Pharmacy Store

SIDHANT KR AGRAWAL(2021495) || PARAS DHIMAN(2021482)
CureCabin

CONFLICTING AND NON-CONFLICTING Queries:

NON-CONFLICTING QUERIES:

1. **First (Creating a Stored Procedure for Updating Contact Numbers with Transactions):**

```
-- Set the delimiter to $$ so that semicolons within the stored procedure don't terminate the CREATE PROCEDURE statement
DELIMITER $$

-- Define the stored procedure
CREATE PROCEDURE example_transaction()
BEGIN
    -- Begin a transaction
    START TRANSACTION;

    -- Update patient's contact number
    UPDATE Patients SET ContactNumber_number = '187654321' WHERE PatientID = 5;

    -- Update doctor's contact number
    UPDATE Doctors SET ContactNumber_number = '123456789' WHERE DoctorID = '3';

    -- Commit the transaction
    COMMIT;
END;
$$ -- End of stored procedure

-- Reset the delimiter to semicolon
```

```
DELIMITER ;

CALL example_transaction();
```

Explanation:
This query defines a MySQL stored procedure named example_transaction which performs a transaction that updates the contact number of a patient and a doctor in their respective tables.

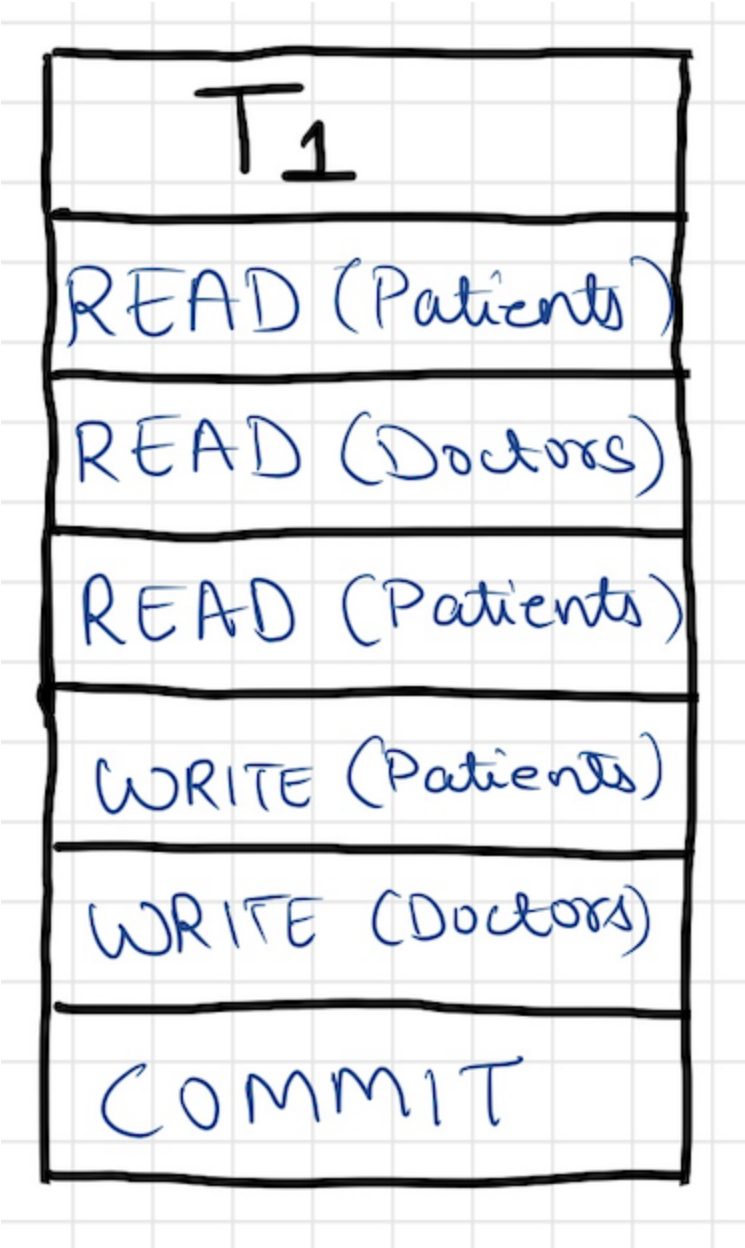
The DELIMITER statement is used to change the delimiter from the default semicolon ; to $$. This is done so that the semicolons within the stored procedure statements do not terminate the CREATE PROCEDURE statement prematurely. Once the stored procedure definition is complete, the delimiter is set back to the semicolon ;.

