Online Auction System Report

Pranjal Jayesh Prajapati, CS23BTECH11048
Roshan Y Singh, CS23BTECH11052
Sujal Banshilal Meshram, CS23BTECH11060

Contents

Introduction **2**

Schema Overview **2**

ER Diagram for the Schema **4**

Functional Dependencies (FDs) **5**

Indexing, Procedures, and Trigger Explanations **10**

Frontend Report **15**

Backend Report **17**

Introduction

Approach to Solving the Problem

This report analyzes the functional dependencies (FDs) for an Online Auction System schema and demonstrates that each table is in Third Normal Form (3NF). It also explains how the decomposition is dependency preserving across the entire schema. A new table, `automated_bids`, has been added to record bids placed automatically. All functional dependencies, including those for weak or associative entities, are identified and analyzed.

Schema Overview

Below is a simplified listing of the main tables with their attributes (primary keys are shown in curly braces):

- **users**({user_id}, username, password, email, address, mobile_number, is_admin, created_at)
- **items**({item_id}, seller_id, title, description, image_path, starting_bid, current_highest_bid, current_highest_bidder, created_at)
- **auctions**({auction_id}, item_id, start_time, end_time, auction_status)
- **bids**({bid_id}, auction_id, buyer_id, bid_amount, bid_time)
- **automated_bids**({bid_id}, auction_id, buyer_id, bid_amount, bid_time)
- **auction_participants**({auction_id, user_id}, user_role, joined_at)
- **transactions**({transaction_id}, auction_id, transaction_date)
- **deliveries**({delivery_id}, transaction_id, delivery_status, delivery_date)
- **payments**({payment_id}, transaction_id, payment_method, payment_date, payment_status)
- **admin_update_log**({log_id}, old_time, new_time, changed_at, changed_by)
- **admin_delete_log**({log_id}, auction_id, changed_at, changed_by)
- **reviews**({review_id}, transaction_id, rating, review_date)

Foreign keys link tables as follows:

- `items.seller_id` and `items.current_highest_bidder` → `users.user_id`
- `auctions.item_id` → `items.item_id`
- `bids.auction_id` → `auctions.auction_id`, `bids.buyer_id` → `users.user_id`
- `automated_bids.auction_id` → `auctions.auction_id`, `automated_bids.buyer_id` → `users.user_id`
- `auction_participants.auction_id` → `auctions.auction_id`, `auction_participants.user_id` → `users.user_id`

- transactions.auction_id → auctions.auction_id
- deliveries.transaction_id → transactions.transaction_id
- payments.transaction_id → transactions.transaction_id
- admin_update_log.changed_by → users.user_id
- admin_delete_log.auction_id → auctions.auction_id, admin_delete_log.changed_by → users.user_id
- reviews.transaction_id → transactions.transaction_id

