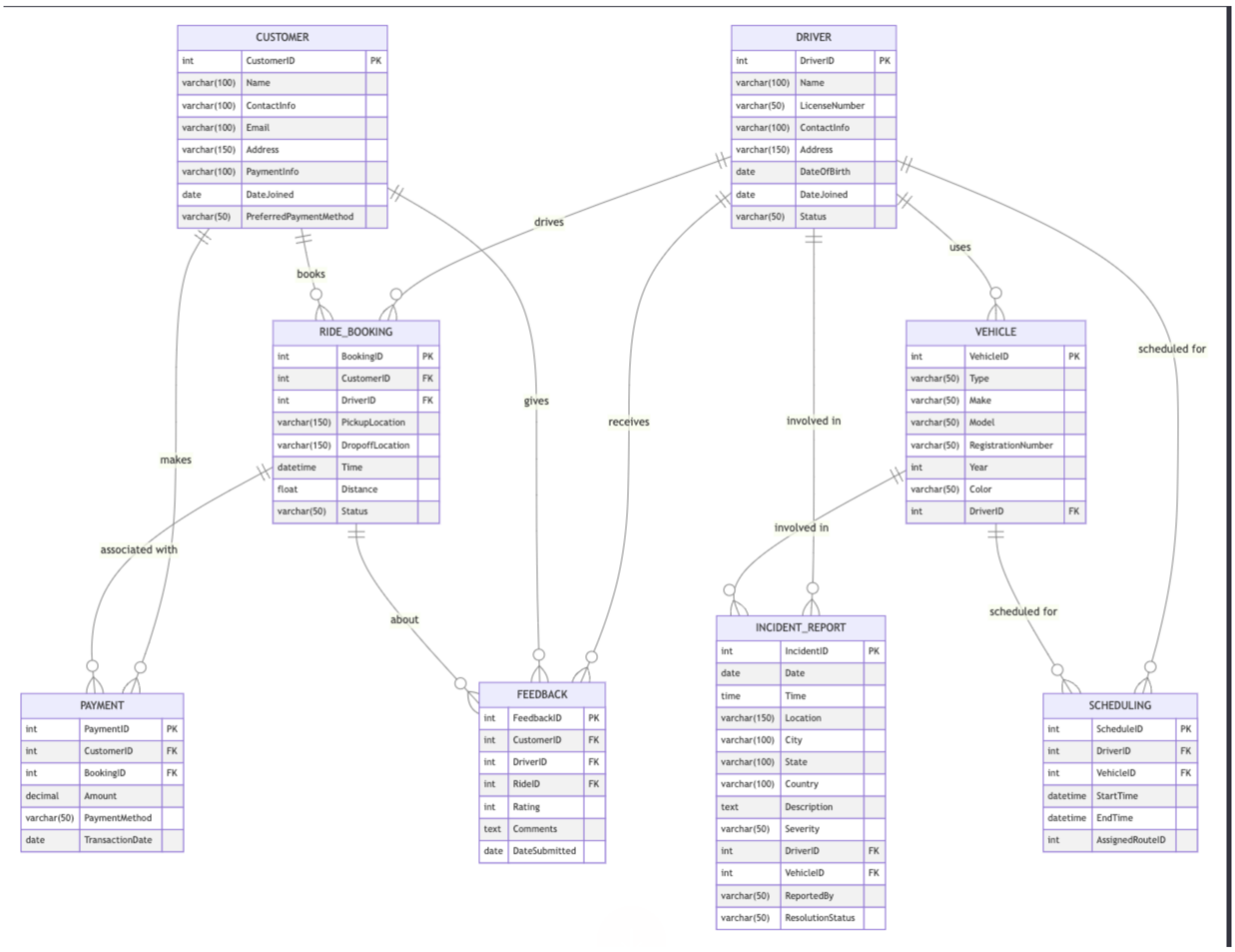
A significant focus was placed on optimizing the database for performance and reliability. This involved strategic indexing of frequently queried columns and fine-tuning the database for efficient data retrieval and storage. The system's functionality and integrity were rigorously tested through a series of queries, ensuring that each component operated as intended and that the system as a whole met the demands of a dynamic ride-booking environment. Advanced features such as stored procedures, triggers, and transaction management were integrated to enhance the database's capabilities. These elements were crucial in automating processes, maintaining data consistency, and ensuring the atomicity of complex transactions.

The report also explores the analytical capabilities of the database. Through targeted SQL queries, valuable insights were extracted regarding driver performance, financial metrics, and customer engagement patterns. These findings are instrumental for strategic decision-making and highlight the database's role in driving operational efficiency and customer satisfaction. Despite its robust design and functionality, the database system faces challenges in scalability, handling complex queries, and real-time data processing. These limitations present opportunities for future enhancements, particularly in adapting the system to handle larger datasets and real-time analytics, which are increasingly important in the fast-paced world of ride-booking services.

DataBase Design

UML Diagram

The UML (Unified Modeling Language) entity-relationship diagram provides a visual representation of the database structure for the ride-booking service. This diagram is crucial for understanding the relationships and dependencies between different entities within the database.



