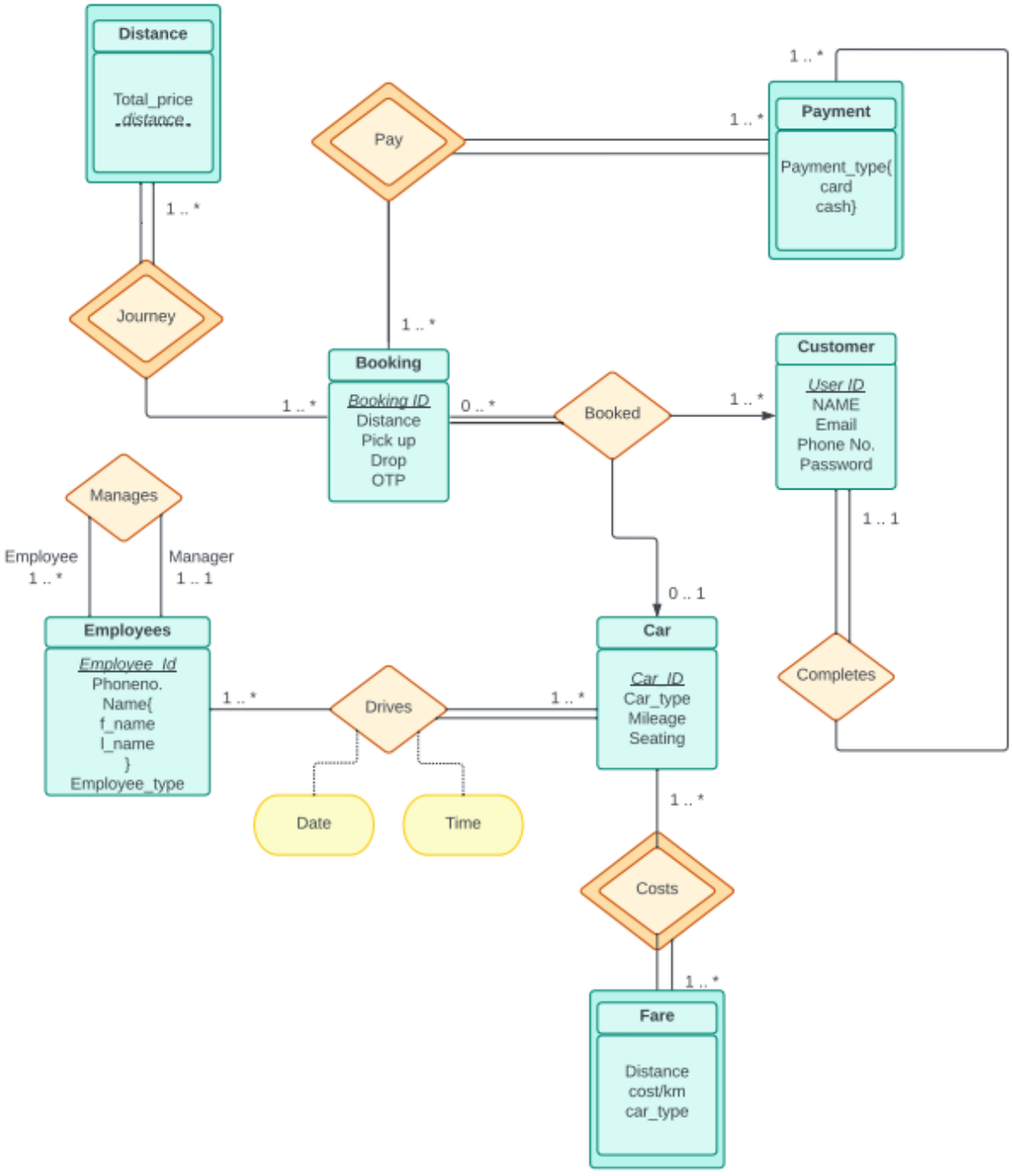
README FILE(Final)

