Transactions

Transaction T1: Update product price and stock
Transaction T2: Buying product, Update product stock

Conflict-Serializable Schedule:

Transaction 1:
START TRANSACTION;
UPDATE product SET Price = 10, **– Write(A)**
Stock = Stock - 1 WHERE idProduct = 123; **– Write(B)**
COMMIT;

Transaction 2:
START TRANSACTION;
SELECT Stock INTO stock FROM product WHERE idProduct = 123; **– Read(B)**

IF stock <= 0 THEN
ROLLBACK;
ELSE
INSERT INTO transaction (idTransaction, Price, Status, date_time, Mode_of_payment) SELECT idProduct, Price, 1, NOW(), mode_of_payment FROM product WHERE idProduct = 123;
INSERT INTO order_details(Product_id, Order_id, Quantity) VALUES (idProduct, Order_id, quantity);
UPDATE product SET Stock = stock - 1 WHERE idProduct = 123; **– Write(B)**
COMMIT;
END IF;

—----------

Explanation:
Transaction 1 writes to data item 'a' and data item 'b' by decreasing the stock of product 123 and updating its price.

Meanwhile, Transaction 2 reads the stock of product 123 (data item 'a') and then writes to data item 'b' (decreasing the stock of product 123 by 1) and then inserts a new order in the order_details table (data item 'c').

Since both transactions access data item 'b' in a conflicting way (Transaction 1 writes to it and Transaction 2 reads from it and then writes to it), the schedule is not conflict serializable.

the precedence graph:
T1 ----> T2

T1	T2
W(a)	
	R(a,b)
W(b)	
COMMIT	
	W(b)
	COMMIT

Non-Conflict Serializable Schedule:

Transaction 1:

START TRANSACTION;
UPDATE product SET Price = 10, Stock = Stock - 1 WHERE idProduct = 123;**-- Write(A,B)**
COMMIT;

Transaction 2:

START TRANSACTION;
SELECT Stock INTO stock FROM product WHERE idProduct = 123 **FOR UPDATE; – Read(a,) putlock**
IF stock <= 0 THEN
ROLLBACK;
ELSE
INSERT INTO transaction (idTransaction, Price, Status, date_time, Mode_of_payment) SELECT idProduct, Price, 1, NOW(), mode_of_payment FROM product WHERE idProduct = 123;
INSERT INTO order_details(Product_id, Order_id, Quantity) VALUES (idProduct, Order_id, quantity);

UPDATE product SET Stock = stock - 1 WHERE idProduct = 123; **– Write(B)**

COMMIT; **— release lock**
END IF;

T1	T2
W(a)	
W(b)	
COMMIT	
	R(a,b)
	W(b)
	COMMIT

Explanation:
transaction 2 reads the stock of product 123 (data item 'a') and then writes to data item 'b' (decreasing the stock of product 123 by 1) and then inserts a new order in the order_details table (data item 'c').

Since transaction 2 performs the write operation after the insert operation, it means that the order details could be inserted into the order_details table before the stock of the product is decreased, which violates the consistency of the data. This order of operations could lead to incorrect information being stored in the database, as a customer could place an order for a product that is not in stock.

Therefore, this schedule is not conflict serializable as the operations of the transactions conflict with each other and could lead to incorrect results. To make this schedule conflict serializable, the operations of the transactions need to be reordered to ensure that the data remains consistent.
++

Transaction T1: decrease product stock , when some order got canceled
Transaction T2: Updates the bought_at field for a product, increase stock when order received

Conflict-Serializable Schedule:

T1: START TRANSACTION : Canceling order status when transaction Failed
T2: START TRANSACTION : checking status and updating stock
T2: SELECT Status_Dispatched FROM orders WHERE idorder = 1 **-- Read(A)**
SET Status_dispatch = “ canceled” **–Write(A)**
T1: SELECT Status_Dispatched FROM order WHERE order_id = 1 **-- Read(A)**

T1: IF Status_Dispatched==1 THEN
UPDATE product SET Stock = stock - 1 WHERE idProduct = 123; **– Write (B)**
COMMIT;
ELSE
ROLLBACK;
END IF;

T2: COMMIT

Explanation

T1	***T2***
	R(A)
	W(A)
R(A)	
W(B)	
COMMIT	
	COMMIT

“The reason for the conflict is that T1 read that just updated by T2 before its commit .”

