Jaypee Institute of Information Technology, Sector - 62, Noida

B.Tech CSE III Semester

DBMS Lab PBL Report

Ecommerce Database Explorer

Submitted to

Dr. Indu Chawla

Dr. Parmeet Kaur Sodhi

Dr. Prateek Soni

Submitted by

Harsh Sharma	2401030232	B5
Karvy Singh	2401030234	B5
Rudra Kumar Singh	2401030237	B5

Letter of Transmittal

Dr. Indu Chawla
Dr. Parmeet Kaur Sodhi
Dr. Prateek Soni

Department of Computer Science & IT

Subject: Submission of Project "Ecommerce Database Explorer"

Respected Madam/Sir,

We are pleased to submit our project titled *"Ecommerce Database Explorer"* as part of our DBMS laboratory course. This report documents the design and implementation of a relational database for a simplified e–commerce platform, along with a web–based interface for exploring tables, routines, and database artifacts.

We have endeavored to cover the complete database life cycle starting from requirement analysis and conceptual design to logical schema, normalization, integrity constraints, and implementation of stored procedures, functions, and triggers using MySQL's procedural extensions (PL/SQL–style constructs). The main focus of the work is on database design quality, normal forms, and the use of relational algebra concepts such as joins, union, intersection, difference, and division, which are demonstrated through various stored routines.

We would like to thank you for your guidance and for providing us the opportunity to apply the theoretical concepts of DBMS to a practical e–commerce scenario.

Sincerely,

Harsh Sharma (2401030232)
Karvy Singh (2401030234)
Rudra Kumar Singh (2401030237)

Date: November 25, 2025

Contents

1 Introduction

Electronic commerce platforms are typical examples of data–intensive applications where database design plays a central role. They involve multiple interacting entities such as customers, sellers, products, orders, reviews, and pricing histories. The quality of the underlying relational schema directly affects correctness, performance, and extensibility of the overall system.

In this project, we designed and implemented an **Ecommerce Database Explorer** using **MySQL** as the relational database management system. On top of the database, we built a lightweight web interface using **FastAPI** and vanilla HTML/JavaScript to browse tables and to experiment with Data Definition Language (DDL), Data Manipulation Language (DML), and stored routines.

The primary emphasis of the project is on:

- Systematic relational database design.
- Achieving higher normal forms and eliminating redundancy.
- Implementing business logic using PL/SQL–style stored procedures, functions and triggers.
- Demonstrating relational algebra concepts such as joins, union, intersection, difference, division, and aggregation using real stored routines.

The web API and front–end are intentionally kept simple and minimal, serving mainly as a convenient explorer for the underlying database design and its artifacts.

2 Objectives

The main objectives of the project are:

1. To model a realistic yet manageable e–commerce domain using an Entity–Relationship (ER) approach.
2. To transform the ER model into a normalized relational schema in MySQL, satisfying at least Third Normal Form (3NF) and, where possible, Boyce–Codd Normal Form (BCNF).
3. To implement integrity constraints using primary keys, foreign keys, unique constraints and `CHECK` constraints.
4. To encapsulate business logic using PL/SQL–style stored procedures, functions, and triggers.

5. To demonstrate relational algebra operations (joins, union, intersection, difference, division) using stored routines.
6. To provide a small web–based interface for exploring tables, modifying data, and invoking database routines.

3 System Overview

At a high level, the system consists of:

- A MySQL database named `dbms` containing all the tables, constraints, stored procedures, functions, and triggers.
- A FastAPI back–end (`app.py`) that connects to the MySQL database and exposes HTTP endpoints for:
  - Listing tables and table contents.
  - Describing table schemas.
  - Creating, dropping, and truncating tables (DDL).
  - Inserting, updating, and deleting rows (DML).
  - Invoking stored procedures and functions.
- A set of static HTML pages:
  - `index.html`: Simple table viewer.
  - `editor.html`: Interactive table editor supporting row update and delete.
  - `ddl.html`: DDL explorer for creating, dropping, and truncating tables.
  - `artifacts.html`: Routine explorer to invoke procedures and functions documented in the `DbArtifacts` table.

