Medical Equipment Ordering System

Shikha Tiwari[1], Nisharg Kamlesh Gosai[1], Shreyashri Vishwanath Athani[1]

[1]Khoury College of Computer Sciences, Northeastern University

tiwari.shi@northeastern.edu, gosai.n@northeastern.edu, athani.sh@northeastern.edu

Keywords: Flask, Web Application, User Authentication, Order Processing, Analytics Dashboard, Stored Procedure, Trigger, Function, SQL Views

INTRODUCTION

The Medical Equipment Ordering System is designed to streamline the procurement process of medical equipment by providing a comprehensive platform for healthcare facilities, suppliers, and customers. Its primary objective is to create an efficient and user-friendly interface that facilitates the ordering, management, and tracking of medical equipment from various suppliers. By offering a centralized system, this database aims to enhance accessibility, accuracy, and efficiency in acquiring essential medical supplies within the healthcare industry.

In response to the growing demand for an organized and optimized approach to medical equipment procurement, the system has been developed to address specific challenges faced by healthcare providers and suppliers. These challenges include the complexities associated with managing multiple suppliers, maintaining accurate inventory records, and ensuring timely equipment deliveries. The system endeavors to alleviate these difficulties by integrating diverse functionalities aimed at improving the overall procurement process.

The database system operates by incorporating a range of entities such as suppliers, customers, equipment details, and order management functionalities. Suppliers are categorized based on business types, specialties, and provider types, while customers are classified into different types to tailor services accordingly. The system's architecture ensures a structured organization of data, allowing for efficient management and retrieval of information related to suppliers, customers, available equipment, and order details.

Moreover, this system's overarching goal is to foster a seamless interaction between customers and suppliers. Customers can register, log in, place orders for various medical equipment, modify order quantities, and deleting existing orders. By facilitating these interactions and functionalities, the database system aims to enhance the overall efficiency and effectiveness of the medical equipment procurement process.

DATABASE DESIGN

The database design for the Medical Equipment Ordering System adopts a structured approach centered on facilitating efficient data management and establishing relationships between key entities. The Entity-Relationship Diagram (ERD) encompasses entities such as 'Supplier,' 'Customer,' 'Equipment,' 'Order,' and additional classification entities like 'Business,' 'Specialty,' and 'ProviderType', 'CustomerType.' Notably, the decision to segment entities like 'Country,' 'State,' and 'City' allows for a hierarchical organization of geographical locations, enhancing data organization and retrieval. The utilization of foreign keys establishes relationships between entities, ensuring referential integrity and consistency in the database. Moreover, the normalization procedures adhere to the principles of First, Second, and Third Normal Forms, eliminating data redundancy, maintaining atomicity, and minimizing transitive dependencies. This structured approach ensures that data remains accurate, readily accessible, and easily manageable, facilitating streamlined order processing, inventory management, and seamless interactions between suppliers, customers, and equipment.

The database design for the Medical Equipment Ordering System underwent rigorous normalization procedures to ensure data integrity and optimize the structure. To adhere to the First Normal Form (1NF), the database tables were organized to guarantee atomicity, ensuring that each field contained only indivisible values. For instance, address details were separated into distinct fields (address1, address2, address3) to fulfill the 1NF requirements. Further normalization efforts focused on achieving the Second Normal Form (2NF) and Third Normal Form (3NF). Partial dependencies were eliminated by segmenting entities like 'Business,' 'Specialty,' and 'ProviderType' into separate tables, ensuring non-key attributes were dependent solely on

the primary key. Additionally, non-transitive dependencies were meticulously managed by establishing appropriate relationships through foreign keys. Redundancy was minimized by using separate tables for geographical locations ('Country,' 'State,' 'City') and connecting them through foreign keys, thus avoiding duplicate data entries. The 'SupplierEquipmentList' table efficiently managed equipment stock per supplier, preventing the need for redundant duplication of equipment details across suppliers. These normalization procedures aimed to eradicate anomalies, maintain data consistency, and optimize the database structure for efficient data storage and retrieval within the Medical Equipment Ordering System.