- Entities and Attributes: Each rectangle in the diagram represents an entity (table) such as Driver, Customer, RideBooking, Vehicle, Payment, Feedback, IncidentReport, and Scheduling. Inside each rectangle, the attributes (columns) of the respective tables are listed, providing a clear view of the data stored in each table.
- Relationships: The lines connecting the entities represent the relationships between them. For instance, the RideBooking entity is connected to both the Driver and Customer entities, indicating that each ride booking is associated with a specific driver and customer. The nature of these relationships is further clarified by symbols at the end of the connecting lines, indicating one-to-many or many-to-one relationships.
- Foreign Keys: The diagram also highlights foreign key relationships, where an attribute in one table references the primary key of another table. This is essential for maintaining data integrity and enabling complex queries that span multiple tables.
- Overall Structure: The overall structure depicted in the diagram is a testament to the database's comprehensive design, ensuring that all aspects of the ride-booking service are captured and interlinked effectively. This structure facilitates efficient data

management, retrieval, and analysis, which are crucial for the smooth operation of the service.

The UML diagram serves as a foundational blueprint for the database, guiding its implementation and providing a clear understanding of its structure and interrelationships. This visual tool is invaluable for database designers, developers, and stakeholders involved in the project.

Creation of Tables for Ride-Booking Service Database

Overview

The development of a database for a ride-booking service involves the creation of several interconnected tables, each designed to store specific information related to the service's operations. These tables form the foundation of the database, enabling efficient data management and retrieval. Below is a detailed overview of each table's creation and purpose.

Driver Table

- Purpose: Stores information about drivers.
- Fields:
    - DriverID: Unique identifier for each driver (Primary Key).
    - Name: Driver's full name.
    - LicenseNumber: Driver's license number.
    - ContactInfo: Contact details of the driver.
    - Address: Residential address of the driver.
    - DateOfBirth: Driver's date of birth.
    - DateJoined: Date when the driver joined the service.
    - Status: Current status of the driver (e.g., Active, Inactive).

Customer Table

- Purpose: Contains details about customers using the service.
- Fields:
    - CustomerID: Unique identifier for each customer (Primary Key).
    - Name: Customer's full name.
    - ContactInfo: Contact details of the customer.

    - Email: Customer's email address.
    - Address: Customer's residential address.
    - PaymentInfo: Payment details or preferences.
    - DateJoined: Date when the customer joined the service.
    - PreferredPaymentMethod: Customer's preferred method of payment.

RideBooking Table

- Purpose: Manages information about ride bookings.
- Fields:
    - BookingID: Unique identifier for each booking (Primary Key).
    - CustomerID: References CustomerID from the Customer table.
    - DriverID: References DriverID from the Driver table.
    - PickupLocation: Location where the customer is picked up.
    - DropoffLocation: Destination location.
    - Time: Date and time of the ride.
    - EstimatedFare: Estimated fare for the ride.
    - Status: Current status of the booking (e.g., Completed, Cancelled).

Vehicle Table

- Purpose: Stores details about vehicles used by drivers.
- Fields:
    - VehicleID: Unique identifier for each vehicle (Primary Key).
    - DriverID: References DriverID from the Driver table.
    - ReportedYear: Year of the report or record.
    - ReportedMonth: Month of the report or record.
    - State: State of vehicle registration.
    - Make: Manufacturer of the vehicle.
    - Model: Model of the vehicle.
    - Color: Color of the vehicle.
    - ModelYear: Year of the vehicle model.
    - NumberOfTrips: Total number of trips completed by the vehicle.
    - MultipleTNPs: Boolean indicating if the vehicle is used for multiple Transport Network Providers.

Payment Table

- Purpose: Manages transaction details for ride bookings.
- Fields:
    - PaymentID: Unique identifier for each payment (Primary Key).
    - CustomerID: References CustomerID from the Customer table.
    - BookingID: References BookingID from the RideBooking table.
    - Amount: Amount paid for the ride.
    - PaymentMethod: Method of payment used.
    - TransactionDate: Date of the transaction.

Feedback Table

- Purpose: Collects feedback from customers about their rides.
- Fields:
    - FeedbackID: Unique identifier for each feedback entry (Primary Key).
    - CustomerID: References CustomerID from the Customer table.
    - DriverID: References DriverID from the Driver table.
    - RideID: References BookingID from the RideBooking table.
    - Rating: Numerical rating given by the customer.
    - Comments: Additional comments provided by the customer.
    - DateSubmitted: Date when the feedback was submitted.

IncidentReport Table

- Purpose: Records incidents related to rides.
- Fields:
    - IncidentID: Unique identifier for each incident report (Primary Key).
    - Date: Date of the incident.
    - Time: Time when the incident occurred.
    - Location: Location where the incident took place.
    - City: City of the incident.
    - State: State of the incident.
    - Country: Country where the incident occurred.
    - Description: Detailed description of the incident.
    - Severity: Severity level of the incident.
    - DriverID: References DriverID from the Driver table.

    - VehicleID: References VehicleID from the Vehicle table.
    - ReportedBy: Person who reported the incident.
    - ResolutionStatus: Current status of the incident resolution.

Scheduling Table

- Purpose: Manages scheduling details for drivers and vehicles.
- Fields:
    - ScheduleID: Unique identifier for each schedule entry (Primary Key).
    - DriverID: References DriverID from the Driver table.
    - VehicleID: References VehicleID from the Vehicle table.
    - StartTime: Start time of the schedule.
    - EndTime: End time of the schedule.
    - AssignedRouteID: Identifier for the assigned route.

Each table is designed with a specific purpose, ensuring that all aspects of the ride-booking service are comprehensively covered. The relationships between these tables are defined through foreign keys, facilitating data integrity and relational database management.

Normalization

First Normal Form (1NF)

Objective: Ensure that each column contains atomic values and each row is unique.

All tables in the database appear to meet the 1NF criteria:

- Driver, Customer, RideBooking, Vehicle, Payment, Feedback, IncidentReport, Scheduling:
    - Each table has a primary key (e.g., DriverID, CustomerID, BookingID).
    - All attributes contain atomic values (no repeating groups or arrays).
    - Each row is unique.

