Design document: Inventory management system

Group Members

1. Nguyen Xuan Truong
2. Le Ngoc toan

Conceptual & Logical Design

Functional Requirements

These specify what the system must do, focusing on the capabilities required for the inventory and order management system.

User Management

- **Create/Update/Delete Users:** Administrators can manage user accounts stored in the `users` table, which includes fields such as `username`, `email`, `password_hash`, `role_name` (`admin` or `staff`), and `warehouse_id`.
- **Authentication:** Users must log in using their email and password. Login attempts are recorded in the `login_logs` table, including `ip_address`, `user_agent`, and `refresh_token`.
- **Role-Based Access:** Admins have full access. Staff users are limited to managing products and orders within their assigned `warehouse_id`.

  Example: An admin creates a user with `role_name = 'staff'` and assigns them to `warehouse_id = 1`. The system logs their login on 2025-05-20 22:17:00 from IP 192.168.1.1.

Warehouse Management

- **Manage Warehouses:** Warehouses are stored and updated in the `warehouses` table with fields like `name` and `address`.
- **Associate Entities:** Warehouses are linked to both users and products via `warehouse_id`.

  Example: A warehouse named "Central Hub" at "123 Main St" is linked to multiple products and users.

Supplier Management

- **Manage Suppliers:** Supplier data is stored in the `suppliers` table, including `name`, `contact_name`, `contact_email`, and `phone`.
- **Link to Products:** Each product is linked to a supplier via `supplier_id`.

  Example: A supplier "TechCorp" with email "contact@techcorp.com" supplies products like "Laptop" and "Mouse".

Category Management

- **Manage Categories:** Categories are stored in the `categories` table with fields `name` and `description`.
- **Link to Products:** Products reference a `category_id` to indicate category.

  Example: A category "Electronics" with description "Gadgets and devices" includes products like "Smartphone".

Product Management

- **Manage Products:** Products are stored in the `products` table with fields `name`, `description`, `price`, `quantity`, and `image_url`, linked to `suppliers`, `warehouses`, and `categories`.
- **Inventory Tracking:** The system tracks quantity per warehouse and updates it when orders are placed.

  Example: A product "Laptop" with price 999.99 and quantity 50 is stored in warehouse "Central Hub" and belongs to the "Electronics" category.

Customer Management

- **Manage Customers:** Customers are stored in the `customers` table with fields such as `name`, `email`, `phone`, and `address`.

  Example: Customer "John Doe" with email "john@example.com" has placed multiple orders.

Order Management

- **Create/Track Orders:** Orders are stored in the `orders` table, including `customer_id`, `order_date`, and `status` (`pending`, `completed`, `cancelled`).
- **Link to Order Items:** Orders are connected to `order_items`, which link to `products` through `product_order_items`.

  Example: Order with `order_id = 1` for customer "John Doe" on 2025-05-20 includes two laptops (quantity = 2, total_price = 1999.98).

Order Item Tracking

- **Manage Order Items:** Order items are tracked in the `order_items` table and linked to products via `product_order_items` including `quantity` and `total_price`.

  Example: `order_item_id = 1` links to `product_id = 1` (Laptop) with quantity = 2.

Non-Functional Requirements

These define system qualities like performance, security, and maintainability.

- **Performance:** Queries (e.g., product searches, order lookups) should return results in under 2 seconds for up to 50 concurrent users.

  Example: A product search by category uses an index on `products.category_id` for fast retrieval.
- **Scalability:** The system must support a modest scale of up to a few thousand products and hundreds of orders without significant performance degradation.

  Example: Partitioning orders by `order_date` ensures efficient access.
- **Security:** Passwords must be hashed (e.g., bcrypt). All login attempts must be logged in `login_logs` to detect suspicious access.

