T.R.

GEBZE TECHNICAL UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

BOOKSTORE MANAGEMENT SYSTEM DATABASE DESIGN AND IMPLEMENTATION

AYKUT SERT

SUPERVISOR
DR. BURCU YILMAZ

GEBZE
2024

T.R.
GEBZE TECHNICAL UNIVERSITY
FACULTY OF ENGINEERING
COMPUTER ENGINEERING DEPARTMENT

BOOKSTORE MANAGEMENT SYSTEM DATABASE DESIGN AND IMPLEMENTATION

AYKUT SERT

COURSE
CSE414 DATABASE

SUPERVISOR
DR. BURCU YILMAZ

2024
GEBZE

ABSTRACT

This project presents the design and implementation of a comprehensive bookstore management system database for the CSE414 Database course. The system encompasses complete database design methodology including entity-relationship modeling, normalization analysis, and advanced database features implementation.

The bookstore management system is designed to handle core business operations including user management, book inventory, order processing, and sales tracking. The database supports two user types: administrators who can manage books, users, and orders, and customers who can browse books and place orders.

The project demonstrates advanced database concepts through implementation of five custom views for data analysis, five stored functions for business logic, and five triggers for automatic data management and logging. The system includes comprehensive join operations (LEFT, RIGHT, and FULL OUTER joins), transaction management for data integrity, and concurrency control mechanisms.

A complete web-based user interface was developed using Flask framework, providing separate dashboards for administrators and customers. The admin interface allows full CRUD operations on books and users, order management, and real-time monitoring of database operations through trigger logs, function testing, and view results.

The database schema follows third normal form (3NF) principles and includes detailed functional dependency analysis. Entity-relationship diagram illustrates the complex relationships between users, books, authors, categories, orders, and order details, including weak entities and many-to-many relationships through junction tables.

Key technical achievements include automatic stock management through triggers, real-time order total calculation, comprehensive logging system for all database operations, and a responsive web interface that demonstrates all database features in real-time. The system successfully integrates theoretical database concepts with practical implementation, providing a complete enterprise-level database solution.

Keywords: Database Design, Entity-Relationship Modeling, Normalization, Triggers, Views, Functions, Web Application, Flask, MySQL

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION AND REQUIREMENTS

1.1. Project Overview

The Bookstore Management System is a comprehensive database application designed to demonstrate advanced database design principles and implementation techniques. This project serves as a practical implementation of the theoretical concepts learned in the CSE414 Database course, showcasing entity-relationship modeling, normalization, and advanced database features.

The system is designed to manage all aspects of a bookstore's operations, from inventory management to customer orders, while providing a user-friendly web interface for both administrators and customers. The project emphasizes the importance of data integrity, efficient query processing, and scalable database design.

1.2. Business Requirements

1.2.1. Functional Requirements

The bookstore management system must satisfy the following functional requirements:

1. **User Management:** The system must support two types of users - administrators and customers. Administrators have full access to system management features, while customers can browse and purchase books.

2. **Book Management:** The system must maintain a comprehensive catalog of books with detailed information including title, ISBN, publication year, price, stock quantity, and category classification.

3. **Author Management:** The system must track author information and support multiple authors per book through many-to-many relationships.

4. **Category Management:** Books must be organized into categories for efficient browsing and searching.

5. **Order Processing:** The system must handle customer orders with detailed order tracking, automatic stock updates, and total calculation.

6. **Inventory Management:** Real-time stock tracking with automatic updates when orders are placed and administrative stock adjustments.

1.2.2. Technical Requirements

1. **Database Views:** Implementation of at least five different views for data analysis and reporting:
    - Books currently in stock
    - Customer order summaries
    - Author-book relationships
    - High-value orders (above 50 TL)
    - Low stock alerts
2. **Stored Functions:** Five custom functions for business logic:
    - Book stock quantity retrieval
    - Order total calculation
    - Books by category listing
    - Author book count
    - Average completed order value
3. **Database Triggers:** Five triggers for automatic operations:
    - Book insertion logging
    - Book update logging
    - Automatic stock updates on orders
    - Order total calculation
    - User registration logging
4. **Advanced SQL Operations:** Implementation of various JOIN types including LEFT, RIGHT, and FULL OUTER joins for comprehensive data retrieval.
5. **Transaction Management:** ACID-compliant transactions for order processing and inventory updates:
    - *Complete Order Processing:* Atomic order creation with stock validation, automatic inventory deduction, and total calculation using row-level locking (FOR UPDATE)

- *Bulk Stock Update:* Batch processing for low-stock items with rollback capability and system logging
- *Order Transfer:* Atomic transfer of pending orders between users with full audit trail
- Real-time transaction history with detailed step-by-step execution logs
- Comprehensive error handling with automatic rollback on failure

6. **Concurrency Control:** Proper isolation levels and locking mechanisms for concurrent user access.

1.2.3. User Interface Requirements

1. **Customer Interface:**
   - Book browsing with detailed information
   - Simple one-click ordering system
   - Order history viewing
   - Real-time stock availability
2. **Administrator Interface:**
   - Complete CRUD operations for books and users
   - Stock management and updates
   - Order status management
   - Real-time view of database operations
   - Function testing interface
   - Trigger log monitoring

1.3. System Architecture

The system follows a three-tier architecture:

1. **Presentation Layer:** Web-based user interface built with HTML, CSS, and JavaScript, providing responsive design for both desktop and mobile access.
2. **Application Layer:** Flask web framework handling business logic, user authentication, and database interactions through a RESTful API design.
3. **Data Layer:** MySQL database with comprehensive schema design, including tables, views, functions, triggers, and stored procedures.