ER Diagram for the Schema

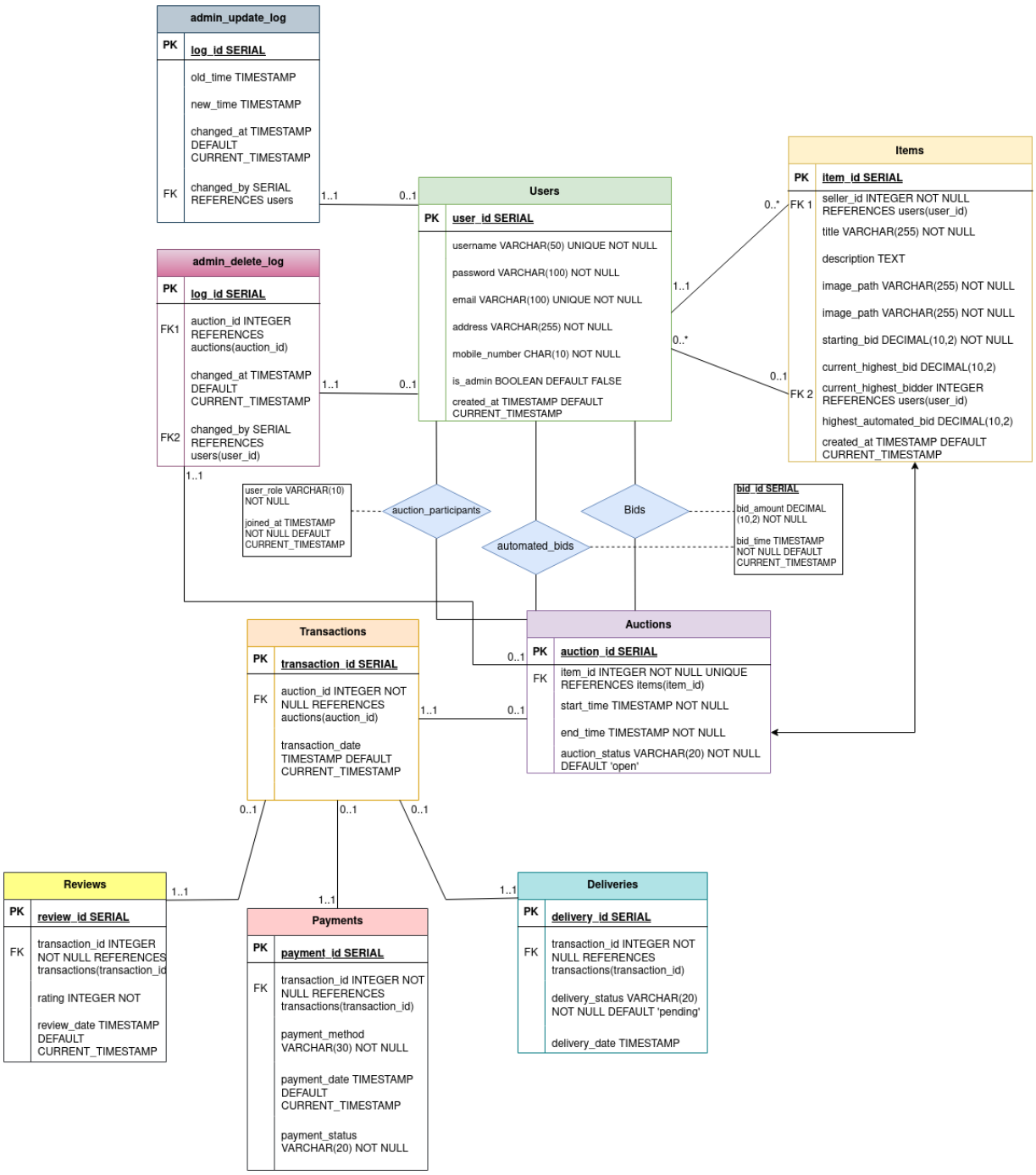

Figure 1: Entity-Relationship Diagram for Online Auction System

Functional Dependencies (FDs)

In this section, we list both trivial and non-trivial functional dependencies for each table, noting candidate keys and superkeys.

Users

Attributes: {user_id, username, password, email, address, mobile_number, is_admin, created_at}

Primary Key: {user_id}
Candidate Keys: {username}, {email}.

$$\{\text{user\_id}\} \rightarrow \{\text{username, password, email, address, mobile\_number, is\_admin, created\_at}\}$$

$$\{\text{username}\} \rightarrow \{\text{user\_id, password, email, address, mobile\_number, is\_admin, created\_at}\}$$

$$\{\text{email}\} \rightarrow \{\text{user\_id, username, password, address, mobile\_number, is\_admin, created\_at}\}$$

Items

Attributes: {item_id, seller_id, title, description, image_path, starting_bid, current_highest_bid, current_highest_bidder, created_at}

Primary Key: {item_id}

$$\{\text{item\_id}\} \rightarrow \{\text{seller\_id, title, description, image\_path, starting\_bid, current\_highest\_bid, current\_highest\_bidder, created\_at}\}$$

Auctions

Attributes: {auction_id, item_id, start_time, end_time, auction_status}

Primary Key: {auction_id}

$$\{\text{auction\_id}\} \rightarrow \{\text{item\_id, start\_time, end\_time, auction\_status}\}$$

Bids

Attributes: {bid_id, auction_id, buyer_id, bid_amount, bid_time}

Primary Key: {bid_id}

$$\{\text{bid\_id}\} \rightarrow \{\text{auction\_id, buyer\_id, bid\_amount, bid\_time}\}$$

Automated_bids

Attributes: {bid_id, auction_id, buyer_id, bid_amount, bid_time}

Primary Key: {bid_id}

$$\{\text{bid\_id}\} \rightarrow \{\text{auction\_id},\ \text{buyer\_id},\ \text{bid\_amount},\ \text{bid\_time}\}$$