Second Normal Form (2NF)

Objective: Remove partial dependencies, where a non-key column is dependent on only part of the primary key in a composite key scenario.

All tables also meet the 2NF criteria, as they are already in 1NF and:

- Driver, Customer, Vehicle, Payment, Feedback, IncidentReport, Scheduling:
    - These tables have single-column primary keys, so there are no partial dependencies on a subset of the primary key.
- RideBooking:
    - Although it has a composite foreign key (CustomerID, DriverID), all non-key attributes are dependent on the whole of the primary key (BookingID), not just a part of it.

Third Normal Form (3NF)

Objective: Remove transitive dependencies, where non-key columns do not depend on other non-key columns.

The tables also conform to 3NF:

- All Tables:
    - Non-key attributes are dependent only on the primary key and not on other non-key attributes.
    - There are no transitive dependencies within the tables. For example, in the Driver table, all non-key attributes like Name, LicenseNumber, etc., are dependent only on DriverID.

Boyce-Codd Normal Form (BCNF)

Objective: Address anomalies not handled by 3NF, especially in tables with complex relationships and overlapping candidate keys.

The database seems to adhere to BCNF as well, which is a stricter version of 3NF. There are no overlapping composite candidate keys leading to anomalies.

- 1NF: Achieved by all tables as they have unique rows and atomic column values.

- 2NF: Achieved by all tables as they either have single-column primary keys or no partial dependency in the case of composite keys.
- 3NF and BCNF: All tables meet these criteria as there are no transitive dependencies and no violations of BCNF are evident.

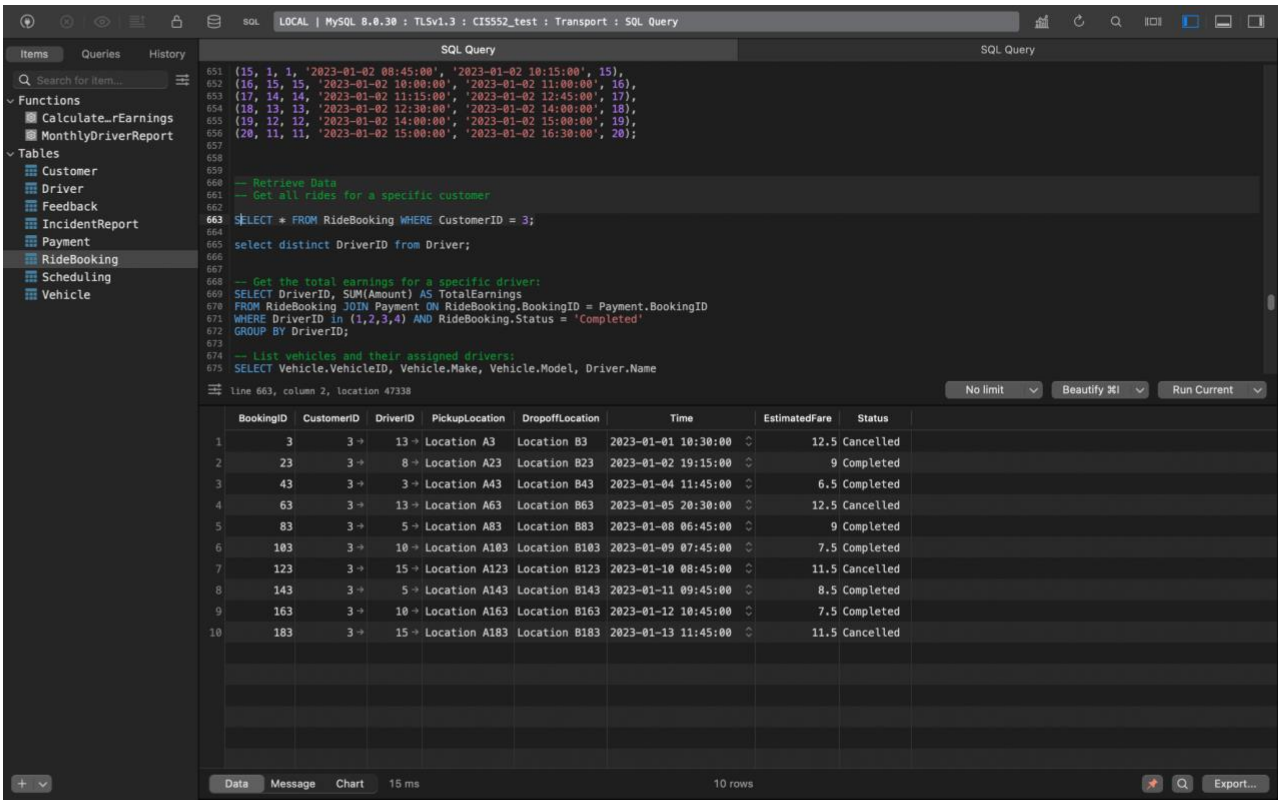
The database design for the ride-booking service efficiently adheres to the principles of normalization up to BCNF, ensuring data integrity, reducing redundancy, and facilitating efficient data management.

Queries: Functioning of the DataBase

Data Retrieval

Select Ride History for Customers: Queries are designed to fetch complete ride histories for individual customers, providing valuable insights into customer behavior and service usage.