The following sections focus primarily on the database design aspects: ER modeling, relational schema, normalization, relational algebra, and PL/SQL constructs.

4 Requirements Analysis

The simplified e–commerce system is designed to support the following core requirements:

- **Manage users**: Two distinct types of users are supported: *customers* and *sellers*. Both have login credentials and contact details.

- **Manage products**: Sellers can list products for sale with name, price, stock quantity, description and rating.
- **Handle orders**: Customers can place orders for products in certain quantities. Stock is decremented accordingly.
- **Shopping cart**: Customers can maintain a cart with one or more items before placing orders.
- **Reviews**: Customers can rate and review products. Each customer can review a product at most once.
- **Price history**: Changes in product price should be logged for auditing.
- **Analytical queries**: The system should support queries such as total spend by a customer, total revenue of a seller, loyal customers who have bought all products of a seller, and aggregated order statistics.

From these requirements, we derived the main entities: `Customers`, `Sellers`, `Products`, `Orders`, `CartItems`, `Reviews`, `ProductPriceHistory`, and `DbArtifacts`.

5 Conceptual Design: ER Diagram

The main cardinalities are:

- One customer can have many cart items and many orders.
- One seller can list many products.
- One product can have many reviews and many price history entries.
- Each review belongs to exactly one customer and one product.
- Each order entry is for exactly one customer and one product.

6 Logical Design: Relational Schema

The ER model is mapped to the following relational tables in MySQL:

