

**GTU Department of Computer Engineering**

**CSE 414 – Spring 2024**

**E-Commerce Company Database Design**

**Hasan Deliktaş**

**1901042625**

# Contents

<b>1. Introduction .....</b>	<b>3</b>
<b>2. User Requirements Outline .....</b>	<b>4</b>
<b>A. Data Storage Needs.....</b>	<b>4</b>
<b>B. Retrieval Capabilities .....</b>	<b>4</b>
<b>C. Specific Functionalities .....</b>	<b>4</b>
<b>3. E-R Diagram for the Database .....</b>	<b>5</b>
<b>4. Functional Dependencies &amp; Normalization.....</b>	<b>6</b>
<b>A. Functional Dependencies .....</b>	<b>6</b>
<b>B. Normalization .....</b>	<b>6</b>
<b>5. Database Schema &amp; Implementation .....</b>	<b>8</b>
<b>6. Query Development .....</b>	<b>13</b>
<b>7. Triggers .....</b>	<b>15</b>
<b>8. Views .....</b>	<b>20</b>
<b>9. Transactions &amp; Concurrency Control.....</b>	<b>22</b>
<b>10. Privileges and Roles .....</b>	<b>26</b>
<b>11. Inheritance .....</b>	<b>27</b>
<b>12. User Interface.....</b>	<b>27</b>
<b>A. Implementation of a query in User Interface.....</b>	<b>27</b>
<b>B. Implementation of a trigger in User Interface .....</b>	<b>27</b>
<b>C. Implementation of a view in User Interface .....</b>	<b>28</b>
<b>D. Implementation of a transaction in User Interface .....</b>	<b>29</b>

# 1. Introduction

The rapid growth of e-commerce opened the way for the development of robust database systems that can efficiently manage complex interactions and vast amounts of data involved in online retail operations. This report outlines the design of a comprehensive database system for an e-commerce company, addressing critical functionalities and the company's operations. The database design aims to create order processing, product management, customer relationship management, promotional activities, vendor coordination, branch operations, delivery logistics, and incident handling.

At the core of the database is the **Order Management** system, which handles the lifecycle of customer orders from creation to delivery. This system ensures accurate tracking of order details, such as order date, delivery date, status, and total price, while linking each order to specific products and customers. Integrated with the order management is the **Product Management** module, which maintains detailed information about the products, including their categorization and associations with vendors and promotional offers.

The **Customer Management** module is designed to store and manage customer profiles comprehensively, capturing essential information such as names, emails, addresses, and phone numbers. This module facilitates tracking customer orders and enables customers to leave reviews and ratings for products, contributing to a richer user experience.

Promotions and discounts play a crucial role in e-commerce, and the **Promotion Management** module efficiently handles these aspects by managing promotional details, discount rates, and validity periods, and applying them to relevant products. Additionally, the **Vendor Management** module maintains records of vendors, including their contact information, ensuring smooth supply chain operations.

Branch operations are managed through the **Branch Management** module, which stores information about various company branches and their associated delivery personnel and customer representatives. This module ensures that each branch operates efficiently, with delivery personnel and representative managed through the **Employee Management** module, which keeps track of personnel details and branch assignments.

The database also includes an **Incident Management** system to handle any issues related to orders, such as delays or product problems. This system records incident details, status, report dates, and resolution dates, facilitating prompt and effective resolution of issues.

Overall, this database design ensures managing an e-commerce business, providing the necessary infrastructure to support operational efficiency.

## 2. User Requirements Outline

### A. Data Storage Needs

- **Order Processing:** Store detailed information about orders, including order date, delivery date, status, total price, and associated customer and branch.
- **Product Information:** Store product details, including name, price, description, category, vendor, and any associated promotions.
- **Customer Information:** Maintain detailed customer profiles, capturing names, email addresses, physical addresses, and phone numbers.
- **Promotion Management:** Track promotions, including discount rates, start dates, and end dates, and link these promotions to relevant products.
- **Vendor Information:** Store vendor details, including name, address, and phone number, and link these vendors to the products they supply.
- **Branch Information:** Maintain branch details, including name, address, and phone number, and associate these branches with delivery personnel.
- **Employee Information:** Store information about employee, including their name and phone number, and link them to the branches they work for.
- **Incident Management:** Track incidents related to orders, including status, report date, resolved date, and description.
- **Product Reviews:** Allow customers to leave reviews and ratings for products.

### B. Retrieval Capabilities

- **Order Retrieval:** Ability to retrieve orders based on date, status, customer, branch, or product.
- **Product Retrieval:** Ability to retrieve product details by category, vendor, or promotion.
- **Customer Retrieval:** Ability to retrieve customer details and their order history.
- **Promotion Retrieval:** Ability to list active and past promotions and their associated products.
- **Vendor Retrieval:** Ability to retrieve vendor details and the products they supply.
- **Branch and Employee Retrieval:** Ability to retrieve branch details and associated employee.
- **Incident Retrieval:** Ability to retrieve incidents associated with specific orders.
- **Review Retrieval:** Ability to retrieve product reviews and ratings.

### C. Specific Functionalities

- **Order Management:** Processing and updating order statuses, handling order incidents.
- **Customer Interaction:** Allowing customers to place orders, leave reviews, and view their order history.
- **Promotion Management:** Applying and managing promotions on products.
- **Vendor Management:** Coordinating with vendors for product supply.
- **Branch Operations:** Managing branch details and employee assignments.

### 3. E-R Diagram for the Database

Here is the detailed E-R diagram of the e-commerce database design depicted in Figure 1. Diagram shows the relationships and attributes of the relations in the database. Relations and references are inspected in detail in section 4.