-Poorvi Kumar 2021343
-Manik Sharma 2021336

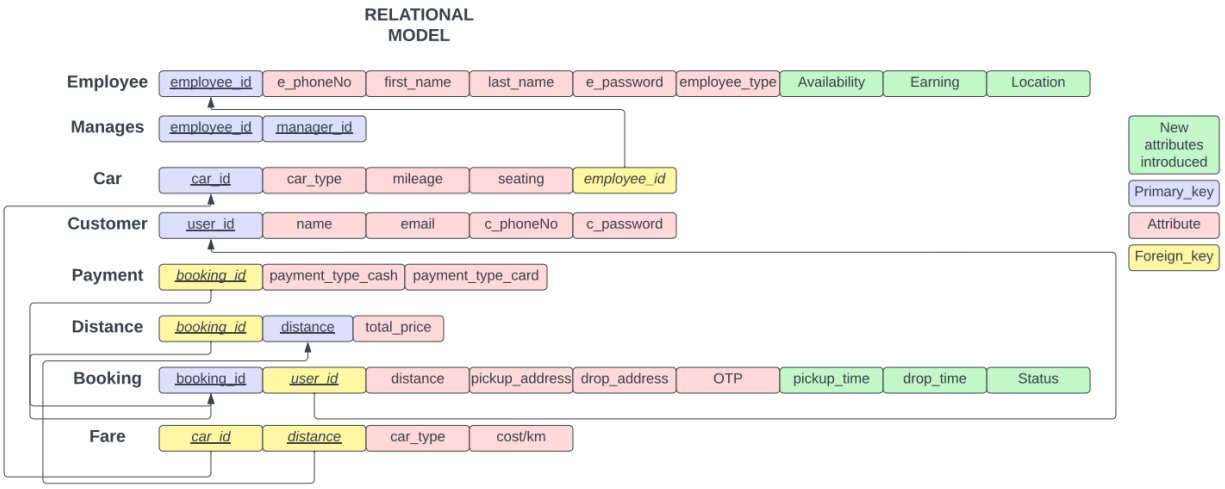
CONTENTS

- **ER Diagram (submission 2)**
- **Relational Model (submission 2)**
- **Create Table Code(submission 3)**
- **Queries (submission 4)**
- **Embedded Queries, Olap Queries and Triggers (submission 5)**
- **CLI (submission 6)**
- **Transactions (submission 6)**

ER Diagram


Distance
Total_price
distance
Pay
1 .. *
Payment
Payment_type{
card
cash}
Journey
Booking
Booking ID
Distance
Pick up
Drop
OTP
0 .. *
Booked
Customer
User ID
NAME
Email
Phone No.
Password
Manages
Employee
1 .. *
Manager
1 .. 1
Employees
Employee_Id
Phoneno.
Name{
f_name
l_name
}
Employee_type
Drives
Car
Car_ID
Car_type
Mileage
Seating
Completes
Date
Time
Costs
Fare
Distance
cost/km
car_type

Relational Model



Create Table Code:

```
DROP DATABASE IF EXISTS `sql_cab`;
CREATE DATABASE `sql_cab`;
USE `sql_cab`;

SET NAMES utf8 ;
SET character_set_client = utf8mb4 ;

CREATE TABLE `Employee` (
`employee_id` tinyint(4) NOT NULL AUTO_INCREMENT,
`e_phoneNo` varchar(50) NOT NULL,
    `first_name` varchar(50) NOT NULL,
    `last_name` varchar(50) NOT NULL,
    `e_password` varchar(20) NOT NULL,
    `employee_type` varchar(50) NOT NULL,
    Primary key(`employee_id`)
);-- ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

The "Employee" table stores information about the employees working in the cab booking system, such as their name, phone number, password, and employee type. It uses the "employee_id" column as the primary key for this table.