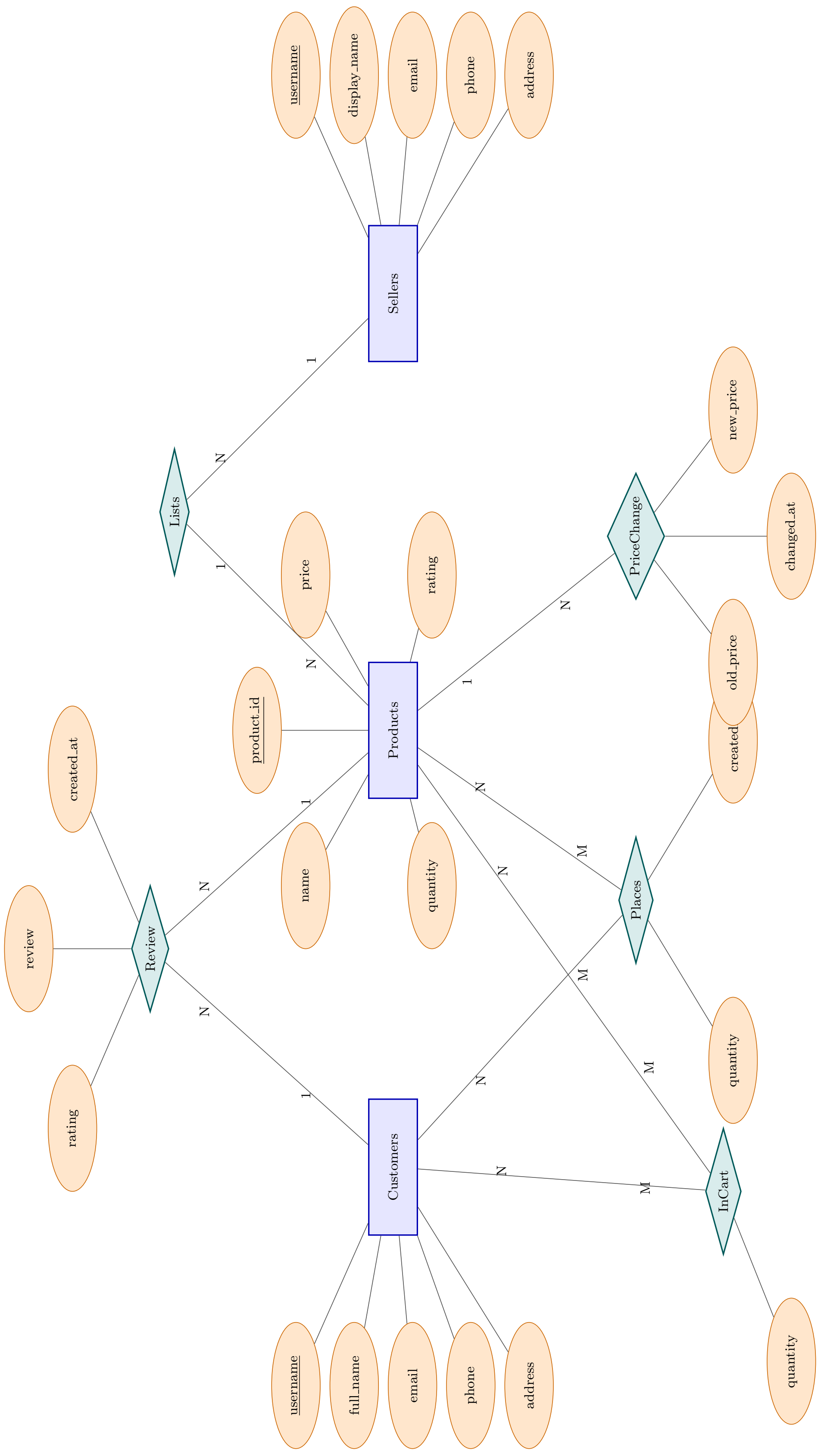

Figure 1: ER diagram for the Ecommerce Database Explorer

6.1 Core Entity Tables

6.1.1 Customers

```
CREATE TABLE Customers (
  username       VARCHAR(64)  PRIMARY KEY,
  password_hash  VARCHAR(255) NOT NULL,
  full_name      VARCHAR(255) NOT NULL,
  email          VARCHAR(255) NOT NULL UNIQUE,
  phone          VARCHAR(32),
  address        TEXT,
  created_at     DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

Key points:

- `username` is the primary key.
- `email` is unique, making it an alternate key.
- All attributes are atomic, satisfying 1NF.

6.1.2 Sellers

```
CREATE TABLE Sellers (
  username       VARCHAR(64)  PRIMARY KEY,
  password_hash  VARCHAR(255) NOT NULL,
  display_name   VARCHAR(255) NOT NULL,
  email          VARCHAR(255) NOT NULL UNIQUE,
  phone          VARCHAR(32),
  address        TEXT,
  created_at     DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

The design is analogous to `Customers`, keeping customer and seller data separate while enforcing similar constraints.

6.1.3 Products

```
CREATE TABLE Products (
  product_id       INT UNSIGNED NOT NULL AUTO_INCREMENT,
  seller_username  VARCHAR(64)  NOT NULL,
  name             VARCHAR(255) NOT NULL,
  rating           INT,
  price            INT UNSIGNED NOT NULL,
  quantity         INT UNSIGNED NOT NULL,
  description      TEXT,
  created_at       DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (product_id),
  FOREIGN KEY (seller_username) REFERENCES Sellers(username)
    ON DELETE CASCADE,
  CHECK (price >= 0),
  CHECK (quantity >= 0),
  CHECK (rating IS NULL OR rating BETWEEN 1 AND 5),
  UNIQUE KEY uq_products_seller_name (seller_username, name)
);
```

Key points:

- product$_{i}distheprimarykey.$
- The pair (seller_username, name) is unique, preventing a seller from listing two products with the same name.
- CHECK constraints enforce data validity for price, quantity, and rating.

6.2 Relationship Tables

6.2.1 Reviews

```
CREATE TABLE Reviews (
  product_id         INT UNSIGNED NOT NULL,
  customer_username  VARCHAR(64)  NOT NULL,
  rating             INT NOT NULL,
  review             TEXT,
  created_at         DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (product_id, customer_username),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
    ON DELETE CASCADE,
  FOREIGN KEY (customer_username) REFERENCES Customers(username)
    ON DELETE CASCADE,
  CHECK (rating BETWEEN 1 AND 5)
);
```

Each customer can review a product at most once, which is enforced by the composite primary key.