The START TRANSACTION statement begins a transaction in MySQL, which ensures that the updates to the database are performed atomically, meaning either all the updates are applied successfully or none of them are applied at all.

The first UPDATE statement modifies the ContactNumber_number field of a patient in the Patients table whose PatientID is 5. The second UPDATE statement modifies the ContactNumber_number field of a doctor in the Doctors table whose DoctorID is 3.

The COMMIT statement ends the transaction and commits the changes made during the transaction to the database. If any of the updates fail, the transaction will be rolled back, and no changes will be made to the database.

Finally, the stored procedure is executed by calling it with the CALL statement. The stored procedure is executed with the previously defined DELIMITER of ;.



2. **Second (Stored procedure for inserting a new prescription and updating stock quantity with a transaction):**

```
DELIMITER $$ -- Set the delimiter to something other than semicolon

CREATE PROCEDURE example_transaction1()
BEGIN
   START TRANSACTION; -- Begin the transaction
```

```
    -- Insert new prescription
    INSERT INTO Prescription (PrescriptionID, DoctorID, PatientID, MedicineID,
Dosage, Duration_Days, StartDate, EndDate, Date, DateWritten)
        VALUES (101, 90, 90, 10, 20, 6, '2020-07-04', '2020-07-10', '2020-07-04',
'2020-07-04');

    -- Update stock quantity
    UPDATE Stock SET Quantity = Quantity - 30 WHERE MedicineID = '1' AND
BranchID = '1';

    COMMIT; -- Commit the transaction
END $$ -- End of stored procedure

DELIMITER ; -- Reset the delimiter back to semicolon

CALL example_transaction1()
```