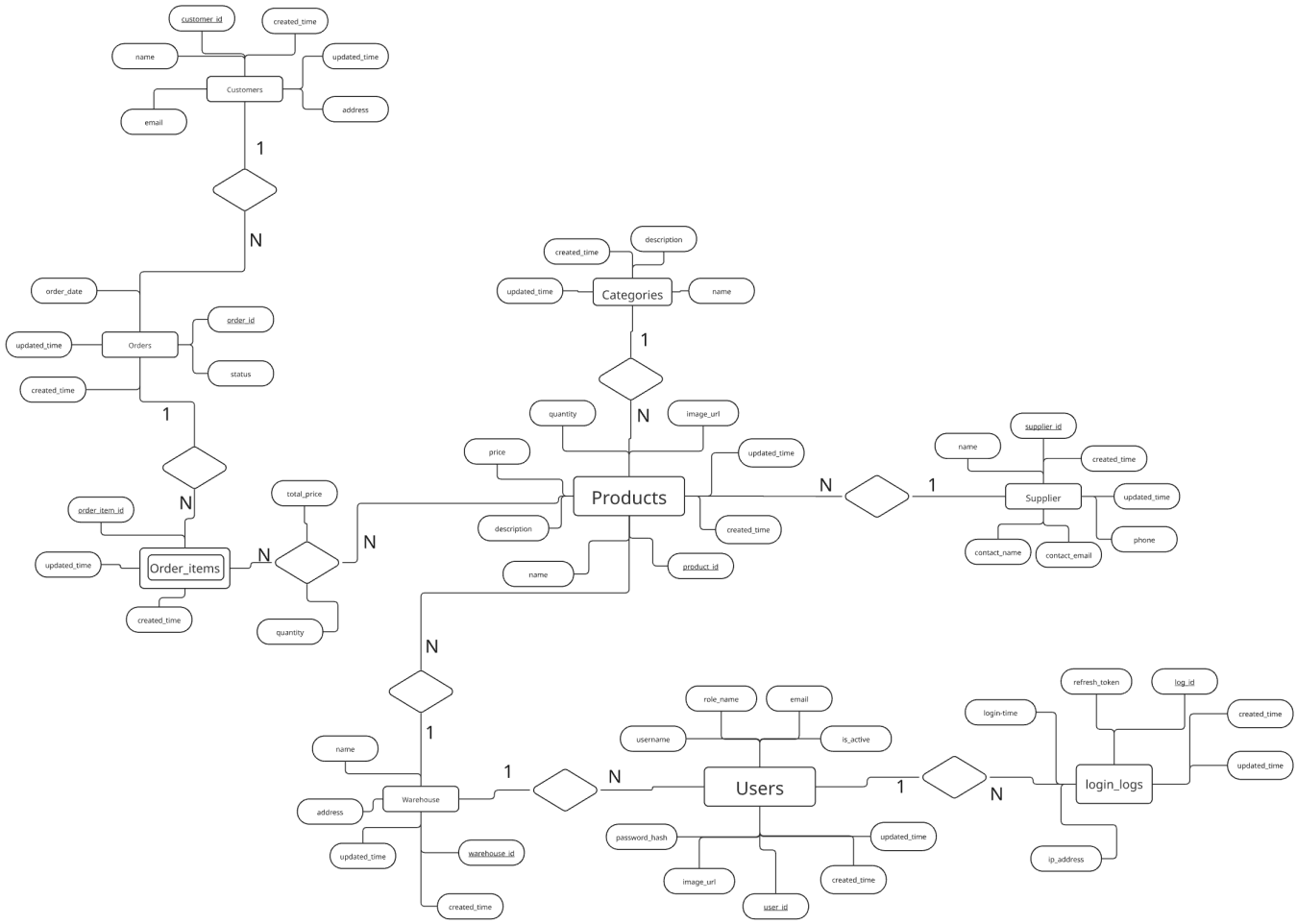
  Example: A failed login attempt for `user_id = 1` is recorded with the `ip_address` and `user_agent`.
- **Reliability:** Foreign key constraints (e.g., `fk_products_supplier`) ensure data integrity. Order updates must be atomic and isolated to prevent race conditions.

  Example: Updating `products.quantity` during an order is handled in a transaction.
- **Availability:** The system should maintain 99.5% uptime. A simple master-slave database setup ensures basic failover support.

  Example: A replicated MySQL setup improves resilience during maintenance.
- **Maintainability:** The schema must be fully normalized (to at least 3NF) to reduce redundancy and simplify modifications.

  Example: Adding a new attribute to a product only requires a column addition in the `products` table.
- **Architecture and Technologies:**
  - Fully normalized schema (3NF)
  - Fast API responses with caching (e.g., Redis)
  - Secure authentication using JWT and bcrypt-hashed passwords
  - ACID-compliant database transactions
  - Indexed queries for optimized performance
  - Modular, containerized architecture using Docker
  - Responsive and intuitive UI (e.g., built with Next.js)
  - CI/CD-enabled development workflow

Entity–Relationship Diagram



- **Warehouses** have a one-to-many relationship with:
  - **Users**
  - **Products**
- **Suppliers** have a one-to-many relationship with **Products**.
- **Categories** have a one-to-many relationship with **Products**.
- **Customers** place multiple **Orders** (one-to-many).
- **Orders** contain multiple **Order_Items** (one-to-many).
- **Products** and **Order_Items** have a many-to-many relationship through the **Product_Order_Items** junction table.