6.2.2 CartItems

```
CREATE TABLE CartItems (
  product_id         INT UNSIGNED NOT NULL,
  customer_username  VARCHAR(64)  NOT NULL,
  quantity           INT UNSIGNED NOT NULL,
  PRIMARY KEY (product_id, customer_username),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
    ON DELETE CASCADE,
  FOREIGN KEY (customer_username) REFERENCES Customers(username)
    ON DELETE CASCADE,
  CHECK (quantity >= 0)
);
```

6.2.3 Orders

```
CREATE TABLE Orders (
  product_id         INT UNSIGNED NOT NULL,
  customer_username  VARCHAR(64)  NOT NULL,
  quantity           INT UNSIGNED NOT NULL,
  created_at         DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (product_id, customer_username, created_at),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
    ON DELETE RESTRICT,
  FOREIGN KEY (customer_username) REFERENCES Customers(username)
    ON DELETE RESTRICT,
  CHECK (quantity >= 0)
);
```

The primary key includes `created_at` so that the same customer can place multiple orders for the same product at different times.

6.3 Supporting Tables

6.3.1 ProductPriceHistory

```
CREATE TABLE ProductPriceHistory (
  id         INT UNSIGNED NOT NULL AUTO_INCREMENT,
  product_id INT UNSIGNED NOT NULL,
  old_price  INT UNSIGNED NOT NULL,
  new_price  INT UNSIGNED NOT NULL,
  changed_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
    ON DELETE CASCADE
);
```

This table records every price change for a product, populated via a trigger (see Section 9).

6.3.2 DbArtifacts

```
CREATE TABLE DbArtifacts (
  name         VARCHAR(128) NOT NULL,
  type         VARCHAR(16)  NOT NULL,
  description  TEXT         NOT NULL,
  param_count  INT          NOT NULL DEFAULT 0,
  PRIMARY KEY (name, type),
  CHECK (type IN ('PROCEDURE','FUNCTION','TRIGGER')),
  CHECK (param_count >= 0)
);
```

This is a documentation table that stores metadata about procedures, functions, and triggers. The `artifacts.html` page reads from this table to dynamically present available routines and their parameter counts.

7 Normalization and Normal Forms

All tables were designed to satisfy at least Third Normal Form (3NF). In many cases the schema also satisfies Boyce–Codd Normal Form (BCNF).

7.1 Customers and Sellers

Functional dependencies in `Customers`:

$$\text{username} \to \{\text{password\_hash}, \text{full\_name}, \text{email}, \text{phone}, \text{address}, \text{created\_at}\}$$

$$\text{email} \to \{\text{username}, \text{password\_hash}, \text{full\_name}, \text{phone}, \text{address}, \text{created\_at}\}$$

Both `username` and `email` are candidate keys. All non–key attributes depend on the whole key and only on the key.

- 1NF: All attributes are atomic (no repeating groups).

- 2NF: There is no composite key, so 2NF is trivially satisfied.
- 3NF: No non–key attribute depends on another non–key attribute; hence 3NF holds.
- BCNF: Every non–trivial functional dependency has a candidate key on the left; therefore `Customers` and `Sellers` are in BCNF.

7.2 Products

Main functional dependencies in `Products`:

$$\text{product\_id} \rightarrow \{\text{seller\_username}, \text{name}, \text{rating}, \text{price}, \text{quantity}, \text{description}, \text{created\_at}\}$$

$$(\text{seller\_username}, \text{name}) \rightarrow \{\text{product\_id}, \text{rating}, \text{price}, \text{quantity}, \text{description}, \text{created\_at}\}$$

Both `product_id` and the pair `(seller_username, name)` are candidate keys due to the unique constraint.

There are no partial or transitive dependencies among non–key attributes. Thus:

- `Products` is in 3NF.
- All determining attributes on the left side of FDs are candidate keys, so it also satisfies BCNF.