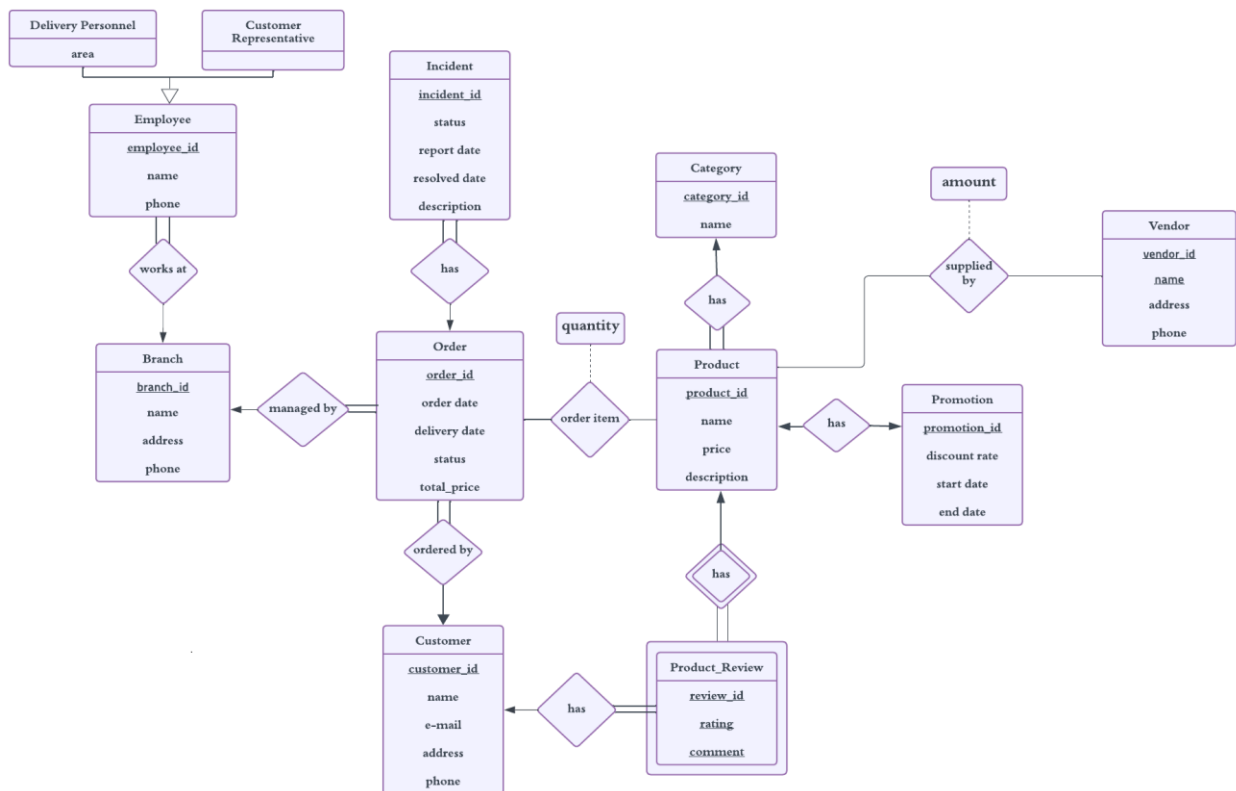


Figure 1: E-R Diagram of the database

## 4. Functional Dependencies & Normalization

### A. Functional Dependencies

Here is the list of functional dependencies for each relation in the database. There are 14 tables in the database and these dependencies comply with the BCNF rule. Because if there is a relation  $\alpha \rightarrow \beta$ ,  $\alpha$  is a super key for all the relations.

#### A. Order

a.  $order\_id \rightarrow order\_date, delivery\_date, status, total\_price, customer\_id, branch\_id$

#### B. Product

a.  $product\_id \rightarrow name, price, description, category\_id, promotion\_id$

#### C. Customer

a.  $customer\_id \rightarrow name, email, address, phone\_number$

#### D. Promotion

a.  $promotion\_id \rightarrow discount\_rate, start\_date, end\_date$

#### E. Vendor

a.  $vendor\_id \rightarrow name, address, phone$

#### F. Branch

a.  $branch\_id \rightarrow name, address, phone$

#### G. Employee

a.  $employee\_id \rightarrow name, phone, branch\_id$

##### i. Delivery Personnel

1.  $personnel\_id \rightarrow area, employee\_id$

##### ii. Customer Representative

1.  $csr\_id \rightarrow employee\_id$

#### H. Incident

a.  $incident\_id \rightarrow status, report\_date, resolved\_date, description, order\_id$

#### I. Product Review

a.  $review\_id \rightarrow rating, comment, customer\_id, product\_id$

#### J. Category

a.  $category\_id \rightarrow name$

#### K. VendorProduct

a.  $vendor\_id, product\_id \rightarrow amount$

#### L. OrderItem

a.  $product\_id, order\_id \rightarrow quantity$

### B. Normalization

Now let's think of two examples where we can say where we can build BCNF form that is applied in the database from decomposition of a hypothetical relation.

#### Example 1: ProductOrder

Initial table structure: ( $order\_id, order\_date, delivery\_date, status, total\_price, product\_id, product\_name, price, quantity$ )

In this scenario, **product\_id** and **order\_id** together form the primary key, meaning each combination of **product\_id** and **order\_id** must be unique.

However, let's say that **order\_id** determines status, meaning that for each **order\_id**, there's only one status. This breaks BCNF because **order\_id** is not a **superkey** for determining status. Therefore, **order\_id** does not fully determine status (i.e.,  $\text{order\_id} \rightarrow \text{status}$  does not hold where **order\_id** is a superkey).

To fix this, we'd need to decompose the **ProductOrder** table into three separate tables where each table satisfies BCNF. For instance, we would have an Order table with **order\_id** as the primary key and status as an attribute. Then, we would have a separate table **OrderItem** to associate products with orders, with **order\_id** and **product\_id** forming the primary key. This decomposition would ensure that each table satisfies BCNF.

(order\_id, order\_date, delivery\_date, status, total\_price)

(product\_id, product\_name, price)

(order\_id, product\_id, quantity)

### Example 2: VendorProduct

Initial table structure: (*vendor\_id, vendor\_name, address, phone, product\_id, product\_name, price, description, amount*)

In this relation, the primary key is {**vendor\_id, product\_id**}. However, neither **vendor\_id** nor **product\_id** alone can determine all other attributes. Hence, it fails to meet the BCNF requirements because there are non-trivial functional dependencies:

- $\text{vendor\_id} \rightarrow \text{vendor\_name}$  where **vendor\_id** is not a superkey.
- $\text{product\_id} \rightarrow \text{product\_name}$  where **product\_id** is not a superkey.

To normalize this relation into BCNF, we should decompose it into three separate relations where each table has a primary key that determines all other attributes.

(vendor\_id, vendor\_name, address, phone)

(vendor\_id, product\_id, amount)

(product\_id, product\_name, price, description)

## 5. Database Schema & Implementation

The e-commerce database is created in MySQL, here are all the tables in the database signifying the attributes and their domains. Moreover, primary keys and foreign keys are identified, **NOT NULL** constraint is implemented to make sure that these values cannot be null and lastly primary keys are based on the automatically incremented integer values. All the necessary SQL statements are in their sections, and the report shows the successful code. Figures 2, 3, 4 and 5 show the creation of the tables and Figure 6 shows the result.

```
-- Create database
CREATE DATABASE EcommerceDB;
USE EcommerceDB;

-- Table: Branch
CREATE TABLE Branch (
    branch_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    address TEXT,
    phone VARCHAR(15)
);

-- Table: Employee
CREATE TABLE Employee (
    employee_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    phone VARCHAR(15),
    branch_id INT,
    FOREIGN KEY (branch_id) REFERENCES Branch(branch_id)
);

-- Table: DeliveryPersonnel
CREATE TABLE DeliveryPersonnel (
    personnel_id INT AUTO_INCREMENT PRIMARY KEY,
    employee_id INT UNIQUE,
    delivery_area VARCHAR(255),
    FOREIGN KEY (personnel_id) REFERENCES Employee(employee_id)
);

-- Table: CustomerServiceRepresentative
CREATE TABLE CustomerServiceRepresentative (
    csr_id INT AUTO_INCREMENT PRIMARY KEY,
    employee_id INT,
    FOREIGN KEY (employee_id) REFERENCES Employee(employee_id)
);

-- Table: Customer
CREATE TABLE Customer (
    customer_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    address TEXT,
    phone_number VARCHAR(15)
);

-- Table: Category
CREATE TABLE Category (
    category_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);

-- Table: Promotion
CREATE TABLE Promotion (
    promotion_id INT AUTO_INCREMENT PRIMARY KEY,
    discount_rate DECIMAL(5, 2),
    start_date DATE,
    end_date DATE
);
```