SELECT * FROM RideBooking WHERE CustomerID = 3;



LOCAL | MySQL 8.0.30 : TLSv1.3 : CIS552_test : Transport : SQL Query

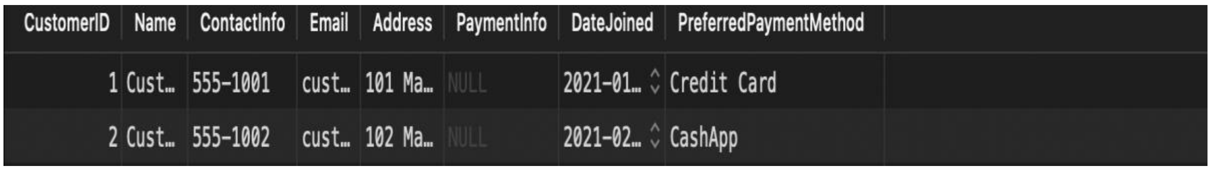
```
(15, 1, 1, '2023-01-02 08:45:00', '2023-01-02 10:15:00', 15),
(16, 15, 15, '2023-01-02 10:00:00', '2023-01-02 11:00:00', 16),
(17, 14, 14, '2023-01-02 11:15:00', '2023-01-02 12:45:00', 17),
(18, 13, 13, '2023-01-02 12:30:00', '2023-01-02 14:00:00', 18),
(19, 12, 12, '2023-01-02 14:00:00', '2023-01-02 15:00:00', 19),
(20, 11, 11, '2023-01-02 15:00:00', '2023-01-02 16:30:00', 20);



-- Retrieve Data
-- Get all rides for a specific customer

SELECT * FROM RideBooking WHERE CustomerID = 3;

select distinct DriverID from Driver;


-- Get the total earnings for a specific driver:
SELECT DriverID, SUM(Amount) AS TotalEarnings
FROM RideBooking JOIN Payment ON RideBooking.BookingID = Payment.BookingID
WHERE DriverID in (1,2,3,4) AND RideBooking.Status = 'Completed'
GROUP BY DriverID;

-- List vehicles and their assigned drivers:
SELECT Vehicle.VehicleID, Vehicle.Make, Vehicle.Model, Driver.Name
```

line 663, column 2, location 47338

	BookingID	CustomerID	DriverID	PickupLocation	DropoffLocation	Time	EstimatedFare	Status
1	3	3	13	Location A3	Location B3	2023-01-01 10:30:00	12.5	Cancelled
2	23	3	8	Location A23	Location B23	2023-01-02 19:15:00	9	Completed
3	43	3	3	Location A43	Location B43	2023-01-04 11:45:00	6.5	Completed
4	63	3	13	Location A63	Location B63	2023-01-05 20:30:00	12.5	Cancelled
5	83	3	5	Location A83	Location B83	2023-01-08 06:45:00	9	Completed
6	103	3	10	Location A103	Location B103	2023-01-09 07:45:00	7.5	Completed
7	123	3	15	Location A123	Location B123	2023-01-10 08:45:00	11.5	Cancelled
8	143	3	5	Location A143	Location B143	2023-01-11 09:45:00	8.5	Completed
9	163	3	10	Location A163	Location B163	2023-01-12 10:45:00	7.5	Completed
10	183	3	15	Location A183	Location B183	2023-01-13 11:45:00	11.5	Cancelled

Update Customer's Preferred Payment Method: This query updates the preferred payment method for a customer, reflecting changes in customer preferences.

UPDATE Customer SET PreferredPaymentMethod = 'CashApp' WHERE CustomerID = 2;



CustomerID	Name	ContactInfo	Email	Address	PaymentInfo	DateJoined	PreferredPaymentMethod
1	Cust…	555-1001	cust…	101 Ma…	NULL	2021-01…	Credit Card
2	Cust…	555-1002	cust…	102 Ma…	NULL	2021-02…	CashApp

Delete a Specific Booking: This query deletes a ride booking, used in cases of cancellation or error correction.

DELETE FROM RideBooking WHERE BookingID = 16;

Stored Procedure:

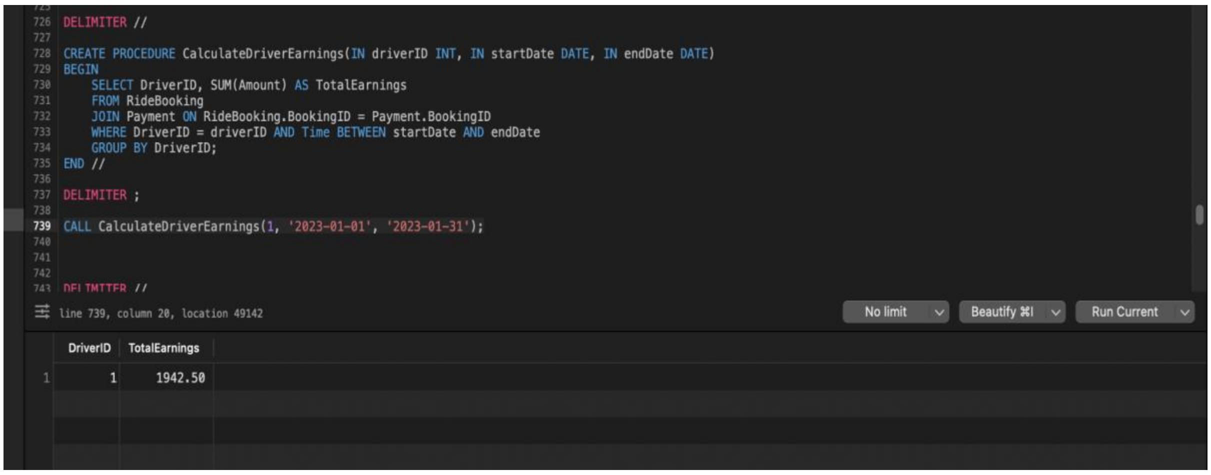
The "CalculateDriverEarnings" stored procedure is a pivotal component of the ride-booking service database, designed to calculate the total earnings of a driver within a specified date range. This procedure takes three parameters: driverID, startDate, and endDate. It performs a join between the RideBooking and Payment tables to correlate rides with their corresponding payments. The SQL query within the procedure filters records based on the provided driver ID and the date range, ensuring that only earnings from the specified period are calculated. The use of SUM(Amount) aggregates the total earnings for each driver, and the results are grouped by DriverID to provide a clear and concise summary of earnings.