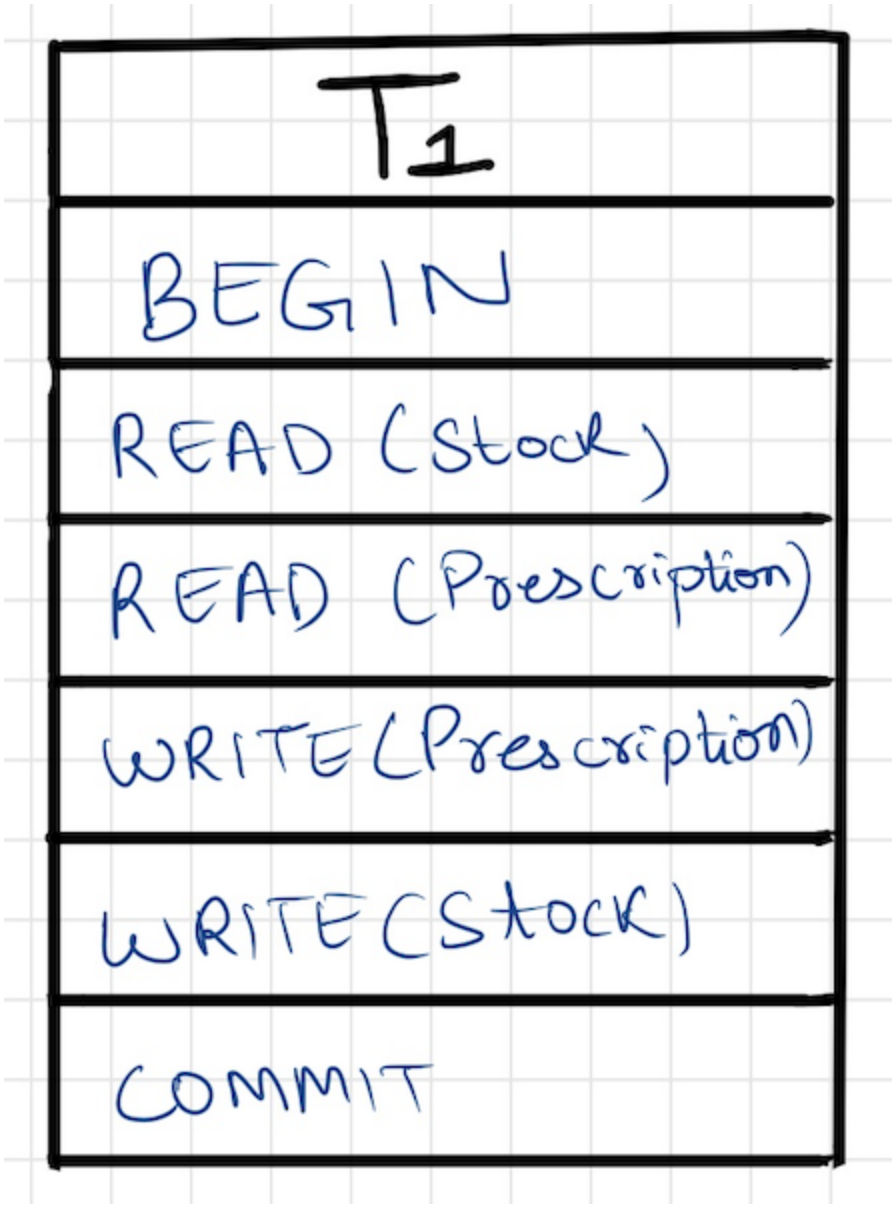
Explanation:

The command you provided is not conflicting because it uses the DELIMITER command to temporarily change the delimiter to $$, which allows the stored procedure to contain semicolons without prematurely ending the command.

The command then defines a stored procedure named example_transaction1() that begins a transaction using START TRANSACTION, inserts a new record into a table named Prescription, updates a record in the Stock table, and finally commits the transaction using COMMIT.

After defining the stored procedure, the DELIMITER command is used again to reset the delimiter back to semicolon, and then the stored procedure is called using the CALL command.

Overall, this command is non-conflicting because it defines a stored procedure and performs a transaction with database tables, which is a common and expected use case in database management. The DELIMITER command is used to avoid conflicts caused by semicolons in the stored procedure definition.



3. **Third (Updating Supplier Address in a Transaction):**

```
-- Set the delimiter to $$ so that semicolons within the stored procedure don't terminate the CREATE PROCEDURE statement
DELIMITER $$

-- Define the stored procedure
CREATE PROCEDURE example_transaction3()
BEGIN
    -- Begin a transaction
```

```
    START TRANSACTION;
    UPDATE Suppliers SET Address = 'New Address' WHERE SupplierID = 1;
    COMMIT;
END;
$$ -- End of stored procedure

-- Reset the delimiter to semicolon
DELIMITER ;

CALL example_transaction3();
```

Explanation:

This MySQL SQL query creates a stored procedure called "example_transaction3" that performs an update operation on a table called "Suppliers" by setting the "Address" column to 'New Address' where the "SupplierID" is 1.