Figure 2: Creation of the Tables



```

-- Table: Product
CREATE TABLE Product (
    product_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    price DECIMAL(10, 2),
    description TEXT,
    category_id INT,
    promotion_id INT,
    FOREIGN KEY (category_id) REFERENCES Category(category_id),
    FOREIGN KEY (promotion_id) REFERENCES Promotion(promotion_id)
);

-- Table: Vendor
CREATE TABLE Vendor (
    vendor_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    address TEXT,
    phone VARCHAR(15)
);

-- Associative Table: VendorProduct (for many-to-many relationship between Vendor and Product)
CREATE TABLE VendorProduct (
    vendor_id INT,
    product_id INT,
    amount DECIMAL(10, 2), -- New column for amount
    PRIMARY KEY (vendor_id, product_id),
    FOREIGN KEY (vendor_id) REFERENCES Vendor(vendor_id),
    FOREIGN KEY (product_id) REFERENCES Product(product_id)
);

```

Figure 3: Creation of the Tables

```

-- Table: Order
CREATE TABLE `Order` (
    order_id INT AUTO_INCREMENT PRIMARY KEY,
    order_date DATE NOT NULL,
    delivery_date DATE,
    status VARCHAR(50),
    total_price DECIMAL(10, 2),
    customer_id INT,
    branch_id INT,
    FOREIGN KEY (customer_id) REFERENCES Customer(customer_id),
    FOREIGN KEY (branch_id) REFERENCES Branch(branch_id)
);

-- Table: Incident
CREATE TABLE Incident (
    incident_id INT AUTO_INCREMENT PRIMARY KEY,
    status VARCHAR(50),
    report_date DATE,
    resolved_date DATE,
    description TEXT,
    order_id INT,
    FOREIGN KEY (order_id) REFERENCES `Order`(order_id)
);

```

Figure 4: Creation of the Tables

```

-- Table: Order Item
CREATE TABLE OrderItem (
    order_id INT,
    product_id INT,
    quantity INT,
    PRIMARY KEY (order_id, product_id),
    FOREIGN KEY (order_id) REFERENCES `Order`(order_id),
    FOREIGN KEY (product_id) REFERENCES Product(product_id)
);

-- Table: Product Review
CREATE TABLE ProductReview (
    review_id INT AUTO_INCREMENT PRIMARY KEY,
    rating INT,
    comment TEXT,
    customer_id INT,
    product_id INT,
    FOREIGN KEY (customer_id) REFERENCES Customer(customer_id),
    FOREIGN KEY (product_id) REFERENCES Product(product_id)
);

CREATE TABLE Notification (
    notification_id INT AUTO_INCREMENT PRIMARY KEY,
    message TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

```

Figure 5: Creation of the Tables

```

83 03:50:41 CREATE TABLE Branch ( branch_id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255) NOT NULL, address TEXT, ... 0 row(s) affected
84 03:50:41 CREATE TABLE Employee ( employee_id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255) NOT NULL, phone VARCH... 0 row(s) affected
85 03:50:41 CREATE TABLE DeliveryPersonnel ( personnel_id INT AUTO_INCREMENT PRIMARY KEY, employee_id INT UNIQUE, delivery_area... 0 row(s) affected
86 03:50:42 CREATE TABLE CustomerServiceRepresentative ( csr_id INT AUTO_INCREMENT PRIMARY KEY, employee_id INT, FOREIGN KEY... 0 row(s) affected
87 03:50:42 CREATE TABLE Customer ( customer_id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255) NOT NULL, email VARCH... 0 row(s) affected
88 03:50:42 CREATE TABLE Category ( category_id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255) NOT NULL) 0 row(s) affected
89 03:50:42 CREATE TABLE Promotion ( promotion_id INT AUTO_INCREMENT PRIMARY KEY, discount_rate DECIMAL(5, 2), start_date DATE, ... 0 row(s) affected
90 03:50:42 CREATE TABLE Product ( product_id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255) NOT NULL, price DECIMAL(1... 0 row(s) affected
91 03:50:42 CREATE TABLE Vendor ( vendor_id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255) NOT NULL, address TEXT, ... 0 row(s) affected
92 03:50:42 CREATE TABLE VendorProduct ( vendor_id INT, product_id INT, amount DECIMAL(10, 2), -- New column for amount PRIMARY KE... 0 row(s) affected
93 03:50:42 CREATE TABLE `Order` ( order_id INT AUTO_INCREMENT PRIMARY KEY, order_date DATE NOT NULL, delivery_date DATE, st... 0 row(s) affected
94 03:50:42 CREATE TABLE Incident ( incident_id INT AUTO_INCREMENT PRIMARY KEY, status VARCHAR(50), report_date DATE, resolve... 0 row(s) affected
95 03:50:42 CREATE TABLE OrderItem ( order_id INT, product_id INT, quantity INT, PRIMARY KEY (order_id, product_id), FOREIGN KEY (or... 0 row(s) affected
96 03:50:42 CREATE TABLE ProductReview ( review_id INT AUTO_INCREMENT PRIMARY KEY, rating INT, comment TEXT, customer_id IN... 0 row(s) affected

```

Figure 6: Results after the Creation

The following insertions are in order and **must be in order**, and they are representatives of the small data. They will show the query results and help to build triggers and concurrency control. Figure 7, 8, 9, 10 and 11 show the insertions and 12 shows the results after the insertions.

```

-- Insert into Branch
INSERT INTO Branch (name, address, phone) VALUES
('Central Branch', '123 Main St', '555-1234'),
('North Branch', '456 North St', '555-5678');

-- Insert into Employee table
INSERT INTO Employee (name, phone, branch_id) VALUES
('Alice Johnson', '555-1234', 1),
('Bob Smith', '555-5678', 2),
('Clark Kent', '555-1537', 1), -- Assuming branch_id 1 exists for Clark Kent
('Bob Dylan', '555-1628', 2); -- Assuming branch_id 2 exists for Bob Dylan

-- Insert into DeliveryPersonnel table
INSERT INTO DeliveryPersonnel (employee_id, delivery_area) VALUES
(1, 'Area A'),
(2, 'Area B');

-- Insert into CustomerServiceRepresentative table
INSERT INTO CustomerServiceRepresentative (employee_id) VALUES
(3), -- Assuming employee_id 3 corresponds to a customer service representative
(4); -- Assuming employee_id 4 corresponds to another customer service representative

```

Figure 7: Insertions into the tables

```

-- Insert into Customer
INSERT INTO Customer (name, email, address, phone_number) VALUES
('Alice Johnson', 'alice@example.com', '789 West St', '555-7890'),
('Bob Brown', 'bob@example.com', '101 East St', '555-1011');

-- Insert into Category
INSERT INTO Category (name) VALUES
('Electronics'),
('Clothing');

-- Insert into Promotion
INSERT INTO Promotion (discount_rate, start_date, end_date) VALUES
(10.00, '2024-01-01', '2024-06-30'),
(20.00, '2024-07-01', '2024-12-31');

-- Insert into Product
INSERT INTO Product (name, price, description, category_id, promotion_id) VALUES
('Smartphone', 699.99, 'Latest model smartphone', 1, 1),
('Laptop', 999.99, 'High performance laptop', 1, 2),
('T-shirt', 19.99, 'Comfortable cotton t-shirt', 2, NULL);

-- Insert into Vendor
INSERT INTO Vendor (name, address, phone) VALUES
('Tech Supplies Inc.', '200 Tech Park', '555-2000'),
('Clothing Co.', '300 Fashion Ave', '555-3000');