```
CREATE TABLE `car`(
`car_id` tinyint(4) NOT NULL AUTO_INCREMENT,
`car_type` varchar(50) NOT NULL,
`mileage_kpl` decimal(4,2) NOT NULL,
`seating` varchar(20) NOT NULL,
`employee_id` tinyint(4) NOT NULL,
PRIMARY KEY (`car_id`),
KEY `fk_employee_id` (`employee_id`),
CONSTRAINT `fk_employee_id` FOREIGN KEY (`employee_id`) REFERENCES `employee` (`employee_id`) ON
);-- ENGINE=InnoDB AUTO_INCREMENT=11 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

The "Car" table stores information about the cars available in the cab booking system, such as the car type, mileage, seating capacity, and the employee who is assigned to that car. It uses the "car_id" column as the primary key for this table, and also creates a foreign key constraint to link the "employee_id" column to the "employee_id" column in the "Employee" table.

```
CREATE TABLE `Customer` (
  `user_id` int(6) NOT NULL,
  `name` VARCHAR(255) NOT NULL,
  `email` VARCHAR(255) NOT NULL,
  `c_phoneNo` VARCHAR(20) NOT NULL,
  `c_password` VARCHAR(255) NOT NULL,
  PRIMARY KEY (`user_id`)
);
```

The "Customer" table stores information about the customers who use the cab booking system, such as their name, email, phone number, and password. It uses the "user_id" column as the primary key for this table.

```
 CREATE TABLE `Booking` (
  `booking_id` VARCHAR(25) NOT NULL,
  `user_id` int(6) NOT NULL,
  `distance_km` INT NOT NULL,
  `pickup` VARCHAR(255) NOT NULL,
  `destination` VARCHAR(255) NOT NULL,
  `otp` INT NOT NULL,
PRIMARY KEY (`booking_id`),
KEY `fk_user_id` (`user_id`),
-- KEY `fk_distance_km`(`distance_km`),
CONSTRAINT `fk_user_id` FOREIGN KEY (`user_id`) REFERENCES `Customer`(`user_id`)
    -- CONSTRAINT `fk_distance_km` FOREIGN KEY (`distance_id`) REFERENCES `Distance`(`distance_
);
```

The "Booking" table stores information about the cab bookings made by customers, such as the booking ID, customer ID, distance, pickup and destination location, and an OTP. It uses the "booking_id" column as the primary key for this table, and also creates

a foreign key constraint to link the "user_id" column to the "user_id" column in the "Customer" table.

```
CREATE TABLE `Distance` (
`booking_id` VARCHAR(25) NOT NULL,
`distance_km` INT NOT NULL,
`total_price` DECIMAL(10, 2) NOT NULL,
-- KEY `fk_booking_id` (`booking_id`),
-- CONSTRAINT `fk_booking_id` FOREIGN KEY (`booking_id`) REFERENCES `Booking`(`booking_id`),
    PRIMARY KEY (`booking_id`),
    FOREIGN KEY (`booking_id`) REFERENCES `Booking`(`booking_id`)
ON UPDATE CASCADE ON DELETE CASCADE
);
```

The "Distance" table stores information about the distance covered in each cab booking, such as the booking ID, distance, and total price. It uses the "booking_id" column as the primary key for this table, and also creates a foreign key constraint to link the "booking_id" column to the "booking_id" column in the "Booking" table.
We create a database named "sql_cab" and set its default character encoding to UTF-8.
By creating a table named "Employee" with the following columns:
employee_id: An auto-incrementing tinyint (4-byte integer) field that acts as the primary key for the table.
e_phoneNo: A varchar (50 characters) field that stores the employee's phone number.
first_name: A varchar (50 characters) field that stores the employee's first name.
last_name: A varchar (50 characters) field that stores the employee's last name.
e_password: A varchar (20 characters) field that stores the employee's password.
employee_type: A varchar (50 characters) field that stores the type of employee.

The table is defined with the primary key constraint on the employee_id column.

Vice-Versa for every Table.

In conclusion, We create a database and tables that can store information required to manage a cab booking system.

Queries

```
USE sql_cab;
```

1) Show all the employees that come under the manager with manager_id=32 via the relation 'manages'.

```
SELECT manager_id,employee_id
FROM manages
WHERE manager_id= 32;
```

-- 2 Show all the employes that are available(i.e. do not have a passenger allotted to them).

```
SELECT employee_id,first_name,last_name,availability
FROM employee
WHERE availability=1;
```

-- 3 List cabs from highest to lowest mileage.

```
SELECT *
FROM car
ORDER BY mileage_kpl DESC;
```

-- 4 Show the number of cars registered under each category(cabSMALL, cabMED, cabXL).

```
SELECT car_type,COUNT(*) number_of_cars
 FROM car
 GROUP BY car_type;
```