Non-Conflict Serializable Schedule:

T1: START TRANSACTION : Canceling order status when transaction Failed
T2: START TRANSACTION : checking status and updating stock
T2: SELECT Status_Dispatched FROM orders WHERE idorder = 1 **FOR UPDATE -- Read(A)**
SET Status_dispatch = “ canceled” **–Write(A)**
T2: COMMIT
T1: SELECT Status_Dispatched FROM order WHERE order_id = 1 **-- Read(A)**

T1: IF Status_Dispatched==1 THEN
UPDATE product SET Stock = stock - 1 WHERE idProduct = 123; **– Write (B)**
COMMIT;
ELSE
ROLLBACK;
END IF;

T1	*T2*
	R(A)
	W(A)
R(A)	*COMMIT*
W(B)	
COMMIT	

==========

Transaction

if there is enough stock for a product, decrements the stock, inserts an order record, creates a transaction record, and then creates an order details record. If there is not enough stock, the transaction is rolled back.

```
  DELIMITER $$
START TRANSACTION;

-- Check if there is enough stock
SELECT Stock INTO @stock FROM product WHERE idProduct = 1 FOR UPDATE;
IF @stock < 5 THEN
   ROLLBACK;
ELSE
   -- Decrement the stock
   UPDATE product SET Stock = Stock - 5 WHERE idProduct =1;

   -- Insert the order details
   INSERT INTO order_details(Product_id,Order_id,Quantity)
   VALUES(1,210,5);

   -- Insert the transaction
   INSERT INTO `transaction` (Amount, Status, date_time, Mode_of_payment)
   VALUES (50, 1, NOW(), "CASH");

   -- Get the transaction ID
   SET @transaction_id = LAST_INSERT_ID();

   -- Insert the order
```

```
    INSERT INTO orders (Transaction_id, Product_id, Status_Dispatched, Status_Received, emp_id,
Order_date)
    VALUES (@transaction_id, 1, 0, 0, 0, NOW());

    COMMIT;
END IF;
END$$
DELIMITER ;
```

==========

we want to insert a new order for a particular customer, but first you need to check that the customer exists in the database. Here's the query:

```
BEGIN TRANSACTION;

DECLARE @customer_exists INT;

SELECT @customer_exists = COUNT(*)
FROM customers
WHERE customer_id = <customer_id>;

IF @customer_exists > 0 THEN
  INSERT INTO orders (order_date, order_quantity, product_id, customer_id)
  VALUES (<order_date>, <order_quantity>, <product_id>, <customer_id>);

  COMMIT TRANSACTION;
ELSE
  ROLLBACK TRANSACTION;
END IF;
```

==========

– If stock is less placing order to supplier

```
SELECT * FROM online_retail_shop.customer;
BEGIN TRANSACTION;
SELECT stock
FROM products
WHERE product_id = <product_id>;
IF stock < 10 THEN
```

```
INSERT INTO orders_to_supplier (Order_to_supplier_Os_ID,
Order_to_supplier_Transaction_idTransaction, Product_idProduct)
VALUES (<Id>,<TN_ID>,<pid>);
UPDATE products
SET stock = stock + <order_quantity>
WHERE product_id = <product_id>;
COMMIT TRANSACTION;
ELSE
ROLLBACK TRANSACTION;
END IF;
```

```
==========
– If user placed order more than per day limit then stopping him

BEGIN TRANSACTION;

SELECT customer_balance
FROM customers
WHERE idCustomer = <customer_id>;

IF <per day limit> - <transaction_amount> >= 0 THEN
  INSERT INTO transactions (customer_id, transaction_type, transaction_amount, transaction_date)
  VALUES (<customer_id>, '<transaction_type>', <transaction_amount>, '<transaction_date>');

  COMMIT TRANSACTION;
ELSE
  ROLLBACK TRANSACTION;
END IF;

==========
– If product is not selling then placing discount in that category

START TRANSACTION;
SELECT SUM(Stock)
FROM product
WHERE category_category_id = 5
FOR UPDATE;
IF @SUM > 1000 THEN
UPDATE product
SET Price = Price * 0.9
WHERE category_category_id = 5;
```

```
ELSE
ROLLBACK;
END IF;
COMMIT;
```