```

Figure 8: Insertions into the tables

```

-- Insert into VendorProduct
INSERT INTO VendorProduct (vendor_id, product_id, amount) VALUES
(1, 1, 30),
(1, 2, 20),
(2, 3, 10);

-- Insert into Order
INSERT INTO `Order` (order_date, delivery_date, status, total_price, customer_id, branch_id) VALUES
('2024-05-01', '2024-05-05', 'Delivered', 719.98, 1, 1),
('2024-06-01', '2024-06-05', 'Pending', 1019.98, 2, 2);

-- Insert into Incident
INSERT INTO Incident (status, report_date, resolved_date, description, order_id) VALUES
('Resolved', '2024-05-10', '2024-05-12', 'Late delivery', 1),
('Open', '2024-06-02', NULL, 'Damaged product', 2);

-- Insert into OrderItem
INSERT INTO OrderItem (order_id, product_id, quantity) VALUES
(1, 1, 1),
(1, 3, 1),
(2, 2, 1);

-- Insert into ProductReview
INSERT INTO ProductReview (rating, comment, customer_id, product_id) VALUES
(5, 'Excellent product!', 1, 1),
(3, 'Average quality', 1, 3),
(4, 'Good performance', 2, 2);

```

Figure 9: Insertions into the tables

Figure 10 and 11 shows some insertions into the tables to show outer join operations deliberately:

```

##### Insertions to show join operations
-- Insert an employee without a branch
INSERT INTO Employee (name, phone, branch_id) VALUES
('John Doe', '555-9999', NULL);

-- Insert a customer without any associated orders
INSERT INTO Customer (name, email, address, phone_number) VALUES
('John Smith', 'john@example.com', '123 Elm St', '555-1234');
#####

```

Figure 10: Inserting Employee without a branch and customer without an order

```

INSERT INTO Vendor (name, address, phone) VALUES
('Tech Gadgets Ltd.', '123 Innovation Blvd', '555-7890');

INSERT INTO Product (name, price, description, category_id, promotion_id) VALUES
('Wireless Earbuds', 39.99, 'High-quality wireless earbuds', 1, NULL);

```

Figure 11: Inserting Vendor without a product and product without a vendor

✓	98	03:54:14	INSERT INTO Branch (name, address, phone) VALUES ('Central Branch', '123 Main St', '555-1234'), ('North Branch', '456 North St', '555-5678')	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0
✓	99	03:54:14	INSERT INTO Employee (name, phone, branch_id) VALUES ('Alice Johnson', '555-1234', 1), ('Bob Smith', '555-5678', 2), ('Clark Kent', '555-153...', 3)	4 row(s) affected Records: 4 Duplicates: 0 Warnings: 0
✓	100	03:54:14	INSERT INTO DeliveryPersonnel (employee_id, delivery_area) VALUES (1, 'Area A'), (2, 'Area B')	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0
✓	101	03:54:14	INSERT INTO CustomerServiceRepresentative (employee_id) VALUES (3), -- Assuming employee_id 3 corresponds to a customer service repr...	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0
✓	102	03:54:14	INSERT INTO Customer (name, email, address, phone_number) VALUES ('Alice Johnson', 'alice@example.com', '789 West St', '555-7890'), ('B...	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0
✓	103	03:54:14	INSERT INTO Category (name) VALUES ('Electronics'), ('Clothing')	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0
✓	104	03:54:14	INSERT INTO Promotion (discount_rate, start_date, end_date) VALUES (10.00, '2024-01-01', '2024-06-30'), (20.00, '2024-07-01', '2024-12-31')	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0
✓	105	03:54:14	INSERT INTO Product (name, price, description, category_id, promotion_id) VALUES ('Smartphone', 699.99, 'Latest model smartphone', 1, 1), (...	3 row(s) affected Records: 3 Duplicates: 0 Warnings: 0
✓	106	03:54:14	INSERT INTO Vendor (name, address, phone) VALUES ('Tech Supplies Inc.', '200 Tech Park', '555-2000'), ('Clothing Co.', '300 Fashion Ave', '...	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0
✓	107	03:54:14	INSERT INTO VendorProduct (vendor_id, product_id, amount) VALUES (1, 1, 30), (1, 2, 20), (2, 3, 10)	3 row(s) affected Records: 3 Duplicates: 0 Warnings: 0
✓	108	03:54:14	INSERT INTO Order (order_date, delivery_date, status, total_price, customer_id, branch_id) VALUES ('2024-05-01', '2024-05-05', 'Delivered', ...	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0
✓	109	03:54:14	INSERT INTO Incident (status, report_date, resolved_date, description, order_id) VALUES ('Resolved', '2024-05-10', '2024-05-12', 'Late deliver...	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0
✓	110	03:54:14	INSERT INTO OrderItem (order_id, product_id, quantity) VALUES (1, 1, 1), (1, 3, 1), (2, 2, 1)	3 row(s) affected Records: 3 Duplicates: 0 Warnings: 0
✓	111	03:54:14	INSERT INTO ProductReview (rating, comment, customer_id, product_id) VALUES (5, 'Excellent product!', 1, 1), (3, 'Average quality', 1, 3), (4, ...	3 row(s) affected Records: 3 Duplicates: 0 Warnings: 0

Figure 12: Results after the insertions into the tables

## 6. Query Development

There are a total of 4 queries in this report, including left outer join, right outer join and full outer join operations. Query development is done firstly in MySQL then in user interface.

Here is the query for getting the review\_id, rating, comment, customer\_name, email, and product name from a specific customer where customer\_id matches the review, for a specific product, and the resulting table after the query. Inner join operations match the three tables based on customer and product IDs.

```
SELECT pr.review_id, pr.rating, pr.comment,
       c.name AS customer_name, c.email AS customer_email,
       p.name AS product_name
FROM ProductReview pr
JOIN Customer c ON pr.customer_id = c.customer_id
JOIN Product p ON pr.product_id = p.product_id
WHERE p.product_id = 1; -- Replace with the desired product_id
```

Figure 13: Get review information

	review_id	rating	comment	customer_name	customer_email	product_name
▶	1	5	Excellent product!	Alice Johnson	alice@example.com	Smartphone

Figure 14: Result of the query

The second query is right outer join operation in MySQL, resulting an employee without a branch when joining two tables on branch id:

```
SELECT Branch.name AS branch_name, Employee.name AS personnel_name
FROM Branch
RIGHT OUTER JOIN Employee ON Branch.branch_id = Employee.branch_id;
```

Figure 14: Right outer join operation

	branch_name	personnel_name
▶	Central Branch	Alice Johnson
	North Branch	Bob Smith
	Central Branch	Clark Kent
	North Branch	Bob Dylan
	NULL	John Doe

