CS157A Final Project Report

Eric Zhao (016886922)

eric.zhao@sjsu.edu

Ellie Chong (017426656)

elaine.chong@sjsu.edu

Ruxin Xie (017060641)

ruxin.xie01@sjsu.edu

Jessica

jessica.fung@sjsu.edu

Rahul

rahul.champaneria@sjsu.edu

Table of Contents

Goals and Description of the Application

Goals

The Auction and Marketplace System aims to provide a user-focused platform where buyers can explore a diverse product catalog, participate in auctions, and efficiently manage orders. Key features include detailed product catalogs, real-time auctions, order tracking, and a review system for feedback on purchased items. Notifications keep users informed at crucial milestones, such as order placement, payment updates, shipping progress, and delivery, ensuring a transparent and intuitive experience tailored for buyers.

Description of the Application

The Auction and Marketplace System is a web-based application connecting to a MySQL database, designed to provide buyers with a user-friendly experience for browsing, purchasing, and participating in auctions. The backend is implemented using Node.js with the Express framework, while the frontend is built with React for dynamic user interactions.

The platform offers a pre-populated product catalog with detailed descriptions, allowing users to explore available items or engage in auctions. Key features include a streamlined purchasing process with secure checkout and order management. Buyers can track orders at every stage, from placement to delivery, with notifications providing updates on order placement, payment status, shipping initiation, and delivery. The system also includes an integrated review feature, enabling buyers to leave 1-5 star ratings and feedback on purchased items.

The application ensures flexibility with two payment methods - credit card and PayPal and shipping options, including standard and express delivery.

Application/Functional Requirements

Functional Requirements

- Registration and Login:
    - Users can register an account with a name, email, and password.
    - Upon registration, users are assigned the role of "Buyer."
    - Users can log in with their email and password for authentication.
- Browsing and Adding to Cart:
    - Buyers can browse a pre-populated product catalog displaying product details such as name, description, and price.
    - Users can add items to their cart using the “Add to Cart” button for easy purchase management.
- Auction Participation:
    - Buyers can browse auctioned items, place bids, and participate in real-time bidding.
    - The auction will automatically end when the auction’s end date has passed, assigning the auctioned item to the highest bidder’s shopping cart; since demonstrating this process is not feasible, a debug button is provided to end the auction manually.
- Order Management:
    - Buyers can place orders for products added to their cart or won in an auction.
    - Orders include Order ID, price, shipping address, shipping method, payment and shipping statuses, tracking number, and order date.
- Payment Options:
    - Buyers can securely complete their purchases using either credit card or PayPal as payment methods.
- Shipping Logistics:
    - Buyers can select between standard and express shipping methods during checkout.
    - Buyers can track shipping status and view updates such as orders shipped, in transit, and delivered.

- Notifications:
    - Notifications are sent to users for key events, including order placement, payment status updates, shipping initiation, and delivery.
    - Users can view and manage notifications from their accounts.
- Review System
    - Buyers can provide feedback by rating purchased products on a scale of 1-5 stars.
    - Reviews are restricted to one per purchased item in each order, ensuring a streamlined feedback process.

Scope and Exclusions (Subsection of Functional Requirements)

- Buyer and seller attributes are included in the database schema, but seller-specific functions (e.g., listing products) are not implemented in this phase.
- Product data is pre-populated for demonstration purposes, eliminating the need for seller actions.

Non-functional Requirements:

- Performance:
    - The system should load product catalogs and cart details within **2 seconds** under normal load conditions.
    - Auction updates, including the highest bid and time remaining, should reflect on the user interface within **1 second** of any change to ensure real-time accuracy.
- Maintainability:
    - The application must use a modular design to enable developers to add or update features without significant rewrites.
    - Bug fixes should be resolved within **24 hours**, supported by clear documentation for seamless maintenance and troubleshooting.
- Availability
    - The system should be available 24/7, with scheduled maintenance taking place during non-peak hours and notifications sent to users 48 hours in advance.

- Security

    - Regular security audits must be conducted, and any discovered vulnerabilities should be addressed within 48 hours.
- Usability
    - The user interface should be intuitive, allowing users to easily navigate between product listings, auction bidding, and account management.
    - Feedback for user actions, such as confirming a bid or purchase, should appear on-screen within **1 second** to ensure a responsive experience

Architecture

We developed a web-based Auction and Marketplace System using a client-server architecture integrated with a MySQL database, leveraging the knowledge and techniques acquired in our Database Management Systems course. The following sections detail the tools and technologies utilized in the development process.

