**Project Deadline - 6**

**Non-Conflicting Transactions:**

/\* Placing a direct order : When directly buying a product (not from the cart), the product should be added to order, the stock quantity updated, the order status updated. \*/

START TRANSACTION;

-- Insert a new order for a specified customer, directly buying a specified product

INSERT INTO orders (customer\_ID, status) VALUES (@customer\_ID, 'Processing');

-- Get the order ID for the newly created order

SET @new\_order\_id = LAST\_INSERT\_ID();

-- Add the purchased product to the order items with specified quantity

INSERT INTO order\_items (order\_ID, product\_ID, quantity, price) VALUES (@new\_order\_id, @product\_ID, @quantity, (SELECT price FROM product WHERE product\_ID = @product\_ID));

-- Update stock quantity for the purchased product

UPDATE product SET stockquantity = stockquantity - @quantity WHERE product\_ID = @product\_ID;

-- Change the order status to a specified status

UPDATE orders SET status = @order\_status WHERE order\_ID = @new\_order\_id;

COMMIT;

/\* Placing an order from cart : When placing an order from the cart, the cart should be emptied, the stock quantity updated, the order status updated. This should happen atomically. In case any step fails, the checkout fails. \*/

START TRANSACTION;

-- Customer places an order for multiple products in their cart

INSERT INTO orders (customer\_ID, status) VALUES (@customer\_ID, 'Processing');

-- Retrieve the newly created order ID

SET @new\_order\_id = LAST\_INSERT\_ID();

-- Transfer all cart items of the specified customer to order items

INSERT INTO order\_items (order\_ID, product\_ID, quantity, price)

SELECT @new\_order\_id, product\_ID, quantity, (SELECT price FROM product WHERE product.product\_ID = cart\_items.product\_ID)

FROM cart\_items

WHERE cart\_ID = (SELECT cart\_ID FROM cart WHERE customer\_ID = @customer\_ID);

-- Update stock quantities based on the ordered quantities

UPDATE product

SET stockquantity = stockquantity - (SELECT quantity FROM order\_items WHERE order\_items.product\_ID = product.product\_ID AND order\_ID = @new\_order\_id)

WHERE EXISTS (

SELECT 1 FROM cart\_items WHERE cart\_ID = (SELECT cart\_ID FROM cart WHERE customer\_ID = @customer\_ID) AND cart\_items.product\_ID = product.product\_ID

);

-- Empty the cart after placing the order

DELETE FROM cart\_items WHERE cart\_ID = (SELECT cart\_ID FROM cart WHERE customer\_ID = @customer\_ID);

-- Update the order status to Placed

UPDATE orders SET status = ‘Placed' WHERE order\_ID = @new\_order\_id;

COMMIT;

/\* Inserting a new customer: When inserting a new customer, a new cart should be created automatically. If this fails, then the new customer should not be added. \*/

START TRANSACTION;

-- Insert a new customer with parameterized details

INSERT INTO customer (fullname, phone\_no, email, address, passwd)

VALUES (@fullname, @phone\_no, @email, @address, @passwd);

-- Get the customer ID

SET @new\_customer\_id = LAST\_INSERT\_ID();

-- Create an empty cart for the new customer

INSERT INTO cart (customer\_ID) VALUES (@new\_customer\_id);

COMMIT;

COMMIT;

/\* Updating information: Editing customer information: Either all the information should be update, or none. If a customer is updating their address and email, both should get updated together, or neither should be updated. \*/

START TRANSACTION;

-- Update profile information for a specified customer

UPDATE customer

SET

fullname = @fullname,

phone\_no = @phone\_no,

email = @email,

address = @address,

passwd = @passwd

WHERE customer\_ID = @customer\_ID;

-- Commit the changes

COMMIT;

**Conflicting Transactions:**

import threading

import pymysql

import time

# Function to simulate purchasing a product

def buy\_product(customer\_id, product\_id, quantity, order\_status):