Normalization proof up to Third Normal Form (3NF)

This section provides a detailed normalization proof for the database schema, demonstrating that all tables conform to the rules of First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF).

First Normal Form (1NF)

Requirement: All attributes must be atomic (no multi-valued attributes or repeating groups), and each table must have a primary key.

Analysis:

- **Atomic Attributes:** Every attribute in all tables contains only atomic values. For instance:
  - `users.email` stores one unique email address per user.
  - `products.price` stores a single decimal value.
  - `customers.address` uses a `TEXT` field but holds a single address per customer, not multiple.
  - `products.quantity` is an integer that represents a single count value.
- **Primary Keys:**
  - **Single-column primary keys:** `warehouses(warehouse_id)`, `users(user_id)`, `products(product_id)`, `suppliers(supplier_id)`, `categories(category_id)`, `orders(order_id)`, `order_items(order_item_id)`, etc.
  - **Composite primary key:** `product_order_items(product_id, order_item_id)` uses a composite key to identify the relationship between a product and an order item.

Conclusion: All tables have primary keys and atomic attributes. Therefore, the schema satisfies 1NF.

Second Normal Form (2NF)

Requirement: The schema must first satisfy 1NF, and all non-key attributes must depend on the entire primary key (i.e., no partial dependencies).

Analysis:

- **Tables with single-column primary keys:**
  - Examples include `users`, `products`, `orders`, etc.
  - All non-key attributes in these tables are fully functionally dependent on the primary key. For instance:
    - In `users`, attributes like `email`, `role_name`, and `is_active` depend entirely on `user_id`.
    - In `products`, attributes such as `name`, `price`, `quantity`, etc., depend entirely on `product_id`.
- **Table with composite key:**
  - `product_order_items(product_id, order_item_id)` uses a composite key.
  - Attributes `quantity` and `total_price` depend on the entire composite key.
  - Example: `quantity = 2` only makes sense when referring to a specific combination of `product_id` and `order_item_id`, not just one of them.

Conclusion: There are no partial dependencies; every non-key attribute depends on the full key. Thus, all tables are in 2NF.

Third Normal Form (3NF)

Requirement: The schema must satisfy 2NF and have no transitive dependencies. That is, no non-key attribute should depend on another non-key attribute.

Analysis:

- **users:** All non-key attributes (e.g., `username`, `email`, `is_active`) depend directly on the primary key `user_id`. Although `warehouse_id` is a foreign key, it does not introduce transitive dependency (e.g., `email` does not depend on `warehouse_id`).
- **products:** Attributes such as `name`, `price`, and `quantity` depend only on `product_id`. Foreign keys like `supplier_id` and `category_id` do not cause transitive dependencies. For example, `price` does not depend on `supplier_id`.
- **orders:** Attributes `order_date` and `status` depend on `order_id`, not on `customer_id`. There is no chaining of dependencies among non-key attributes.
- **Avoidance of transitive storage:** If, for instance, `supplier_name` were stored directly in the `products` table, it would be transitively dependent via `supplier_id`. However, the schema correctly avoids this by referencing the name through a foreign key.

Conclusion: No transitive dependencies exist in any table. Therefore, the schema satisfies 3NF.

Final Remark: Since the schema adheres to 1NF, 2NF, and 3NF, it is properly normalized up to **Third Normal Form (3NF)**.