Frontend: React, CSS, and JavaScript

The user interface is built with React, CSS, and JavaScript to ensure a smooth and interactive user experience. When users access the website, the frontend sends requests to the backend through RESTful APIs to fetch data for products, auctions, cart items, orders, reviews, and more. React's component-based architecture dynamically renders this data and manages state changes efficiently. The frontend features intuitive navigation through a dynamic Navbar, allowing users to seamlessly access key sections such as Products, Auctions, ShoppingCart, Orders, Notifications, and Reviews. User interactions such as adding items to the cart, placing bids, or submitting reviews trigger API calls, dynamically retrieving and displaying results to provide an engaging and seamless browsing experience.

Backend: Node.js and Express Framework

The backend is built using Node.js and the Express framework, creating a powerful and flexible server-side environment. The backend handles all HTTP requests from the frontend, processes them, and communicates with the MySQL database to retrieve, insert,

update, or delete data as required. For example, when users view products, place bids, or manage orders, the backend constructs and executes SQL queries to interact with the database. The server then formats the results as **JSON responses** and sends them back to the frontend, ensuring fast and efficient communication between the client and server.

Database: MySQL

The MySQL database stores all marketplace and auction-related data, structured in tables with attributes such as Product_ID, User_ID, Auction_ID, Order_ID, and more. When a request is made to the backend, SQL queries are executed to retrieve the relevant data from the database. For example, when a user views products, participates in auctions, or places an order, the backend constructs SQL queries to match the criteria and fetch the corresponding data, such as available products, highest bids, or order details. The database executes these queries and returns the results to the backend, which processes and formats them for display on the front end. This interaction ensures that the marketplace data is efficiently managed and retrieved based on user actions, providing a seamless and interactive experience for the users.

By integrating these components, the system is designed to provide an engaging and intuitive platform for users to browse products, participate in auctions, place orders, and track their purchases. This architecture supports the goal of delivering a robust, user-focused e-commerce experience, where the client, server, and database work in harmony to provide real-time updates and seamless transactions.

ER Data Model

The ERD represents a well-structured database design for the Marketplace and Auction System, organizing data into distinct tables with clear relationships. It efficiently manages core functionalities such as product listings, auctions, orders, reviews, payments, and shipping updates, ensuring accurate and scalable data handling for a seamless platform experience.