Auction_Participants

Attributes: {auction_id, user_id, user_role, joined_at}

Primary Key: {auction_id, user_id}

$$\{\text{auction\_id},\ \text{user\_id}\} \rightarrow \{\text{user\_role},\ \text{joined\_at}\}$$

Transactions

Attributes: {transaction_id, auction_id, transaction_date}

Primary Key: {transaction_id}

$$\{\text{transaction\_id}\} \rightarrow \{\text{auction\_id},\ \text{transaction\_date}\}$$

Deliveries

Attributes: {delivery_id, transaction_id, delivery_status, delivery_date}

Primary Key: {delivery_id}

$$\{\text{delivery\_id}\} \rightarrow \{\text{transaction\_id},\ \text{delivery\_status},\ \text{delivery\_date}\}$$

Payments

Attributes: {payment_id, transaction_id, payment_method, payment_date, payment_status}

Primary Key: {payment_id}

$$\{\text{payment\_id}\} \rightarrow \{\text{transaction\_id},\ \text{payment\_method},\ \text{payment\_date},\ \text{payment\_status}\}$$

Admin_Update_Log

Attributes: {log_id, old_time, new_time, changed_at, changed_by}

Primary Key: {log_id}

$$\{\text{log\_id}\} \rightarrow \{\text{old\_time, new\_time, changed\_at, changed\_by}\}$$

Admin_Delete_Log

Attributes: {log_id, auction_id, changed_at, changed_by}

Primary Key: {log_id}

$$\{\text{log\_id}\} \rightarrow \{\text{auction\_id, changed\_at, changed\_by}\}$$

Reviews

Attributes: {review_id, transaction_id, rating, review_date}

Primary Key: {review_id}

$$\{\text{review\_id}\} \rightarrow \{\text{transaction\_id, rating, review\_date}\}$$

Cross-Table Dependencies

In addition to the dependencies that hold within each individual table, our schema also preserves cross-table dependencies through foreign key relationships. Such dependencies can be viewed as chains of functional dependencies spanning two or more tables. Below, we list these chains:

- **Auctions to Items:**
  In **auctions**, the primary key {auction_id} functionally determines {item_id}. In **items**, the primary key {item_id} determines attributes such as {current_highest_bid}, as well as:
  $$\left\{ \begin{array}{l} \text{seller\_id, title, description, image\_path,} \\ \text{starting\_bid, current\_highest\_bidder, created\_at} \end{array} \right\}$$
  Thus, the dependency
  $$\{\text{auction\_id}\} \rightarrow \{\text{item\_id}\} \rightarrow \{\text{current\_highest\_bid}\}$$
  is preserved.

- **Bids to Auctions to Items:**
  In **bids**, {bid_id} determines {auction_id}. Then, in **auctions**, {auction_id} determines {item_id}. Finally, in **items**, {item_id} determines attributes such as {title, starting_bid, current_highest_bid}. Hence, the chain
  $$\{\text{bid\_id}\} \rightarrow \{\text{auction\_id}\} \rightarrow \{\text{item\_id}\} \rightarrow \{\text{title, starting\_bid, current\_highest\_bid}\}$$
  is preserved.

- **Automated Bids to Auctions to Items:**
  Similarly, in **automated_bids**, {bid_id} determines {auction_id}, which in turn determines {item_id} in **auctions**, and then the attributes of the item in **items** are determined. Thus, automated bids also inherit the dependency chain.

- **Transactions to Auctions to Items:**
  In **transactions**, {transaction_id} determines {auction_id}. Then, as before, {auction_id} in **auctions** determines {item_id}, which in **items** determines various item attributes. Therefore:

  $$\{\text{transaction\_id}\} \to \{\text{auction\_id}\} \to \{\text{item\_id}\} \to \{\text{item attributes}\}$$

  is preserved. Thus, the real world dependencies of the from transaction_id to any attribute in auctions or items table is preserved.

- **Deliveries to Transactions to Auctions to Items:**
  In **deliveries**, {delivery_id} determines {transaction_id}. The dependency chain continues through **transactions** (i.e., {transaction_id} $\to$ {auction_id}) and then through **auctions** to **items**. Thus, attributes of the associated item are indirectly determined by the delivery's primary key.

