Court Kart: E-Commerce Platform

Database Implementation Project

Submitted by:
HADJ ARAB Adel
Student ID: 222231482117

May 15, 2025

TABLE OF CONTENTS

1 Introduction

Court Kart is a specialized e-commerce platform designed for basketball enthusiasts. The platform offers a wide range of basketball-related products including footwear, apparel, gear, and merchandise. This report documents the database implementation of the platform, focusing on its relational schema, stored procedures, triggers, and the overall integration with the web application.

The database design follows best practices for e-commerce applications, with particular attention to inventory management, order processing, and user account management. The implementation leverages MySQL's advanced features such as stored procedures and triggers to ensure data integrity, automate processes, and enhance application performance.

2 Database Design

2.1 Relational Schema and Relationships

Court Kart's database is structured around the following core entities and relationships:

- **users**: Stores user account information including authentication details, profile data, and role assignment (user/admin)
- **products**: Contains comprehensive product details including inventory levels, pricing information, categorization, and media assets
- **cart_items**: Represents the shopping cart functionality, linking users to products they intend to purchase
- **orders**: Records order transactions with status tracking and financial details
- **order_items**: Contains the line items within each order, maintaining historical pricing data
- **canceled_orders**: Records history and reasons for canceled transactions
- **logs**: Maintains a comprehensive audit trail of system operations

2.1.1 Key Relationships

- One user can have many cart items (one-to-many)
- One user can place many orders (one-to-many)
- One order contains many order items (one-to-many)
- Each product can appear in many carts and orders (many-to-many through junction tables)
- One canceled order references exactly one order (one-to-one)
- System logs can reference users and orders (many-to-one)

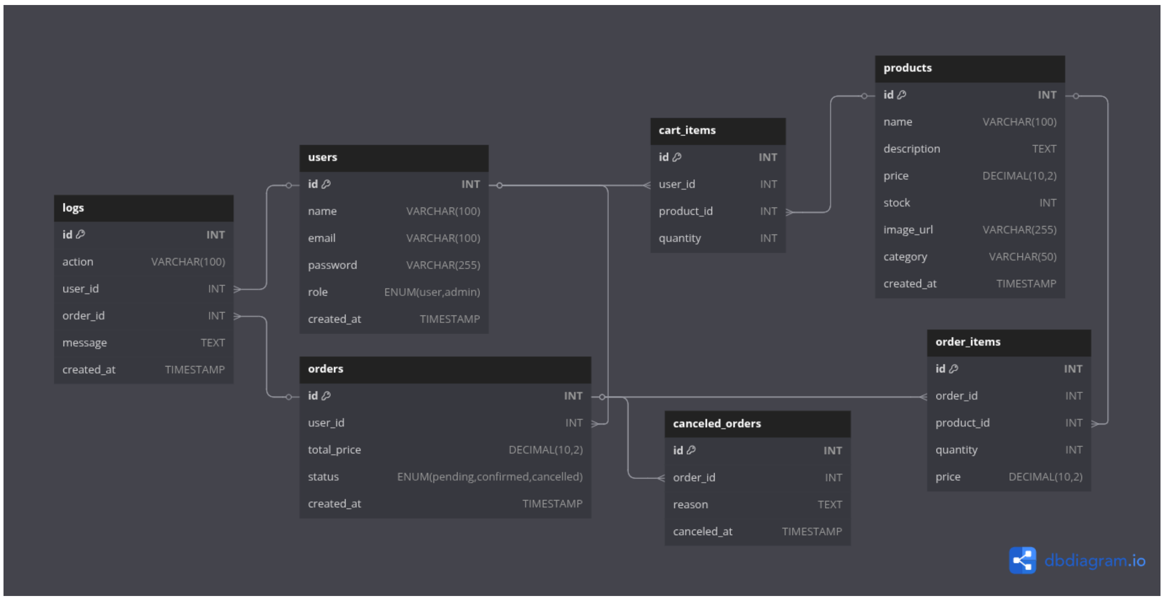

Figure 1: Entity-Relationship Diagram for Court Kart Database

2.2 Schema Optimization and Constraints

The database schema incorporates several optimization strategies:

- **Appropriate data types**: Selected to minimize storage requirements while ensuring data integrity
- **Foreign key constraints**: Implemented to maintain referential integrity across related tables
- **CHECK constraints**: Applied to validate data (e.g., ensuring discount values are between 0 and 1)
- **ENUM types**: Used for fields with predefined values to enforce data consistency
- **Indexing**: Applied to frequently queried columns to improve performance
- **Default values**: Provided where appropriate to simplify data insertion

3 Stored Procedures Implementation

Stored procedures are utilized to encapsulate complex business logic and improve application performance. The following key procedures have been implemented:

3.1 GetOrderDetails Procedure

This procedure retrieves comprehensive details about a specific order, joining multiple tables to provide a complete view of the transaction.

```
CREATE PROCEDURE GetOrderDetails (IN p_order_id INT)
BEGIN
    SELECT
        o.id AS order_id,
        o.created_at AS order_date,
        o.status,
        u.name AS customer_name,
        u.email AS customer_email,
        p.id AS product_id,
        p.name AS product_name,
        p.image_url,
        oi.quantity,
        oi.price AS unit_price,
        (oi.quantity * oi.price) AS subtotal,
        o.total_price AS total_amount
    FROM
        orders o
        JOIN users u ON o.user_id = u.id
        JOIN order_items oi ON o.id = oi.order_id
        JOIN products p ON oi.product_id = p.id
    WHERE
        o.id = p_order_id;
END
```

3.1.1 Implementation Notes

This procedure:

- Joins four tables to compile complete order information
- Calculates subtotals for each line item
- Returns both individual product details and aggregate order information
- Supports administrative and customer-facing order viewing

3.2 FinalizeOrder Procedure

This procedure handles the critical order checkout process, ensuring data consistency through transaction management.

```
CREATE PROCEDURE FinalizeOrder (
    IN p_order_id INT,
    IN p_user_id INT
)
BEGIN
    DECLARE v_order_exists INT;

    START TRANSACTION;

    SELECT COUNT(*) INTO v_order_exists
    FROM orders
    WHERE id = p_order_id AND user_id = p_user_id AND status = 'pending';

    IF v_order_exists = 1 THEN
        UPDATE orders
        SET status = 'confirmed'
        WHERE id = p_order_id;

        DELETE FROM cart_items
        WHERE user_id = p_user_id;

        INSERT INTO logs (action, user_id, order_id, message)
        VALUES ('CHECKOUT', p_user_id, p_order_id, 'Order finalized and cart emptied'
    );
```

```
        COMMIT;
    ELSE
        ROLLBACK;
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Invalid or non-pending order for this user';
    END IF;
END
```

3.2.1 Implementation Notes

This procedure:

- Validates order existence and ownership before processing
- Uses transactions to ensure atomicity of multi-step operations
- Clears the user's cart after successful order confirmation
- Logs the checkout action for audit purposes
- Provides error handling with descriptive messages

3.3 GetCustomerOrderHistory Procedure

This procedure retrieves a customer's complete order history with aggregated product information.

```
CREATE PROCEDURE GetCustomerOrderHistory (
    IN p_user_id INT
)
BEGIN
    SELECT
        o.id AS order_id,
        o.created_at AS order_date,
        o.total_price,
        o.status,
        COUNT(oi.id) AS item_count,
        GROUP_CONCAT(p.name SEPARATOR ', ') AS products
    FROM
        orders o
        LEFT JOIN order_items oi ON o.id = oi.order_id
        LEFT JOIN products p ON oi.product_id = p.id
    WHERE
        o.user_id = p_user_id
    GROUP BY
        o.id, o.created_at, o.total_price, o.status
    ORDER BY
        o.created_at DESC;
END
```

3.3.1 Implementation Notes

This procedure:

- Uses LEFT JOINs to include all orders, even if they have no items

- Employs GROUP_CONCAT to create a comma-separated list of product names
- Counts items per order for quick reference
- Orders results chronologically for intuitive display

4 Triggers Implementation

Triggers automate critical business logic and maintain data integrity across the database. The following key triggers have been implemented:

4.1 AfterOrderConfirmed Trigger

This trigger automatically updates product stock quantities when an order status changes to 'confirmed'.

```sql
CREATE TRIGGER AfterOrderConfirmed
AFTER UPDATE ON orders
FOR EACH ROW
BEGIN
    DECLARE v_done INT DEFAULT 0;
    DECLARE v_product_id INT;
    DECLARE v_quantity INT;
    DECLARE cur CURSOR FOR
        SELECT product_id, quantity FROM order_items WHERE order_id = NEW.id;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done = 1;

    IF OLD.status != 'confirmed' AND NEW.status = 'confirmed' THEN
        INSERT INTO logs (action, user_id, order_id, message)
        VALUES ('CHECKOUT', NEW.user_id, NEW.id, CONCAT('Order #', NEW.id, '
    confirmed'));

        OPEN cur;
        read_loop: LOOP
            FETCH cur INTO v_product_id, v_quantity;
            IF v_done THEN
                LEAVE read_loop;
            END IF;
            UPDATE products
            SET stock = stock - v_quantity
            WHERE id = v_product_id;
        END LOOP;
        CLOSE cur;

        INSERT INTO logs (action, user_id, order_id, message)
        VALUES ('PRODUCT_UPDATE', NEW.user_id, NEW.id, CONCAT('Stock updated for
    order #', NEW.id));
    END IF;
END
```

4.1.1 Implementation Notes

This trigger:

- Uses a cursor to iterate through all order items

- Only activates when an order status specifically changes to 'confirmed'
- Automatically decrements product stock levels
- Creates log entries before and after stock updates for audit purposes