7.3 Relationship Tables: Reviews, CartItems, Orders

`Reviews` has primary key `(product_id, customer_username)` and other attributes `rating, review, created_at`. The only relevant FD is:

$$(\text{product\_id},\ \text{customer\_username}) \rightarrow \{\text{rating}, \text{review}, \text{created\_at}\}$$

Similar reasoning applies to `CartItems` and `Orders`.

- All attributes are atomic: 1NF.
- All non–key attributes depend on the full composite key: 2NF.
- There are no transitive dependencies: 3NF.
- Left side of each FD is a key: hence BCNF.

7.4 Supporting Tables

`ProductPriceHistory` and `DbArtifacts` are also in at least 3NF:

- Each table has a simple primary key (`id` for `ProductPriceHistory`, and `(name, type)` for `DbArtifacts`).
- All other attributes depend solely on that key.
- No non–key attribute determines any other non–key attribute.

Thus the overall schema is free of update, insertion, and deletion anomalies related to redundancy.

8 Integrity Constraints and Business Rules

Beyond primary and foreign keys, the database enforces various integrity rules:

- **Domain constraints**: `CHECK` constraints ensure non–negative prices and quantities and enforce valid rating ranges.
- **Referential integrity**: Foreign keys connect `Orders`, `CartItems`, and `Reviews` to `Products`, `Customers`, and `Sellers` with appropriate `ON DELETE` actions.
- **Uniqueness constraints**: Unique keys on `email` and on `(seller_username, name)` avoid duplicates.
- **Triggers**: Before/after triggers check business rules such as stock availability, rating range, non–zero cart quantities, and automatic stock decrement and rating updates.

9 PL/SQL Layer: Procedures, Functions and Triggers

MySQL procedural SQL (PL/SQL–style syntax) is used to encapsulate business logic inside the database.

9.1 Stored Procedures

9.1.1 `create_customer` and `create_seller`

These procedures insert new rows into `Customers` and `Sellers` respectively:

```
CREATE PROCEDURE create_customer(
  IN p_username      VARCHAR(64),
  IN p_password_hash VARCHAR(255),
  IN p_full_name     VARCHAR(255),
  IN p_email         VARCHAR(255),
  IN p_phone         VARCHAR(32),
  IN p_address       TEXT
)
BEGIN
  INSERT INTO Customers(username, password_hash, full_name,
                        email, phone, address)
  VALUES (p_username, p_password_hash, p_full_name,
          p_email, p_phone, p_address);
END;
```

These procedures support controlled creation of users, centralizing validation and allowing reuse.

9.1.2 add_product

```
CREATE PROCEDURE add_product(
  IN p_seller_username VARCHAR(64),
  IN p_name            VARCHAR(255),
  IN p_price           INT UNSIGNED,
  IN p_quantity        INT UNSIGNED,
  IN p_description     TEXT
)
BEGIN
  INSERT INTO Products(seller_username, name, rating,
                       price, quantity, description)
  VALUES (p_seller_username, p_name, NULL,
          p_price, p_quantity, p_description);
END;
```

9.1.3 place_order with Transactional Logic

```
CREATE PROCEDURE place_order(
  IN p_customer_username VARCHAR(64),
  IN p_product_id        INT UNSIGNED,
  IN p_quantity          INT UNSIGNED
)
BEGIN
  DECLARE v_stock INT;

  SELECT quantity INTO v_stock
  FROM Products
  WHERE product_id = p_product_id
  FOR UPDATE;

  IF v_stock IS NULL THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Product not found';
  ELSEIF v_stock < p_quantity THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Insufficient stock';
  ELSE
    INSERT INTO Orders(product_id, customer_username, quantity)
    VALUES (p_product_id, p_customer_username, p_quantity);

    UPDATE Products
    SET quantity = quantity - p_quantity
    WHERE product_id = p_product_id;
  END IF;
END;
```

The `FOR UPDATE` clause locks the product row to prevent race conditions while checking and updating stock.