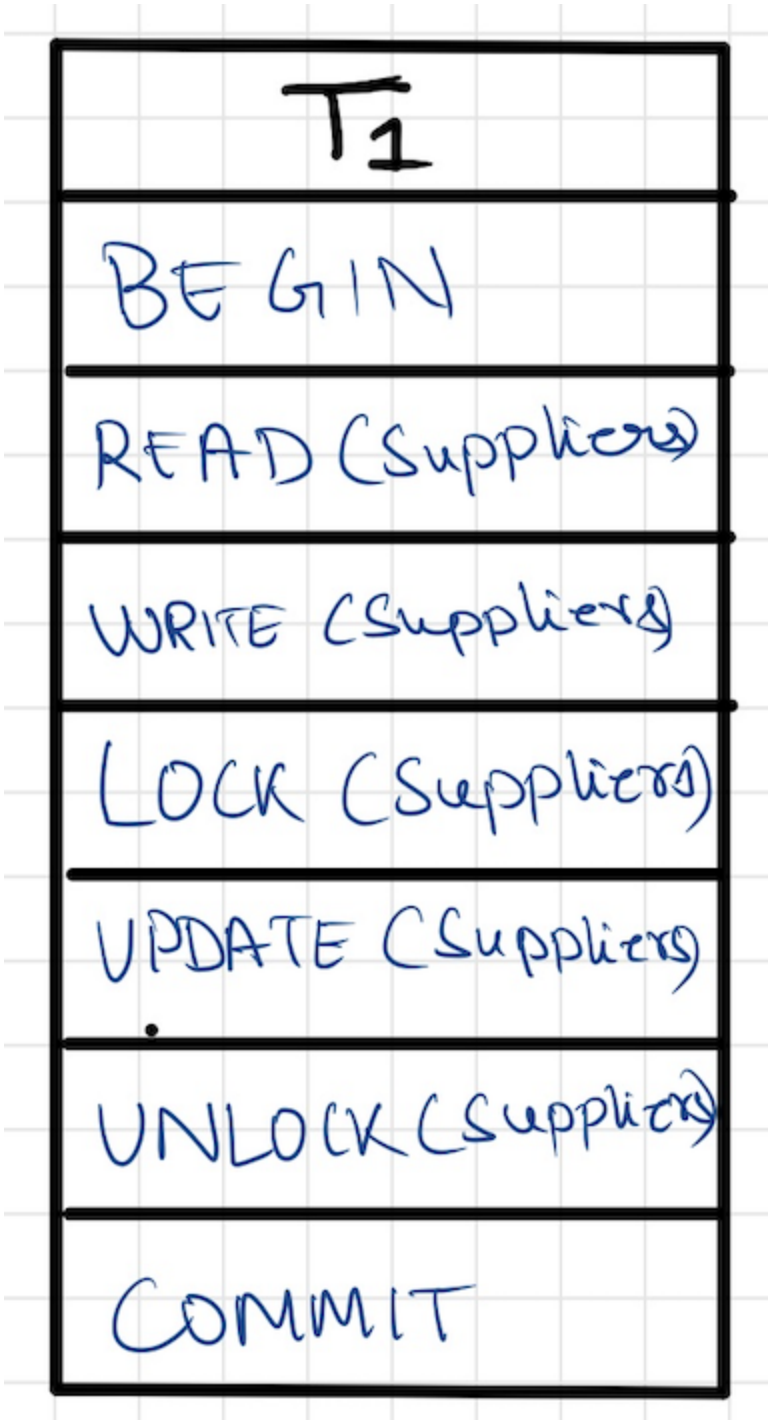
To ensure that semicolons within the stored procedure don't terminate the CREATE PROCEDURE statement, the delimiter is temporarily set to "$$". The CREATE PROCEDURE statement begins with the "CREATE PROCEDURE" statement followed by the name of the stored procedure and the "BEGIN" keyword which indicates the start of the stored procedure's code block.

Inside the code block, the "START TRANSACTION" statement is used to begin a new transaction, which ensures that the update operation is performed atomically, meaning it will either complete in its entirety or not at all. The "UPDATE" statement modifies the data in the "Suppliers" table by setting the "Address" column to 'New Address' where the "SupplierID" is 1.

The "COMMIT" statement is used to commit the changes made during the transaction to the database.