4.2 BeforeOrderItemInsert Trigger

This trigger prevents adding items to orders if the requested quantity exceeds available stock.

```
CREATE TRIGGER BeforeOrderItemInsert
BEFORE INSERT ON order_items
FOR EACH ROW
BEGIN
    DECLARE available_stock INT;
    DECLARE v_user_id INT;

    SELECT stock INTO available_stock
    FROM products
    WHERE id = NEW.product_id;

    SELECT user_id INTO v_user_id
    FROM orders
    WHERE id = NEW.order_id;

    IF NEW.quantity > available_stock THEN
        INSERT INTO logs (action, user_id, order_id, message)
        VALUES ('PRODUCT_UPDATE', v_user_id, NEW.order_id,
                CONCAT('Failed to add product #', NEW.product_id, ' to order #', NEW.
    order_id,
                       ': Requested ', NEW.quantity, ', Available ', available_stock)
    );

        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Cannot insert order item: requested quantity exceeds
    available stock';
    END IF;
END
```

4.2.1 Implementation Notes

This trigger:

- Acts as a gatekeeper before inserting order items
- Compares requested quantity against available stock
- Creates detailed log entries when stock limitations are encountered
- Throws a descriptive error message when insertion fails

4.3 AfterOrderCancelled and LogCanceledOrder Triggers

These complementary triggers restore product stock when orders are canceled and maintain a history of canceled orders.

```
-- Only showing part of these triggers for brevity
CREATE TRIGGER AfterOrderCancelled
AFTER UPDATE ON orders
FOR EACH ROW
BEGIN
    -- ... existing code ...

    IF OLD.status != 'cancelled' AND NEW.status = 'cancelled' THEN
        -- Log the order cancellation
        INSERT INTO logs (action, user_id, order_id, message)
        VALUES ('ORDER_CANCEL', NEW.user_id, NEW.id, CONCAT('Order #', NEW.id, '
   canceled'));

        -- Restore product stock using cursor
        -- ... existing code ...
    END IF;
END
```

4.3.1 Implementation Notes

These triggers:

- Work together to handle order cancellation workflows
- Restore product stock quantities automatically
- Create audit trail entries
- Maintain historical records of cancellations for reporting

5 Product Data Structure and Management

The Court Kart database maintains a comprehensive product catalog with carefully structured data:

5.1 Product Categories

Products are organized into six primary categories:

- **Footwear**: Basketball shoes from major brands (Nike, Adidas, Under Armour)
- **Apparel**: Jerseys, shorts, and other clothing items
- **Gear**: Support equipment like ankle braces and protective gear
- **Accessories**: Complementary items like headbands, water bottles, and bags
- **Equipment**: Core basketball equipment like balls, pumps, and training aids
- **Merchandise**: Collectibles, posters, and fan items

5.2 Inventory Management

The platform implements robust inventory management through:

- Real-time stock tracking via database triggers

- Automatic stock deduction upon order confirmation
- Stock restoration when orders are canceled
- Prevention of orders that exceed available stock
- Comprehensive logging of inventory transactions

6 Logging and Auditing

The Court Kart database implements comprehensive logging capabilities:

6.1 Log Types

The system tracks various event types:

- User authentication events (USER_REGISTER, USER_LOGIN, USER_LOGOUT)
- Cart operations (CART_ADD, CART_REMOVE)
- Order processing (CHECKOUT, ORDER_UPDATE, ORDER_CANCEL)
- Inventory management (PRODUCT_ADD, PRODUCT_UPDATE, PRODUCT_DELETE)

6.2 Log Structure and Implementation

Each log entry contains:

- Action type (using ENUM for consistency)
- User ID (when applicable)
- Order ID (when applicable)
- Detailed message explaining the event
- Timestamp for chronological tracking

The logs table is indexed on action type and timestamp to support efficient querying for administrative reporting and troubleshooting.

7 Website Implementation Overview

The Court Kart web application layers on top of the database infrastructure to provide a complete e-commerce experience:

7.1 Key Features

- **Product browsing and search**: Allows customers to navigate the catalog with filtering options
- **User account management**: Supports registration, authentication, and profile management
- **Shopping cart system**: Provides temporary storage of selected items before checkout
- **Order management**: Facilitates the complete order lifecycle including tracking and history
- **Admin dashboard**: Offers inventory management, order processing, and reporting for administrators

7.2 Security Measures

The application implements several security measures:

- Parameterized queries to prevent SQL injection
- Password hashing for secure credential storage
- Input validation at application and database levels
- Role-based access control for administrative functions
- Comprehensive logging for audit and security monitoring

8 Conclusion

The Court Kart database implementation demonstrates best practices for e-commerce data management with special attention to:

- Data integrity through constraints, triggers, and stored procedures
- Performance optimization through appropriate indexing and query design
- Business logic encapsulation in database operations
- Security and auditing through comprehensive logging
- Scalable design to support future growth

The combination of a well-structured relational database with automated processes creates a robust foundation for the Court Kart e-commerce platform, enabling efficient operations and a seamless customer experience.