E-commerce System Design

1. Problem Statement

Design an e-commerce system that manages products, orders, customers, and payments while addressing scalability, fault tolerance, and real-time updates.

---

2. System Requirements and Solutions

1. **Product Management**
   - *Solution*: The system should support the addition, updating, and removal of products, and track inventory for each product.
2. **Customer Management**
   - *Solution*: Customers should be able to create accounts, view their order history, and maintain personal details (e.g., shipping address, payment methods).
3. **Order Management**
   - *Solution*: The system should process customer orders, track order status, and handle returns and cancellations.
4. **Shopping Cart Management**
   - *Solution*: Customers should be able to add, remove, or update products in their shopping cart.
5. **Payment and Checkout**
   - *Solution*: The system should integrate with various payment gateways (e.g., PayPal, credit card processing) and securely process transactions.
6. **Inventory Management**
   - *Solution*: Track product stock levels and notify customers or admins when items are out of stock or running low.
7. **Discounts and Promotions**
   - *Solution*: Allow for flexible pricing mechanisms like discounts, promotions, and coupon codes.
8. **Product Search and Filtering**
   - *Solution*: Implement efficient search and filtering by product categories, price, rating, and availability.
9. **Recommendation Engine**
   - *Solution*: Provide personalized recommendations to customers based on their browsing history and past orders.

10. **Scalability**
    - *Solution*: The system should handle large traffic spikes during events like sales or holidays by using load balancing, caching, and database replication.

---

3. Key Classes, Functions, and Relationships

1. **Class:** `Product`
    - Represents a product with attributes like name, category, price, inventory, and discount information.
2. **Class:** `Customer`
    - Stores customer details, including name, email, address, and order history.
3. **Class:** `ShoppingCart`
    - Manages the current shopping session for a customer, including the list of products to be purchased.
4. **Class:** `Order`
    - Represents an order placed by a customer, including products, payment status, and delivery details.
5. **Class:** `Payment`
    - Handles payment processing and integration with payment gateways.
6. **Class:** `Inventory`
    - Manages stock levels of products, ensuring accurate data on available quantities.

---

4. Data Structures Used

- **Hash Map** (used in `Product` for product categories and inventory):
    - For fast lookup of products by category, price, or other attributes.
    - **Complexity**: O(1) for lookups and updates.
- **Vector** (used in `ShoppingCart` for storing product details):
    - Efficient for dynamic manipulation of products in the cart.
    - **Complexity**: O(n) for traversing and managing the cart.

---

5. High-Level Design

- **Product**: Manages product details like category, price, stock, and promotions.
- **Customer**: Stores customer-related information, including account details and order history.

- **ShoppingCart**: Maintains the current list of items a customer wishes to purchase.
- **Order**: Records order information, including purchased products and delivery status.
- **Payment**: Coordinates with payment gateways to handle transactions.
- **Inventory**: Tracks stock levels for each product, ensuring customers can't buy out-of-stock items.

---

6. Clarifications to Ask the Interviewer (with Solutions)

1. **What happens when an item is out of stock during checkout?**
   - *Solution*: Determine whether the item should be placed on backorder or removed from the order.
2. **Should the system support guest checkouts (without registration)?**
   - *Solution*: Decide if customers can make purchases without creating an account.
3. **How are returns and refunds handled?**
   - *Solution*: Clarify the refund policy and process for returned products.

---

7. Open-Ended Questions from the Interviewer

1. **How would you ensure that product recommendations are accurate and personalized?**
   - *Solution*: Use machine learning algorithms to suggest products based on past customer behavior, similar products, and collaborative filtering.
2. **How would you handle flash sales or large spikes in user traffic?**
   - *Solution*: Use caching (e.g., Redis), load balancers, and horizontal scaling for handling large volumes of traffic.

