BLG317E Database Systems
E-Commerce Website Documentation

Group Information

Yusuf Karamustafa 150210319

Emir Arda Eker 150220331

Project Description

In this project, we designed and implemented a robust database system tailored for an E-Commerce website. The primary goal was to consolidate all core operations—such as customer transactions, inventory management, and order fulfillment—into a single, efficient system. This streamlined approach improves operational efficiency by enabling quick data retrieval and manipulation, ensuring a seamless experience for both customers and administrators.

Tables

Table Name: User

- user_id (primary key, INT, auto increment)
- name (VARCHAR(100), Not Null)
- email (VARCHAR(100), Not Null, Unique)
- phone_number(INT, Unique)
- role (VARCHAR(100), Default = “user”)

The User table represents the users of the website and has information related to them.

Table Name: Address

- user_id (foreign key, INT, Not Null)
- address_id (primary key, INT, auto increment)
- country (VARCHAR(100), Not Null)
- city (VARCHAR(100), Not Null)
- zip_code (INT, Not Null)

The Address table has addresses saved by the users in the website.

Table Name: Order

- order_id (primary key, INT, auto increment)
- user_id (foreign key, INT, Not Null)
- product_manufacturer_id (foreign key, INT, Not Null)
- order_quantity (INT, Not Null, CHECK > 0)
- order_date (DATE, Not Null)
- status (VARCHAR(50), Default = “Pending”)

In this design an order can only contain a single type of product.

Table Name: Product

- product_id (primary key, INT, auto increment)
- name (VARCHAR(100), Not Null)
- description (VARCHAR(255)
- rating (FLOAT, Default = 0, CHECK between 0 and 5)

The Product table contains information about the products in the website, it has various information about the product itself.

Table Name: Manufacturer

- manufacturer_id (primary key, INT, auto increment)
- name (VARCHAR(100), Not Null)
- rating (FLOAT, Default = 0, CHECK between 0 and 5)

The Manufacturer table contains information about manufacturers of the products on the website, it is connected to products via junction table ProductManufacturer.

Table Name: ProductManufacturer

- product_manufacturer_id (primary_key, INT, auto increment)
- product_id (foreign key, INT, Not Null)
- manufacturer_id (foreign key, INT, Not Null)
- price (FLOAT, Not Null, CHECK > 0)
- stock (INT, Default = 0)

The Productmanufacturer table establishes the connection between products and their manufacturer, it also contains the price and stock information since every manufacturer can have different pricings and amount of stocks. This table is necessary to represent the many-to-many relationship between manufacturer and product.

Table Name: Cart

- cart_id (primary key, INT, auto increment)
- product_manufacturer_id (foreign key, INT, Not Null)
- user_id (foreign key, INT, Not Null)

The Cart table contains users and the products in their carts. It can have repeating product_manufacturer_ids and user_ids. It has two foreign keys product_manufacturer_id and user_id pointing to their tables.

Table Name: Payment

- payment_id (primary_key, INT, auto increment)
- order_id (foreign key, INT, Not Null)
- payment_date (DATE, Not Null)
- amount_paid (FLOAT, Not Null, CHECK > 0)
- payment_method (VARCHAR(50), Not Null)

The Payment table contains the payment information about an order. It has order_id as a foreign key.

Table Name: Review

- review_id (primary, INT, auto increment)
- user_id (foreign key, INT, Not Null)
- product_id (foreign key, INT, Not Null)
- rating (INT, Not Null, CHECK between 0 and 5)
- review_text (VARCHAR(100))
- review_date (DATE, Not Null)

The Review table contains the reviews made by users about products. It has two foreign keys pointing to the User and the Product tables.

Table Name: Shipping

- shipping_id (primary_key, INT, auto increment)
- order_id (foreign_key, INT, Not Null)
- address_id (foreign_key, INT, Not Null)
- shipping_date (DATE)
- estimated_delivery (DATE)
- status (VARCHAR(100), Not Null)

The Shipping table contains the information about an order's shipment and its address. It has two foreign keys order_id and address_id which point to the Order and the Address tables.

Relationships

1. **User -> Address:**
- **Type:** One-to-Many, a user can have multiple addresses.
- **Cardinality:** (1, N) for address (N addresses per user), (1, 1) for user (an address can have only one user).
- **Modality:** Mandatory for address (each address must belong to a user), optional for user (a user might not have an address saved).