-- 5 Join booking table and customer table mapped according to the user_id.

```
SELECT book.booking_id, book.user_id , cust.name, book.status, book.destination, book.pickup, book.otp
FROM customer cust
LEFT JOIN booking book
ON cust.user_id = book.user_id;
```

-- 6 Alter the booking table such that those bookings with booking status as 'dropped' get removed from the table.

```
DELETE
FROM booking
WHERE `status`='dropped';
SELECT * FROM booking;
```

-- 7 Print the information of those employess that are present in a particular area so the user knows who are the drivers
-- available in a specific area.

```
SELECT * FROM employee
WHERE  availability=TRUE
AND location = 'New York';
```

-- 8 Return all te bookings that are on the way and will take some time to reach the pick up spot.

```
SELECT * FROM booking
WHERE `status`='on the way'
AND pickup_time > NOW();
```

-- 9 Listdown all the bookings of a particular driver.

```
SELECT *
FROM booking
WHERE employee_id = 15;
```

-- 10 Query to listdown all the customers who are either on the way or have just been picked up.

```
SELECT c.*
FROM customer c
JOIN booking b ON c.user_id = b.user_id
WHERE b.status <> 'dropped'
GROUP BY c.user_id;
```

-- 11 Desc command

DESC booking

Field	Type	Null	Key	Default	Extra
booking_id	varchar(25)	NO	PRI	NULL	
user_id	int	NO	MUL	NULL	
distance_km	int	NO		NULL	
pickup	varchar(255)	NO		NULL	
destination	varchar(255)	NO		NULL	
otp	int	NO		NULL	
status	varchar(90)	NO		NULL	
pickup_time	datetime	YES		NULL	
drop_time	datetime	YES		NULL	
employee_id	tinyint	NO		NULL	

Embedded Queries, Olap Query & Triggers

Embedded Queries

A simple menu-driven program that displays options for each query and accepts user input to execute the desired query.

PREFERRED LANGUAGE USED: C++

This C++ program connects to a MySQL database and provides a menu of four options for the user.

Option 1: Display the number of bookings made by each customer. This query selects the user ID, counts the number of bookings for each user from the "Booking" table, and groups the results by user ID. The program then retrieves and displays the results.

Option 2: Display the total distance and price for each booking. This query selects the booking ID, sums the distance and total price for each booking from the "Distance" table, and groups the results by booking ID. The program then retrieves and displays the results.

Option 3:
Display all the bookings. This query selects all columns from the "Booking" table and retrieves and displays all the results.

Option 4: Display all the cars. This query selects all columns from the "Car" table and retrieves and displays all the results.

Option 5,6,7,8,9,10,..... And many more.

The program uses the MySQL C API to connect to the database and execute the queries. It prompts the user to input a menu option and executes the corresponding query. After executing each query, it retrieves and displays the results using a while loop to iterate over each result set row. To run this program, the MySQL C API must be installed and linked to your C++ project. You will also need to replace the placeholder values for the database host, username, and password with the appropriate values for your MySQL database.

OLAP Queries

-- 1) Show the total cars registered at the given time(such a query can be used by the higher authorities to check the total number of cars registered)

```
SELECT car_type,COUNT(*) number_of_cars
 FROM car
 GROUP BY car_type WITH ROLLUP;
```

-- 2)This query uses the **ROLLUP** function to generate average distance traveled by an employee (driver) and the final average.

```
SELECT employee_id,  AVG(distance_km) AS avg_distance
FROM booking
GROUP BY (employee_id) WITH ROLLUP;
```

-- 3) Query to show total distance traveled and the number of bookings made by each user, grouped by **user_id** and also show the grand total at the end:

```
SELECT user_id, SUM(distance_km) AS total_distance, COUNT(*) AS num_bookings
FROM Booking
GROUP BY user_id WITH ROLLUP;
```

-- 4) Query to show the total distance traveled and the total price paid for each booking, grouped by **booking_id**, and also show the grand total at the end:

```
SELECT booking_id, SUM(distance_km) AS total_distance, SUM(total_price) AS total_price
FROM Distance
GROUP BY booking_id WITH ROLLUP;
```