This procedure is particularly useful for financial reporting and payroll processing, as it allows for quick and accurate calculations of driver earnings over any given period. By encapsulating this functionality in a stored procedure, the database ensures efficiency, reusability, and consistency in how driver earnings are computed. The ability to call this procedure with different parameters, as demonstrated in the example CALL

CalculateDriverEarnings(1, '2023-01-01', '2023-01-31'), highlights its flexibility and ease of use in various operational contexts.

DELIMITER //

```
CREATE PROCEDURE CalculateDriverEarnings(IN driverID INT, IN startDate DATE, IN
endDate DATE)
BEGIN
    SELECT DriverID, SUM(Amount) AS TotalEarnings
    FROM RideBooking
    JOIN Payment ON RideBooking.BookingID = Payment.BookingID
    WHERE DriverID = driverID AND Time BETWEEN startDate AND endDate
    GROUP BY DriverID;
END //

DELIMITER ;

CALL CalculateDriverEarnings(1, '2023-01-01', '2023-01-31');
```



```
DELIMITER //

CREATE PROCEDURE CalculateDriverEarnings(IN driverID INT, IN startDate DATE, IN endDate DATE)
BEGIN
    SELECT DriverID, SUM(Amount) AS TotalEarnings
    FROM RideBooking
    JOIN Payment ON RideBooking.BookingID = Payment.BookingID
    WHERE DriverID = driverID AND Time BETWEEN startDate AND endDate
    GROUP BY DriverID;
END //

DELIMITER ;

CALL CalculateDriverEarnings(1, '2023-01-01', '2023-01-31');


DELIMITER //
```

```
line 739, column 20, location 49142
```

```
   DriverID  TotalEarnings
1         1        1942.50
```

This Procedure gives the total earning report for Driver 1.

Trigger:

The "UpdateNumberOfTripsAfterRide" trigger is an essential feature in the ride-booking service database, designed to automatically update the trip count for vehicles after each ride. This trigger is activated after a new record is inserted into the RideBooking table.

Functionality and Workflow

- Trigger Activation: The trigger is set to execute after a new row is inserted into the RideBooking table. It checks each new booking entry as it's added.
- Condition Check: The core of this trigger lies in its conditional check, IF NEW.Status = 'Completed'. This ensures that the vehicle's trip count is updated only when a ride is marked as completed, thus maintaining accurate records.
- Updating Trip Count: Upon meeting the condition, the trigger executes an update command on the Vehicle table. It increments the NumberOfTrips by 1 for the vehicle associated with the completed ride. The association is determined by the VehicleID matching the DriverID in the new ride booking entry.

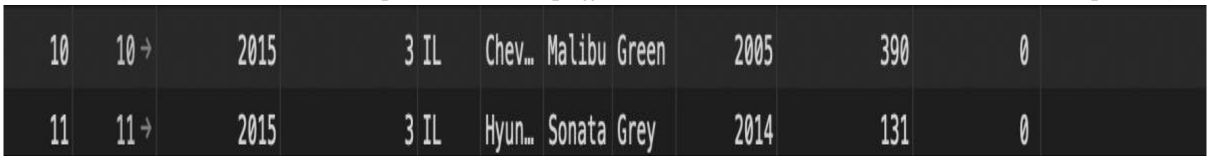
Practical Application

- Automated Data Management: This trigger automates the process of updating trip counts, reducing manual data entry and potential human errors.
- Real-Time Updates: As the trigger operates in real-time, it ensures that vehicle usage data is always current, which is crucial for operational decisions, maintenance scheduling, and performance analysis.
- Efficiency and Accuracy: By automating this process, the database not only saves time but also ensures accuracy in the vehicle usage records.

Query:

```
DELIMITER //

CREATE TRIGGER UpdateNumberOfTripsAfterRide
AFTER INSERT ON RideBooking
FOR EACH ROW
BEGIN
   IF NEW.Status = 'Completed' THEN
      UPDATE Vehicle
      SET NumberOfTrips = NumberOfTrips + 1
      WHERE VehicleID = NEW.DriverID;
   END IF;
END //

DELIMITER ;



INSERT INTO RideBooking (BookingID, CustomerID, DriverID, PickupLocation, DropoffLocation,
Time, EstimatedFare, Status)
```

VALUES (205, 5, 10, '123 Pickup St', '456 Dropoff Ave', '2023-12-15 08:00:00', 25.50, 'Completed');



10	10→	2015	3	IL	Chev…	Malibu	Green	2005	390	0
11	11→	2015	3	IL	Hyun…	Sonata	Grey	2014	131	0

For the Vehicle ID 10 the ride rose to 390 from 389

Transaction:

Transaction script for the ride-booking database exemplifies a streamlined approach to managing complex operations within a transactional context. This script begins with START TRANSACTION;, signaling the start of a transaction block, which ensures that the following operations are executed as a single unit. The first operation is an INSERT into the RideBooking table, recording a new ride booking with specific details like customer ID, driver ID, locations, time, fare, and status. The success of this insertion is crucial as it represents the initiation of a service request by a customer.