2. **User -> Order:**
- **Type:** One-to-many, a user can have multiple orders.
- **Cardinality:** (1, N) for order (N orders per user), (1, 1) for user (an order can have only one user).
- **Modality:** Mandatory for order (each order needs to have a user), optional for user.

3. **User -> Review:**
- **Type:** One-to-many, a user can have multiple reviews.
- **Cardinality:** (1, N) for review, (1, 1) for user.
- **Modality:** Mandatory for review, optional for user.

4. **User -> Carts:**
- **Type:** One-to-one, a user can have only one shopping cart.
- **Cardinality:** (1, 1) a user has only one cart and a cart has only one user.
- **Modality:** Mandatory for carts, optional for users.

5. **ProductManufacturer -> Cart:**
- **Type:** Many-to-many, a product can appear in multiple carts and a cart can have multiple products.
- **Cardinality:** (N, N).
- **Modality:** Mandatory for cart (a cart must have products in it), optional for product (a product might not be in any cart).

6. **Product -> Review:**
- **Type:** One-to-many, a product might have multiple reviews but a review has only one product.
- **Cardinality:** (1, N) a product can have N reviews.
- **Modality:** Mandatory for review (a review must have a product), optional for product (a product might not have any review).

7. **ProductManufacturer -> Order:**
- **Type:** One-to-many, an order has only one product but a product can be in multiple orders.
- **Cardinality:** (1, N) a product can be in N orders.
- **Modality:** Mandatory for order (an order must have a product), optional for product (a product might not be in any orders).

8. **Order -> Payment:**
- **Type:** One-to-one, an order can have a single payment and a payment can have a single order.
- **Cardinality:** (1, 1)
- **Modality:** Mandatory for both (an order must have a payment and a payment must be done for an order).

9. **Order -> Shipping:**
- **Type:** One-to-one, an order can have a single shipment and a shipment contains a single order.
- **Cardinality:** (1,1)
- **Modality:** Mandatory for both.

10. **Product -> ProductManufacturer and Manufacturer -> ProductManufacturer:**
- **Type:** Many-to-many, a product can have multiple manufacturers and a manufacturer can have multiple products.
- **Cardinality:** (N, N)
- **Modality:** Optional for both, a product can have no manufacturers (no stock), and a manufacturer can have no products at any given time.

Example Queries

Names of the manufacturer(s) that have the most products listed:

```
SELECT M.name FROM Manufacturer M

JOIN ProductManufacturer PM ON M.manufacturer_id = PM.manufacturer_id

GROUP BY M.manufacturer_id, M.name

HAVING COUNT(PM.product_id) = (

        SELECT MAX(product_count) FROM (

                SELECT COUNT(PM2.product_id) AS product_count

                FROM ProductManufacturer PM2

                GROUP BY PM2.manufacturer_id

                ) AS subquery );
```