9.1.4 `get_invoice`: Example with UNION ALL

```
CREATE PROCEDURE get_invoice(IN p_user VARCHAR(10))
BEGIN
  SELECT p.name AS product, o.quantity AS quantity,
         p.price * o.quantity AS price
  FROM Products p, Orders o
  WHERE o.customer_username = p_user
    AND o.product_id = p.product_id
  UNION ALL
  SELECT 'TOTAL' AS product,
         SUM(o.quantity) AS quantity,
         SUM(p.price * o.quantity) AS price
  FROM Products p, Orders o
  WHERE o.customer_username = p_user
    AND o.product_id = p.product_id;
END;
```

The invoice procedure demonstrates a combination of join and aggregation plus union of detail rows with a total row.

9.2 Stored Functions

Several functions compute derived values:

- `product_avg_rating(product_id)`: Returns average rating for a product.
- `customer_total_spent(customer_username)`: Total amount spent by a customer.
- `customer_order_count(customer_username)`: Number of orders placed by a customer.
- `product_stock(product_id)`: Current stock quantity.
- `seller_total_revenue(seller_username)`: Total revenue for a seller.

Example:

```
CREATE FUNCTION customer_total_spent(
  p_customer_username VARCHAR(64)
)
RETURNS INT
READS SQL DATA
BEGIN
  DECLARE v_total INT;
  SELECT IFNULL(SUM(o.quantity * p.price),0) INTO v_total
  FROM Orders o
  JOIN Products p ON p.product_id = o.product_id
  WHERE o.customer_username = p_customer_username;
  RETURN v_total;
END;
```

This function corresponds to a relational algebra expression applying a join and aggregation (see Section 10).

9.3 Triggers

9.3.1 Validation Triggers

- `reviews_before_insert_rating` ensures rating is between 1 and 5.

```
CREATE TRIGGER reviews_before_insert_rating
BEFORE INSERT ON Reviews
FOR EACH ROW
BEGIN
  IF NEW.rating < 1 OR NEW.rating > 5 THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Rating must be between 1 and 5';
  END IF;
END;
```

- `cartitems_before_insert_quantity` enforces strictly positive cart quantities.

9.3.2 Stock Management Triggers

- `orders_before_insert_stock` checks stock before inserting an order.
- `orders_after_insert_decrement_stock` decrements product stock after each order.

These triggers cooperate with the `place_order` procedure to ensure that stock never goes negative and that each order corresponds to available inventory.

9.3.3 Price History Trigger

```
CREATE TRIGGER products_after_update_price
AFTER UPDATE ON Products
FOR EACH ROW
BEGIN
  IF NEW.price <> OLD.price THEN
    INSERT INTO ProductPriceHistory(product_id, old_price, new_price)
    VALUES (NEW.product_id, OLD.price, NEW.price);
  END IF;
END;
```

This automatically logs every price change in `ProductPriceHistory`.

9.3.4 Rating Aggregation Trigger

```
CREATE TRIGGER reviews_after_insert_update_product_rating
AFTER INSERT ON Reviews
FOR EACH ROW
BEGIN
```

```
  UPDATE Products p
  SET p.rating = (
    SELECT ROUND(AVG(r.rating))
    FROM Reviews r
    WHERE r.product_id = NEW.product_id
  )
  WHERE p.product_id = NEW.product_id;
END;
```

The product's `rating` column is maintained as the rounded average of all review ratings for that product.

10 Relational Algebra and Joins

Several procedures are designed specifically to demonstrate relational algebra operations using SQL.

10.1 Join Operations

10.1.1 Inner Join: `join_orders_products_inner`

```
CREATE PROCEDURE join_orders_products_inner(
  IN p_customer_username VARCHAR(64)
)
BEGIN
  SELECT o.product_id, p.name, o.quantity,
         p.price * o.quantity AS total_price
  FROM Orders o
  JOIN Products p ON p.product_id = o.product_id
  WHERE o.customer_username = p_customer_username;
END;
```

In relational algebra, this corresponds to:

$$\pi_{\text{product\_id, name, quantity, total\_price}}(\sigma_{\text{customer\_username}=c}(\text{Orders}) \bowtie_{\text{Orders.product\_id}=\text{Products.product\_id}} \text{Products})$$

10.1.2 Left Join: `join_products_reviews_left`

```
CREATE PROCEDURE join_products_reviews_left()
BEGIN
  SELECT p.product_id, p.name, r.customer_username, r.rating
  FROM Products p
  LEFT JOIN Reviews r ON r.product_id = p.product_id;
END;
```

This shows all products, including those with no reviews (outer join).