Post the booking insertion, the script assumes that the application code checks the success of this operation. If successful, the transaction proceeds with another critical step: inserting a payment record into the Payment table. This step is vital as it links the financial transaction to the specific ride booking, ensuring coherence in financial records.

The final part of the script involves a decision-making process, controlled by the application code, to either commit or roll back the transaction. This decision is based on the success of both the booking and payment insertions. If both operations are successful, the transaction is committed using COMMIT;, making all changes permanent. Conversely, if either operation fails, the transaction is rolled back using ROLLBACK;, ensuring that no partial changes are left in the database.

This transaction script demonstrates a robust approach to handling multi-step operations in a database, ensuring data integrity and consistency. By leveraging application-level logic for decision-making, it offers flexibility and control, while still maintaining the atomicity and reliability essential in transactional database operations.

Query:

START TRANSACTION;

-- Insert a new ride booking
INSERT INTO RideBooking (BookingID, CustomerID, DriverID, PickupLocation, DropoffLocation, Time, EstimatedFare, Status)
VALUES (203, 1, 2, '123 Start St', '456 End St', '2023-12-15 10:00:00', 30.00, 'Booked');

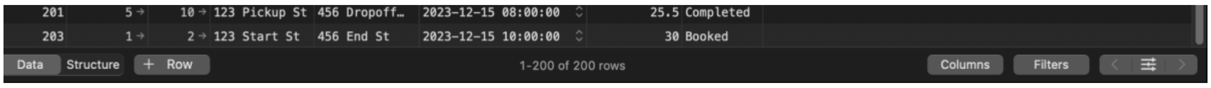
-- Application code checks if the above query was successful
-- If successful, proceed with the following insert

INSERT INTO Payment (PaymentID, CustomerID, BookingID, Amount, PaymentMethod, TransactionDate)
VALUES (201, 1, 203, 30.00, 'Credit Card', CURRENT_DATE);

-- Application code decides whether to commit or rollback based on the success of both operations

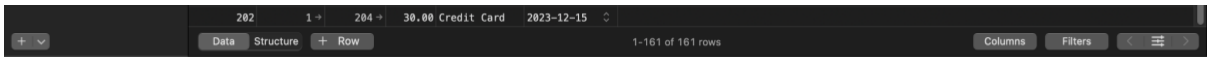
COMMIT; -- or ROLLBACK; based on the success of the operations

Ride Booking



```
201    5 →   10 → 123 Pickup St  456 Dropoff…  2023-12-15 08:00:00    25.5 Completed
203    1 →    2 → 123 Start St   456 End St    2023-12-15 10:00:00      30 Booked
```

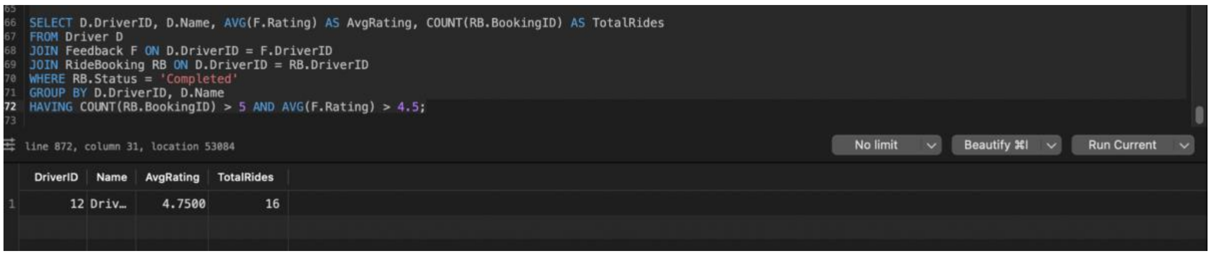
Payment Updated



```
202    1 →   204 →   30.00 Credit Card   2023-12-15
```

Analytical Questions

Question 1: Which drivers have the highest average ratings and have completed more than a specific number of rides?



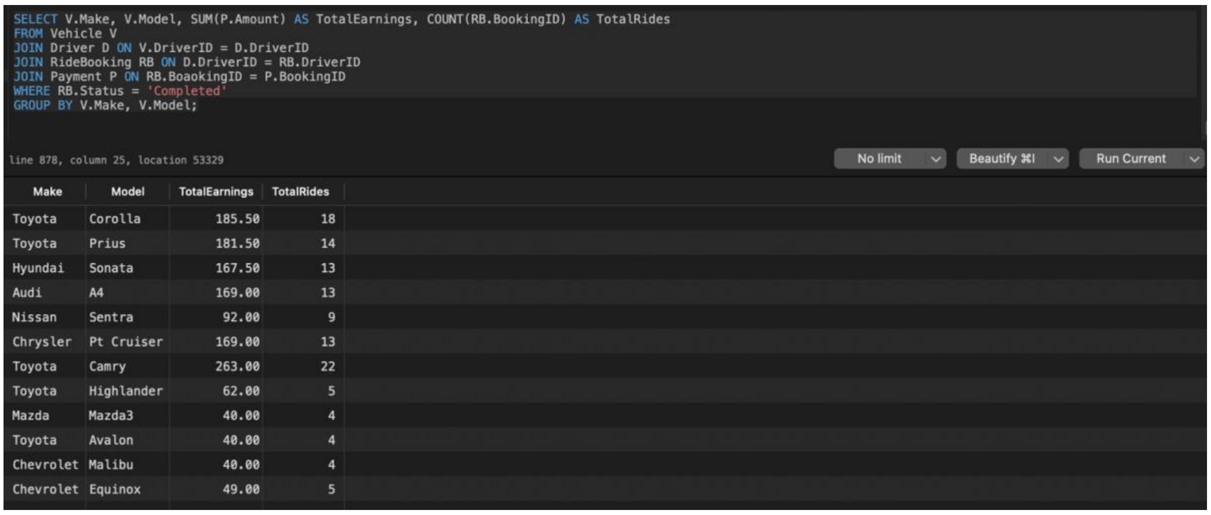
```sql
SELECT D.DriverID, D.Name, AVG(F.Rating) AS AvgRating, COUNT(RB.BookingID) AS TotalRides
FROM Driver D
JOIN Feedback F ON D.DriverID = F.DriverID
JOIN RideBooking RB ON D.DriverID = RB.DriverID
WHERE RB.Status = 'Completed'
GROUP BY D.DriverID, D.Name
HAVING COUNT(RB.BookingID) > 5 AND AVG(F.Rating) > 4.5;
```

```
line 872, column 31, location 53084
```

```
  DriverID  Name   AvgRating  TotalRides
1       12  Driv…     4.7500          16
```