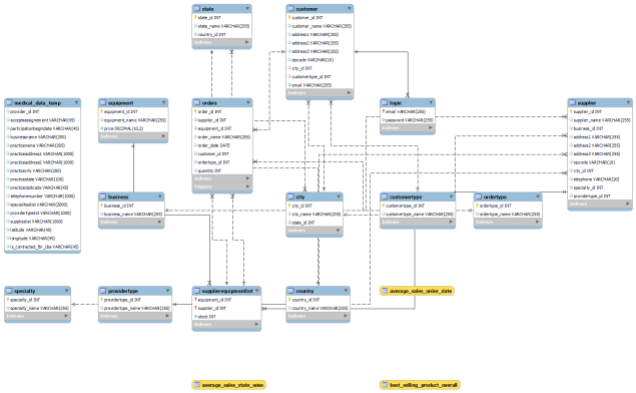

Figure 1: Entity Relationship Diagram

DATA COLLECTION

The external data used to populate the database of the Medical Equipment Ordering System was acquired from the **Centers for Medicare & Medicaid Services (CMS) dataset,** **https://data.cms.gov/provider-data/dataset/ct36-nrcq**. This dataset served as a comprehensive repository of information pertaining to various medical equipment suppliers, encompassing critical details such as supplier names, addresses, contact information, specialized equipment offered, provider types, and specialties. The acquisition involved accessing the dataset available on the CMS platform, ensuring compliance with their data usage terms and permissions**.** Upon acquisition, the dataset underwent a series of meticulous processing procedures-

1. Data Cleaning: Rigorous cleaning procedures were applied to rectify inconsistencies, standardize formats, handle missing values, and eliminate duplicates within the dataset. This involved normalizing naming conventions, addresses, and contact details to ensure uniformity and accuracy across entries.

2. Extraction and Aggregation: Relevant fields containing crucial information such as supplier names, addresses, zip codes, phone numbers, specialized equipment lists, provider types, and specialties were extracted from the dataset. Aggregation methods were employed to compile and organize these essential details for subsequent integration into the database.

3. Normalization and Deduplication: The extracted data underwent normalization processes to align with the database schema. Address fields were separated into distinct components to conform to normalization rules. Additionally, measures were taken to remove duplicate entries, ensuring data integrity and consistency within the database.

4. Data Integration: Python scripts leveraging libraries like pandas were employed to handle data manipulation tasks, such as splitting delimited fields ('supplieslist', 'providertypelist', 'specialitieslist'), removing duplicates, and generating SQL 'INSERT' queries for efficient integration into the respective tables ('Equipment', 'ProviderType', 'Specialty') within the database.

The comprehensive data processing procedures encompassed cleaning, extraction, normalization, aggregation, and integration methodologies, ensuring that the acquired external data was refined, standardized, and prepared for seamless utilization within the Medical Equipment Ordering System's database. These procedures upheld data accuracy, consistency, and compatibility with the system's architecture and requirements.

APPLICATION DESCRIPTION

For building medical equipment ordering system we have utilized-
Database system- MySQL
Front End Language - HTML, CSS, AJAX
Back End Language - Python FLASK
Dashboarding - Plotly and Dash

1. Customer Registration Page-

The Customer Registration Page serves as a platform for new customers such as Healthcare Providers, Hospitals,

Clinics, etc., to register before initiating equipment orders.

Functionality:

User Input: Users input their details including Name, Address information, City, State, Country, and Zip Code.

Account credentials: During registration, users are required to set up their login credentials by providing their email and creating a password. These credentials enable access to the system upon successful registration.

This page facilitates the initial setup for users by capturing their pertinent details and establishing secure login credentials, paving the way for seamless access to the system's functionalities.