Figure 15: Resulting table

The third query is left outer join operation in MySQL, resulting a customer without an order when joining two tables on customer id:

```
SELECT Customer.name AS customer_name, `Order`.order_id
FROM Customer
LEFT OUTER JOIN `Order` ON Customer.customer_id = `Order`.customer_id;
```

Figure 16: Left outer join operation

	customer_name	order_id
▶	Alice Johnson	1
	Bob Brown	2
	John Smith	NULL

Figure 17: Resulting table

The last query example is full Outer Join, since we cannot get full outer join with MySQL directly, we combine two left joins and take union of them. Here is the code for getting both vendors and associative products even if they do not hold the corresponding values:

```

SELECT Vendor.name AS vendor_name, Product.name AS product_name
FROM Vendor
LEFT JOIN VendorProduct ON Vendor.vendor_id = VendorProduct.vendor_id
LEFT JOIN Product ON VendorProduct.product_id = Product.product_id
UNION
SELECT Vendor.name AS vendor_name, Product.name AS product_name
FROM Product
LEFT JOIN VendorProduct ON Product.product_id = VendorProduct.product_id
LEFT JOIN Vendor ON VendorProduct.vendor_id = Vendor.vendor_id;

```

Figure 18: Full outer join operation

	vendor_name	product_name
▶	Tech Supplies Inc.	Smartphone
	Tech Supplies Inc.	Laptop
	Clothing Co.	T-shirt
	New Vendor	NULL
	NULL	New Product

Figure 19: Resulting Table

## 7. Triggers

These triggers in the EcommerceDB database serve various critical functions to maintain data integrity and automate certain processes. The **Update Order Total Price** trigger ensures that the total price of an order accurately reflects any changes in the items included. Upon insertion of a new product review, the **Update Product Average Rating** trigger recalculates and updates the average rating of the corresponding product, providing valuable feedback for customers. **Notify Low Stock Products** triggers proactive alerts when stock levels for products fall below a predefined threshold, allowing for timely restocking. The **Delete Vendor Products** trigger maintains referential integrity by removing associated products when a vendor is deleted from the database. Additionally, the **Adjust Product Price on Promotion** trigger automatically adjusts product prices when promotions are applied, ensuring accurate pricing for customers. Finally, the **Adjust Product Price on Promotion End** trigger reverts product prices to their original values after a promotion ends, maintaining consistency and transparency in pricing. Together, these triggers contribute to the smooth operation and integrity of the EcommerceDB database.

Declaration of the **update\_order\_total\_price()** trigger in MySQL, it updates the total price of an order after insertion on order item table:

```

DELIMITER $$
CREATE TRIGGER update_order_total_price AFTER INSERT ON OrderItem
FOR EACH ROW
BEGIN
    UPDATE `Order`
    SET total_price = (SELECT SUM(quantity * price)
                      FROM OrderItem
                      JOIN Product ON OrderItem.product_id = Product.product_id
                      WHERE OrderItem.order_id = NEW.order_id)
    WHERE order_id = NEW.order_id;
END$$

```

Figure 20: Trigger for updating the total price after insert on order item

To check whether the trigger is working or not following queries are implemented to get the result:

```
SELECT total_price FROM `Order` WHERE order_id = 1;
```

Query 1: Get total price of the order which has the id 1

	total_price
▶	719.98

Table 1: Resulting Table

```

INSERT INTO OrderItem (order_id, product_id, quantity) VALUES (1, 2, 1);
SELECT total_price FROM `Order` WHERE order_id = 1;

```

Query 2: Insert an order item to the database and show the total price again

	total_price
▶	1719.97

Table 2: Resulting Table

The second trigger updates average rating after a product review is added:



```

DELIMITER $$
CREATE TRIGGER update_product_avg_rating AFTER INSERT ON ProductReview
FOR EACH ROW
BEGIN
    UPDATE Product
    SET average_rating = (SELECT AVG(rating)
                        FROM ProductReview
                        WHERE product_id = NEW.product_id)
    WHERE product_id = NEW.product_id;
END$$
DELIMITER ;

```

Figure 21: Update average rating after a product review insertion

To check whether the trigger is working or not following queries are implemented to get the result:

```

SELECT average_rating FROM Product
WHERE Product.product_id = 1

```

Query 1: Get average rating from a product

	average_rating
▶	5.00

Table 1: Resulting Table

```

INSERT INTO ProductReview (rating, comment, customer_id, product_id)
VALUES (4, 'Great product!', 1, 1); -- Assuming the product_id is 1 and customer_id is 1

SELECT average_rating FROM Product
WHERE Product.product_id = 1

```

Query 2: Add a new product review and check the product average rating again

	average_rating
▶	4.50

Table 2: Resulting Table

Declaration of the third trigger in MySQL, it adds a notification to the notification table if a vendor has less product, the logic is here states that vendor can't have stock that is lower than 20:

```

CREATE TRIGGER notify_low_stock_products AFTER INSERT ON VendorProduct
FOR EACH ROW
BEGIN
    DECLARE product_stock INT;
    SET product_stock = (SELECT amount FROM VendorProduct WHERE product_id = NEW.product_id AND vendor_id = NEW.vendor_id);
    IF product_stock < 20 THEN
        INSERT INTO Notification (message, created_at)
        VALUES (CONCAT('Low stock alert for product: ', NEW.product_id), NOW());
    END IF;
END$$

```

Figure 22: Notify the user if the stock is low for a particular vendor

To check whether the trigger is working or not following queries are implemented to get the result:

```

-- Insert into VendorProduct
INSERT INTO VendorProduct (vendor_id, product_id, amount) VALUES
(1, 1, 30),
(1, 2, 20),
(2, 3, 10);

```

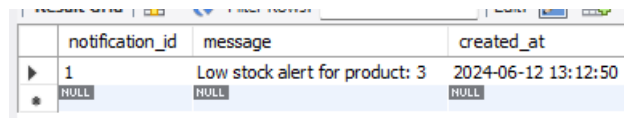
Query 1: Insertions to vendor product relationship set

```

SELECT * FROM Notification

```

Query 2: Check if any notification exists



notification_id	message	created_at
1	Low stock alert for product: 3	2024-06-12 13:12:50
NULL	NULL	NULL

Table 1: Here is the notification for the third insertion

The fourth trigger is responsible for handling promotion assignment for a product in MySQL, it regulates the price when a promotion is defined for that product:

```

CREATE TRIGGER update_product_price_on_promotion_assignment
BEFORE INSERT ON Product
FOR EACH ROW
BEGIN
    -- Update product price if a promotion is associated with the product
    IF NEW.promotion_id IS NOT NULL THEN
        SET NEW.price = NEW.price * (1 - (SELECT discount_rate FROM Promotion WHERE promotion_id = NEW.promotion_id) / 100);
    END IF;
END$$