1.4. Development Environment

The project utilizes the following technology stack:

- **Database:** MySQL 8.0 for robust ACID compliance and advanced features
- **Backend:** Python Flask framework for rapid development and scalability
- **Frontend:** HTML5, CSS3, and vanilla JavaScript for cross-browser compatibility
- **Database Connectivity:** mysql-connector-python for efficient database operations
- **Development Tools:** Modern code editors and database management tools

1.5. Project Scope and Limitations

1.5.1. Scope

This project covers:

- Complete database design and implementation
- Advanced database features (views, functions, triggers)
- Web-based user interface
- Real-time data operations and monitoring
- Comprehensive testing of all database features

1.5.2. Limitations

The following features are outside the project scope:

- Payment processing integration
- Advanced reporting and analytics
- Mobile application development
- Multi-language support
- Advanced security features (encryption, OAuth)

2. DATABASE DESIGN

2.1. Entity-Relationship Diagram

The Entity-Relationship (E-R) diagram forms the foundation of our database design, illustrating the relationships between different entities in the bookstore management system. Figure 2.1 presents the complete E-R diagram with all entities, attributes, and relationships.

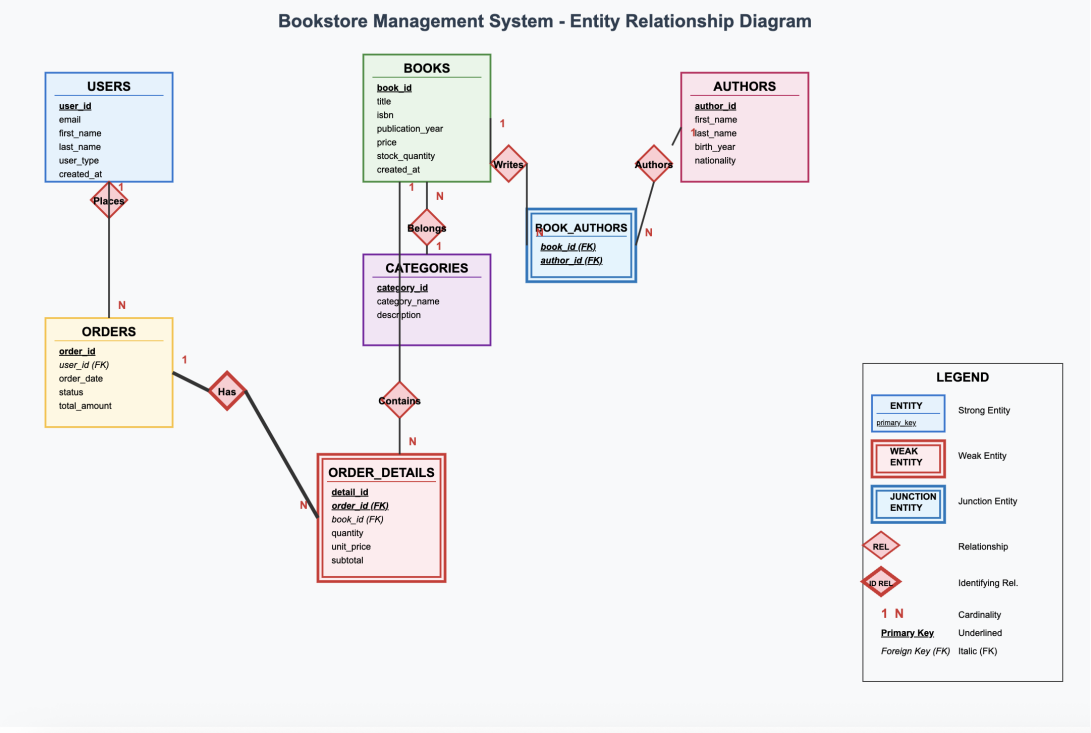

Figure 2.1: Bookstore Management System E-R Diagram

2.1.1. Entity Descriptions

2.1.1.1. Strong Entities

1. **USERS:** Represents both administrators and customers in the system
   - Primary Key: user_id
   - Attributes: email, first_name, last_name, user_type, created_at

2. **BOOKS:** Central entity representing book inventory
    - Primary Key: book_id
    - Attributes: title, isbn, publication_year, price, stock_quantity, created_at
3. **AUTHORS:** Information about book authors
    - Primary Key: author_id
    - Attributes: first_name, last_name, birth_year, nationality
4. **CATEGORIES:** Book classification system
    - Primary Key: category_id
    - Attributes: category_name, description
5. **ORDERS:** Customer order information
    - Primary Key: order_id
    - Attributes: user_id (FK), order_date, status, total_amount

2.1.1.2. Weak Entities

1. **ORDER_DETAILS:** Detailed information about items in each order
    - Primary Key: detail_id, order_id (FK)
    - Attributes: book_id (FK), quantity, unit_price, subtotal
    - Depends on: ORDERS entity

2.1.1.3. Junction Entities

1. **BOOK_AUTHORS:** Resolves many-to-many relationship between books and authors
    - Primary Key: book_id (FK), author_id (FK)
    - Purpose: Supports multiple authors per book

2.1.2. Relationship Analysis

1. **Places (USERS → ORDERS):** One-to-Many
    - A user can place multiple orders