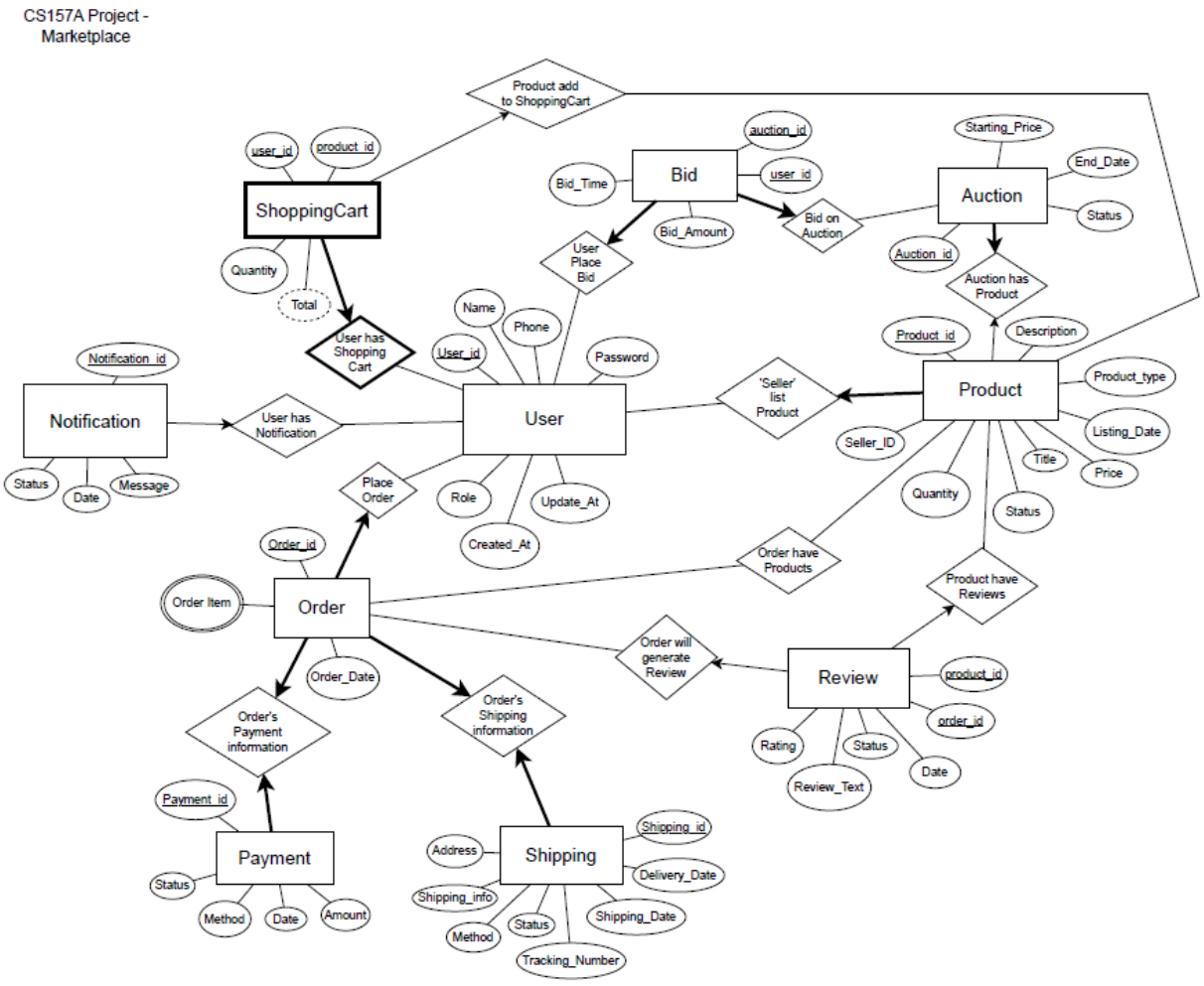

Figure 1. ER Diagram

Database Design

The database design for the Auction and Marketplace System is structured to manage key platform functionalities efficiently, including product listings, auctions, orders, payments, reviews, and notifications. This design follows normalization principles to ensure data integrity, minimize redundancy, and facilitate seamless data retrieval for dynamic user interactions.

Normalization and Data Integrity

The database effectively uses primary keys (e.g., User_ID, Product_ID, Order_ID, Auction_ID, Bid_ID) to uniquely identify each record in their respective tables. This ensures that every entity

(such as a user, product, or auction) is distinctly identifiable, reducing the risk of duplication and maintaining data integrity.

By separating different aspects of the system into dedicated tables (e.g., User, Product, Auction, Orders), the design adheres to normalization principles. Each table stores information specific to a single entity, avoiding redundancy and creating an organized structure that facilitates efficient data handling. For detailed attributes, functional dependencies, and normalization steps, please refer to the appendix.

Relationships and Foreign Keys
For a comprehensive overview of these relationships, please refer to the ER Diagram.

Major Design Decisions

Normalization of Tables: Core entities such as users, products, orders, and auctions are organized into separate tables, reducing redundancy and maintaining data integrity.

Use of Foreign keys: Relationships between tables are established using foreign keys to centralize and maintain consistency in the data.

Management of Many-to-Many Relationships: The Bid table connects users to auctions, enabling multiple users to bid on the same auction and each user to participate in multiple auctions. This structure ensures flexibility and accurate data representation without redundancy.

Modularity for Future Expansion: The design supports adding new features or entities without requiring significant structural changes.

Optimization for Query Performance: Indexing of primary and foreign keys ensures fast and efficient data retrieval.

Incorporation of Business Rules: Triggers and constraints enforce rules such as limiting reviews to one per product per order and automatically closing auctions.

Implementation Details

This section outlines the technical configurations and software implementations used in developing the Marketplace and Auction System. The application leverages a Node.js server, a MySQL database, and a front-end designed with React, CSS, and JavaScript.