Triggers

1)This trigger will fire after an update is made on the **Distance** table, and it will update the **status** column of the corresponding booking in the **booking** table to 'completed' based on the **booking_id** column.

```
DELIMITER $$
CREATE TRIGGER update_booking_status
AFTER UPDATE ON Distance
FOR EACH ROW
BEGIN
   UPDATE booking
   SET status = 'completed'
   WHERE booking_id = NEW.booking_id;
END $$
DELIMITER ;
```

2)This trigger will be activated after a row is deleted from the **booking table**, and it will update the **car_status** column of the corresponding car in the **car table** to 'available' based on the **employee_id** column.

```
DELIMITER $$
CREATE TRIGGER update_car_status AFTER DELETE ON booking
FOR EACH ROW
BEGIN
   UPDATE car
   SET car_status = 'available'
   WHERE employee_id = car.employee_id;
END$$
DELIMITER ;
```

Command Line Interface

A simple menu-driven program that displays options for each query and accepts user input to execute the desired query.

PREFERRED LANGUAGE USED: C++

This programme is a console-based C++ application that communicates with a MySQL database via the MySQL C API. The user is given a number of choices when the programme asks them to log in or register as a customer, enter developer mode, or quit. Once logged in, a user may examine their ride history, book a ride, and leave the customer menu.

The developer mode option offers a number of database queries that may be run, such as showing all reservations, showing all cars, showing the amount of bookings made by each client, and more.

This C++ program connects to a MySQL database and provides a menu of four options for the user.
Macros related to connection is

```
MYSQL* conn;
MYSQL_ROW row;
MYSQL_RES* res;

conn = mysql_init(NULL);
```

```
if (mysql_real_connect(conn, "localhost", "root", "root", "sql_cab", 3306, NULL, 0) == NULL)
{
    cerr << "Error: " << mysql_error(conn) << endl;
    exit(1);
}
```

Components are divided into 4 sections:
Sign Up:
Inserting values Query.
Sign In:
Using a Query to open a particular user Database.

Developer options:

Option 1: Display the number of bookings made by each customer. This query selects the user ID, counts the number of bookings for each user from the "Booking" table, and groups the results by user ID. The program then retrieves and displays the results.

Option 2: Display the total distance and price for each booking. This query selects the booking ID, sums the distance and total price for each booking from the "Distance" table, and groups the results by booking ID. The program then retrieves and displays the results.

Option 3:
Display all the bookings. This query selects all columns from the "Booking" table and retrieves and displays all the results.

Option 4: Display all the cars. This query selects all columns from the "Car" table and retrieves and displays all the results.

Option 5,6,7,8,9,10,..... And many more.

The program uses the MySQL C API to connect to the database and execute the queries. It prompts the user to input a menu option and executes the corresponding query. After executing each query, it retrieves and displays the results using a while loop to iterate over each result set row. To run this program, the MySQL C API must be installed and linked to the C++ project. You will also need to replace the placeholder values for the database host, username, and password with the appropriate values for your MySQL database.

* USE MICROSOFT VISUAL STUDIO 2022

Exit:

Note:
By using Google Map API, we can do bookings from multiple locations.

Transactions

Transaction 1:

START TRANSACTION;
UPDATE employee SET earning =earning +200 WHERE employee_id=10;
UPDATE booking SET status='dropped' WHERE employee_id=10;
COMMIT;

Transaction 2:

START TRANSACTION;
UPDATE employee SET location = 'Central Park, New York' WHERE employee_id = 11;
UPDATE booking SET employee_id = 11 WHERE booking_id = '44-067-8359';
COMMIT;