Finally, the delimiter is reset to the semicolon using the "DELIMITER ;" statement. The stored procedure is then called using the "CALL" statement, which executes the stored procedure and performs the update operation.

This non-conflicting SQL query ensures that the semicolons within the stored procedure don't interfere with the CREATE PROCEDURE statement by temporarily changing the delimiter. It also demonstrates the use of transactions to ensure that the database is updated atomically.



4. **Fourth (Updating Email Address for a Pharmacist in the Database Using a Transaction):**

-- Set the delimiter to $$ so that semicolons within the stored procedure don't terminate the CREATE PROCEDURE statement

```
DELIMITER $$

-- Define the stored procedure
CREATE PROCEDURE example_transaction4()
BEGIN
    -- Begin a transaction
    START TRANSACTION;
    UPDATE Pharmacists SET Email_Address = 'newemail@example.com'
WHERE PharmacistID = 1;
    COMMIT;
END;
$$ -- End of stored procedure

-- Reset the delimiter to semicolon
DELIMITER ;

CALL example_transaction4();
```

Explanation:

This MySQL SQL query creates a stored procedure named "example_transaction4" that performs an update operation on the "Pharmacists" table. The update operation sets the "Email_Address" column to 'newemail@example.com' where the "PharmacistID" is 1.

To prevent semicolons within the stored procedure from terminating the CREATE PROCEDURE statement, the delimiter is temporarily set to "$$".

The CREATE PROCEDURE statement starts with the "CREATE PROCEDURE" statement followed by the name of the stored procedure and the "BEGIN" keyword, which indicates the start of the stored procedure's code block.

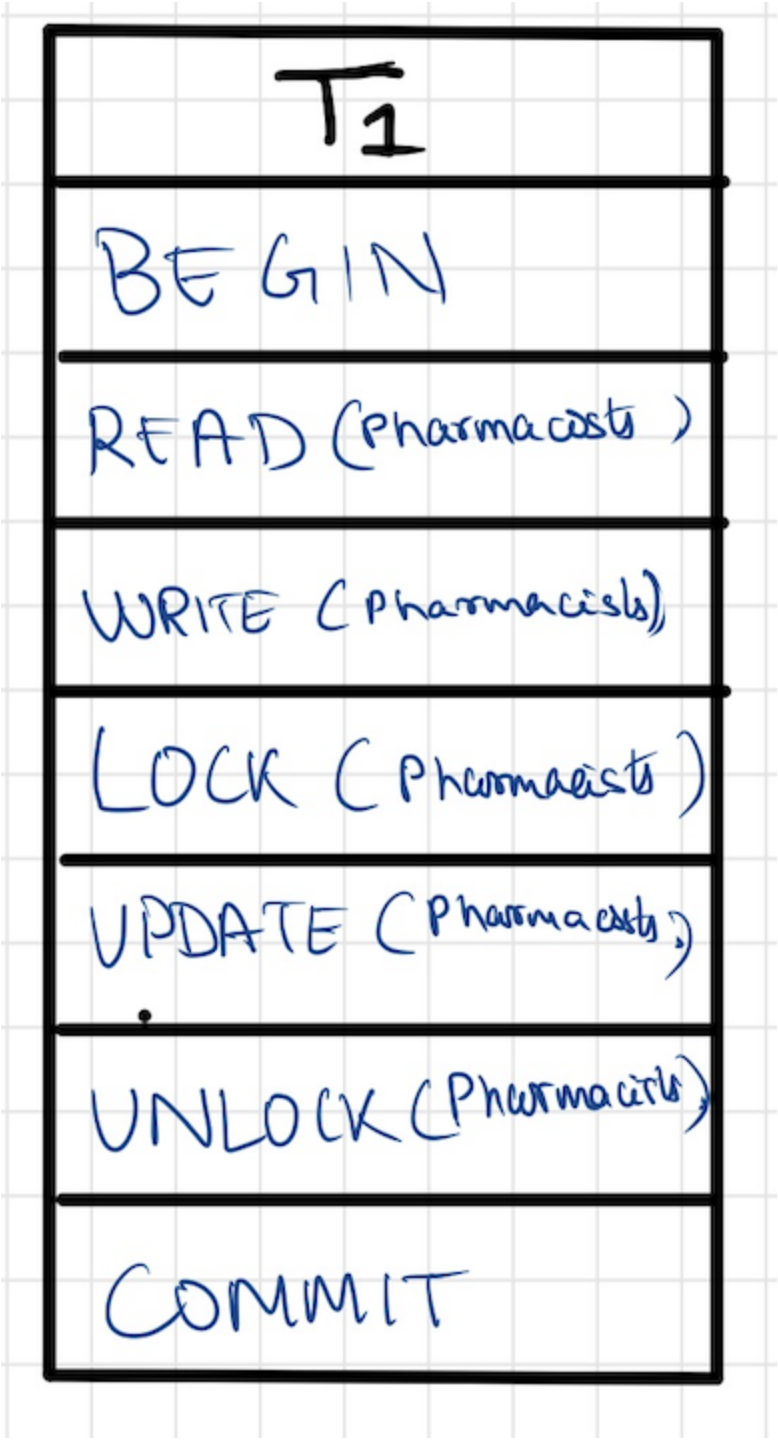
Inside the code block, the "START TRANSACTION" statement is used to start a new transaction, which ensures that the update operation is performed