```

Figure 23: Trigger that updates price if it has a promotion

To check whether the trigger is working or not following queries are implemented to get the result:

```
-- Insert into Promotion
INSERT INTO Promotion (discount_rate, start_date, end_date) VALUES
(10.00, '2024-01-01', '2024-06-30'),
(20.00, '2024-07-01', '2024-12-31');

-- Insert into Product
INSERT INTO Product (name, price, description, category_id, promotion_id, average_rating) VALUES
('Smartphone', 699.99, 'Latest model smartphone', 1, 1, NULL),
('Laptop', 999.99, 'High performance laptop', 1, 2, NULL),
('T-shirt', 19.99, 'Comfortable cotton t-shirt', 2, NULL, NULL);
```

Insertions: Here are the respective insertions for different products

```
SELECT * FROM PRODUCT
```

Query 1: Show the products

	product_id	name	price	description	category_id	promotion_id	average_rating
1	1	Smartphone	629.99	Latest model smartphone	1	1	5.00
2	2	Laptop	799.99	High performance laptop	1	2	4.00
3	3	T-shirt	19.99	Comfortable cotton t-shirt	2	NULL	3.00
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Table 1: Product prices changed after the insertion

The last trigger is responsible for deleting vendor and product associativity after a vendor is deleted in MySQL:

```
CREATE TRIGGER delete_vendor_products
BEFORE DELETE ON Vendor
FOR EACH ROW
BEGIN
    -- Delete records from VendorProduct table referencing the deleted vendor
    DELETE FROM VendorProduct WHERE vendor_id = OLD.vendor_id;
END$$
```

Figure 24: Deletion of a vendor-product relationship after a vendor deletion

To check whether the trigger is working or not following queries are implemented to get the result:

```
-- Check products linked to the vendor
• SELECT * FROM VendorProduct WHERE vendor_id = 1;
```

Query 1: Look for the vendors

	vendor_id	product_id	amount
▶	1	1	30.00
	1	2	20.00
✱	NULL	NULL	NULL

Table 1: Before deletion

```
-- Delete the vendor
DELETE FROM Vendor WHERE vendor_id = 1;

-- Check the product table to ensure the products linked to the vendor are deleted
SELECT * FROM VendorProduct
```

Query 2: Delete vendor and look for the relationship table

	vendor_id	product_id	amount
✎	S	3	10.00
✱	NULL	NULL	NULL

Table 2: There are no corresponding vendors

## 8. Views

- **OrderDetails View:** This view provides comprehensive details about orders within the system, including the order ID, date, status, customer name, and customer email. It's useful for various stakeholders such as customer service representatives, managers, and analysts who need to track and manage orders efficiently. By joining the Order table with the Customer table, it presents a consolidated view of order information along with customer details. This view enhances customer service operations, enables order tracking, and facilitates communication with customers regarding their orders.
- **ProductReviewsInfo View:** This view consolidates information about product reviews, including the review ID, rating, comment, and associated product name. It's valuable for product managers, marketers, and analysts who need insights into product performance and customer feedback. By joining the ProductReview table with the Product table, it provides a concise overview of product reviews alongside the corresponding product

names. This view aids in understanding customer sentiment, identifying product strengths and weaknesses, and guiding product improvement strategies.

- **OrdersByBranch View:** This view presents a summary of orders categorized by branch, displaying details such as order ID, date, status, and branch name. It's essential for branch managers, logistics teams, and executives who oversee operations across different branches. By joining the Order table with the Branch table, it offers visibility into order distribution among branches and helps monitor branch-specific performance. This view supports decision-making related to resource allocation, inventory management, and branch-level analysis.
- **ProductsByCategory View:** This view organizes products by category, providing information such as product ID, name, price, and category name. It's beneficial for merchandisers, inventory managers, and marketing teams who need to segment and manage products effectively. By joining the Product table with the Category table, it offers a structured view of products grouped by their respective categories. This view assists in product assortment planning, category management, and targeted marketing efforts.
- **TotalSalesByProduct View:** This view calculates the total sales amount for each product by aggregating the sales generated from orders. It includes details such as product ID, name, and total sales amount. It's crucial for sales analysts, finance teams, and business executives who monitor product performance and revenue generation. By joining the Product table with the OrderItem table and applying aggregation functions, it provides insights into product profitability and popularity. This view supports strategic decisions related to pricing, inventory stocking, and product promotion.

Here is the creation of the views in the MySQL database and the respective uses in various scenarios. Figures show the creation

```
CREATE VIEW OrderDetails AS
SELECT `Order`.order_id, `Order`.order_date, `Order`.status, Customer.name AS customer_name, Customer.email
FROM `Order`
JOIN Customer ON `Order`.customer_id = Customer.customer_id;

CREATE VIEW ProductReviewsInfo AS
SELECT ProductReview.review_id, ProductReview.rating, ProductReview.comment, Product.name AS product_name
FROM ProductReview
JOIN Product ON ProductReview.product_id = Product.product_id;
```

Figure 25: Order Details and Product Reviews Info Views

```

CREATE VIEW OrdersByBranch AS
SELECT `Order`.order_id, `Order`.order_date, `Order`.status, Branch.name AS branch_name
FROM `Order`
JOIN Branch ON `Order`.branch_id = Branch.branch_id;

CREATE VIEW ProductsByCategory AS
SELECT Product.product_id, Product.name, Product.price, Category.name AS category_name
FROM Product
JOIN Category ON Product.category_id = Category.category_id;

CREATE VIEW TotalSalesByProduct AS
SELECT Product.product_id, Product.name, SUM(OrderItem.quantity * Product.price) AS total_sales
FROM Product
JOIN OrderItem ON Product.product_id = OrderItem.product_id
GROUP BY Product.product_id, Product.name;

```

Figure 26: Order by Branch, Products by Category and Total Sales By Product Views

Here are two examples of using views:

```

SELECT review_id, rating, comment, product_name
FROM ProductReviewsInfo
WHERE product_name = 'Smartphone';

SELECT order_id, order_date, status, customer_name, email
FROM OrderDetails
WHERE order_date > '2024-05-15';

```

review_id	rating	comment	product_name	order_id	order_date	status	customer_name	email
1	5	Excellent product!	Smartphone	2	2024-06-01	Pending	Bob Brown	bob@example.com

## 9. Transactions & Concurrency Control

These transactions serve critical roles in maintaining data integrity and ensuring smooth operational flow within an e-commerce database. In the first transaction, a new order is inserted into the system along with its associated order items. This transaction begins with the assurance that all subsequent operations are part of a cohesive unit. By inserting the order and its items within a single transaction, data consistency is preserved, preventing partial updates or inconsistencies that could arise if operations were performed individually.

Similarly, the second transaction underlines the importance of atomicity by combining an update operation on the delivery date of an existing order with the insertion of an incident record documenting the change. This ensures that any alterations to existing data are carried out in a coordinated manner, reducing the risk of errors or discrepancies. By encapsulating these operations within a transaction, the system can guarantee that either all changes are successfully applied or none, thus safeguarding data integrity and providing a reliable audit trail for tracking updates and modifications. Overall, these transactions play pivotal roles in maintaining the reliability and consistency of an e-commerce database, ensuring smooth operations and accurate record-keeping.