Conflict Serializable Schedule:
Conflicting
Non-Conflicting

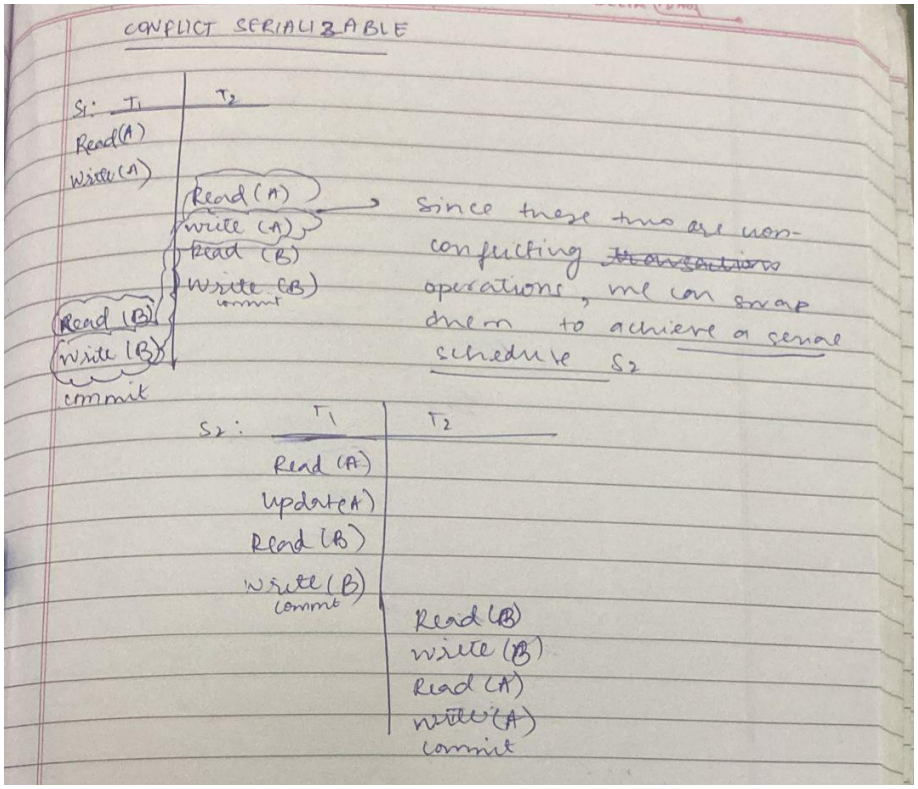
S1

T1	T2
READ(employee)	
WRITE(employee)	
	READ(employee)
	WRITE(employee)
	READ(booking)
	WRITE(booking)
READ(booking)	COMMIT
WRITE(booking)	
COMMIT	

Since the yellow operations in S1 are non-conflicting operations, we can swap the to achieve a serial schedule S2.

S2

T1	T2
READ(employee)	
WRITE(employee)	
READ(booking)	
WRITE(booking)	
COMMIT	READ(employee)
	WRITE(employee)
	READ(booking)
	WRITE(booking)
	COMMIT



Non-Conflict Serializable Schedule:
S1

T1	T2
READ(employee)	
WRITE(employee)	
	READ(employee)
	WRITE(employee)
READ(booking)	
WRITE(booking)	
COMMIT	READ(booking)
	WRITE(booking)
	COMMIT

Non-Conflict Resolution Schedule using locks:
S2
Red 'WRITES' after locking will not work.

T1	T2
READ(employee)	
lock	
WRITE(employee)	
	READ(employee)
	unlock
	WRITE(employee)
WRITE(employee)	
READ(booking)	
lock	
WRITE(booking)	
	READ(booking)
	unlock
	WRITE(booking)
WRITE(booking)	
COMMIT	COMMIT

There is no conflict resolution using locks between the transactions, and they can be executed without any issues.

Conflict Resolution Schedule:

T1	T2
START TRANSACTION;	START TRANSACTION;

SELECT * FROM employee WHERE employee_id = 10 FOR UPDATE;	
UPDATE employee SET earning = earning + 200 WHERE employee_id = 10;	
SELECT * FROM booking WHERE employee_id = 10 FOR UPDATE;	
UPDATE booking SET status = 'dropped' WHERE employee_id = 10;	
COMMIT;	
	SELECT * FROM employee WHERE employee_id = 11 FOR UPDATE;

	UPDATE employee SET location = 'Central Park, New York' WHERE employee_id = 11;
	SELECT * FROM booking WHERE booking_id = '44-067-8359' FOR UPDATE;
	UPDATE booking SET employee_id = 11 WHERE booking_id = '44-067-8359';
	COMMIT;

In this schedule, T1 and T2 acquire an exclusive lock on the rows it updates and a shared lock on the rows it reads. This ensures that there are no conflicts.