atomically. The "UPDATE" statement modifies the data in the "Pharmacists" table by setting the "Email_Address" column to 'newemail@example.com' where the "PharmacistID" is 1.

The "COMMIT" statement is used to commit the changes made during the transaction to the database.

Finally, the delimiter is reset to semicolon using the "DELIMITER ;" statement. The stored procedure is then called using the "CALL" statement, which executes the stored procedure and performs the update operation.

This non-conflicting SQL query ensures that semicolons within the stored procedure don't interfere with the CREATE PROCEDURE statement by temporarily changing the delimiter. It also demonstrates the use of transactions to ensure that the database is updated atomically.



CONFLICTING QUERIES:

1. **First (Update stock and add a new order):**

-- Set the delimiter to $$ so that semicolons within the stored procedure don't terminate the CREATE PROCEDURE statement

```
DELIMITER $$

-- Define the stored procedure
CREATE PROCEDURE example_transaction5()
BEGIN
    -- Begin a transaction
   START TRANSACTION;

   -- Query 1: Update stock of medicine with ID=1 at branch ID=1
   UPDATE Stock SET Quantity = Quantity - 20 WHERE MedicineID = 1 AND
BranchID = 1;

   -- Query 2: Add a new order
   INSERT INTO Orders (OrderID,PatientID, MedicineID, PharmacistID, Quantity,
Date, DeliveryStatus)
    VALUES (102, 1, 2, 5,20, '2023-04-19', 'Pending');

   COMMIT;
END;
$$ -- End of stored procedure

-- Reset the delimiter to semicolon
DELIMITER ;

CALL example_transaction5();
```

2. **Second (Updating Stock and Adding New Order with Quantity Conflict):**

```
-- Set the delimiter to $$ so that semicolons within the stored procedure don't
terminate the CREATE PROCEDURE statement
DELIMITER $$
```

```
-- Define the stored procedure
CREATE PROCEDURE example_transaction6()
BEGIN
    -- Begin a transaction
   START TRANSACTION;

   -- Query 1: Update stock of medicine with ID=1 at branch ID=1
   UPDATE Stock SET Quantity = Quantity - 30 WHERE MedicineID = 1 AND
BranchID = 1;

   -- Query 2: Add a new order
   INSERT INTO Orders (OrderID,PatientID, MedicineID, PharmacistID, Quantity,
Date, DeliveryStatus)
   VALUES (102,  1, 2, 5,20, '2023-04-19', 'Pending');

   COMMIT;
END;
$$ -- End of stored procedure

-- Reset the delimiter to semicolon
DELIMITER ;

CALL example_transaction6();
```