---

8. C++ Code Implementation

```cpp
#include <iostream>
#include <vector>
#include <unordered_map>
#include <string>

using namespace std;

// Product class to represent each item in the store
class Product {
public:
    string name;
    string category;
    double price;
    int stock;
    double discount;
    Product(string n, string c, double p, int s, double d = 0)
        : name(n), category(c), price(p), stock(s), discount(d) {}

    double getDiscountedPrice() {
        return price - (price * discount / 100);
    }
};

// Customer class to represent the user
class Customer {
public:
    string name;
    string email;
    string address;
    vector<string> orderHistory;

    Customer(string n, string e, string a) : name(n), email(e), address(a) {}

    void addOrderToHistory(string orderId) {
        orderHistory.push_back(orderId);
    }
};

// ShoppingCart class to manage cart operations
class ShoppingCart {
    unordered_map<string, pair<Product, int>> cartItems; // product name, (product object, quantity)
public:
    void addToCart(Product& product, int quantity) {
        if (product.stock >= quantity) {
            cartItems[product.name] = make_pair(product, quantity);
        } else {
            cout << "Product out of stock!\n";
        }
    }

    void removeFromCart(string productName) {
        if (cartItems.find(productName) != cartItems.end()) {
            cartItems.erase(productName);
        }
    }

    double calculateTotal() {
        double total = 0;
        for (auto& [name, item] : cartItems) {
            total += item.first.getDiscountedPrice() * item.second;
        }
        return total;
    }
};

// Order class to represent a customer order
class Order {
public:
    string orderId;
    Customer customer;
    vector<pair<Product, int>> products; // list of product and quantity
    string status;
    double totalAmount;

    Order(string id, Customer c) : orderId(id), customer(c), status("Processing")
{}

    void addProduct(Product& product, int quantity) {
        products.push_back(make_pair(product, quantity));
    }

    void completeOrder() {
        status = "Shipped";
        totalAmount = calculateTotal();
        customer.addOrderToHistory(orderId);
    }

    double calculateTotal() {
        double total = 0;
        for (auto& [product, quantity] : products) {
            total += product.getDiscountedPrice() * quantity;
        }
        return total;
    }
};

// Payment class to manage payment operations
class Payment {
public:
    bool processPayment(double amount) {
        // Simulate payment processing
        cout << "Payment of $" << amount << " processed successfully!\n";
        return true;
    }
};

// Inventory class to manage product stock levels
class Inventory {
    unordered_map<string, Product> products; // product name, product object

public:
    void addProduct(Product product) {
        products[product.name] = product;
    }

    Product* getProduct(string name) {
        if (products.find(name) != products.end()) {
            return &products[name];
        }
        return nullptr;
    }
    bool updateStock(string name, int quantity) {
        if (products.find(name) != products.end()) {
            products[name].stock -= quantity;
            return true;
        }
        return false;
    }
};

// Main function to simulate the e-commerce operations
int main() {
    // Create some products
    Product laptop("Laptop", "Electronics", 1200.99, 10, 10);  // 10% discount
    Product phone("Smartphone", "Electronics", 799.49, 15);

    // Create inventory and add products
    Inventory inventory;
    inventory.addProduct(laptop);
    inventory.addProduct(phone);

    // Create a customer
    Customer customer("John Doe", "john.doe@example.com", "123 Main St");

    // Shopping cart for customer
    ShoppingCart cart;
    cart.addToCart(laptop, 1);
    cart.addToCart(phone, 2);

    // Checkout process
    double totalAmount = cart.calculateTotal();
    cout << "Total Amount: $" << totalAmount << endl;

    // Process payment
    Payment payment;
    if (payment.processPayment(totalAmount)) {
        // Create order and complete
        Order order("ORD12345", customer);
        order.addProduct(laptop, 1);
        order.addProduct(phone, 2);
        order.completeOrder();

        // Update inventory stock
        inventory.updateStock("Laptop", 1);
        inventory.updateStock("Smartphone", 2);
    }

    return 0;
}
```

Explanation of Code

- **Product Class**: Represents product details including price, discount, and stock.

- **Customer Class**: Holds customer information and order history.
- **ShoppingCart Class**: Manages items in the cart and calculates total amounts.
- **Order Class**: Manages customer orders, tracks order

This C++ e-commerce system design allows for product management, customer registration, order processing, shopping cart functionality, payment handling, and inventory management. It uses key classes such as `Product`, `Customer`, `Order`, `ShoppingCart`, and `Payment` to implement essential e-commerce operations. The system supports operations like adding products to a cart, calculating total costs, processing payments, updating inventory, and storing customer order history.

To handle scalability, this design can incorporate microservices, load balancing, and caching for large-scale e-commerce operations.