Physical Schema Definition

Schema Definition

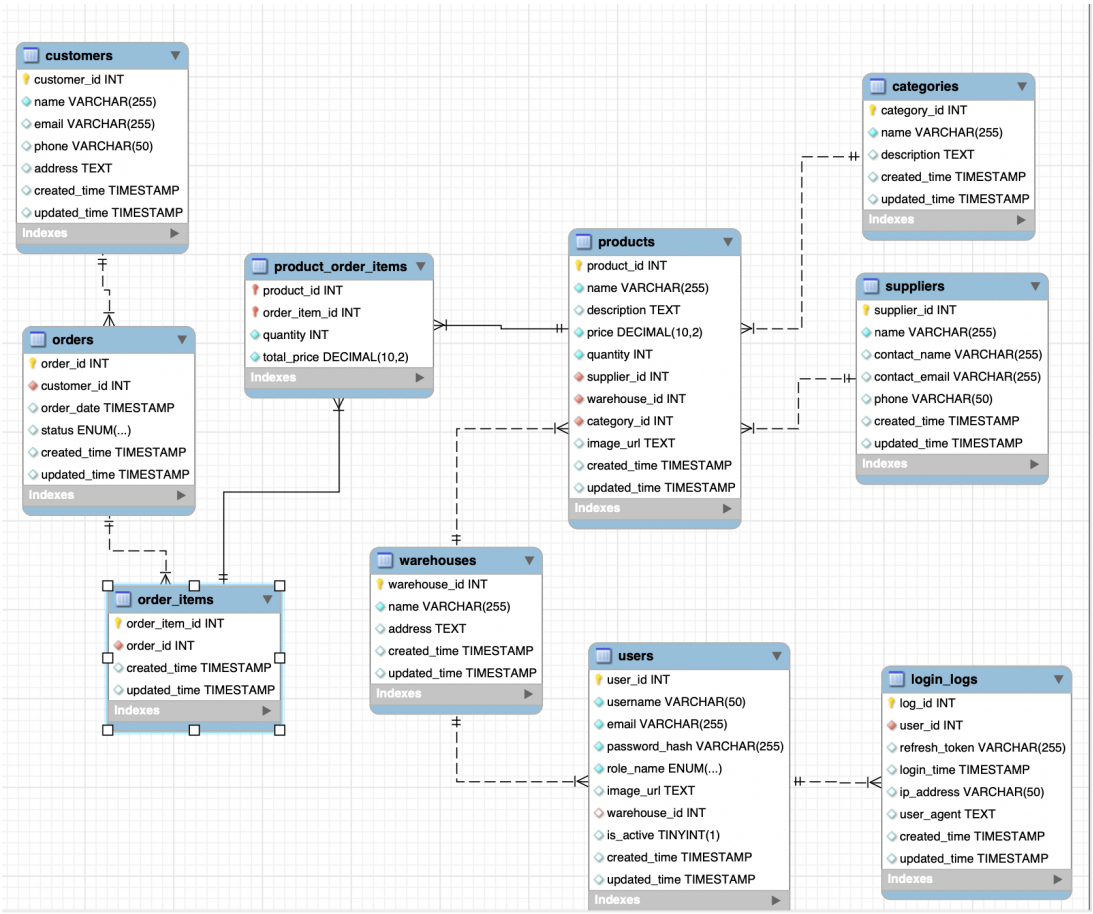
This database schema models a warehouse inventory and order management system. It includes 10 core tables with clearly defined relationships, data types, and constraints. The schema supports user roles, product tracking, supplier integration, order management, and auditing.

Tables Overview:

- **warehouses**: Physical storage locations
- **users**: Authenticated personnel (admin/staff)
- **login_logs**: Tracks login activity
- **suppliers**: Sources of products
- **categories**: Product groupings
- **products**: Inventory items
- **customers**: Buyers of products
- **orders**: Customer order records
- **order_items**: Line items in an order
- **product_order_items**: Product-Order line item mapping

Data Types and Constraints:

- **Primary Keys**: Each table includes an `INT AUTO_INCREMENT` primary key.
- **Foreign Keys**: Enforce referential integrity between related tables.
- **ENUM**: Used for roles (`admin`, `staff`) and order status (`pending`, `completed`, `cancelled`).
- **Timestamps**: `created_time` and `updated_time` automatically manage record lifecycle.
- **Defaults**: e.g., `users.is_active = FALSE`, `products.quantity = 0`.