# Establishing the database connection

connection = pymysql.connect(host='localhost',

user='your\_username',

password='your\_password',

db='your\_database',

charset='utf8mb4',

cursorclass=pymysql.cursors.DictCursor)

try:

with connection.cursor() as cursor:

# Start transaction

cursor.execute("START TRANSACTION;")

print(f"Transaction started for Customer {customer\_id} buying Product {product\_id}")

# Insert a new order

cursor.execute(f"INSERT INTO orders (customer\_ID, status) VALUES ({customer\_id}, 'Processing');")

new\_order\_id = connection.insert\_id()

print(f"Order ID {new\_order\_id} created for Customer {customer\_id}")

# Add the purchased product to the order items with specified quantity

cursor.execute(f"SELECT price FROM product WHERE product\_ID = {product\_id};")

price = cursor.fetchone()['price']

cursor.execute(f"INSERT INTO order\_items (order\_ID, product\_ID, quantity, price) VALUES ({new\_order\_id}, {product\_id}, {quantity}, {price});")

print(f"Product {product\_id} added to Order {new\_order\_id}")

# Update stock quantity for the purchased product

cursor.execute(f"UPDATE product SET stockquantity = stockquantity - {quantity} WHERE product\_ID = {product\_id};")

print(f"Stock updated for Product {product\_id}")

# Change the order status

cursor.execute(f"UPDATE orders SET status = '{order\_status}' WHERE order\_ID = {new\_order\_id};")

print(f"Order {new\_order\_id} status changed to {order\_status}")

# Commit transaction

connection.commit()

print(f"Transaction committed for Customer {customer\_id} buying Product {product\_id}")

except Exception as e:

print(f"An error occurred: {str(e)}")

connection.rollback()

print("Transaction rolled back due to an error")

finally:

connection.close()

print("Database connection closed")

# Parameters for the transactions

customer\_id1 = 1

product\_id = 3 # Assuming both customers are trying to buy the same product

quantity = 1

order\_status = 'Completed'

# Creating threads

thread1 = threading.Thread(target=buy\_product, args=(customer\_id1, product\_id, quantity, order\_status))

thread2 = threading.Thread(target=buy\_product, args=(customer\_id1 + 1, product\_id, quantity, order\_status)) # Increment customer ID for simplicity

# Starting threads

thread1.start()

thread2.start()

# Adding a delay to make sure conflicts can occur

time.sleep(1)

# Joining threads to the main thread

thread1.join()

thread2.join()

# Function to simulate an admin updating the stock quantity of a product

def update\_stock(admin\_id, product\_id, new\_stock\_quantity):

# Establishing the database connection

connection = pymysql.connect(host='localhost',

user='your\_username',

password='your\_password',

db='your\_database',

charset='utf8mb4',

cursorclass=pymysql.cursors.DictCursor)

try:

with connection.cursor() as cursor:

# Start transaction

cursor.execute("START TRANSACTION;")

print(f"Transaction started for Admin {admin\_id} updating stock quantity of Product {product\_id}")

# Update stock quantity for the product

cursor.execute(f"UPDATE product SET stockquantity = {new\_stock\_quantity} WHERE product\_ID = {product\_id};")

print(f"Stock quantity updated for Product {product\_id}")

# Commit transaction

connection.commit()

print(f"Transaction committed for Admin {admin\_id} updating stock quantity")

except Exception as e:

print(f"An error occurred: {str(e)}")

connection.rollback()

print("Transaction rolled back due to an error")

finally:

connection.close()

print("Database connection closed")

# Parameters for the transactions

product\_id = 3 # Assuming both transactions involve Product 3

quantity = 1

order\_status = 'Completed'

new\_stock\_quantity = 10 # New stock quantity to be set by the admin

# Creating threads for the customer buying and admin updating stock

thread1 = threading.Thread(target=buy\_product, args=(1, product\_id, quantity, order\_status)) # Customer buying

thread2 = threading.Thread(target=update\_stock, args=(1, product\_id, new\_stock\_quantity)) # Admin updating stock

# Starting threads

thread1.start()

thread2.start()

# Joining threads to the main thread

thread1.join()

thread2.join()