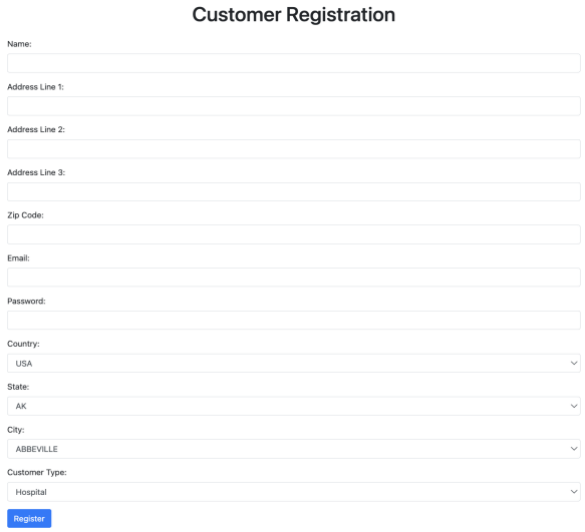

Figure 2: Customer Registration Page

a) Name - This text box allows users to enter their name. The data provided here will be inserted into the Customer table in the database.

b) Address Line 1- Another text box where users can input the first line of their address. The data entered will be stored in the Customer table.

c) Address Line 2- Users can input additional address details in this text box. Like Address Line 1, the data will be inserted into the Customer table.

d) Address Line 3- This text box allows for further address information if needed. Like previous fields, the data will be stored in the Customer table.

e) Zip Code- Users can enter their zip code here. The provided data will be inserted into the Customer table.

f) Email - A text box for users to input their email address. This data will be inserted into both the Customer table and the Login table. The Login table has a foreign key referencing the Customer table's email field. The system validates the email format, throwing an error if the format is incorrect, ensuring accurate registration.

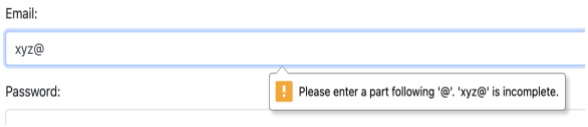

Figure 3: Customer Registration Page
(In this when a user enters incorrect email system will validate and will not permit to register.)

g) Password - Users can input their password here. This data will also be inserted into both the Customer table and the Login table. The passwords are hashed using bcrypt for secure storage and comparison during user authentication.

Figure 4: Password stored in database.
(Whenever a new user register then password is stored in login table in hashed format.)

h) Country- A dropdown menu populated from the Country table. Users can select their country here. The system will store the City_id in the Customer table. By retrieving the State_id from the State table, it can then determine the Country_id for data integrity.

i) State - Another dropdown menu, populated from the State table. Users can select their state here. Similar to Country, the system stores the City_id in the Customer table and fetches the State_id from the State table.

j) City - This dropdown menu is populated from the City table. Users can select their city here. The Customer table will store the City_id instead of the city_name for data consistency.

k) Customer Type- A dropdown menu sourced from the Customer Type table. Users can select their customer type here. The system stores only the

Customer_type_id in the Customer table instead of the customer type name for data integrity purposes.

Upon successful collection of data from all input fields, the Flask web application establishes a connection with the MySQL database. It utilizes this connection to execute an INSERT query, enabling the storage of gathered data into the Customer and Login tables within the database. This INSERT query executed by the Flask application is designed to efficiently organize and save the user-provided information. It ensures that the collected details, including the user's name, address details, zip code, email, password, selected country, state, city, and customer type, are appropriately stored in their respective fields within the Customer table. Simultaneously, the provided email and password are securely encrypted and stored in the Login table. This two-step data insertion process maintains data integrity and security within the application's database architecture.

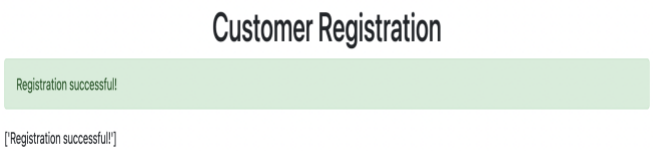

Figure 5: Customer Registration Page
(When user register system will display Registration successful message)

2.Customer Login Page-

The Login Page is an integral component of the Medical Equipment Ordering System, acting as the gateway for users to access their accounts and initiate interactions with the database. It is designed to provide a secure and user-friendly authentication process, allowing registered users to seamlessly log in and utilize the system's features.