SQL Schema:

```
-- DELETE ALL TABLES BEFORE RECREATING
DROP TABLE IF EXISTS product_order_items;
DROP TABLE IF EXISTS order_items;
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS products;
DROP TABLE IF EXISTS categories;
DROP TABLE IF EXISTS suppliers;
DROP TABLE IF EXISTS warehouses;
DROP TABLE IF EXISTS customers;
DROP TABLE IF EXISTS users;
DROP TABLE IF EXISTS login_logs;

-- CORE TABLES

CREATE TABLE warehouses (
    warehouse_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    address TEXT,
    created_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

CREATE TABLE users (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
```

```
    password_hash VARCHAR(255) NOT NULL,
    role_name ENUM('admin', 'staff') NOT NULL DEFAULT 'staff',
    image_url TEXT,
    warehouse_id INT,
    is_active BOOLEAN DEFAULT FALSE,
    created_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    CONSTRAINT fk_users_warehouse FOREIGN KEY (warehouse_id) REFERENCES warehouses(warehouse_id)
);

CREATE TABLE login_logs (
    log_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    refresh_token VARCHAR(255),
    login_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    ip_address VARCHAR(50),
    user_agent TEXT,
    created_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    CONSTRAINT fk_loginlogs_user FOREIGN KEY (user_id) REFERENCES users(user_id)
);

CREATE TABLE suppliers (
    supplier_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    contact_name VARCHAR(255),
    contact_email VARCHAR(255),
    phone VARCHAR(50),
    created_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

CREATE TABLE categories (
    category_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    created_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

-- PRODUCT SYSTEM

CREATE TABLE products (
    product_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    price DECIMAL(10, 2) NOT NULL,
    quantity INT NOT NULL DEFAULT 0,
    supplier_id INT NOT NULL,
    warehouse_id INT NOT NULL,
    category_id INT NOT NULL,
    image_url TEXT,
    created_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    CONSTRAINT fk_products_supplier FOREIGN KEY (supplier_id) REFERENCES suppliers(supplier_id),
    CONSTRAINT fk_products_warehouse FOREIGN KEY (warehouse_id) REFERENCES warehouses(warehouse_id),
    CONSTRAINT fk_products_category FOREIGN KEY (category_id) REFERENCES categories(category_id)
```

```
);

-- ORDER SYSTEM

CREATE TABLE customers (
    customer_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255),
    phone VARCHAR(50),
    address TEXT,
    created_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

CREATE TABLE orders (
    order_id INT AUTO_INCREMENT PRIMARY KEY,
    customer_id INT NOT NULL,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status ENUM('pending', 'completed', 'cancelled') DEFAULT 'pending',
    created_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    CONSTRAINT fk_orders_customer FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

CREATE TABLE order_items (
    order_item_id INT AUTO_INCREMENT PRIMARY KEY,
    order_id INT NOT NULL,
    created_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    CONSTRAINT fk_orderitems_order FOREIGN KEY (order_id) REFERENCES orders(order_id)
);

CREATE TABLE product_order_items (
    product_id INT NOT NULL,
    order_item_id INT NOT NULL,
    quantity INT NOT NULL,
    total_price DECIMAL(10, 2) NOT NULL,
    PRIMARY KEY (product_id, order_item_id),
    CONSTRAINT fk_poi_product FOREIGN KEY (product_id) REFERENCES products(product_id),
    CONSTRAINT fk_poi_order_item FOREIGN KEY (order_item_id) REFERENCES order_items(order_item_id)
);
```

Definitions of Views, Indexes, and Partitioning Strategy

Views

Views simplify complex queries, provide abstracted data access, and support reporting needs. Below are four suggested views to address common use cases in the inventory and order management system:

Product Summary View **Purpose:** Aggregates product details with related supplier, category, and warehouse names for inventory reporting.

Use Case: Enables users to retrieve a comprehensive product list without complex joins, useful for inventory dashboards or stock analysis.

SQL:

```
CREATE VIEW product_summary AS
SELECT
    p.product_id,
```

```
    p.name AS product_name,
    p.price,
    p.quantity,
    c.name AS category_name,
    s.name AS supplier_name,
    w.name AS warehouse_name
FROM products p
JOIN categories c ON p.category_id = c.category_id
JOIN suppliers s ON p.supplier_id = s.supplier_id
JOIN warehouses w ON p.warehouse_id = w.warehouse_id;
```

Example: `SELECT * FROM product_summary WHERE category_name = 'Electronics';`

Order Details View **Purpose:** Combines order, customer, and product information to provide a detailed view of orders for tracking and reporting.

Use Case: Useful for generating order reports or customer purchase histories.

SQL:

```
CREATE VIEW order_details AS
SELECT
    o.order_id,
    o.order_date,
    o.status,
    c.name AS customer_name,
    p.name AS product_name,
    poi.quantity,
    poi.total_price
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN order_items oi ON o.order_id = oi.order_id
JOIN product_order_items poi ON oi.order_item_id = poi.order_item_id
JOIN products p ON poi.product_id = p.product_id;
```

Example: `SELECT * FROM order_details WHERE status = 'pending';`

User Activity View **Purpose:** Tracks user login activity by combining users and login_logs data.

Use Case: Supports security auditing and monitoring user access patterns.

