Group 173: ApnaMart

User Guide:

ApnaMart:

Investigating Products:

- ApnaMart has a large selection of goods in many different categories, including electronics, home appliances, clothing, food, cosmetic products, and more.
- To discover the product they're looking for, customers can browse through the many categories and use filters like price range, brand, ratings, and more.
- Customers can look for a specific product by typing its name or associated keywords into the website's search bar at the top.

Login:

To access the website's backend, administrators must first log in with their own user ID and password.
Admins can manage products, alter prices, handle orders, and more once they are logged in.
Customers can log in with their user ID and password to access extra features like order tracking, purchase history viewing, address saving, and more.

Signup:

- You can register to create an account if you're not currently a customer.
- Just click the "Sign up" button in the top right corner of the page and enter the required information, including your name, email address, phone number, and password.
- You can begin adding things to your cart and checking out to place an order once you've registered an account.

After login as customers
Browsing options:
User can browse and add to cart the items he want to purchase.Then he/she needed to select a quantity.If the available quantity is more and he has that much amount in his/her wallet then the order will be placed.

Viewing vouchers:

By visiting the website's "Vouchers" area, customers can view the available vouchers.
The available vouchers are shown here together with the relevant terms and conditions, including the minimum order amount, the expiration date, and more.
If you have a coupon code, you can use it to get a discount at the checkout.

Manage Cart:

Click the 'Manage cartt' button when you're ready to finish your order and make payment.
On the checkout page, you can examine your order, add or delete goods, choose the delivery address, and select a payment option.
An email detailing the order's specifics will be sent to you once you have confirmed the order.

Customers can manage their cart by adding or removing items, updating the quantity of items, and viewing the total price of the order.
You can also save items for later, which will remove them from the cart but save them for future reference.
The cart will also display the estimated delivery date, the shipping cost, and the total price of the order.

Now ADMIN:
Executing OLAP Inquiries:

- Administrators can analyze information about sales, clients, and products using OLAP (Online Analytical Processing) queries.
- Making data-driven judgements about product offers, marketing tactics, and other matters is aided by this.

Queries Embedded:

-
- Admins can also access particular data about items, clients, and orders via embedded queries.
- This makes it easier to swiftly and effectively retrieve pertinent information.
- Update Product Information:

☐

Administrators

can change a product's price, description, photos, and other details.
This assists in maintaining the accuracy and currency of the product information.
Delete Customer:

A customer's account may be deleted from the website by admins.
This might be required in specific circumstances, such as when a consumer has transgressed the terms of service or committed fraud.

Transactions Used:
– If a Categories' sale is low then lowering amount in that category

```
START TRANSACTION;
SELECT SUM(Quantity) FROM product WHERE
idCategory = 5 FOR UPDATE;
IF @Quantity > 1000 THEN UPDATE product SET Price = Price * 0.95
WHERE category_category_id = 5;
ELSE ROLLBACK;
END IF;
COMMIT;
```

-IF a Product is running low on Stock, Placing order to Supplier and Updating Inventory

```
BEGIN TRANSACTION;
SELECT quantity FROM products WHERE product_id =id ;
IF quantity < 5 THEN INSERT INTO  Inventory_History ( idAdmin, idSupplier, idProduct,
quantity, date)
VALUES (idAdmin, idSupplier, idProduct, quantity, date)
UPDATE products SET quantity = quantity + Inventory_History.quantity
WHERE product_id = id;
COMMIT TRANSACTION;
ELSE ROLLBACK TRANSACTION;
END IF;
```

Schedule conflict serializable and non-conflict serializable:

Transaction 1 : add2Cart(id,cursor,choice, quantity): This will add items in the cart.

Transaction 2 :transaction_payment() : This will update the payment table,customer wallet,product quantity,order_statuses and empty the cart.

Read Write operations
1.The add2Cart transaction reads from the carts table, the products table, and the orders table. It writes to the carts table, the orders table, the order_items table.

2.The transaction_payment transaction reads from the orders table, the customers table, and the order_items table. It writes to the customers table, the payments table, the order_statuses table, and the products table.And then it will empty the cart by writing in it.

Conflicting operations:
1.Transaction 1 reads and then writes in cart table.Transaction 2 empties cart table after payment is done successfully by writing in it.So,transaction2 should not write before transaction 1.We will use lock so,that after transaction1 finishes writing then only transaction 2 can access it.
2.Transaction 2 uses total_amount value from orders table which is calculated and updated by transaction 1.So,it should not read while transaction 1 is still writing.Again we will use lock for this issue.

Non-Conflicting Schedule : Transaction 2 starts after transaction 1 commits.

```
def non_conflict_serializable_schedule():
    # Start transaction 1
    with conn:
        with conn.cursor() as cursor:
            add2Cart(1, cursor, "P1", 2)
            conn.commit()

    # Start transaction 2
    with conn:
        with conn.cursor() as cursor:
                    transaction_payment(1, 1, cursor)
                    conn.commit()
```

Conflicting Schedule : Here before transaction 1 writes the complete quantity in cart, transaction 2 starts reading,and ends up reading old value.Transaction 2 acquires a **row-level lock on the cart row using the FOR UPDATE clause** in the **SELECT statement,** which prevents transaction 1 from committing until transaction 2 completes.

```
def conflict_serializable_schedule():
    with conn:
        with conn.cursor() as cursor:
            add2Cart(1, cursor, "P1", 2)
            conn.commit()

    # Start transaction 2
    with conn:
        with conn.cursor() as cursor:
            query = "SELECT * FROM carts WHERE idCustomer=%s AND idProduct=%s
FOR UPDATE"
            cursor.execute(query, (1, "P1"))
            result = cursor.fetchone()
            if result:
                new_quantity = result[2] + 1
                query = "UPDATE carts SET quantity=%s WHERE idCustomer=%s AND
idProduct=%s"
                cursor.execute(query, (new_quantity, 1, "P1"))
                print(f"\n Quantity updated in cart.\n")
            else:
                query = "INSERT INTO carts (idCustomer, idProduct, quantity) VALUES (%s,
%s, %s)"
                cursor.execute(query, (1, "P1", 1))
                print(f"\n Added to cart.\n")

            query = "SELECT price FROM products WHERE idProduct=%s"
            cursor.execute(query, ("P1",))
            price = cursor.fetchone()[0]

            query = "SELECT MAX(idOrder) FROM orders"
            cursor.execute(query)
            last_order_id = cursor.fetchone()[0]

            total_amount = price * 1

            if last_order_id is None:
                new_order_id = 1
            else:
                new_order_id = last_order_id + 1
            query = "INSERT INTO orders (idOrder, idCustomer, totalAmount, orderDate,
deliveryDate) VALUES (%s, %s, %s, %s, %s)"
            cursor.execute(query, (new_order_id, 1, total_amount, datetime.datetime.now(),
datetime.datetime.now() + datetime.timedelta(days=7)))
            conn.commit()

            query = "INSERT INTO order_items (idOrder, idProduct, quantity, unitPrice)
VALUES (%s, %s, %s, %s)"
            cursor.execute(query, (new_order_id, "P1", 1, price))

            conn.commit()

    # Commit transaction 2
    with conn:
        with conn.cursor() as cursor:
            transaction_payment(1, 1, cursor)
            conn.commit()
```