10.1.3 Right Join: `join_sellers_products_right`

```
CREATE PROCEDURE join_sellers_products_right()
BEGIN
  SELECT s.username AS seller_username,
         s.display_name, p.product_id, p.name
  FROM Products p
  RIGHT JOIN Sellers s ON p.seller_username = s.username;
END;
```

Each seller appears even if they have no products, illustrating a right outer join.

10.1.4 Full Join via UNION: `join_products_reviews_full`

```
CREATE PROCEDURE join_products_reviews_full()
BEGIN
  SELECT p.product_id, p.name, r.customer_username, r.rating
  FROM Products p
  LEFT JOIN Reviews r ON r.product_id = p.product_id
  UNION
  SELECT p.product_id, p.name, r.customer_username, r.rating
  FROM Products p
  RIGHT JOIN Reviews r ON r.product_id = p.product_id;
END;
```

Since MySQL does not have a native full outer join, this procedure simulates it using a union of left and right joins.

10.1.5 Cross Join: `join_customers_sellers_cross`

```
CREATE PROCEDURE join_customers_sellers_cross(
  IN p_limit INT
)
BEGIN
  SELECT c.username AS customer_username,
         s.username AS seller_username
  FROM Customers c
  CROSS JOIN Sellers s
  LIMIT p_limit;
END;
```

This corresponds to the Cartesian product of `Customers` and `Sellers`, limited to a subset of pairs.

10.2 Set Operations: Union, Intersection, Difference

10.2.1 Union: `ra_union_customers_orders_reviews`

```
CREATE PROCEDURE ra_union_customers_orders_reviews()
BEGIN
  SELECT DISTINCT customer_username
  FROM Orders
  UNION
  SELECT DISTINCT customer_username
  FROM Reviews;
END;
```

Relational algebra:

$$\pi_{\text{customer\_username}}(\text{Orders}) \cup \pi_{\text{customer\_username}}(\text{Reviews})$$

10.2.2 Intersection: `ra_intersection_customers_orders_reviews`

```
CREATE PROCEDURE ra_intersection_customers_orders_reviews()
BEGIN
  SELECT DISTINCT o.customer_username
  FROM Orders o
  JOIN Reviews r
    ON r.customer_username = o.customer_username;
END;
```

Relational algebra:

$$\pi_{\text{customer\_username}}(\text{Orders}) \cap \pi_{\text{customer\_username}}(\text{Reviews})$$

10.2.3 Difference: `ra_difference_customers_orders_not_reviews`

```
CREATE PROCEDURE ra_difference_customers_orders_not_reviews()
BEGIN
```

```
  SELECT DISTINCT o.customer_username
  FROM Orders o
  LEFT JOIN Reviews r
    ON r.customer_username = o.customer_username
  WHERE r.customer_username IS NULL;
END;
```

Relational algebra:

$$\pi_{\text{customer\_username}}(\text{Orders}) \; - \; \pi_{\text{customer\_username}}(\text{Reviews})$$

10.3 Division: Loyal Customers

10.3.1 Relational Division: `ra_division_loyal_customers`

```
CREATE PROCEDURE ra_division_loyal_customers(
  IN p_seller_username VARCHAR(64)
)
BEGIN
  SELECT c.username
  FROM Customers c
  WHERE NOT EXISTS (
    SELECT 1
    FROM Products p
    WHERE p.seller_username = p_seller_username
      AND NOT EXISTS (
        SELECT 1
        FROM Orders o
        WHERE o.customer_username = c.username
          AND o.product_id = p.product_id
      )
  );
END;
```

This procedure finds customers who have bought *all* products from a given seller. In relational algebra terms, it corresponds to division:

$$\pi_{\text{customer}}(O) \div \pi_{\text{product}}(P_s)$$

where O is the relation of customer–product orders and P_s is the set of products of seller s.