SQL:

```
CREATE VIEW user_activity AS
SELECT
    u.user_id,
    u.username,
    u.email,
    u.role_name,
    w.name AS warehouse_name,
    ll.login_time,
    ll.ip_address,
    ll.user_agent
FROM users u
LEFT JOIN login_logs ll ON u.user_id = ll.user_id
LEFT JOIN warehouses w ON u.warehouse_id = w.warehouse_id;
```

Example: `SELECT * FROM user_activity WHERE login_time >= '2025-05-01';`

Low Stock Alert View **Purpose:** Identifies products with low inventory (e.g., quantity < 10) to support restocking decisions.

Use Case: Helps warehouse managers prioritize restocking by highlighting low-stock products.

SQL:

```
CREATE VIEW low_stock_alert AS
SELECT
    p.product_id,
    p.name AS product_name,
    p.quantity,
    w.name AS warehouse_name,
    s.name AS supplier_name
FROM products p
JOIN warehouses w ON p.warehouse_id = w.warehouse_id
JOIN suppliers s ON p.supplier_id = s.supplier_id
WHERE p.quantity < 10;
```

Example: `SELECT * FROM low_stock_alert;`

Indexes

Indexes improve query performance for frequent operations. Below are five suggested indexes to optimize common queries:

- **Users Email Index**
  SQL: `CREATE INDEX idx_users_email ON users(email);`
  Rationale: Speeds up login queries and email-based lookups.
- **Products Name Index**
  SQL: `CREATE INDEX idx_products_name ON products(name);`
  Rationale: Accelerates product searches by name.
- **Orders Date Index**
  SQL: `CREATE INDEX idx_orders_date ON orders(order_date);`
  Rationale: Optimizes queries filtering orders by date.
- **Product Order Items Composite Index**
  SQL: `CREATE INDEX idx_poi_product_order ON product_order_items(product_id, order_item_id);`
  Rationale: Enhances performance of joins involving `product_order_items`.
- **Customers Email Index**
  SQL: `CREATE INDEX idx_customers_email ON customers(email);`
  Rationale: Speeds up customer lookups by email.

Partitioning Strategy

Partitioning improves performance and manageability for large tables. Below are strategies for two key tables:

Orders Table **Strategy:** Range partitioning by `order_date` year.
SQL:

```
ALTER TABLE orders
PARTITION BY RANGE (YEAR(order_date)) (
    PARTITION p0 VALUES LESS THAN (2023),
    PARTITION p1 VALUES LESS THAN (2024),
    PARTITION p2 VALUES LESS THAN (2025),
    PARTITION p3 VALUES LESS THAN (2026),
    PARTITION p_future VALUES LESS THAN MAXVALUE
);
```

Rationale: Orders grow over time, and queries often filter by date. Range partitioning reduces scanned data for time-based queries.
Example: `SELECT * FROM orders WHERE YEAR(order_date) = 2025;` scans only the `p2` partition.

Products Table **Strategy:** Hash partitioning by `warehouse_id`.
SQL:

```
ALTER TABLE products
PARTITION BY HASH (warehouse_id)
PARTITIONS 4;
```

Rationale: Distributes products evenly across partitions for warehouse-specific queries.
Example: `SELECT * FROM products WHERE warehouse_id = 1;` accesses only one partition.

Partitioning Notes

- **Orders:** Range partitioning suits time-series data and enables efficient queries and maintenance.
- **Products:** Hash partitioning improves scalability in multi-warehouse environments.
- **Future Consideration:** As data grows, sub-partitioning (e.g., by `category_id`) may be considered.

Task Division & Project Plan

Task Division

The responsibilities of the team members in the project are clearly divided to ensure smooth collaboration and efficient development. Below is a detailed breakdown of the tasks assigned to each team member.

Nguyen Xuan Truong (Software Developer)