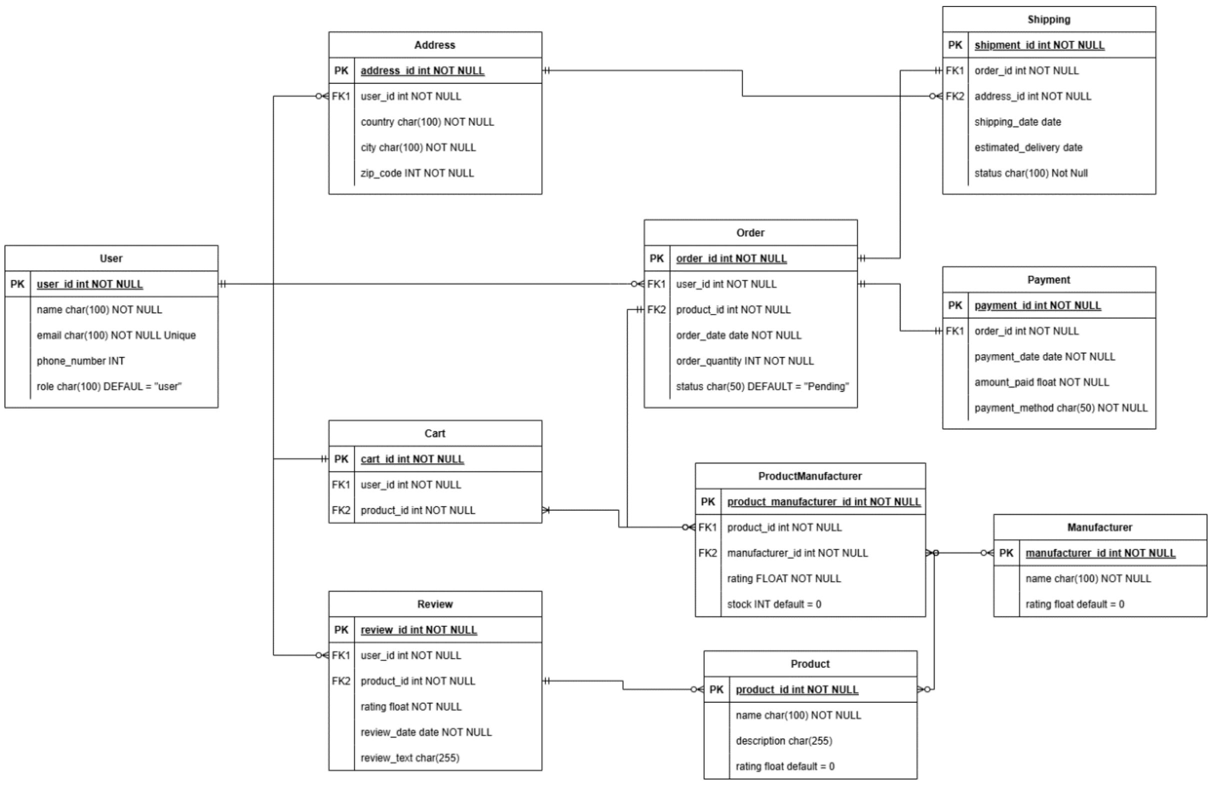
Explanation: The subquery calculates the product count for each manufacturer, the max function selects the maximum number. And then the main query filters manufacturers with equal number.

List the best-selling product for each manufacturer based on total sales quantity.

```
WITH ProductSales AS (
  SELECT
    pm.manufacturer_id,
    pm.product_id,
    SUM(o.order_quantity) AS total_sales
  FROM Order o
  JOIN ProductManufacturer pm ON o.product_manufacturer_id = pm.product_manufacturer_id
  GROUP BY pm.manufacturer_id, pm.product_id
),
RankedSales AS (
  SELECT
    ps.manufacturer_id,
    ps.product_id,
    ps.total_sales,
    RANK() OVER (PARTITION BY ps.manufacturer_id ORDER BY ps.total_sales DESC) AS rank
  FROM ProductSales ps
)
SELECT
  rs.manufacturer_id,
  rs.product_id,
  rs.total_sales
FROM RankedSales rs
WHERE rs.rank = 1;
```