Backend Configuration

- Server Setup: The backend is developed using **Node.js** with the **Express** framework, enabling efficient handling of HTTP requests and middleware functionality. The server is initiated by running node app.js, which sets it to listen on **port 5000** for incoming requests.
- Database Connection: The application uses **MySQL** as the database management system. The connection to the market database is established using the **mysql2** Node.js client, with credentials and host details configured for the development environment.
- API Routes:
    - The backend exposes various API endpoints to handle operations such as:
        - **/products:** Retrieves the list of all products, including their details and availability.
        - **/products/:id:** Fetches details for a specific product based on its ID.
        - **/orders:** Handles order creation and retrieval.
        - **/bids:** Processes auction bids, validating and updating the database with the latest bid.

Frontend Design

- User Interface: The frontend is developed using **React**, styled with CSS, and enhanced with JavaScript. It provides a responsive and dynamic interface, ensuring accessibility across devices and screen sizes. The interface features a navigation bar that offers easy access to key sections like **Products**, **Auctions**, **Shopping Cart**, **Orders**, **Notifications**, and **Reviews**.

- Interactive Elements: User interactions such as adding items to the cart, placing bids, or submitting reviews trigger RESTful API calls to the backend. The data is dynamically fetched and displayed, allowing for real-time updates without reloading the page.

Database Integration

- Schema: The database schema supports the core functionalities of the system, including product listings, auctions, orders, reviews, and notifications. Key tables include Product, Auction, Orders, OrderItems, User, and Notification.
- Normalization: The database is normalized to **Boyce-Codd Normal Form (BCNF)** to reduce redundancy and ensure data integrity. Relationships between tables, such as Orders and OrderItems, are managed using foreign keys to maintain referential integrity. For complete normalization, please refer to the appendix.

Demonstration of example system run

To set up the system, start by ensuring that both MySQL Workbench and Node.js are installed. If MySQL Workbench is not already installed, use Homebrew to install it with the command brew install mysql, and start the MySQL server using brew services start mysql. Confirm the server is running by connecting via the terminal with mysql -u root -p. Additionally, download and install the latest LTS version of Node.js from the official website, and verify the installation using the commands node -v and npm -v.

Next, clone the project repository by running git clone <repository-url> in your terminal, then navigate to the project folder using cd <repository-folder>. To set up the database, open MySQL Workbench and execute the provided SQL scripts in order. Start with db_createtable.sql to create the necessary tables, followed by db_trigger.sql to set up triggers, db_insert.sql to populate the tables with sample data, and db_indexing.sql to optimize the database with indexing. In MySQL Workbench, navigate to the "Users and Privileges" section to create a new user account with the username root and password 1234567. Assign this user full privileges for the market database to ensure seamless backend communication.

After setting up the database, configure the backend by updating the server.js file with the database credentials for the root user. Install the required Node.js dependencies by running npm

install in the project directory. Once the dependencies are installed, start the backend server by navigating to the folder containing server.js and executing node server.js. A successful connection to the database will display the message “Connected to MySQL Database.”

In a new terminal, start the frontend development server by running npm start. After both the backend and frontend are running, access the application by opening a browser and navigating to http://localhost:3306.

By following these steps, the system can be fully set up and its core functionalities demonstrated, highlighting its ease of use and integration of backend and frontend technologies. This setup also addresses the project’s non-functional requirements, such as query optimization through indexing and triggers.

```
const express = require('express');
const mysql = require('mysql2');
const cors = require('cors');

const app = express();
app.use(cors());
app.use(express.json()); // Modern way to handle JSON requests

// MySQL Connection
const db = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: '1234567', // Replace with your MySQL password
    database: 'market',
    port: 3306
});

db.connect((err) => {
    if (err) {
        console.error('Error connecting to the database:', err);
        return;
    }
    console.log('Connected to MySQL Database.');
});
```

Figure 2. This figure displays the backend configuration for the web app System.

Conclusion

The Marketplace and Auction System provides a strong foundation for an efficient and user-friendly platform where buyers can browse products, participate in auctions, and manage their orders. Future improvements include integrating seller functionalities, enhancing auction features with real-time notifications, and incorporating analytics to personalize the user experience and track activity trends.

This project demonstrated the effectiveness of using MySQL for managing complex workflows, ensuring data integrity, and automating processes like notifications and inventory updates. The experience highlights the importance of structured database systems for scalability and efficiency, while also paving the way for future expansions, such as predictive analytics and AI-driven features, to enhance the platform further.