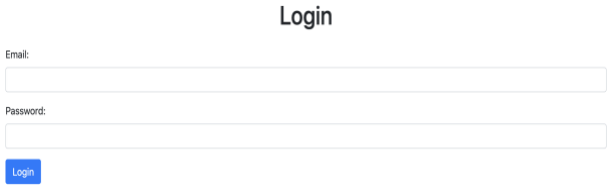

Figure 6: Customer Login Page

Functionality:

User Authentication: The page prompts users to enter their registered email and password. Upon submission, the system verifies the credentials against the stored records in the database using a secure authentication process.

Email Verification: The system checks if the entered email is formatted correctly and is present within the Login table of the database. If the email does not exist or is incorrectly formatted, an error message is displayed, guiding the user to either re-enter the information or register if they haven't done so. The Python application queries the Login table in the database, specifically the email column. The SQL command executed looks like: SELECT email FROM Login WHERE email = %s.

Password Validation: If the email exists, the system then validates the entered password. The password stored in the database is hashed for security purposes. Using bcrypt, the system compares the hashed password against the one provided by the user at login. The application retrieves the hashed password from the database for the given email. The SQL command looks like: SELECT password FROM Login WHERE email = %s. The bcrypt library is used to compare the hashed password from the database with the plaintext password entered by the user. If they match, authentication is successful.

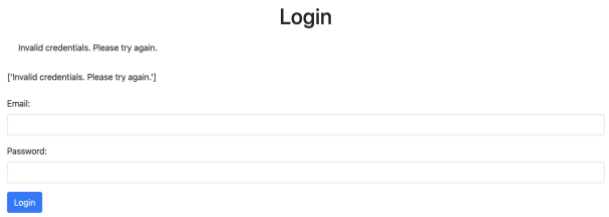

Figure 7: Customer Login with Invalid Credentials

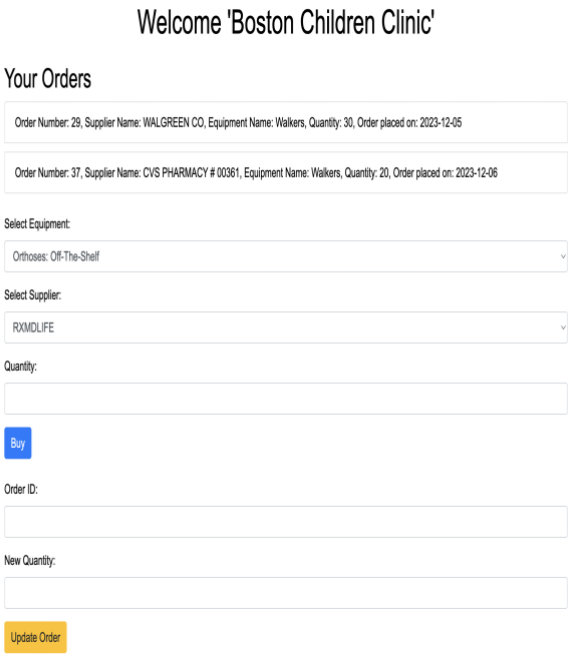

Figure 7: Customer Login with Valid Credentials (When user login with valid credential it will land on Customer Order page which user can further use for order management.)

Session Management: Upon successful authentication, the system creates a session for the user. It stores essential session variables such as email and customer_id, which are used throughout the user's interaction with the system to personalize the experience and maintain the security of transactions. A new session record is created in the server's session storage, not directly in the database. However, user-related data such as customer_id might be fetched from the Customer table at this point. The Flask application uses its session object to store user-specific data like customer_id and email, which are essential for personalizing the user's experience.

3. Customer Dashboard Page-

The Customer Dashboard Page is the core of the Medical Equipment Ordering System for authenticated customers. Once the supplier has successfully logged in, this page serves as the personalized hub for managing orders, browsing equipment, and interacting with customers.