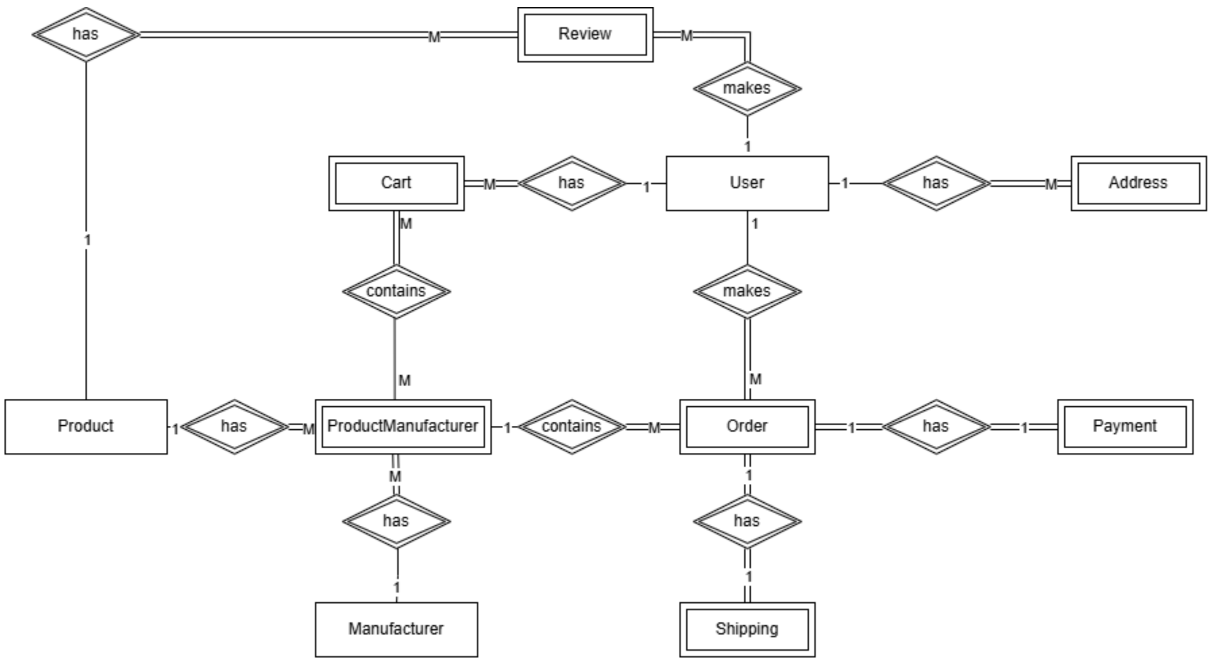
Explanation: The query calculates total sales for each product-manufacturer pair using a CTE (ProductSales). A second CTE (RankedSales) ranks products within each manufacturer based on total sales using RANK(). The final query selects the top product (rank = 1) for each manufacturer.

Data Model



Address
PK address_id int NOT NULL
FK1 user_id int NOT NULL
country char(100) NOT NULL
city char(100) NOT NULL
zip_code INT NOT NULL
Shipping
PK shipment_id int NOT NULL
FK1 order_id int NOT NULL
FK2 address_id int NOT NULL
shipping_date date
estimated_delivery date
status char(100) Not Null
User
PK user_id int NOT NULL
name char(100) NOT NULL
email char(100) NOT NULL Unique
phone_number INT
role char(100) DEFAUL = "user"
Order
PK order_id int NOT NULL
FK1 user_id int NOT NULL
FK2 product_id int NOT NULL
order_date date NOT NULL
order_quantity INT NOT NULL
status char(50) DEFAULT = "Pending"
Payment
PK payment_id int NOT NULL
FK1 order_id int NOT NULL
payment_date date NOT NULL
amount_paid float NOT NULL
payment_method char(50) NOT NULL
Cart
PK cart_id int NOT NULL
FK1 user_id int NOT NULL
FK2 product_id int NOT NULL
ProductManufacturer
PK product_manufacturer_id int NOT NULL
FK1 product_id int NOT NULL
FK2 manufacturer_id int NOT NULL
rating FLOAT NOT NULL
stock INT default = 0
Manufacturer
PK manufacturer_id int NOT NULL
name char(100) NOT NULL
rating float default = 0
Review
PK review_id int NOT NULL
FK1 user_id int NOT NULL
FK2 product_id int NOT NULL
rating float NOT NULL
review_date date NOT NULL
review_text char(255)
Product
PK product_id int NOT NULL
name char(100) NOT NULL
description char(255)
rating float default = 0

ER DIAGRAM



API and Swagger Documentation

The API for this project was constructed in Python using Flask and additional helper libraries. The Swagger documentation was developed using Flasgger library, as such the documentation is accessible using {link_to}/apidocs. All of the CRUD operations were handled in the API. All the endpoints are defined in their own .py files (e.g., order.py, product.py) and the documentation are stored in YAML files.

We used JWT tokens that are given to the user upon login, all the operations in the system require a valid JWT token.

There are also two user roles: admin, user. While users can only see their data and have limited interactions, an admin can see the data of all the users and have much wider access. Some actions are exclusive to the admin's such as retrieving the list of all users.

Example CRUD Operations

Create (POST)

- **Purpose:** Add new data to the database (e.g., create a new user, product, or order).
- **Example Endpoint:** /users (Create a New User)
  - **Method:** POST
  - **Request Parameters:**

    ```
    {
      "name": "John Johnson",
      "email": "john.johnson@example.com",
      "phone_number": "1234567890"
    }
    ```

  - **Response:**

    ```
    {
      "User Created Successfully."
    }
    ```

  - **Implementation Logic:**
    - Validate the input (e.g., check for email uniqueness).
    - Insert the new record into the User table.
    - Return a success message.