Appendix

User :

Attribute : user_id (PK), name, email, password, phone, role, created_at, updated_at

relation : user can be buyer/seller (seller include all function of buyer, consider buyer/seller as normal user/Premium user concept), user can add product to shopping cart. user can place order. user can get notification. user can write review for a product that was ordered. user can bid on products that is in auction. user login use email and password.

FD : user_id → name, email, password, phone, role, created_at, updated_at
email, password → user_id

atomic value : password, phone, role, created_at, updated_at
name can be break into firstname, lastname. since name isn't used anywhere firstname/lastname separately, therefore name is atomic value.
email is candidate key. It's possible to use email as primary key, however, email can be changed. ① if email as user PK and changed, then every table use user PK as FK will need modified. for simplicity, user entity will use user_id as PK, user id will not change upon register to avoid possible database inconsistence. ② if email as user PK and changed, it will increase complexity of backend implementation. Backend have to update multiple tables and might possibly involve data loss.

1NF : user (user_id, name, email, password, phone, rde, created-at, update_at)

2NF : every non-candidate key attributes are functionally dependent on user_id.

3NF : No non-candidate key attribute is transitively dependent on email.

BCNF : email, password → user_id does not violate BCNF, because email, password together are composite candidate key.

Therefore : user (user_id, name, email, password, phone, role, created_at, updated_at) is in Boyce-Codd Normal Form.

product:

Attribute: product_id (PK), title, description, price, seller_id (FK), quantity, product_type, listing_date, status

relation: product could be directly purchase or auction, it determin with product_type product will not open for purchase or auction if status is sold.

FD: product_id → title, description, price, seller_id, quantity, product_type, listing_date, status

product_id, product_type, status → title, description, price, quantity

atomic value: title, description, price, seller_id, quantity, product_type, listing_date, status

1NF: already in 1NF

2NF: every non-candidate key attributes are fully functional dependent on product_id

3NF: No non-candidate key attributes is transitively dependent on candidate key. seller_id is a user, since seller could list multiple product, all other attribute could be repeat or same. therefore, seller_id with other attribute can be a composite candidate key.

BCNF: every functional dependency contain product_id. no other non-candidate key determine any attributes.

Therefore: product(product_id (PK), title, description, price, seller_id (FK), quantity, product_type, listing_date, status)

Shopping cart:

Attributes: user_id, product_id, quantity (user_id, product_id: PK)

Candidate key: (user_id, product_id) as composite primary key.
Shopping cart is weak entity, without user_id and product_id, shopping cart would not be exist and meaningless.

Relation: one shopping cart associate with one user. one user can place multiple product in shopping cart but same user won't have multiple same product, alternatively shopping cart will increase product quantities.

FD: user_id, product_id → quantity

atomic value: user_id, product_id, quantity

1NF: already in 1NF.

2NF: quantity is fully dependent on composite candidate key (user_id, product_id)

3NF: No non-candidate key attribute is transitively dependent on a candidate key (user_id, product_id) is the only composite primary key.

BCNF: no other functional dependency exist. already in BCNF

Therefore: Shopping cart (user_id, product_id, quantity)
(user_id, product_id: PK)

Order:

Attribute: order_id (PK), user_id (FK), order_date, orderItems, payment_status, payment_method, payment_date, payment_amount, shipping_address, shipping_info, shipping_method, shipping_status, tracking_number, shipping_date, delivery_date

Relation: use order_id to display information for a user.

note: since payment_status, payment_method, payment_date, payment_amount could be a set of information, separate into another table. same reason for information about shipping, separate into another table. orderItems will contain which product and quantity of product, so it is a multi-value attribute. separate into another table.

Attribute: order_id (PK), user_id (FK), order_date

FD: order_id → user_id, order_date

atomic value: order_id, user_id, order_date

1NF: already in 1NF

2NF: every non-candidate key attributes are fully functional dependent on order_id

3NF: no transitive dependency

BCNF: no non-candidate key function determine another attribute.

