QUERY 1:

Let's consider a scenario where two customers, Alice and Bob, are trying to book the same cab at the same time, but with a serializable transaction isolation level. Here's how a conflicting transaction could occur:

Alice calls the cab booking service and requests a ride from her current location to a nearby shopping mall. The operator checks the available cabs and assigns cab #123 to Alice's request. The operator writes down the details of the ride, including the cab number, the pickup location, and the destination, on a paper form.

At the same time, Bob uses the cab booking app to request a ride from his office to the same shopping mall. The app shows him that cab #123 is available and he confirms the booking. The app sends the booking details, including the cab number, the pickup location, and the destination, to the cab booking service's backend server.

The server receives Bob's booking request and starts a serializable transaction to assign cab #123 to his request. The server checks the available cabs and sees that cab #123 is free, so it locks the cab record to prevent other transactions from accessing it.

However, at the same time, the operator completes Alice's ride booking form and hands it over to the cab driver. The driver checks the form and sees that the pickup location and destination match Alice's request. The driver starts driving towards Alice's pickup location.

The server tries to insert Bob's booking details into the database, but it detects a conflict with the record that the operator has already inserted for Alice's booking. Since the server is using a serializable isolation level, it rolls back Bob's transaction and releases the lock on the cab record.

Bob's app receives an error message saying that his booking could not be confirmed, and he has to retry the booking. Meanwhile, Alice's ride proceeds as planned, and she reaches the shopping mall on time.

As a result of this conflicting transaction, Bob's ride gets canceled, but Alice's ride proceeds as expected, and there are no inconsistencies in the cab booking system. This shows how a serializable transaction isolation level can prevent conflicting transactions and ensure the consistency and correctness of the system.

Non-conflicting serializable transaction query:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- First operation: INSERT a record for Alice's booking
INSERT INTO rides_data (c_id, pickup, destination, cab_no, status) VALUES ('Alice',
'Home', 'Shopping mall', '123', 'Pending');

-- Second operation: UPDATE the status of Alice's booking to 'Confirmed'
UPDATE rides_data SET status = 'Confirmed' WHERE c_id = 'Alice' AND status =
'Pending';

COMMIT;
```

In this example, the first operation inserts a record for Alice's booking with the details of her pickup location, destination, and the cab assigned to her. The second operation updates the status of Alice's booking from 'Pending' to 'Confirmed'. These two operations do not conflict with each other, and can be executed in any order without affecting the final result.

Conflicting serializable transaction query:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- First operation: SELECT the cab record and lock it
SELECT * FROM cabs WHERE cab_no = '123' FOR UPDATE;

-- Second operation: INSERT a record for Bob's booking
INSERT INTO rides_data (c_id, pickup, destination, cab_no, status) VALUES ('Bob',
'Office', 'Shopping mall', '123', 'Pending');

COMMIT;
```

In this example, the first operation selects the cab record with cab number '123' and locks it for update, preventing other transactions from accessing it until this transaction

is completed. The second operation tries to insert a record for Bob's booking with the same cab number, but it detects a conflict with the lock held by the first operation. Since the transaction is using a serializable isolation level, the second operation will be rolled back and the lock on the cab record will be released. As a result, Bob's booking will not be confirmed, and he will have to retry the booking.

T1	T2
Reads(whether driver A is free)	
	Reads(whether driver A is free)
Write(Books a drive with A)	
Commit	
	Write(Books a drive with A)
	Commit

T1	T2
Reads(whether driver A is free)	
	Lock(Reads(whether driver A is free))
Write(Books a drive with A)	Wait
Commit	Wait
	Write(Books a drive with A)
	Abort

QUERY 2:

Here's how a conflicting serializable transaction could occur:

1. The administrator of the cab booking service logs into the system and wants to retrieve the details of a particular driver, say Driver A, who has been working with the service for the past year. The administrator starts a serializable transaction to ensure that no other transaction can modify the data while the query is running.
2. At the same time, a customer, say Customer X, opens the booking app to book a ride. The app shows the available cabs in the area, including the ones that Driver A is driving. Customer X selects a cab driven by Driver A and confirms the booking.
3. The app sends the booking details, including the driver's ID, to the backend server, which starts a serializable transaction to assign the booking to the driver.
4. The server queries the driver's information to check if the driver is available and eligible to take the booking. However, the server cannot access the driver's information as it is locked by the administrator's transaction.
5. Meanwhile, the administrator's transaction queries the driver's information and retrieves it successfully. The administrator then updates the driver's information with the latest performance data.
6. The server's transaction then tries to insert the booking details into the database, but it detects a conflict with the record that the administrator has already updated. Since the server is using a serializable isolation level, it rolls back the transaction and releases the lock on the driver's record.
7. The customer's booking request fails with an error message saying that the booking could not be confirmed due to a conflict. The customer has to retry the booking with another driver.
8. The administrator's transaction commits successfully, and the driver's information is updated.

As a result of this conflicting serializable transaction, Customer X's booking request gets canceled, and the customer has to retry the booking. However, the cab booking system remains consistent and correct, and the administrator's update to the driver's information is committed without any issues. This shows how a serializable transaction isolation level can prevent conflicting transactions and ensure the consistency and correctness of the system.

Here's an example of a conflicting serializable transaction for the above scenario:

```
-- Transaction 1: Admin reads driver's information
START TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SELECT * FROM drivers WHERE driver_id = 123;

-- while transaction 1 is still running, transaction 2 starts

-- Transaction 2: Customer books a cab
START TRANSACTION ISOLATION LEVEL SERIALIZABLE;

INSERT INTO rides_data(c_id, d_id, car_no, status) VALUES(456, 123, 'ABC123',
'ongoing');

COMMIT;
```

In the above conflicting serializable transaction, the admin and the customer are accessing the same driver's information and the same cab at the same time. Transaction 1 locks the driver's record and starts reading the information, while transaction 2 tries to book a ride with that driver.
As a result of the conflict, one of the transactions will have to roll back. If transaction 2 rolls back, the customer's ride won't be booked, and if transaction 1 rolls back, the admin won't get the information they need.

Here's an example of a non-conflicting serializable transaction for the above scenario:

```
-- Transaction 1: Admin reads driver's information
START TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SELECT * FROM drivers WHERE driver_id = 123;

COMMIT;

-- Transaction 2: Customer books a cab
START TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

INSERT INTO rides_data(c_id, d_id, car_no, status) VALUES(456, 789, 'XYZ789', 'ongoing');

COMMIT;

In the above non-conflicting serializable transaction, the admin and the customer are accessing different drivers and cabs. Therefore, both transactions can run in parallel without conflicting with each other.

T1(By Customer)	T2(By administrator)
Read(whether driver is free)	
	Read(whether driver is free)
Write(Books driver for a ride)	
Commit	
	Reads (Reading the rides by driver to calculate total earning)
	Commit

T1(By Customer)	T2(By administrator)
Read(whether driver is free)	
	Lock(Read(whether driver is free))
Write(Books driver for a ride)	wait
Commit	wait
	Reads(Reading the rides by driver)
	Abort

QUERIES:

1.)

T1(By Customer)	T2(By administrator)
Read(whether driver A is free)	
	Read(whether driver B is free)
Write(Books driver A for a ride)	
Commit	
	Reads (Reading the rides by driver B to calculate total earning)
	Commit

2.)

T1(By Customer A)	T2(By Customer B)
Read(whether driver A is free)	
	Read(whether driver B is free)
Write(Books driver A for a ride)	
Commit	
	Write(Books driver B for a ride)
	Commit

3.)

T1(By Administrator X)	T2(By Administrator Y)
Read(whether taxi A is free)	
	Read(whether taxi B is free)
Write(read rides_data for taxi A)	
Commit	
	Writes(Removes taxi B)
	Commit