- Each order belongs to exactly one user
- Cardinality: 1:N

2. **Has (ORDERS → ORDER_DETAILS):** One-to-Many (Identifying)
    - An order can have multiple detail records
    - Each detail record belongs to exactly one order
    - Identifying relationship: ORDER_DETAILS depends on ORDERS
    - Cardinality: 1:N
3. **Contains (BOOKS → ORDER_DETAILS):** One-to-Many
    - A book can appear in multiple order details
    - Each order detail references exactly one book
    - Cardinality: 1:N
4. **Belongs (BOOKS → CATEGORIES):** Many-to-One
    - Multiple books can belong to the same category
    - Each book belongs to exactly one category
    - Cardinality: N:1
5. **Writes (AUTHORS BOOKS):** Many-to-Many
    - An author can write multiple books
    - A book can have multiple authors
    - Resolved through BOOK_AUTHORS junction table
    - Cardinality: M:N

2.2. Functional Dependencies

Functional dependencies define how attributes in our database relate to each other. Understanding these dependencies is crucial for proper normalization and maintaining data integrity.

2.2.1. USERS Table Dependencies

$$user\_id \rightarrow email, first\_name, last\_name, user\_type, created\_at \tag{2.1}$$

$$email \rightarrow user\_id, first\_name, last\_name, user\_type, created\_at \tag{2.2}$$

The user_id uniquely determines all other attributes. Email is also a candidate key as it must be unique.

2.2.2. BOOKS Table Dependencies

$$book\_id \rightarrow title, isbn, publication\_year, price, stock\_quantity, category\_id, created\_at \tag{2.3}$$

$$isbn \rightarrow book\_id, title, publication\_year, price, stock\_quantity, category\_id, created\_at \tag{2.4}$$

Both book_id and isbn can serve as primary keys, with isbn being a natural candidate key.

2.2.3. AUTHORS Table Dependencies

$$author\_id \rightarrow first\_name, last\_name, birth\_year, nationality \tag{2.5}$$

The author_id uniquely determines all author attributes.

2.2.4. CATEGORIES Table Dependencies

$$category\_id \rightarrow category\_name, description \tag{2.6}$$

$$category\_name \rightarrow category\_id, description \tag{2.7}$$

Category names are unique, making category_name a candidate key.

2.2.5. ORDERS Table Dependencies

$$order\_id \rightarrow user\_id, order\_date, status, total\_amount \tag{2.8}$$

Each order is uniquely identified by order_id.

2.2.6. ORDER_DETAILS Table Dependencies

$$detail\_id \rightarrow order\_id, book\_id, quantity, unit\_price, subtotal \tag{2.9}$$

$$order\_id, book\_id \rightarrow detail\_id, quantity, unit\_price, subtotal \tag{2.10}$$

The detail_id is the primary key. Additionally, the combination of order_id and book_id forms a candidate key, ensuring each book appears only once per order.

2.2.7. BOOK_AUTHORS Table Dependencies

$$book\_id, author\_id \rightarrow \emptyset \tag{2.11}$$

Pure junction table with no additional attributes beyond the foreign key pair.

2.3. Normalization Analysis

2.3.1. First Normal Form (1NF)

All tables satisfy 1NF requirements:

- All attributes contain atomic values
- No repeating groups or arrays
- Each column contains values of a single type
- All entries in a column are of the same kind

Example: Instead of storing multiple authors in a single field, we use the BOOK_AUTHORS junction table to properly represent the many-to-many relationship.

2.3.2. Second Normal Form (2NF)

All tables satisfy 2NF requirements:

- Already in 1NF
- All non-key attributes are fully functionally dependent on the primary key
- No partial dependencies exist

Example: In ORDER_DETAILS table, attributes like quantity and unit_price depend on the complete primary key (detail_id), ensuring no partial dependencies exist.

2.3.3. Third Normal Form (3NF)

All tables satisfy 3NF requirements:

- Already in 2NF
- No transitive dependencies exist
- All non-key attributes depend directly on the primary key

Example Decomposition:
Before normalization (violates 3NF):

book_id	title	category_id	category_name

Problem: category_name depends on category_id, not directly on book_id (transitive dependency).

After normalization (satisfies 3NF):
BOOKS table:

book_id	title	category_id

CATEGORIES table:

category_id	category_name

2.3.4. Boyce-Codd Normal Form (BCNF)

Our database also satisfies BCNF requirements:

- Already in 3NF
- For every functional dependency X $\rightarrow$ Y, X is a superkey
- No anomalies exist from candidate key dependencies

All our tables have simple primary keys or well-designed composite keys that eliminate BCNF violations.

2.4. Database Schema

The final database schema consists of eight main tables with carefully designed relationships and constraints.

2.4.1. Table Specifications

Table	Attribute	Type	Constraints
6***USERS**	user_id	INT	PRIMARY KEY, AUTO_INCREMENT
	email	VARCHAR(255)	UNIQUE, NOT NULL
	password	VARCHAR(255)	NOT NULL
	first_name	VARCHAR(100)	NOT NULL
	last_name	VARCHAR(100)	NOT NULL
	user_type	ENUM	'admin', 'customer', DEFAULT 'customer'
	created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP
7***BOOKS**	book_id	INT	PRIMARY KEY, AUTO_INCREMENT
	title	VARCHAR(255)	NOT NULL
	isbn	VARCHAR(20)	UNIQUE
	publication_year	INT	
	price	DECIMAL(10,2)	NOT NULL
	stock_quantity	INT	DEFAULT 0
	category_id	INT	FOREIGN KEY $\rightarrow$ categories(category_id)