First Transaction makes sure the order and order item tables are implemented together being **atomic**.

```
START TRANSACTION;
-- Insert into Order table
INSERT INTO `Order` (order_date, delivery_date, status, total_price, customer_id, branch_id)
VALUES ('2024-06-12', '2024-06-20', 'Pending', 100.00, 1, 1);
-- Insert into OrderItem table
INSERT INTO OrderItem (order_id, product_id, quantity)
VALUES (LAST_INSERT_ID(), 1, 2),
       (LAST_INSERT_ID(), 2, 1);
COMMIT;
```

Figure 27: The first transaction

919	13:55:57	INSERT INTO 'Order' (order_date, delivery_date, status, total_price, customer_id, branch_id) VALUES ('2024-06-12', '2024-06-20', 'Pending', N...	1 row(s) affected
920	13:55:57	INSERT INTO OrderItem (order_id, product_id, quantity) VALUES (LAST_INSERT_ID(), 1, 2), (LAST_INSERT_ID(), 2, 1)	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0
921	13:55:57	COMMIT	0 row(s) affected

Figure 28: The result of the first transaction

Second transaction updating incident table with the order delivery date change, commits together:

```
-- Update the delivery date of an order
UPDATE `Order`
SET delivery_date = '2024-06-25'
WHERE order_id = 1;

-- Insert an incident record for the change in delivery date
INSERT INTO Incident (status, report_date, description, order_id)
VALUES ('Change in Delivery Date', CURDATE(), 'The delivery date of the order has been updated to 2024-06-25.', 1);

-- Check if the update and insertion were successful
SELECT * FROM `Order` WHERE order_id = 1 AND delivery_date = '2024-06-25';
SELECT * FROM Incident WHERE order_id = 1;

COMMIT;
```

Figure 29: The second transaction

923	14:02:31	START TRANSACTION	0 row(s) affected
924	14:02:31	UPDATE 'Order' SET delivery_date = '2024-06-25' WHERE order_id = 1	1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0
925	14:02:31	INSERT INTO Incident (status, report_date, description, order_id) VALUES ('Change in Delivery Date', CURDATE(), 'The delivery date of the or...	1 row(s) affected
926	14:02:31	SELECT * FROM 'Order' WHERE order_id = 1 AND delivery_date = '2024-06-25' LIMIT 0, 1000	1 row(s) returned
927	14:02:31	SELECT * FROM Incident WHERE order_id = 1 LIMIT 0, 1000	2 row(s) returned
928	14:02:31	COMMIT	0 row(s) affected

Figure 30: The result of the second transaction

The last transaction ensures that when an employee is assigned it is a delivery personnel, the same logic can be implemented for customer representatives:

```
-- Start the transaction
START TRANSACTION;

-- Create the employee
INSERT INTO Employee (name, phone, branch_id) VALUES ('Arthur Solomon', '2312312', 1);
SET @employee_id := LAST_INSERT_ID();

-- Assign the employee as delivery personnel
INSERT INTO DeliveryPersonnel (employee_id, delivery_area) VALUES (@employee_id, 'Downtown');

-- Check for errors
-- If an error occurs, rollback the transaction
-- Otherwise, commit the transaction
COMMIT;
```

Figure 31: The last transaction

✓	24	18:43:10	START TRANSACTION	0 row(s) affected
✓	25	18:43:10	INSERT INTO Employee (name, phone, branch_id) VALUES ('Arthur Solomon', '2312312', 1)	1 row(s) affected
✓	26	18:43:10	SET @employee_id := LAST_INSERT_ID()	0 row(s) affected
✓	27	18:43:10	INSERT INTO DeliveryPersonnel (employee_id, delivery_area) VALUES (@employee_id, 'Downtown')	1 row(s) affected
✓	28	18:43:10	COMMIT	0 row(s) affected

Figure 32: The result of the last transaction

After researching, it is concluded that there are multiple ways to do concurrency control in the database excluding timestamp method, because tables are needed to be created again, there are 3 ways to ensure concurrency in the database; lock, isolation levels and optimistic concurrency control, here are the examples for each of them.

### Using Locks:

```
START TRANSACTION;

-- Acquire an exclusive lock on the order row
SELECT * FROM `Order` WHERE order_id = 1 FOR UPDATE;

-- Update the delivery date of the order
UPDATE `Order`
SET delivery_date = '2024-06-25'
WHERE order_id = 1;

-- Insert an incident record for the change in delivery date
INSERT INTO Incident (status, report_date, description, order_id)
VALUES ('Change in Delivery Date', CURDATE(), 'The delivery date of the order has been updated to 2024-06-25.', 1);

-- Check if the update and insertion were successful
SELECT * FROM `Order` WHERE order_id = 1 AND delivery_date = '2024-06-25';
SELECT * FROM Incident WHERE order_id = 1;

COMMIT;
```

Figure 33: Lock based concurrency control in the database



## Using isolation levels:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;

-- Update the delivery date of an order
UPDATE `Order`
SET delivery_date = '2024-06-25'
WHERE order_id = 1;

-- Insert an incident record for the change in delivery date
INSERT INTO Incident (status, report_date, description, order_id)
VALUES ('Change in Delivery Date', CURDATE(), 'The delivery date of the order has been updated to 2024-06-25.', 1);

-- Check if the update and insertion were successful
SELECT * FROM `Order` WHERE order_id = 1 AND delivery_date = '2024-06-25';
SELECT * FROM Incident WHERE order_id = 1;

COMMIT;
```

Figure 34: Isolation level-based concurrency control

## Using Optimistic Concurrency Control:

```
DELIMITER //
CREATE PROCEDURE UpdateOrderWithOptimisticConcurrency()
BEGIN
    DECLARE retrieved_delivery_date DATE;
    DECLARE retrieved_status VARCHAR(50);

    -- Step 1: Read the record
    SELECT delivery_date, status INTO retrieved_delivery_date, retrieved_status
    FROM `Order`
    WHERE order_id = 1;

    -- Step 2: Update the record
    UPDATE `Order`
    SET delivery_date = '2024-06-25', status = 'Updated'
    WHERE order_id = 1 AND delivery_date = retrieved_delivery_date AND status = retrieved_status;

    -- Step 3: Check for concurrency conflicts
    IF ROW_COUNT() = 0 THEN
        -- Handle concurrency conflict (e.g., raise an error, log, or notify the user)
        SELECT 'Concurrency conflict detected. Please retry the operation.' AS Message;
    ELSE
        -- Commit the transaction if no conflicts
        SELECT 'Order successfully updated.' AS Message;
    END IF;
END //
DELIMITER ;
CALL UpdateOrderWithOptimisticConcurrency();
```

Figure 35: Optimistic Concurrency Control

## 10. Privileges and Roles

To create users and roles in MySQL database the following syntax is used. There are two roles created, the first user is supervisor and the second is customer representative. Respectively supervisor can select everything as well as insertion to incident, meaning it can report something. While customer representatives can only select on the database and see the ongoing processes in e-commerce database

```
CREATE ROLE supervisor;  
GRANT SELECT ON ecommercedb.* TO supervisor;  
GRANT INSERT ON ecommercedb.incident TO supervisor;  
CREATE USER 'clark_kent' IDENTIFIED BY 'superman' DEFAULT ROLE 'supervisor';  
  