- **Payments to Transactions to Auctions to Items:**
  Similarly, {payment_id} in **payments** determines {transaction_id}, and the chain continues as above.

- **Reviews to Transactions to Auctions to Items:**
  In **reviews**, {review_id} determines {transaction_id}, which then maps to an auction and subsequently to the corresponding item attributes.

- **Admin Delete Log to Auctions to Items:**
  In **admin_delete_log**, {log_id} determines {auction_id}. Since {auction_id} in **auctions** determines {item_id} (and hence attributes of the item), the dependency chain is preserved.

Dependency Preservation & 3NF Justification

Dependency Preservation Each table's non-trivial FDs are fully contained within that table. There are no cross-table FDs that have been split during normalization. Foreign keys ensure referential integrity without introducing FDs that violate 3NF. Thus, the decomposition is *dependency preserving*.

Third Normal Form (3NF) Criteria A relation R is in 3NF if for every functional dependency $X \to Y$ in R, either:

1. The dependency is trivial ($Y \subseteq X$)
2. X is a superkey, or
3. Each attribute in $Y - X$ is contained within a candidate key.

Additionally, 3NF requires that there be no partial dependencies or transitive dependencies among non-key attributes.

3NF for Each Table

users:

- Primary key: $\{user\_id\}$, alternate keys: $\{username\}$ and $\{email\}$.
- Every FD has a key on the left; hence, `users` is in 3NF.

items:

- Primary key: $\{item\_id\}$ determines all other attributes.
- No partial or transitive dependencies exist; thus, `items` is in 3NF.

auctions:

- Primary key: $\{auction\_id\}$ determines all other attributes.
- `auctions` is in 3NF.

bids:

- Primary key: $\{bid\_id\}$ determines $\{auction\_id, buyer\_id, bid\_amount, bid\_time\}$.
- `bids` is in 3NF.

automated_bids:

- Primary key: $\{bid\_id\}$ determines $\{auction\_id, buyer\_id, bid\_amount, bid\_time\}$.
- `automated_bids` is in 3NF.

auction_participants:

- Composite primary key: $\{auction\_id, user\_id\}$ determines $\{user\_role, joined\_at\}$.
- `auction_participants` is in 3NF.

transactions:

- Primary key: $\{transaction\_id\}$ determines $\{auction\_id, transaction\_date\}$.
- `transactions` is in 3NF.

deliveries:

- Primary key: $\{delivery\_id\}$ determines $\{transaction\_id, delivery\_status, delivery\_date\}$.
- `deliveries` is in 3NF.

payments:

- Primary key: $\{payment\_id\}$ determines $\{transaction\_id, payment\_method, payment\_date, payment\_status\}$.
- `payments` is in 3NF.

admin_update_log:

- Primary key: $\{log\_id\}$ determines $\{old\_time, new\_time, changed\_at, changed\_by\}$.
- `admin_update_log` is in 3NF.

admin_delete_log:

- Primary key: $\{log\_id\}$ determines $\{auction\_id, changed\_at, changed\_by\}$.
- `admin_delete_log` is in 3NF.

reviews:

- Primary key: $\{review\_id\}$ determines $\{transaction\_id, rating, review\_date\}$.
- `reviews` is in 3NF.

Entire Schema in 3NF

Since each table individually satisfies 3NF (every non-trivial FD has a superkey on the left and there are no partial or transitive dependencies), the entire schema is in 3NF. Foreign keys enforce referential integrity without introducing additional FDs that would violate 3NF. Thus, the design maintains 3NF and preserves all dependencies.

Each table's non-trivial FDs are fully contained within that table. In addition, the schema preserves *cross-table* dependencies through the use of foreign keys. For example, the dependency:

$$\{\text{auction\_id}\} \rightarrow \{\text{item\_id}\} \rightarrow \{\text{current\_highest\_bid}\}$$

is preserved because:

- In **auctions**, {auction_id} determines {item_id}.
- In **items**, {item_id} determines {current_highest_bid}.

Thus, even though the dependency spans two tables, the breakdown preserves it via foreign key relationships without violating 3NF.