Explanation For BOTH 1 and 2:

The first query creates a stored procedure named "example_transaction5" that performs two queries as part of a transaction. The first query updates the quantity of a medicine with ID=1 at branch ID=1 by subtracting 20 from the current quantity. The second query inserts a new order into the "Orders" table with OrderID=102, PatientID=1, MedicineID=2, PharmacistID=5, Quantity=20, Date='2023-04-19', and DeliveryStatus='Pending'.

The second query creates a stored procedure named "example_transaction6" that is similar to the first one but has a conflicting quantity in the second query. Specifically, the first query updates the quantity of the medicine with ID=1 at branch ID=1 by subtracting 30 from the current quantity. However, the second query tries to insert a new order with Quantity=20, which conflicts with the previous update query. This conflict can cause inconsistencies in the database.

To resolve this conflict, the stored procedure can either be modified to update the stock and add the new order with compatible quantities or to throw an error and roll back the transaction if the quantities conflict. This can be achieved using control flow statements such as IF-ELSE and EXCEPTION handling.

one possible solution could be to modify the second stored procedure to first check if the requested order quantity is less than or equal to the current stock quantity before updating the stock and adding the new order. If the requested quantity is greater than the current stock quantity, then the procedure can throw an error and roll back the transaction. Here's an example modification to the second stored procedure:
-- Set the delimiter to $$ so that semicolons within the stored procedure don't terminate the CREATE PROCEDURE statement

```
DELIMITER $$

-- Define the stored procedure
CREATE PROCEDURE example_transaction6()
BEGIN
    -- Begin a transaction
```

```
  START TRANSACTION;

  -- Query 1: Check if requested quantity is less than or equal to current stock quantity
  SELECT Quantity FROM Stock WHERE MedicineID = 1 AND BranchID = 1 INTO @current_quantity;
  IF @current_quantity >= 30 THEN
      -- Query 2: Update stock of medicine with ID=1 at branch ID=1
      UPDATE Stock SET Quantity = Quantity - 30 WHERE MedicineID = 1 AND BranchID = 1;

      -- Query 3: Add a new order
      INSERT INTO Orders (OrderID,PatientID, MedicineID, PharmacistID, Quantity, Date, DeliveryStatus)
      VALUES (102,  1, 2, 5,20, '2023-04-19', 'Pending');
  ELSE
      -- Requested quantity is greater than current stock quantity
      -- Roll back transaction and throw error
      ROLLBACK;
      SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Requested quantity is greater than current stock quantity';
  END IF;

  COMMIT;
END;
$$ -- End of stored procedure

-- Reset the delimiter to semicolon
DELIMITER ;

CALL example_transaction6();
```

This modification adds a SELECT statement to check the current stock quantity and assigns the result to a user-defined variable "@current_quantity". Then, an IF statement is used to check if the requested quantity is less than or equal to the current stock quantity. If the condition is true, then the stock is updated and a new order is added as before. If the condition is false, then the procedure rolls back the transaction and throws an error with a custom message.

T1	T2
READ (Stock)	
Quantity = Quantity -20	
WRITE(Stock)	
LOCK(Stock)	
	READ (Stock)
	Quantity = Quantity -30
	WRITE(Stock)
	LOCK(Stock)
READ (Orders)	
INSERT (Orders)	
WRITE(Orders)	
LOCK(Orders)	
	READ (Orders)
	INSERT (Orders)
	WRITE(Orders)
	LOCK(Orders)
COMMIT	
	COMMIT