Table	Attribute	Type	Constraints
	created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP
5***AUTHORS**	author_id	INT	PRIMARY KEY, AUTO_INCREMENT
	first_name	VARCHAR(100)	NOT NULL
	last_name	VARCHAR(100)	NOT NULL
	birth_year	INT	
	nationality	VARCHAR(100)	
3***CATEGORIES**	category_id	INT	PRIMARY KEY, AUTO_INCREMENT
	category_name	VARCHAR(100)	NOT NULL, UNIQUE
	description	TEXT	
5***ORDERS**	order_id	INT	PRIMARY KEY, AUTO_INCREMENT
	user_id	INT	FOREIGN KEY → users(user_id)
	order_date	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP
	status	ENUM	'pending', 'completed', 'cancelled'
	total_amount	DECIMAL(10,2)	DEFAULT 0
6***ORDER_DETAILS**	detail_id	INT	PRIMARY KEY, AUTO_INCREMENT
	order_id	INT	FOREIGN KEY → orders(order_id)
	book_id	INT	FOREIGN KEY → books(book_id)
	quantity	INT	NOT NULL
	unit_price	DECIMAL(10,2)	NOT NULL
	subtotal	DECIMAL(10,2)	NOT NULL
2***BOOK_AUTHORS**	book_id	INT	PRIMARY KEY, FOREIGN KEY → books(book_id)
	author_id	INT	PRIMARY KEY, FOREIGN KEY → authors(author_id)
6***SYSTEM_LOGS**	log_id	INT	PRIMARY KEY, AUTO_INCREMENT
	table_name	VARCHAR(100)	
	operation	VARCHAR(50)	
	record_id	INT	
	old_values	TEXT	
	new_values	TEXT	
	timestamp	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP

Table 2.1: Complete Database Schema Specification

3. DATABASE IMPLEMENTATION

3.1. Views Implementation

Database views provide abstracted access to complex queries and enhance data security by limiting direct table access. Our system implements five specialized views for different business requirements.

3.1.1. View 1: Books in Stock

This view displays all books currently available in inventory with author information.

```
CREATE VIEW books_in_stock AS
SELECT
    b.book_id,
    b.title,
    b.isbn,
    b.price,
    b.stock_quantity,
    c.category_name,
    GROUP_CONCAT(CONCAT(a.first_name, ' ', a.last_name) SEPARATOR ',
        ') AS authors
FROM books b
JOIN categories c ON b.category_id = c.category_id
LEFT JOIN book_authors ba ON b.book_id = ba.book_id
LEFT JOIN authors a ON ba.author_id = a.author_id
WHERE b.stock_quantity > 0
GROUP BY b.book_id, b.title, b.isbn, b.price, b.stock_quantity, c.
   category_name;
```

Listing 3.1: Books in Stock View

Purpose: Provides customer-facing book catalog with availability information.
Key Features:

- Filters out books with zero stock
- Aggregates multiple authors per book
- Includes category information for better organization

3.1.2. View 2: Customer Orders Summary

This view aggregates customer order statistics for business analytics.

```
CREATE VIEW customer_orders_summary AS
SELECT
    u.user_id,
    CONCAT(u.first_name, ' ', u.last_name) AS customer_name,
    u.email,
    COUNT(o.order_id) AS total_orders,
    COALESCE(SUM(o.total_amount), 0) AS total_spent,
    MAX(o.order_date) AS last_order_date
FROM users u
LEFT JOIN orders o ON u.user_id = o.user_id
WHERE u.user_type = 'customer'
GROUP BY u.user_id, u.first_name, u.last_name, u.email;
```

Listing 3.2: Customer Orders Summary View

Purpose: Enables customer relationship management and sales analysis.

Business Value:

- Identifies high-value customers
- Tracks customer engagement patterns
- Supports targeted marketing campaigns

3.1.3. View 3: Author Books

This view shows the relationship between authors and their published books.

```
CREATE VIEW author_books AS
SELECT
    a.author_id,
    CONCAT(a.first_name, ' ', a.last_name) AS author_name,
    a.nationality,
    b.title,
    b.publication_year,
    c.category_name
FROM authors a
JOIN book_authors ba ON a.author_id = ba.author_id
JOIN books b ON ba.book_id = b.book_id
JOIN categories c ON b.category_id = c.category_id
ORDER BY a.last_name, a.first_name, b.publication_year;
```

Listing 3.3: Author Books View

Purpose: Facilitates author research and bibliography creation.

Applications:

- Academic research support
- Author portfolio tracking
- Publication timeline analysis

3.1.4. View 4: High Value Orders

This view identifies orders exceeding 50 TL for business intelligence.

```
CREATE VIEW high_value_orders AS
SELECT
    o.order_id,
    CONCAT(u.first_name, ' ', u.last_name) AS customer_name,
    o.order_date,
    o.total_amount,
    o.status,
    COUNT(od.order_detail_id) AS item_count
FROM orders o
JOIN users u ON o.user_id = u.user_id
LEFT JOIN order_details od ON o.order_id = od.order_id
WHERE o.total_amount > 50
GROUP BY o.order_id, u.first_name, u.last_name, o.order_date, o.
   total_amount, o.status
ORDER BY o.total_amount DESC;
```

Listing 3.4: High Value Orders View

Purpose: Supports premium customer identification and revenue analysis.