In fact, the method of creating our schema first involved listing all attributes and FD's within a single table, then using the 3NF decomposition algorithm, we were efficiently able to split the schema as required. To make this process more streamlined, we left out the trivial FD's and ones contained under superkeys. In the end, we checked whether these trivial and superkey-based FD's were satisfied. Thus, the entire schema remains in 3NF. This can be verified by checking for each $X \implies Y$ in F (the set of all FDs), X is either a superkey, or Y is contained within a candidate key.

Indexing, Procedures, and Trigger Explanations

Indexing

PostgreSQL primarily uses B-tree indexes (a variant of B-trees similar in behavior to B^+ trees) for indexing. Now, an index is created implicitly on the primary key and unique attributes of a relation, so these do not have to be included here specifically. The other indices here are unclustered by default, meaning the physical order of rows is independent of the index order (on primary key, we already have a clustered index, so secondary indices must be unclustered). The following indexes are recommended to optimize key queries:

- **Items:**
  - `idx_items_seller` on `seller_id` – speeds up queries tracking seller activity.
  - `idx_items_highest_bidder` on `current_highest_bidder` – facilitates lookup of current bids.
- **Auctions:**
  - Composite index `idx_auctions_time_status` on `auction_status, end_time` – optimizes time-based queries and status checks.
  - Index on `item_id` reinforces lookups by item (even though a UNIQUE constraint exists).
- **Bids:**
  - Composite index `idx_bids_auction_amount` on `auction_id, bid_amount DESC` – helps in quickly identifying the highest bids.
  - Composite index `idx_bids_buyer_time` on `buyer_id, bid_time` – aids in retrieving a buyer's bidding history.
- **Automated Bids:**
  - Composite index `idx_automated_bids_auction_user` on `auction_id, buyer_id, bid_time DESC` – accelerates analysis of automated bid activity.
- **Auction Participants:**
  - Index `idx_participants_user` on `user_id` – supports user-centric queries.
- **Transactions:**
  - Index `idx_transactions_buyer_date` on `transaction_date` – speeds up historical transaction lookups.
- **Admin Logs:**
  - Index `idx_admin_log_table_deleted` on `changed_at` in **admin_delete_log** and `idx_admin_log_table_updated` on `changed_at` in **admin_update_log** – these indexes optimize audit trail queries.

- **Reviews:**
  - Index `idx_reviews_reviewee` on `transaction_id` – enables fast lookups for review-based reputation checks.

Auction Closing Procedure and Trigger

Auction Closing Procedure (`close_auction`):

- **Purpose:** Automatically finalizes an auction when its end time is reached.
- **Internal Workflow:**
  1. Retrieves the `item_id` corresponding to the given `auction_id`.
  2. Checks whether the auction exists; if not, an exception is raised.
  3. Updates the auction's status to `'closed'`.
  4. Fetches the highest bid details (if any) and the seller's ID from the associated item.
  5. If a valid highest bid exists, records the transaction by inserting a new row into **transactions**. Note that only the `auction_id` is inserted as the `transaction_id` is auto-generated and the date defaults.
  6. Calls the `send_notification` procedure to inform relevant parties of the auction's closure and outcome.

Auction End Trigger (`check_auction_end`):

- **Purpose:** Monitors the **auctions** table for updates to `end_time` and `auction_status`.
- **Internal Workflow:**
  1. After an update to an auction, the trigger checks if the auction is still marked as `'open'` and whether the current time has passed the auction's `end_time`.
  2. If these conditions are met, the trigger uses dynamic SQL to call the `close_auction` procedure.
  3. Finally, the trigger returns the updated row, ensuring that the update process continues as expected.
- **Integration with Backend:** In our backend, we have set up a cronjob to check if an auction has ended every minute. If at any point, we see that the auction has ended, again this logic is implemented for closing an auction.

Notification Procedure (`send_notification`)

Purpose: Notifies users about the outcome of an auction.

- **Internal Workflow:**

1. Retrieves details of the auction (such as status and end time) from **auctions**.
2. Retrieves item details (title, highest bid, seller, highest bidder) from **items**.
3. Looks up the seller's contact information (email and mobile number) from **users**.
4. If a winning bid exists, fetches the buyer's email.
5. Uses `RAISE NOTICE` to simulate sending notifications to the seller and buyer, including details like the final bid amount and auction closing time.
6. If no bid was placed, notifies only the seller.