Finding: Driver 12 is the most valuable driver from ratings and rides

Question 2: What are the total earnings and number of rides for each vehicle type, categorized by vehicle make and model?



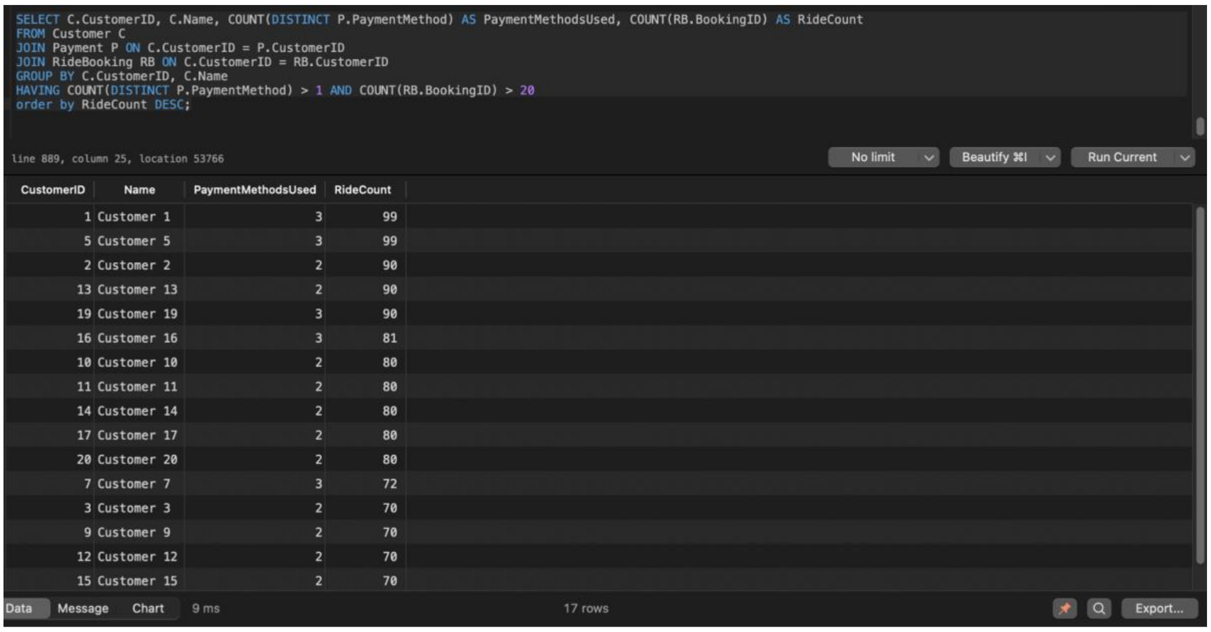
```sql
SELECT V.Make, V.Model, SUM(P.Amount) AS TotalEarnings, COUNT(RB.BookingID) AS TotalRides
FROM Vehicle V
JOIN Driver D ON V.DriverID = D.DriverID
JOIN RideBooking RB ON D.DriverID = RB.DriverID
JOIN Payment P ON RB.BoaokingID = P.BookingID
WHERE RB.Status = 'Completed'
GROUP BY V.Make, V.Model;
```

```
line 878, column 25, location 53329
```

Make	Model	TotalEarnings	TotalRides
Toyota	Corolla	185.50	18
Toyota	Prius	181.50	14
Hyundai	Sonata	167.50	13
Audi	A4	169.00	13
Nissan	Sentra	92.00	9
Chrysler	Pt Cruiser	169.00	13
Toyota	Camry	263.00	22
Toyota	Highlander	62.00	5
Mazda	Mazda3	40.00	4
Toyota	Avalon	40.00	4
Chevrolet	Malibu	40.00	4
Chevrolet	Equinox	49.00	5

Finding: Toyota Corolla and Prius have highest earning in terms of earning and rides consecutively

Question 3: Identify customers who have used multiple payment methods and have a high frequency of ride bookings.



```sql
SELECT C.CustomerID, C.Name, COUNT(DISTINCT P.PaymentMethod) AS PaymentMethodsUsed, COUNT(RB.BookingID) AS RideCount
FROM Customer C
JOIN Payment P ON C.CustomerID = P.CustomerID
JOIN RideBooking RB ON C.CustomerID = RB.CustomerID
GROUP BY C.CustomerID, C.Name
HAVING COUNT(DISTINCT P.PaymentMethod) > 1 AND COUNT(RB.BookingID) > 20
order by RideCount DESC;
```

```
line 889, column 25, location 53766
```