Therefore: order (order_id (PK), user_id (FK), order_date)

OrderItems:

Attribute: Order_id, product_id, quantity

relation: orderItems is multi-value attribute of order entity.

composite primary key: order_id, product_id

FD: order_id, product_id → quantity

atomic value: order_id, product_id, quantity

1NF: already in 1NF

2NF: quantity is fully functional dependent on primary key (ord_id, product_id)

3NF: No transitive dependency

BCNF: no non-candidate key attribute functional determine another attribute

Therefore, OrderItems (order_id, product_id, quantity)

PK

payment :

Attribute: payment_id, order_id, payment_status, payment_method, payment_date, payment_amount

FD: payment_id → order_id, payment_status, payment_method, payment_date, payment_amount

relation: once a user place an order, payment_method will be set to user's selection, won't change later. payment_status will set to pending by default, upon paid (bank send info about it, for simplicity, we make a debug button to set payment to paid) payment_date will be update.

atomic value: payment_id, order_id, payment_status, payment_method, payment_date, payment_amount.

payment_status original design with pending, paid, cancelled. for simplicity, we delete cancelled and not consider situation that payment reject to cause order cancelled.

1NF: already in 1NF

2NF: no partial dependency

3NF: no transitive dependency

BCNF: no non-candidate key attribute functional determine another attribute.

Therefore, payment (payment_id (PK), order_id (FK), payment_status, payment_method, payment_date, payment_amount)

Shipping:

Attribute: shipping_id, order_id, shipping_address, shipping_info, shipping_method, shipping_status, tracking_number, shipping_date, delivery_date

FD: shipping_id → order_id, shipping_address, shipping_info, shipping_method, shipping_status, tracking_number, shipping_date, delivery_date

relation: similar to payment.

atomic value: shipping_id, order_id, shipping_info, shipping_method, shipping_status, tracking_number, shipping_date, delivery_date

shipping_address could be break down into street, city, zip code. however, there isn't use case for street, city, zipcode separately. Therefore, we will keep shipping_address as whole for project simplicity.

1NF: already in 1NF

2NF: no partial dependency

3NF: no transitive dependency

BCNF: no non-candidate key attribute functional determine another attribute

Therefore, shipping (shipping_id (PK), order_id (FK), shipping_address, shipping_info, shipping_method, shipping_status, tracking_number, shipping_date, delivery_date)

Auction:

Attribute: auction_id, product_id, starting_price, end_date, auction_status

relation: for simplicity of our project, we changed user freely add product as auction items to upon product record with product_type is auction will trigger DBMS to add new auction record. Auction end_date will be set for 3 days in trigger. For presentation purpose, we made a debug button to end auction, so we can show case our project.

FD: auction_id → product_id, starting_price, end_date, auction_status

atomic value: auction_id, product_id, starting_price, end_date, auction_status

1NF: already in 1NF

2NF: no partial dependency

3NF: no transitive dependency

BCNF: No non-candidate key attribute functional determine another attribute

Therefore, Auction (PK auction_id, FK product_id, starting_price, end_date, auction_status)

Bid:

Attribute: auction_id, user_id, bid_amount, bid_time

relation: user bid in auction, database will put product into winning user's shopping cart when auction end.

composite primary key: (auction_id, user_id)

FD: auction_id, user_id → bid_amount, bid_time

atomic value: auction_id, user_id, bid_amount, bid time

1NF: already in 1NF

2NF: no partial dependency

3NF: no transitive dependency

BCNF: No non-candidate key attribute functional determine another attribute

Therefore, Bid (auction_id, user_id, bid_amount, bid_time)

PK

Review:

Attribute: user_id, product_id, order_id, rating, review_text, review_status, review_date

relation: after order complete which means product delivered in shipping table, a review for each product will be created as pending review. since it is complicated to implement in the backend, I wrote trigger to create review after shipping status update to delivered.

Note: since order table has user_id as attribute, remove user_id.

composite primary key: (product_id, order_id)

FD: product_id, order_id → rating, review_text, review_status, review_date

atomic value: product_id, order_id, rating, review_text, review_status, review_date

1NF: already in 1NF

2NF: no partial dependency

3NF: no transitive dependency

BCNF: No non-candidate key attribute functional determine another attribute

Therefore, Review (product_id, order_id, rating, review_text, review_status, review_date)

PK

Notification:

Attribute: notification_id (PK), user_id (FK), message, notification_date, status

relation: notification contain information about user's order, it handle by DBMS trigger, after order created, payment status change, shipping status change, trigger will create a new notification with unread status to notify user.

FD: notification_id → user_id, message, notification_date, status

atomic value: notification_id, user_id, message, notification_date, status

1NF: already in 1NF

2NF: no partial dependency

3NF: no transitive dependency

BCNF: No non-candidate key attribute functional determine another attribute

Therefore, Notification (notification_id (PK), user_id (FK), message, notification_date, status)