Procedure for Updating the Highest Bid

Trigger Function: `update_highest_bid` This trigger function is designed to update the highest bid for an item when a new bid is inserted into the **bids** table. Its internal workings are as follows:

- **Row Locking and Data Retrieval:** The function first retrieves the associated `item_id` and `auction_status` from the **auctions** table by using the `FOR UPDATE` clause. This clause locks the specific auction row, ensuring that concurrent transactions cannot modify it until the current transaction completes.
- **Auction Status Check:** Once the auction row is locked, the function checks whether the auction is still marked as `'open'`. If the auction is not open, it raises a notice and exits without making any changes. This prevents updates to auctions that have already been closed.
- **Locking the Item Row:** Next, the function retrieves the current highest bid (or, if no bid exists, the starting bid) from the **items** table using another `FOR UPDATE` clause. Locking the item row ensures that only one transaction can update the highest bid at a time, which is essential for handling concurrent bid submissions.
- **Updating Highest Bid and Recording Participation:** If the new bid exceeds the current highest bid, the function updates the item's record with the new highest bid and the buyer's ID as the new highest bidder. Additionally, it attempts to insert a record into the **auction_participants** table to record the buyer's participation in the auction, using the `ON CONFLICT DO NOTHING` clause to avoid duplicate entries.
- **Return:** Finally, the function returns the new row, allowing the insertion in the **bids** table to complete successfully.

Concurrency and Row Locking Overview Both the trigger function and the procedure play critical roles in managing concurrent transactions:

- **Row-Level Locking:** The use of `FOR UPDATE` in the trigger function ensures that rows in the **auctions** and **items** tables are locked during the critical section of updating the highest bid. This prevents race conditions where multiple bids might try to update the same record simultaneously.

- **Sequential Processing:** With the locks in place, transactions that attempt to update the same auction or item must wait until the lock is released, ensuring that bid updates occur sequentially and that the final state of the highest bid is consistent.
- **Maintaining Data Integrity:** By ensuring that only one transaction can update the auction or item at a time, the system avoids issues like lost updates or inconsistent bid records, thus preserving data integrity in a concurrent multi-user environment.

In summary, the `update_highest_bid` trigger function ensures that bid updates, auction finalizations, and payment status changes are handled in a reliable and concurrent-safe manner. The trigger uses row locking to enforce sequential processing of bids, while the procedure ensures that payment statuses are updated consistently. This integrated approach helps maintain data integrity and responsiveness in the Online Auction System.

Transactions and Concurrency Control

In our GO code, transactions are used to ensure that database operations are executed atomically, thus preserving data consistency even under concurrent access. The key aspects of our transaction and concurrency control strategy include:

- **Transaction Initiation:** When an operation requires multiple related queries (e.g., inserting a bid, updating an automated bid, or updating an auction's end time), we begin a transaction using `Begin()`. This starts an isolated context in which all subsequent operations are treated as a single unit.
- **Deferred Rollback:** Immediately after starting a transaction, we use the `defer` keyword to schedule a `Rollback()` call. This deferred rollback ensures that if any error occurs during the transaction, the entire set of operations will be reverted, preventing partial updates. For example, in the `CreateBid` function:

  ```
  defer tx.Rollback(c)
  ```

  This guarantees that if an error occurs before the transaction is explicitly committed, all changes are undone.
- **Commit on Success:** After all intended queries are successfully executed within the transaction, a `Commit()` call finalizes the changes. The commit is only reached if no errors occur. This pattern ensures that only complete and consistent state updates are written to the database.
- **Row-Level Locking:** In critical sections such as the `update_highest_bid` trigger function, the use of the `FOR UPDATE` clause locks specific rows (e.g., in the **auctions** and **items** tables). This locking mechanism prevents concurrent transactions from simultaneously updating the same row, thereby avoiding race conditions and ensuring that updates (such as bid submissions) are processed sequentially.

- **Isolation and Consistency:** The combination of transaction isolation (provided by PostgreSQL's transaction management) and explicit row locks ensures that even in a multi-user environment, all database modifications occur in a controlled manner. This prevents issues such as lost updates or dirty reads, maintaining the integrity of the auction process.
- **Example in Code:** In functions like `CreateBid` and `UpdateAutomatedBid`, transactions are used as follows:
    1. A transaction is started with `Begin()`.
    2. Database operations (inserts or updates) are executed.
    3. A deferred rollback is set to revert changes in case of an error.
    4. Finally, if all operations succeed, `Commit()` is called to save the changes.