CustomerID	Name	PaymentMethodsUsed	RideCount
1	Customer 1	3	99
5	Customer 5	3	99
2	Customer 2	2	90
13	Customer 13	2	90
19	Customer 19	3	90
16	Customer 16	3	81
10	Customer 10	2	80
11	Customer 11	2	80
14	Customer 14	2	80
17	Customer 17	2	80
20	Customer 20	2	80
7	Customer 7	3	72
3	Customer 3	2	70
9	Customer 9	2	70
12	Customer 12	2	70
15	Customer 15	2	70

```
9 ms    17 rows
```

Finding: Customer 1 and Customer 5 have most rides with highest payment method count

Findings:

1. Driver 12 is the most valuable driver from ratings and rides
2. Toyota Corolla and Prius have highest earning in terms of earning and rides consecutively
3. Customer 1 and Customer 5 have most rides with highest payment method count

Tuple Relational Calculus of Queries

TRC of Query 1

```
{ t | ∃d ∈ Driver, ∃f ∈ Feedback, ∃rb ∈ RideBooking (
    d.DriverID = f.DriverID ∧
    d.DriverID = rb.DriverID ∧
    rb.Status = 'Completed' ∧
    t.DriverID = d.DriverID ∧
    t.Name = d.Name ∧
    t.AvgRating = AVG(f.Rating) ∧
    t.TotalRides = COUNT(rb.BookingID) ∧
    COUNT(rb.BookingID) > 5 ∧
    AVG(f.Rating) > 4.5
  )
}
```

In this TRC query:

- The tuple **t** is the output tuple.
- **∃** denotes the existence of a tuple in the relation.
- The conditions within the parentheses represent the join, selection, and aggregation conditions similar to those in the SQL query.
- **t.DriverID**, **t.Name**, **t.AvgRating**, and **t.TotalRides** are the attributes of the output tuple.

TRC of Query 2

```
{ t | ∃v ∈ Vehicle, ∃d ∈ Driver, ∃rb ∈ RideBooking, ∃p ∈ Payment (
    v.DriverID = d.DriverID ∧
    d.DriverID = rb.DriverID ∧
    rb.BookingID = p.BookingID ∧
    rb.Status = 'Completed' ∧
    t.Make = v.Make ∧
    t.Model = v.Model ∧
    t.TotalEarnings = SUM(p.Amount) ∧
    t.TotalRides = COUNT(rb.BookingID)
  )
  GROUP BY v.Make, v.Model
}
```

In this TRC query:

- **t** is the output tuple representing the results.
- **∃** denotes the existence of a tuple in each of the respective relations.
- The conditions within the parentheses specify the joins and the selection criteria.
- The attributes **t.Make**, **t.Model**, **t.TotalEarnings**, and **t.TotalRides** are the fields in the output.
- The **GROUP BY** clause is represented at the end, aligning with how the output should be grouped.

It's important to note that TRC is a theoretical model and doesn't directly support aggregation functions like SUM and COUNT.

TRC of Query 3

```
{ t | ∃c ∈ Customer, ∃p ∈ Payment, ∃rb ∈ RideBooking (
    c.CustomerID = p.CustomerID ∧
    c.CustomerID = rb.CustomerID ∧
    t.CustomerID = c.CustomerID ∧
    t.Name = c.Name ∧
    t.PaymentMethodsUsed = COUNT(DISTINCT p.PaymentMethod) ∧
    t.RideCount = COUNT(rb.BookingID) ∧
    COUNT(DISTINCT p.PaymentMethod) > 1 ∧
    COUNT(rb.BookingID) > 20
  )
  ORDER BY t.RideCount DESC
}
```

In this TRC query:

- **t** is the output tuple representing the results.
- **∃** denotes the existence of a tuple in the respective relations.
- The conditions inside the parentheses specify the joins, the selection criteria, and the calculations for distinct payment methods and ride count.
- The **ORDER BY t.RideCount DESC** clause is included to represent the ordering of the results by the total number of rides in descending order.

Note that in actual TRC, operations like **COUNT**, **DISTINCT**, and **ORDER BY** are not directly represented as they are in SQL.

Limitations

Synthetic Data Generation: The data used in the database is artificially generated, which may not accurately reflect real-world scenarios. This can lead to unrealistic insights and patterns that do not hold in actual operational environments.

Lack of Real-World Validation: Without real-world data, it's challenging to validate the database design and query outputs. This limitation can affect the reliability of the conclusions drawn from the data analysis.

Scalability and Performance Testing: The database, while functional with generated data, has not been tested under real-world data volumes and usage patterns. Its performance and scalability in a live environment remain unverified.

Simplified Assumptions: The database design and queries are based on simplified assumptions about the ride-booking business model. They may not account for complex business rules, diverse customer behaviors, and operational exceptions.

Data Privacy and Security: Since the focus was on data generation and query execution, aspects like data privacy, security, and compliance with regulations (e.g., GDPR) were not considered. These are critical for real-world applications.

Conclusion

The database created for the ride-booking service, with self-generated data, serves as a valuable prototype to demonstrate the potential capabilities and functionalities of a ride-booking system. The SQL queries formulated provide insights into various aspects of the business, such as driver performance, vehicle utilization, and customer engagement. These insights are instrumental in understanding the operational dynamics of a ride-booking service.

However, the limitations stemming from the use of synthetic data highlight the need for real-world data to validate and refine the database. Future enhancements should focus on incorporating actual operational data, rigorous performance testing, and addressing privacy and security concerns. This will ensure that the database not only supports the current operational requirements but is also scalable, secure, and adaptable to the evolving needs of a real-world ride-booking service.