3.1.5. View 5: Low Stock Books

This view alerts administrators about books requiring restocking.

```
CREATE VIEW low_stock_books AS
SELECT
    b.book_id,
    b.title,
    b.stock_quantity,
    b.price,
    c.category_name,
    GROUP_CONCAT(CONCAT(a.first_name, ' ', a.last_name) SEPARATOR ',
        ') AS authors
FROM books b
```

```
JOIN categories c ON b.category_id = c.category_id
LEFT JOIN book_authors ba ON b.book_id = ba.book_id
LEFT JOIN authors a ON ba.author_id = a.author_id
WHERE b.stock_quantity < 5
GROUP BY b.book_id, b.title, b.stock_quantity, b.price, c.
   category_name
ORDER BY b.stock_quantity ASC;
```

Listing 3.5: Low Stock Books View

Purpose: Inventory management and automated restocking alerts.

3.2. Functions Implementation

Stored functions encapsulate business logic and provide reusable computation modules throughout the application.

3.2.1. Function 1: Get Book Stock

Returns the current stock quantity for a specific book.

```
DELIMITER //
CREATE FUNCTION get_book_stock(book_id_param INT)
RETURNS INT
READS SQL DATA
DETERMINISTIC
BEGIN
    DECLARE stock_count INT DEFAULT 0;
    SELECT stock_quantity INTO stock_count
    FROM books
    WHERE book_id = book_id_param;
    RETURN COALESCE(stock_count, 0);
END//
DELIMITER ;
```

Listing 3.6: Get Book Stock Function

Usage: Pre-order validation and real-time inventory checks.

3.2.2. Function 2: Calculate Order Total

Computes the total amount for a given order.

```
DELIMITER //
CREATE FUNCTION calculate_order_total(order_id_param INT)
RETURNS DECIMAL(10,2)
```

```
READS SQL DATA
DETERMINISTIC
BEGIN
    DECLARE total_amount DECIMAL(10,2) DEFAULT 0;
    SELECT SUM(subtotal) INTO total_amount
    FROM order_details
    WHERE order_id = order_id_param;
    RETURN COALESCE(total_amount, 0);
END//
DELIMITER ;
```

Listing 3.7: Calculate Order Total Function

Usage: Order processing and financial reporting.

3.2.3. Function 3: Get Books by Category

Returns a comma-separated list of book titles for a specific category.

```
DELIMITER //
CREATE FUNCTION get_books_by_category(category_name_param VARCHAR
    (100))
RETURNS TEXT
READS SQL DATA
DETERMINISTIC
BEGIN
    DECLARE book_titles TEXT DEFAULT '';
    SELECT GROUP_CONCAT(b.title SEPARATOR ', ') INTO book_titles
    FROM books b
    JOIN categories c ON b.category_id = c.category_id
    WHERE c.category_name = category_name_param;
    RETURN COALESCE(book_titles, '');
END//
DELIMITER ;
```

Listing 3.8: Get Books by Category Function

Usage: Category-based searching and catalog organization.

3.2.4. Function 4: Get Author Book Count

Returns the total number of books written by a specific author.

```
DELIMITER //
CREATE FUNCTION get_author_book_count(author_id_param INT)
RETURNS INT
READS SQL DATA
```

```
DETERMINISTIC
BEGIN
    DECLARE book_count INT DEFAULT 0;
    SELECT COUNT(*) INTO book_count
    FROM book_authors
    WHERE author_id = author_id_param;
    RETURN book_count;
END//
DELIMITER ;
```

Listing 3.9: Get Author Book Count Function

Usage: Author productivity analysis and bibliography statistics.

3.2.5. Function 5: Get Average Completed Order Value

Calculates the average value of all completed orders.

```
DELIMITER //
CREATE FUNCTION get_average_completed_order_value()
RETURNS DECIMAL(10,2)
READS SQL DATA
DETERMINISTIC
BEGIN
    DECLARE avg_value DECIMAL(10,2) DEFAULT 0;
    SELECT AVG(total_amount) INTO avg_value
    FROM orders
    WHERE status = 'completed';
    RETURN COALESCE(avg_value, 0);
END//
DELIMITER ;
```

Listing 3.10: Average Order Value Function

Usage: Business intelligence and performance metrics.

3.3. Triggers Implementation

Database triggers ensure data integrity and automate business processes without application intervention.

3.3.1. Trigger 1: Book Insert Log

Logs all new book insertions for audit purposes.

```
DELIMITER //
CREATE TRIGGER book_insert_log
AFTER INSERT ON books
FOR EACH ROW
BEGIN
    INSERT INTO system_logs (table_name, operation, record_id,
       new_values)
    VALUES ('books', 'INSERT', NEW.book_id,
            CONCAT('Title: ', NEW.title, ', Price: ', NEW.price,
                   ', Stock: ', NEW.stock_quantity));
END//
DELIMITER ;
```

Listing 3.11: Book Insert Log Trigger

3.3.2. Trigger 2: Book Update Log

Tracks modifications to existing book records.

```
DELIMITER //
CREATE TRIGGER book_update_log
AFTER UPDATE ON books
FOR EACH ROW
BEGIN
    INSERT INTO system_logs (table_name, operation, record_id,
       old_values, new_values)
    VALUES ('books', 'UPDATE', NEW.book_id,
            CONCAT('Old Stock: ', OLD.stock_quantity, ', Old Price:
               ', OLD.price),
            CONCAT('New Stock: ', NEW.stock_quantity, ', New Price:
               ', NEW.price));
END//
DELIMITER ;
```

Listing 3.12: Book Update Log Trigger

3.3.3. Trigger 3: Order Stock Update

Automatically decreases book stock when orders are placed.

```
DELIMITER //
CREATE TRIGGER order_stock_update
AFTER INSERT ON order_details
FOR EACH ROW
BEGIN
    UPDATE books
```

```
    SET stock_quantity = stock_quantity - NEW.quantity
    WHERE book_id = NEW.book_id;

    INSERT INTO system_logs (table_name, operation, record_id,
        new_values)
    VALUES ('order_details', 'STOCK_UPDATE', NEW.book_id,
            CONCAT('Stock reduced by: ', NEW.quantity, ' for book_id
                : ', NEW.book_id));
END//
DELIMITER ;
```