Read (GET)

- **Purpose:** Retrieve data from the database (e.g., fetch user details, list products, get order details).
- **Example Endpoint:** /users/<user_id> (Get User Details)
  - **Method:** GET
  - **Request Parameters:**
    - Path parameter: user_id (integer)
  - **Response:**

    ```
    {
      "user_id": 1,
      "name": "John Johnson",
      "email": "john.johnson@example.com",
      "phone_number": "1234567890"
    }
    ```

    - **Implementation Logic:**
        - Query the User table using the user_id.
        - If found, return the user details as JSON.
        - If not found, return a 404 Not Found error.

Update (PUT)

- **Purpose:** Modify existing data in the database (e.g., update user information, change order status).
- **Example Endpoint:** /orders/<order_id> (Update Order)
    - **Method:** PUT
    - **Request Parameters:**

      ```
      {

        "status": "Shipped"
      }
      ```

    - **Response:**

      ```
      {
        "Order successfully updated."
      }
      ```

    - **Implementation Logic:**
        - Validate that the order_id exists.
        - Update the status field in the Order table.
        - Return a success message.

Delete (DELETE)

- **Purpose:** Remove data from the database (e.g., delete a user, remove a product from the cart).
- **Example Endpoint:** /users/<user_id> (Delete a User)
    - **Method:** DELETE
    - **Request Parameters:**
        - Path parameter: user_id (integer, required)
    - **Response:**

      ```
      {
        "message": "User deleted successfully."
      }
      ```

    - **Implementation Logic:**
        - Check if the user_id exists.
        - If found, delete the user record.
        - Return a success message.
        - If not found, return a 404 Not Found error.

Other Example Documentation of Methods

Because the documentation is so large, approximately 70 pages when in pdf format, we will show a few examples as nearly all of them are similar.

Clear All Products from a Cart

- **Purpose:** Clear all the products in a user's cart.
- **Endpoint:** /cart/clear
- **HTTP Method:** DELETE
- **Structure of the Swagger API**:

```
Clear all items from the cart
---
tags:
  - Cart
security:
  - Bearer: []
responses:
  200:
    description: Cart cleared successfully
    schema:
      type: object
      properties:
        message:
          type: string
          example: "Cart cleared successfully"
  401:
    description: Unauthorized
  500:
    description: Internal server error
```

Get the Payment of an Order

- **Purpose:** Get the payment information of a certain order with it's ID
- **Endpoint:** /payment/order/<order_id>
- **HTTP Method:** GET
- **Structure of the Swagger API**:

```
Get payment details for an order
---
tags:
  - Payments
security:
  - Bearer: []
parameters:
  - name: order_id
    in: path
    type: integer
    required: true
    description: ID of the order to get payment details for
```

```yaml
responses:
 200:
  description: Payment details retrieved successfully
  schema:
   type: object

     properties:
       payment_id:
         type: integer
         example: 1
       order_id:
         type: integer
         example: 1
       payment_date:
         type: string
         format: date-time
         example: "2024-12-15T10:00:00Z"
       amount_paid:
         type: number
         format: float
         example: 199.99
       payment_method:
         type: string
         example: "Credit Card"
 403:
  description: Forbidden
  schema:
   type: object
   properties:
    message:
     type: string
     example: "You can only view your own payments"
 404:
  description: Payment not found for this order
  schema:
   type: object
   properties:
    message:
     type: string
     example: "Payment not found for this order"
 401:
  description: Unauthorized
 500:
  description: Internal server error
```

Challenges and Difficulties

There were many difficulties when working on the project:

1. **Database Design:**
- At the start of the project, coming up with logical tables was a bit difficult. Also adding a ProductManufacturer table and figuring its configuration was challenging.

2. **API Design and JWT Integration:**
- The most difficult part was designing the API and constructing the Swagger Documentation, this is because it was time consuming to figure out how JWT works and how to handle request and data inputs.

3. **Swagger Documentation:**
- Preparing the Swagger Documentation was the most time-consuming part by far, although most of the request are similar and were not hard to write, it was still a chore to write down the yml files for every single method in the API.