Overall, the use of transactions combined with row-level locking (via `FOR UPDATE`) and the proper use of `defer` to handle rollbacks ensures that our system processes concurrent operations in a safe, consistent, and atomic manner. This approach is critical in a high-concurrency environment like an online auction system, where multiple users may be bidding simultaneously.

Frontend Report

The frontend of the Online Auction System is a modern, scalable, and modular React-based application. It leverages several libraries and tools to provide a seamless user experience, efficient state management, and responsive design. Below is a comprehensive breakdown of the frontend.

1. Libraries and Tools Used

React:

- **Purpose:** Provides a component-based architecture and a virtual DOM for efficient UI rendering.
- **Usage:** Reusable components, hooks (e.g., `useState`, `useEffect`), and custom hooks for state and side-effect management.

React Router:

- **Purpose:** Enables client-side routing for seamless navigation without full page reloads.
- **Usage:** Routes are defined using `<Routes>` and `<Route>` components; redirection is handled by `<Navigate>`.

Zustand:

- **Purpose:** A lightweight state management library for managing global state.
- **Usage:** Custom hooks (e.g., `useAuthStore`, `useThemeStore`, `useProfileStore`) manage authentication, theming, and profile state.

Tailwind CSS and DaisyUI:

- **Purpose:** Provide utility-first styling and pre-styled UI components.
- **Usage:** Layouts and designs are quickly developed using Tailwind's classes and DaisyUI components.

Axios:

- **Purpose:** A promise-based HTTP client for making API requests.
- **Usage:** Centralized API calls, handling authentication tokens, and error responses.

React Hot Toast:

- **Purpose:** Provides toast notifications for immediate user feedback.
- **Usage:** Displays notifications for events such as successful logins and auction updates.

Vite:

- **Purpose:** A fast build tool for development and production.
- **Usage:** Offers a fast development server with hot module replacement (HMR) and optimizes production builds.

Lucide React:

- **Purpose:** Provides customizable icons for enhancing the UI.
- **Usage:** Icons are used in navigation components and other parts of the interface.

2. File and Folder Structure

Root Files:

- **.env:** Stores environment variables.
- **.gitignore:** Specifies files and directories to be ignored.
- **eslint.config.js:** Configures code linting.
- **index.html:** The entry point for the application.
- **package.json:** Manages dependencies and scripts.
- **vite.config.js:** Configures the build tool.

Source Directory (src/):

- **App.jsx:** Defines the main application structure and routes.
- **main.jsx:** The entry point for rendering the React app.
- **index.css:** Contains global styles.
- Subdirectories for components, constants, helpers, libraries (e.g., Axios configuration), pages, and store (custom hooks for state management).

3. Key Features

- **Authentication:** Dedicated login and signup pages with state management via `useAuthStore`.
- **Profile Management:** A tab-based profile page displaying personal details, bidding history, and other user-specific data.
- **Auction Management:** Real-time auction updates, detailed auction pages, and functionality to create and manage auctions.
- **Responsive Design:** Tailwind CSS ensures that the application is responsive across various devices.
- **Notifications:** React Hot Toast displays feedback for user actions.

4. Development Tools

- **Vite:** Provides a fast development server with hot module replacement and efficient production builds.
- **React Hot Toast:** Manages user notifications.

Backend Report

The backend of the Online Auction System is a robust, efficient, and secure Go-based application. It implements a RESTful API architecture with real-time communication capabilities, database integration, and scheduled tasks. This section provides a comprehensive breakdown of the backend, including the libraries used, their purposes, the file and folder structure, key features, and the API endpoints.

1. Libraries and Tools Used

Gin (Web Framework):

- **Purpose:** A high-performance HTTP web framework providing robust features for building web applications and APIs.
- **Usage:** Router configuration, middleware integration, request handling, parameter binding, response formatting, and static file serving for uploaded images.

PostgreSQL (Database):

- **Purpose:** A relational database for storing application data with support for complex queries and transactions.
- **Driver:** jackc/pgx/v5.
- **Usage:** Storing user information, auction data, bids, and transaction records; implementing business logic via SQL functions and triggers; ensuring data integrity through foreign key constraints.