3. **Third (Update stock of medicine and order delivery status.):**

```
-- Set the delimiter to $$ so that semicolons within the stored procedure don't terminate the CREATE PROCEDURE statement
DELIMITER $$
```

```
-- Define the stored procedure
CREATE PROCEDURE example_transaction7()
BEGIN
    -- Begin a transaction
   START TRANSACTION;

   -- Query 1: Update stock of medicine with ID=1 at branch ID=1
   UPDATE Stock SET Quantity = Quantity - 20 WHERE MedicineID = 1 AND
BranchID = 1;

   -- Query 2: Update a order
   UPDATE Orders SET Quantity = 20, DeliveryStatus = 'Delivered' WHERE
OrderID = 1;

   COMMIT;
END;
$$ -- End of stored procedure

-- Reset the delimiter to semicolon
DELIMITER ;

CALL example_transaction7();
```

4. **Fourth (conflicting transaction as both queries are trying to modify the same row in the Orders table):**

```
-- Set the delimiter to $$ so that semicolons within the stored procedure don't
terminate the CREATE PROCEDURE statement
DELIMITER $$

-- Define the stored procedure
CREATE PROCEDURE example_transaction8()
BEGIN
```

```
    -- Begin a transaction
   START TRANSACTION;

   -- Query 1: Update stock of medicine with ID=1 at branch ID=1
   UPDATE Stock SET Quantity = Quantity - 30 WHERE MedicineID = 1 AND
BranchID = 1;

   -- Query 2: Update a order
   UPDATE Orders SET Quantity = 30, DeliveryStatus = 'Delivered' WHERE
OrderID = 1;

   COMMIT;
END;
$$ -- End of stored procedure

-- Reset the delimiter to semicolon
DELIMITER ;

CALL example_transaction8();
```

Explanation For BOTH 3 and 4:

The third example_transaction7() is a non-conflicting SQL query. It begins a transaction, updates the stock of a medicine with ID=1 at branch ID=1, and updates an order with OrderID=1 to have a Quantity of 20 and a DeliveryStatus of "Delivered". Then, it commits the transaction. This is an example of a transaction that executes two queries that do not conflict with each other.

The fourth example_transaction8(), on the other hand, is an example of a conflicting transaction. It begins a transaction, updates the stock of a medicine with ID=1 at branch ID=1 by reducing the quantity by 30, and updates an order with OrderID=1 to have a Quantity of 30 and a DeliveryStatus of "Delivered". The

problem with this transaction is that both queries are trying to modify the same row in the Orders table. This can lead to data inconsistencies and incorrect results.

A solution to this conflicting transaction is to use a locking mechanism, such as the FOR UPDATE clause, to ensure that only one query can modify the row at a time. Here is an example of a modified transaction:

```
-- Define the stored procedure
CREATE PROCEDURE example_transaction8()
BEGIN
-- Begin a transaction
START TRANSACTION;

-- Query 1: Update stock of medicine with ID=1 at branch ID=1
UPDATE Stock SET Quantity = Quantity - 30 WHERE MedicineID
= 1 AND BranchID = 1;

-- Query 2: Update a order
SELECT * FROM Orders WHERE OrderID = 1 FOR UPDATE;
UPDATE Orders SET Quantity = 30, DeliveryStatus = 'Delivered'
WHERE OrderID = 1;

COMMIT;
END;
```

This command updates the stock quantity by 30 units and sets the order quantity and delivery status to 30 and 'Delivered', respectively, all within a single transaction. This ensures that

either all or none of the changes are applied, maintaining data consistency and avoiding conflicts.

T1	T2
READ (Stock)	
Quantity = Quantity -20	
UPDATE (Stock)	
LOCK (Stock)	
	READ (Stock)
	Quantity = Quantity -30
	UPDATE (Stock)
	LOCK (Stock)
READ (Orders)	
Quantity = 20	
UPDATE (Orders)	
LOCK (Orders)	
	READ (Orders)
	Quantity = 30
	UPDATE (Orders)
	LOCK (Orders)
COMMIT	
	COMMIT