10.4 Aggregation: Grouping and Aggregate Functions

10.4.1 Aggregated Orders: `agg_orders_by_customer`

```
CREATE PROCEDURE agg_orders_by_customer(
  IN p_customer_username VARCHAR(64)
)
BEGIN
  SELECT
    p_customer_username AS customer_username,
    COUNT(*)                      AS order_count,
    SUM(o.quantity * p.price)     AS total_amount,
    AVG(o.quantity * p.price)     AS avg_order_value,
    MIN(o.quantity * p.price)     AS min_order_value,
    MAX(o.quantity * p.price)     AS max_order_value
  FROM Orders o
  JOIN Products p ON p.product_id = o.product_id
  WHERE o.customer_username = p_customer_username;
END;
```

This uses `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`, which map directly to relational algebra's aggregation operator γ.

11 Database Explorer Front–End (Brief Description)

Although the project centers on the database, a minimal web interface has been implemented for convenience:

- **Table viewer** (`index.html`): Allows users to select any table and display its contents as an HTML table.
- **Table editor** (`editor.html`): Provides interactive editing of table rows. Primary key columns are detected via `DESC` and used for `UPDATE` and `DELETE` operations through `/api/update_table` and `/api/delete_table`.
- **DDL explorer** (`ddl.html`): Enables creation of new tables with chosen columns and types, and supports dropping and truncating existing tables.
- **Routine explorer** (`artifacts.html`): Reads procedure/function metadata from `DbArtifacts` and dynamically renders input fields for parameters, allowing easy experimentation with stored routines demonstrating relational algebra and PL/SQL.

The front–end is intentionally simple and generic so that the design and behavior of the database layer remain the main focus.

12 Testing and Sample Scenarios

To validate the design and implementation, the following test scenarios were considered:

- **User creation**: Creating customers and sellers using `create_customer` and `create_seller`, ensuring uniqueness of usernames and emails.
- **Product listing**: Adding products via `add_product` and verifying foreign key relationships and check constraints.
- **Order placement**: Calling `place_order` under different stock conditions:
  - Sufficient stock: Order inserted and product quantity decremented.
  - Insufficient stock: Error raised by trigger or procedure.
  - Non–existent product: `SIGNAL` raised.
- **Review insertion**: Attempting to insert valid and invalid ratings to test the rating validation trigger.
- **Price update**: Updating product price and verifying that entries are created in `ProductPriceHistory`.
- **Relational algebra routines**: Executing join and set operation procedures via `artifacts.html` and checking correctness of results (union, intersection, difference, division and aggregations).

13 Conclusion

The **Ecommerce Database Explorer** project demonstrates how theoretical concepts of relational database design can be applied to a realistic application domain. Starting from requirements, we constructed an ER model and a normalized relational schema in MySQL, enforced various integrity constraints, and implemented business rules and analytical queries using PL/SQL–style stored procedures, functions, and triggers.

The project also illustrates how core relational algebra operations—joins, union, intersection, difference, division, and aggregation—can be expressed and executed through stored routines. The minimal web interface built with FastAPI and HTML/JavaScript acts as a convenient front–end to explore and interact with the database.

Overall, the project helped us to:

- Practically apply normal forms and evaluate the quality of a schema.
- Use MySQL's procedural capabilities to keep business logic close to the data.
- Bridge the gap between relational algebra theory and SQL implementations.

14 Future Enhancements

Possible improvements and extensions include:

- Introducing a dedicated `Orders` header table with an `order_id` and a separate `OrderItems` table.
- Adding more complex constraints, such as preventing multiple reviews by the same customer for the same product at the application level.
- Implementing indexing strategies and query optimization for large data volumes.
- Extending the web explorer with authentication and role–based access (customer vs. seller).
- Adding reporting dashboards using stored views and additional aggregation routines.

References

[1] R. Elmasri and S. Navathe. *Fundamentals of Database Systems.* Pearson.

[2] MySQL 8.0 Reference Manual, `https://dev.mysql.com/doc/`.