JWT (Authentication):

- **Purpose:** JSON Web Tokens are used for secure authentication and session management.
- **Usage:** Generating tokens upon login, verifying user identity for protected routes, and managing session expiration.

WebSockets (gorilla/websocket):

- **Purpose:** Provides real-time, bidirectional communication between clients and the server.
- **Usage:** Broadcasting new bids, notifying users about auction status changes, and maintaining active connections.

UUID (google/uuid):

- **Purpose:** Generates unique identifiers for resources (e.g., uploaded files).
- **Usage:** Creating unique filenames to avoid collisions in the upload directory.

Godotenv:

- **Purpose:** Loads environment variables from a .env file for configuration.
- **Usage:** Loading database connection strings and other settings without hardcoding values.

Email (jordan-wright/email):

- **Purpose:** Sends email notifications to users.
- **Usage:** Notifying users about outbids and auction end events, often using HTML templates.

2. File and Folder Structure

Root Files:

- **.env:** Contains environment variables (e.g., DB connection strings, JWT secrets).
- **.gitignore:** Specifies files and directories to ignore (e.g., uploads, .env).
- **go.mod & go.sum:** Manage dependencies for the project.
- **main.go:** The entry point that initializes components and starts the server.

Config Directory (config/):

- **db.go:** Sets up the PostgreSQL database connection pool.
- **env.go:** Loads environment variables.
- **init.go:** Initializes configuration components.

Internal Directory (internal/):

- **Controllers (controller/):** Handle HTTP requests for auctions, authentication, profiles, transactions, and uploads.
- **Database Layer (db/):** Contains operations for auctions, users, profiles, and transactions.
- **Helpers (helpers/):** Utility functions for cookies, JWT tokens, email notifications, and password management.
- **Middlewares (middlewares/):** Contains authentication middleware to verify JWT tokens.
- **Router (router/):** Configures the Gin router, API endpoints, and WebSocket handlers.
- **Schema (schema/):** Defines data structures for API requests and responses.
- **WebSockets (websockets/):** Manages WebSocket connections and message broadcasting.
- **Cronjobs (cronjob/):** Implements scheduled tasks for auction status checks.
- **Templates (templates/):** Contains HTML templates for email notifications.

SQL Directory (sql/):

- **init.sql:** Contains the database schema, indexes, triggers, and stored procedures.

Uploads Directory (uploads/):

- **images/:** Storage for uploaded auction item images.

3. Key Features

- **Authentication:** JWT-based authentication with secure token management and protected routes.
- **Auction Management:** Full auction lifecycle management from creation, bidding, to automatic closure and transaction recording.
- **Real-time Communication:** WebSocket integration for instant bid notifications and auction updates.
- **Bidding System:** Supports both manual and automated bidding with corresponding notifications.
- **Background Processing:** Scheduled tasks for processing auctions and sending notifications.
- **Email Notifications:** Templated emails for outbid alerts and auction closure notifications.

4. API Endpoints

Authentication:

- `POST /auth/signup`: Register a new user.
- `POST /auth/login`: Authenticate a user.
- `GET /auth/logout`: Log out the current user.
- `GET /auth/check`: Verify authentication status.

Auctions:

- `GET /api/auctions`: List all active auctions.
- `GET /api/auctions/:id`: Get details of a specific auction.
- `POST /api/auctions`: Create a new auction.
- `DELETE /api/auctions/:id`: Delete an auction.
- `PUT /api/auctions/:id/update-end-time`: Update auction end time.
- `POST /api/auctions/:id/bid`: Place a bid.
- `POST /api/auctions/:id/automated-bid`: Place an automated bid.
- `GET /api/auctions/:id/bids`: Get all bids for an auction.
- `POST /api/auctions/upload`: Upload an auction item image.

Profile:

- `GET /api/profile`: Get current user profile.
- `PUT /api/profile`: Update user profile.
- `GET /api/profile/auctions`: Get auctions created by the user.
- `GET /api/profile/bids`: Get user's bid history.
- `GET /api/profile/sold`: Get items sold by the user.
- `GET /api/profile/bought`: Get items bought by the user.

Reviews:

- `POST /api/reviews`: Submit a review for a completed transaction.

Background Processing:

- `POST /api/cronjob`: Trigger processing of auction status.