CREATE ROLE representative;  
GRANT SELECT ON ecommercedb.* TO representative;  
CREATE USER 'bob_dylan' IDENTIFIED BY 'musiclegend' DEFAULT ROLE 'representative';
```

Figure 35: Creating users and roles

Here is an example of using **customer service representative** on ecommerce database:

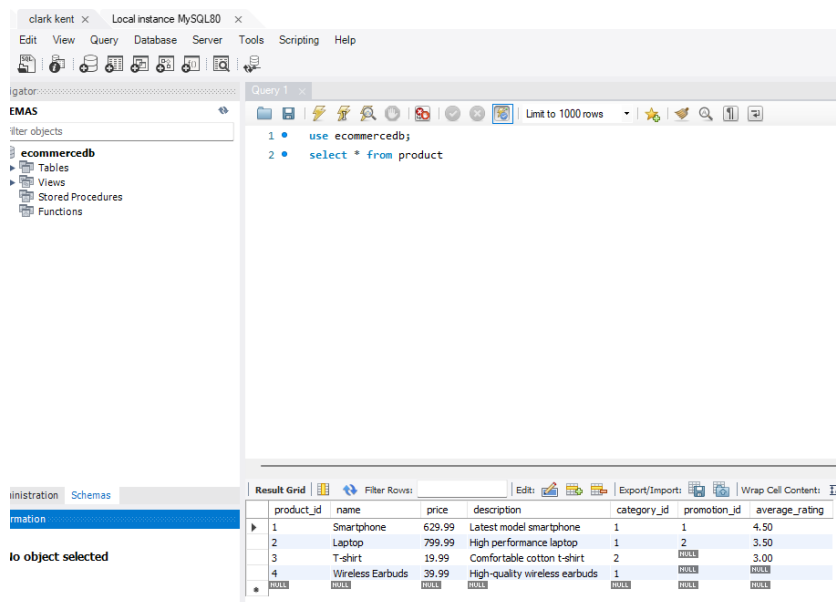
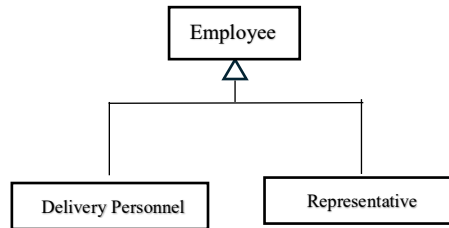


Figure 36: Using supervisor user in database

## 11. Inheritance

There is only one inheritance in the project, and it is built on the relationship between Delivery Personnel, Customer Service Representative and Employee objects. Inheritance is created through the ID of the employee object, and there is a joint inheritance in this time with a total participation, meaning that employee cannot be any other object than a delivery personnel or a representative.



## 12. User Interface

The user interface for this project does not necessarily cover all the aspects but it shows the basic functionalities and how they work in a simple design. The user interface is implemented as a Python project. It connects the database to the models in Django, but **all the tables are fetched with raw SQL queries**. There are two dependencies for this project **mysqlclient** and **Django** for Python.

### A. Implementation of a query in User Interface

In user interface a **left outer join** query is covered to show the results:

Customer Name	Email	Phone Number	Order ID	Status	Total Price
John Smith	john@example.com	555-1234	7	Pending	100.00
John Smith	john@example.com	555-1234	3	Pending	2059.97
John Smith	john@example.com	555-1234	1	Updated	649.98
Alice Johnson	alice@example.com	555-7890	2	Pending	799.99
Bob Brown	bob@example.com	555-1011	None	None	None

Figure 37: Table showing the left outer join query

### B. Implementation of a trigger in User Interface

In this user interface module, here is the initial product list in the database, a user can add reviews or can see the reviews as a basic user interface design, if user tries to add a review, he or she will be redirected to page in Figure 39

Product ID	Name	Price	Description	Average Rating	Reviews
1	Smartphone	\$629.99	Latest model smartphone	4.50	<a href="#">Add Review</a> <a href="#">View Reviews</a>
2	Laptop	\$799.99	High performance laptop	4.00	<a href="#">Add Review</a> <a href="#">View Reviews</a>
3	T-shirt	\$19.99	Comfortable cotton t-shirt	3.00	<a href="#">Add Review</a> <a href="#">View Reviews</a>
4	Wireless Earbuds	\$39.99	High-quality wireless earbuds	None	<a href="#">Add Review</a> <a href="#">View Reviews</a>

Figure 38: Table showing initial process list query

### Add Review for Product 2

**Rating:**

**Comment:**

It is good for daily usage.

Figure 39: Adding a review

If the user adds the review this way, the trigger **update\_product\_avg\_rating()** will be triggered in the database and the average rating will be changed as it can be seen below:

Product ID	Name	Price	Description	Average Rating	Reviews
1	Smartphone	\$629.99	Latest model smartphone	4.50	<a href="#">Add Review</a> <a href="#">View Reviews</a>
2	Laptop	\$799.99	High performance laptop	3.50	<a href="#">Add Review</a> <a href="#">View Reviews</a>
3	T-shirt	\$19.99	Comfortable cotton t-shirt	3.00	<a href="#">Add Review</a> <a href="#">View Reviews</a>
4	Wireless Earbuds	\$39.99	High-quality wireless earbuds	None	<a href="#">Add Review</a> <a href="#">View Reviews</a>

Figure 40: Product list after the review is added

## C. Implementation of a view in User Interface

Here is the product by category view in a simple user interface, it is classified based on the categories like in a regular website.

#### Electronics

Product ID	Name	Price
1	Smartphone	629.99
2	Laptop	799.99
4	Wireless Earbuds	39.99

#### Clothing

Product ID	Name	Price
3	T-shirt	19.99

Figure 41: Products by Category View in User Interface

### D. Implementation of a transaction in User Interface

Implementation of a transaction in User Interface is simple Django helps us to build such transactions, here we are assigning a delivery personnel, first it will assign it to employee table then it will assign it to the delivery personnel table. Figure 43 shows the transaction assignment in Django to mimic the MySQL behavior. Users will be redirected to success or failure pages.

### Add a Delivery Personnel

**Employee Name:**

**Phone Number:**

**Branch ID:**

**Delivery Area:**

Figure 42: Adding a delivery personnel

```

try:
    with transaction.atomic():
        # Start the transaction
        with connection.cursor() as cursor:
            # Create the employee
            cursor.execute("""
                INSERT INTO Employee (name, phone, branch_id)
                VALUES (%s, %s, %s)
            """, [employee_name, phone_number, branch_id])

            # Get the last inserted employee_id
            cursor.execute("SELECT LAST_INSERT_ID()")
            employee_id = cursor.fetchone()[0]

            # Assign the employee as delivery personnel
            cursor.execute("""
                INSERT INTO DeliveryPersonnel (employee_id, delivery_area)
                VALUES (%s, %s)
            """, [employee_id, delivery_area])

        # If no errors, commit the transaction
        return render(request, 'transaction_success.html')

except Exception as e:
    # If there's an error, rollback the transaction
    transaction.rollback()
    return render(request, 'transaction_error.html', {'error': str(e)})

```

Figure 43: Code showing how atomic transactions are handled in Django

- Video on how the database works: [Video](#)