Listing 3.13: Order Stock Update Trigger

3.3.4. Trigger 4: Order Total Calculation

Automatically calculates and updates order totals.

```
DELIMITER //
CREATE TRIGGER order_total_calculation
AFTER INSERT ON order_details
FOR EACH ROW
BEGIN
    UPDATE orders
    SET total_amount = (
        SELECT SUM(subtotal)
        FROM order_details
        WHERE order_id = NEW.order_id
    )
    WHERE order_id = NEW.order_id;

    INSERT INTO system_logs (table_name, operation, record_id,
        new_values)
    VALUES ('order_details', 'TOTAL_CALCULATION', NEW.order_id,
            CONCAT('Order total calculated for order_id: ', NEW.
                order_id,
                    ', Amount: ', (SELECT total_amount FROM orders
                        WHERE order_id = NEW.order_id)));
END//
DELIMITER ;
```

Listing 3.14: Order Total Calculation Trigger

3.3.5. Trigger 5: User Insert Log

Records new user registrations for security auditing.

```
DELIMITER //
CREATE TRIGGER user_insert_log
AFTER INSERT ON users
FOR EACH ROW
BEGIN
    INSERT INTO system_logs (table_name, operation, record_id,
       new_values)
    VALUES ('users', 'INSERT', NEW.user_id,
            CONCAT('Email: ', NEW.email, ', Type: ', NEW.user_type))
               ;
END//
DELIMITER ;
```

Listing 3.15: User Insert Log Trigger

3.4. Advanced SQL Queries

The system demonstrates various JOIN operations and complex queries for comprehensive data retrieval.

3.4.1. LEFT JOIN Example

Retrieves all books with their order statistics, including books never ordered.

```
SELECT b.title, b.price, b.stock_quantity,
       COALESCE(SUM(od.quantity), 0) as total_sold
FROM books b
LEFT JOIN order_details od ON b.book_id = od.book_id
GROUP BY b.book_id, b.title, b.price, b.stock_quantity;
```

Listing 3.16: LEFT JOIN - Books with Order Statistics

3.4.2. RIGHT JOIN Example

Shows all orders with customer information, ensuring no orders are missed.

```
SELECT o.order_id, o.order_date, o.total_amount,
       CONCAT(u.first_name, ' ', u.last_name) as customer_name
FROM books b
RIGHT JOIN order_details od ON b.book_id = od.book_id
RIGHT JOIN orders o ON od.order_id = o.order_id
RIGHT JOIN users u ON o.user_id = u.user_id;
```

Listing 3.17: RIGHT JOIN - Orders with Customer Details

3.4.3. FULL OUTER JOIN Simulation

MySQL doesn't support FULL OUTER JOIN directly, so we simulate it using UNION.

```
SELECT b.title, 'Book' as type, b.price as amount, NULL as
   order_date
FROM books b
UNION
SELECT CONCAT('Order #', o.order_id), 'Order' as type,
       o.total_amount as amount, o.order_date
FROM orders o;
```

Listing 3.18: FULL OUTER JOIN Simulation

3.5. Sample Data

The database is populated with comprehensive sample data to demonstrate all features effectively.

3.5.1. Users Data

- 1 Administrator account for system management
- 4 Customer accounts for testing various scenarios
- Email-based authentication system
- Different user types with appropriate permissions

3.5.2. Books Data

- 10 books across 6 categories
- Turkish and international literature
- Varied stock levels to demonstrate low-stock alerts
- Different price ranges for order value testing

3.5.3. Orders Data

- 5 sample orders with different statuses

- Multiple items per order to test calculations
- Various order values to demonstrate high-value filtering
- Historical order dates for timeline analysis

4. APPLICATION DEVELOPMENT

4.1. Flask Web Application Architecture

The web application follows the Model-View-Controller (MVC) architectural pattern, providing a clean separation of concerns and maintainable code structure.

4.1.1. Application Structure

Listing 4.1: Flask Application Structure

```
bookstore/
        app.py                          # Main Flask application
        templates/
            base.html                   # Base template with common elem
            login.html                  # User authentication interface
            customer_dashboard.html     # Customer main interface
            admin_dashboard.html        # Admin main interface
            admin_books.html            # Book management interface
            admin_users.html            # User management interface
            admin_orders.html           # Order management interface
            admin_views.html            # Database views interface
            admin_functions.html        # Function testing interface
            admin_triggers.html         # Trigger monitoring interfa
```

4.1.2. Database Connection Management

The application uses mysql-connector-python for efficient database operations with proper connection handling.

Listing 4.2: Database Connection Function

```
def get_db_connection():
    return mysql.connector.connect(
        host='localhost',
        user='root',
        database='bookstore_db'
    )
```

4.2. User Authentication System

4.2.1. Login Implementation

The authentication system supports email-based login with role-based access control.