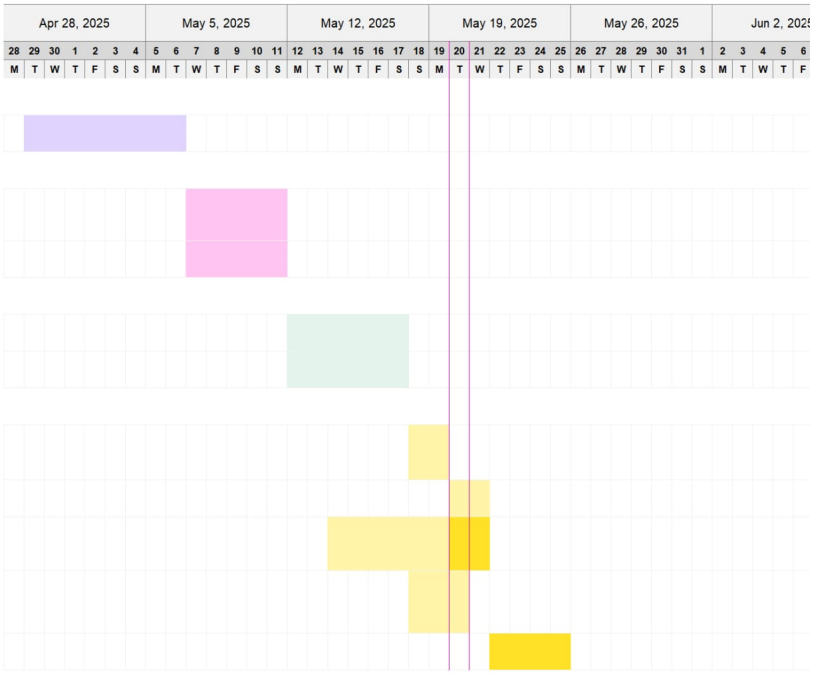
- **Database Schema & DDL Implementation:**
  - Designed the overall relational schema for the system.
  - Implemented the data definition language (DDL) statements for creating tables, primary and foreign keys, constraints, and indexes.
  - Ensured proper normalization of the schema up to Third Normal Form (3NF).
- **Triggers & Stored Procedures:**
  - Developed triggers to enforce data integrity, such as cascading updates or automatic timestamp updates.
  - Wrote stored procedures to encapsulate complex business logic within the database layer.
- **Backend API (Database-Related):**
  - Implemented backend API endpoints that interact directly with the database.
  - Built functionalities such as retrieving product lists, creating and updating orders, and managing users and inventory.
- **Deployment:**
  - Managed containerized deployment using Docker.
  - Wrote Dockerfiles and Docker Compose scripts for environment setup and orchestration of backend, database, and other services.

Le Ngoc Toan (Software Developer)

- **Backend API (Redis & MinIO Integration):**
  - Integrated Redis caching to improve API performance and reduce database load.
  - Implemented file and image management functionality using MinIO, including secure upload and retrieval.
- **Frontend UI (Next.js & API Integration):**
  - Developed the frontend of the application using the Next.js framework.
  - Designed and implemented responsive UI components and integrated them with backend APIs for dynamic content.
- **Test Data Insertion & SQL Queries:**
  - Created realistic test data to simulate user and transaction behavior.
  - Wrote complex SQL queries for validation, reporting, and system testing purposes.

Project Plan

TASK	ASSIGNED TO	PROGRESS	START	END
Planning & Design				
ER Diagram Design	Toan, Truong	100%	4/29/25	5/6/25
DDL Implementation				
Warehouse, Category, Supplier, Product, User table	Truong	100%	5/7/25	5/11/25
Customer, Order, Order_item table	Toan	100%	5/7/25	5/11/25
Triggers, Stored Procedures & Views				
AuditLog for Product, User table changes	Truong	100%	5/12/25	5/17/25
AuditLog for Supplier, Customer, Order table	Toan	100%	5/12/25	5/17/25
Backend + Frontend				
Setup Base FastAPI Project, Auth (JWT), Pydantic Models	Truong	100%	5/18/25	5/19/25
Implement Product Endpoints	Truong	100%	5/20/25	5/21/25
Implement Order, Customer & User Endpoints	Toan	75%	5/14/25	5/21/25
Setup Base Next.js Project, Layout, Routing, Auth Context	Truong	100%	5/18/25	5/20/25
Develop UI	Both	0%	5/22/25	5/25/25



Supporting Documentation

Design Rationale

Our database design prioritizes integrity, efficiency, and scalability for the inventory and order management system:

- **Normalization (3NF)**: The schema is normalized to Third Normal Form to eliminate redundancy and ensure data consistency. Entities such as `products`, `suppliers`, `categories`, `warehouses`, `customers`, and `orders` are separated into distinct tables, avoiding duplication (e.g., storing `supplier_name` in `products` is avoided by using `supplier_id`).
- **Primary & Foreign Keys**: Auto-incrementing `INT` primary keys (e.g., `product_id`, `order_id`) ensure unique identification and simplify joins. Foreign keys (e.g., `fk_products_supplier`, `fk_orders_customer`) enforce referential integrity, ensuring relationships like a product's link to a valid `supplier_id` or `warehouse_id` are maintained.
- **Data Types & Constraints**:
  - **Numeric**: `INT` for IDs (e.g., `warehouse_id`) and `DECIMAL(10,2)` for monetary fields (e.g., `products.price`, `product_order_items.total_price`) to handle precise financial calculations.
  - **Text**: `VARCHAR` for fixed-length fields (e.g., `users.username` at 50, `suppliers.name` at 255) and `TEXT` for flexible fields (e.g., `products.description`, `customers.address`).
  - **Enumerated**: `ENUM('admin', 'staff')` for `users.role_name` and `ENUM('pending', 'completed', 'cancelled'` for `orders.status` to restrict values and improve clarity.
  - **Temporal**: `TIMESTAMP` for `created_time` and `updated_time` with `CURRENT_TIMESTAMP` defaults and `ON UPDATE` for automatic audit tracking.
  - **Constraints**: `NOT NULL` ensures required fields (e.g., `products.name`, `users.email`) are populated. `UNIQUE` on `users.email` prevents duplicates. Defaults (e.g., `users.is_active = FALSE`) streamline data entry.
- **Audit Timestamps**: `created_time` and `updated_time` in all tables provide a basic audit trail for tracking record creation and modifications, essential for debugging and compliance.
- **Specialized Tables for Core Logic**:
  - `Products`: Centralizes product details (`name`, `price`, `quantity`) with links to `suppliers`, `warehouses`, and `categories` for flexible inventory management.
  - `Orders and Order Items`: The `orders`, `order_items`, and `product_order_items` tables model a many-to-many relationship between orders and products, allowing multiple products per order with quantities and total prices tracked in `product_order_items`.
  - `Login Logs`: The `login_logs` table tracks user authentication events (`user_id`, `ip_address`, `user_agent`, `refresh_token`), supporting security and auditing requirements.
  - `Users and Warehouses`: The `users` table links to `warehouses` to restrict staff access to specific locations, enhancing operational control.
- **Indexing**: Primary keys (e.g., `product_id`, `order_id`) and foreign keys (e.g., `products.supplier_id`) are automatically indexed in MySQL (InnoDB). Additional indexes on frequently queried columns (e.g., `users.email`, `products.name`, `orders.order_date`) are added to optimize performance for searches, logins, and order filtering. Partitioning is considered for future scalability on large tables like `orders` and `products`.

Sample Data Loading Approach

We will populate the database with realistic sample data using SQL `INSERT` scripts to support testing and demonstration of the inventory and order management system.

- **Scripting Order**: Scripts will respect foreign key dependencies to avoid constraint violations. The order is:
  1. `warehouses`
  2. `suppliers`, `categories`, `users`

3. `products`, `customers`, `login_logs`
4. `orders`, `order_items`, `product_order_items`

- **Data Variety**: Sample data will cover diverse scenarios to test functionality and edge cases:
  - `Users`: Multiple roles (`admin`, `staff`), active/inactive accounts, and users linked to different warehouses.
  - `Products`: Various categories (e.g., "Electronics", "Clothing"), suppliers, and warehouses with different quantities and prices.
  - `Orders`: Different statuses (`pending`, `completed`, `cancelled`), multiple customers, and orders with varying numbers of items.
  - **Edge Cases**: Empty warehouses, orders with no items, or products with zero quantity.
  - **Example**: Insert a user (`role_name = 'admin', email = 'admin@example.com'`), a product (`name = 'Laptop', price = 999.99, quantity = 50`), and an order with two items.
- **Tools**: SQL `INSERT` scripts will be used for data loading due to their simplicity and repeatability. For `products.image_url`, sample URLs or references to files stored in an external system (e.g., MinIO for product images) will be included. A script to populate `login_logs` will simulate login events with varied `ip_address` and `user_agent` values.
- **Repeatability**: Scripts will be organized in a modular structure (e.g., separate files for each table) and executable via a single command (e.g., a Makefile target or shell script) to reset and repopulate the database consistently, ensuring a predictable state for development, testing, and demonstrations.
- **Example Script Snippet**:

```
INSERT INTO warehouses (name, address) VALUES ('Central Hub', '123 Main St');
INSERT INTO suppliers (name, contact_email) VALUES ('TechCorp', 'contact@techcorp.com');
INSERT INTO products (name, price, quantity, supplier_id, warehouse_id, category_id)
VALUES ('Laptop', 999.99, 50, 1, 1, 1);
```