Listing 4.3: Login Route Implementation

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']

        conn = get_db_connection()
        cursor = conn.cursor(dictionary=True)

        cursor.execute("SELECT * FROM users WHERE email = %s AND pa
                       (email, password))
        user = cursor.fetchone()

        if user:
            session['user_id'] = user['user_id']
            session['user_type'] = user['user_type']
            session['user_name'] = f"{user['first_name']} {user['las

            if user['user_type'] == 'admin':
                return redirect(url_for('admin_dashboard'))
            else:
                return redirect(url_for('customer_dashboard'))
        else:
            flash('Invalid email or password!')

        cursor.close()
        conn.close()

    return render_template('login.html')
```

4.2.2. Session Management

Flask sessions maintain user state throughout the application lifecycle with proper security measures.

4.3. Customer Interface

4.3.1. Book Browsing

The customer interface displays available books with real-time stock information and easy ordering functionality.

Listing 4.4: Customer Dashboard Implementation

```
@app.route('/customer')
def customer_dashboard():
    if 'user_id' not in session or session['user_type'] != 'custome
        return redirect(url_for('login'))

    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)

    # Get available books using the books_in_stock view
    cursor.execute("SELECT * FROM books_in_stock")
    books = cursor.fetchall()

    # Get customer orders with proper item aggregation
    cursor.execute("""
        SELECT o.*,
               COALESCE(GROUP_CONCAT(CONCAT(b.title, ' (', od.quant
                        SEPARATOR ', '), 'No items') as order_items
        FROM orders o
        LEFT JOIN order_details od ON o.order_id = od.order_id
        LEFT JOIN books b ON od.book_id = b.book_id
        WHERE o.user_id = %s
        GROUP BY o.order_id
        ORDER BY o.order_date DESC
    """, (session['user_id'],))
    orders = cursor.fetchall()
```

```
    cursor.close()
    conn.close()

    return render_template('customer_dashboard.html', books=books,
```

4.3.2. Order Placement

One-click ordering system with automatic stock validation and trigger execution.

Listing 4.5: Order Placement Implementation

```
@app.route('/place_order', methods=['POST'])
def place_order():
    if 'user_id' not in session or session['user_type'] != 'custome
        return redirect(url_for('login'))

    book_id = request.form['book_id']
    quantity = int(request.form.get('quantity', 1))

    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)

    try:
        # Validate stock availability
        cursor.execute("SELECT * FROM books WHERE book_id = %s", (b
        book = cursor.fetchone()

        if book and book['stock_quantity'] >= quantity:
            # Create order
            cursor.execute("""
                INSERT INTO orders (user_id, status, total_amount)
                VALUES (%s, 'pending', 0)
            """, (session['user_id'],))
            order_id = cursor.lastrowid

            # Add order details (triggers will handle stock and tot
            subtotal = book['price'] * quantity
            cursor.execute("""
                INSERT INTO order_details (order_id, book_id, quant
```

```
                VALUES (%s, %s, %s, %s, %s)
            """, (order_id, book_id, quantity, book['price'], subto

            conn.commit()
            flash('Order placed successfully!')
        else:
            flash('Insufficient stock available!')

    except Exception as e:
        conn.rollback()
        flash(f'Error: {str(e)}')

    cursor.close()
    conn.close()

    return redirect(url_for('customer_dashboard'))
```

4.4. Administrator Interface

4.4.1. Dashboard Overview

The admin dashboard provides comprehensive system statistics and quick access to management functions.

Listing 4.6: Admin Dashboard Implementation

```
@app.route('/admin')
def admin_dashboard():
    if 'user_id' not in session or session['user_type'] != 'admin':
        return redirect(url_for('login'))

    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)

    # Generate system statistics
    cursor.execute("SELECT COUNT(*) as total FROM books")
    total_books = cursor.fetchone()['total']

    cursor.execute("SELECT COUNT(*) as total FROM users WHERE user_t
```

```
    total_customers = cursor.fetchone()['total']

    cursor.execute("SELECT COUNT(*) as total FROM orders")
    total_orders = cursor.fetchone()['total']

    cursor.execute("SELECT SUM(total_amount) as revenue FROM orders
    total_revenue = cursor.fetchone()['revenue'] or 0

    cursor.close()
    conn.close()

    return render_template('admin_dashboard.html',
                           total_books=total_books,
                           total_customers=total_customers,
                           total_orders=total_orders,
                           total_revenue=total_revenue)
```

4.4.2. Book Management

Comprehensive CRUD operations for book inventory management.

Listing 4.7: Book Addition Implementation

```
@app.route('/admin/add_book', methods=['POST'])
def add_book():
    if 'user_id' not in session or session['user_type'] != 'admin':
        return redirect(url_for('login'))

    # Extract form data
    title = request.form['title']
    isbn = request.form['isbn']
    publication_year = request.form['publication_year']
    price = request.form['price']
    stock_quantity = request.form['stock_quantity']
    category_id = request.form['category_id']
    author_id = request.form['author_id']

    conn = get_db_connection()
    cursor = conn.cursor()
```

```
    try:
        # Insert book (triggers book_insert_log automatically)
        cursor.execute("""
            INSERT INTO books (title, isbn, publication_year, price
            VALUES (%s, %s, %s, %s, %s, %s)
        """, (title, isbn, publication_year, price, stock_quantity,

        book_id = cursor.lastrowid

        # Link author to book
        cursor.execute("""
            INSERT INTO book_authors (book_id, author_id) VALUES (%
        """, (book_id, author_id))

        conn.commit()
        flash('Book added successfully!')

    except Exception as e:
        conn.rollback()
        flash(f'Error: {str(e)}')

    cursor.close()
    conn.close()

    return redirect(url_for('admin_books'))
```

4.4.3. View Management Interface

Real-time display of database views with formatted output.

Listing 4.8: Views Interface Implementation

```
@app.route('/admin/views')
def admin_views():
    if 'user_id' not in session or session['user_type'] != 'admin':
        return redirect(url_for('login'))

    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)
```

```
    views_data = {}

    # Execute all five views
    cursor.execute("SELECT * FROM books_in_stock LIMIT 10")
    views_data['books_in_stock'] = cursor.fetchall()

    cursor.execute("SELECT * FROM customer_orders_summary")
    views_data['customer_orders_summary'] = cursor.fetchall()

    cursor.execute("SELECT * FROM author_books LIMIT 10")
    views_data['author_books'] = cursor.fetchall()

    cursor.execute("SELECT * FROM high_value_orders")
    views_data['high_value_orders'] = cursor.fetchall()

    cursor.execute("SELECT * FROM low_stock_books")
    views_data['low_stock_books'] = cursor.fetchall()

    cursor.close()
    conn.close()

    return render_template('admin_views.html', views_data=views_data
```

4.4.4. Function Testing Interface

Interactive testing environment for database functions with parameter input.

Listing 4.9: Function Testing Implementation

```
@app.route('/admin/test_function', methods=['POST'])
def test_function():
    if 'user_id' not in session or session['user_type'] != 'admin':
        return redirect(url_for('login'))

    function_name = request.form['function_name']
    param = request.form.get('param', '')

    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)
```

```
    try:
        # Execute function based on name and parameters
        if function_name == 'get_book_stock':
            cursor.execute(f"SELECT get_book_stock({param}) as resu
        elif function_name == 'calculate_order_total':
            cursor.execute(f"SELECT calculate_order_total({param})
        elif function_name == 'get_books_by_category':
            cursor.execute(f"SELECT get_books_by_category('{param}'
        elif function_name == 'get_author_book_count':
            cursor.execute(f"SELECT get_author_book_count({param})
        elif function_name == 'get_average_completed_order_value':
            cursor.execute("SELECT get_average_completed_order_valu

        result = cursor.fetchone()
        flash(f'Function Result: {result["result"]}')

    except Exception as e:
        flash(f'Error: {str(e)}')

    cursor.close()
    conn.close()

    return redirect(url_for('admin_functions'))
```

4.4.5. Trigger Monitoring

Real-time monitoring of trigger execution through system logs.

Listing 4.10: Trigger Monitoring Implementation

```
@app.route('/admin/triggers')
def admin_triggers():
    if 'user_id' not in session or session['user_type'] != 'admin':
        return redirect(url_for('login'))

    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)

    # Retrieve system logs organized by trigger type
    cursor.execute("""
```

```
        SELECT * FROM system_logs
        ORDER BY timestamp DESC
        LIMIT 50
    """)
    logs = cursor.fetchall()

    # Categorize logs by trigger type
    triggers_data = {
        'book_logs': [],
        'user_logs': [],
        'order_logs': [],
        'stock_logs': []
    }

    for log in logs:
        if log['table_name'] == 'books':
            triggers_data['book_logs'].append(log)
        elif log['table_name'] == 'users':
            triggers_data['user_logs'].append(log)
        elif log['table_name'] == 'order_details' and 'STOCK_UPDATE
            triggers_data['stock_logs'].append(log)
        elif log['table_name'] == 'order_details' and 'TOTAL_CALCUL
            triggers_data['order_logs'].append(log)

    cursor.close()
    conn.close()

    return render_template('admin_triggers.html', triggers_data=trig
```

4.5. Frontend Implementation

4.5.1. Responsive Design

The user interface employs responsive CSS design principles for cross-device compatibility.

Listing 4.11: Base Template Structure

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale
    <title>{% block title %}Bookstore Management System{% endblock %
    <style>
        * { margin: 0; padding: 0; box-sizing: border-box; }
        body { font-family: Arial, sans-serif; line-height: 1.6; ba
        .container { max-width: 1200px; margin: 0 auto; padding: 20
        .grid { display: grid; grid-template-columns: repeat(auto-f
        .card { background: white; border-radius: 8px; box-shadow: 
        .btn { background: #4CAF50; color: white; padding: 10px 20p
    </style>
</head>
<body>
    {% block content %}{% endblock %}
</body>
</html>
```

4.5.2. Interactive Features

JavaScript enhances user experience with dynamic content and real-time updates.

Listing 4.12: Tab Navigation Implementation

```
function showTab(tabName) {
    // Hide all tab contents
    document.querySelectorAll('.tab-content').forEach(content => {
        content.classList.remove('active');
    });

    // Remove active class from all tabs
    document.querySelectorAll('.tab').forEach(tab => {
        tab.classList.remove('active');
    });

    // Show selected tab content
    document.getElementById(tabName).classList.add('active');
    event.target.classList.add('active');
}
```

4.6. Error Handling and Validation

4.6.1. Form Validation

Client-side and server-side validation ensure data integrity and user experience.

Listing 4.13: Error Handling Example

```
try:
    cursor.execute(sql_query, parameters)
    conn.commit()
    flash('Operation completed successfully!')
except mysql.connector.Error as err:
    conn.rollback()
    if err.errno == errorcode.ER_DUP_ENTRY:
        flash('Duplicate entry - record already exists!')
    else:
        flash(f'Database error: {str(err)}')
except Exception as e:
    conn.rollback()
    flash(f'Unexpected error: {str(e)}')
finally:
    cursor.close()
    conn.close()
```

4.7. Security Considerations

4.7.1. SQL Injection Prevention

Parameterized queries prevent SQL injection attacks throughout the application.

4.7.2. Session Security

Secure session management with proper timeout and validation mechanisms.

4.7.3. Access Control

Role-based access control ensures users can only access appropriate functionality.