Upon accessing the page, it checks if the customer is authenticated by looking for an 'email' in the session. If not found, it redirects to the login page. Once authenticated, it connects to the database to fetch customer details, available equipment, suppliers' information, and the customer's order history. These details are then passed to the HTML template, which renders the page displaying a personalized dashboard for the customer.

Functionality:

Order Management: The dashboard provides an overview of the user's active orders and a historical log of past transactions. Users can track the status of their orders, update quantities, or cancel orders as needed. User accesses their dashboard to view current and past orders. The Orders table is queried to retrieve the user's orders, joining with Equipment and Supplier tables to provide detailed views. Flask routes fetch and display order data, allowing users to manage their orders within their dashboard view.

Equipment Information: The dashboard features a comprehensive list of available medical equipment. Users can browse the catalog and select items for purchase based on their needs. To populate Equipment dropdown, we have used stored procedure “up_read_equipment”.

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `up_read_equipment`()
BEGIN

select * from Equipment
order by equipment_id;

END
```

Figure 8: up_read_equipment Stored Procedure

Supplier Information: Supplier is again a dropdown, but we have made sure to list only those suppliers which are available in same city as that of customer. We have used session variable customer_id to get city_id and pass that to

stored procedure "up_read_supplier_by_city" as input parameter. Stored procedure will return list of suppliers.
Also, to get supplier we have used SupplierEquipmentList table so that we only get suppliers which supplies an equipment.

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `up_read_supplier_by_city`(
 IN cityid int
)
BEGIN

 select  distinct SEL.supplier_id,supplier_name from SupplierEquipmentList SEL
 inner join Supplier S on SEL.supplier_id = S.supplier_id
 where  city_id  = cityid ;

 END
```

Figure 9: up_read_supplier_by_city Stored Procedure

Placing New Orders: The page includes a streamlined process for placing new orders. Users can select the desired equipment, quantity, and supplier, and confirm their order with a single click. Users select equipment, specify quantity, choose a supplier, and place an order. Once, user has selected all the details and click on "Buy" button system will pass all the parameters to "up_write_order" stored procedure which will insert data into Orders table.

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `up_write_order`(
supplierid Int,
equipmentid Int,
ordername varchar(255),
orderdate date,
customerid int,
ordertypeid int,
quantity int
)
BEGIN

INSERT INTO `Medical_equipment_system`.`Orders`
(`supplier_id`,`equipment_id`,`order_name`,`order_date`,`customer_id`,`ordertype_id`,`quantity`)
VALUES
(supplierid ,equipmentid ,ordername ,orderdate ,customerid ,ordertypeid ,quantity
);

END
```

Figure 10: up_write_order Stored Procedure

Also, whenever a new order is inserted, a trigger is being called which will update the remaining quantity of the equipment in SupplierEquipmentList table.

```
BEGIN


 UPDATE SupplierEquipmentList
 set stock = stock - new.quantity
 where supplier_id = new.supplier_id
 and equipment_id = new.equipment_id;

 END
```

Figure 11: After insert trigger on Orders Table

Updating Orders: The page allows users to update their orders based on the order id. It asks user to input their order id and the new quantity that user wants to add or delete. When the user clicks on the update button the system will pass the order_id and the quantity to the stored procedure "up_update order". The stored procedure retrieves the equipment and supplier for the given order. It then checks if the equipment is available in SupplierEquipmentList with sufficient stock or not. It then updates the order quantity if the requested quantity is less than or equal to available stock.

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `up_update_order`(
 IN orderid int,
 IN quantitynew int)
BEGIN

 DECLARE stock_available INT;

     -- Retrieve equipment and supplier for the given order
 DECLARE equipment_id_val INT;
 DECLARE supplier_id_val INT;
 SELECT equipment_id, supplier_id INTO equipment_id_val, supplier_id_val FROM `Orders` WHERE order_id = orderid;

     -- Check if the equipment is available in SupplierEquipmentList with sufficient stock
 SELECT stock INTO stock_available
 FROM SupplierEquipmentList
 WHERE equipment_id = equipment_id_val AND supplier_id = supplier_id_val;

     -- Update order quantity if the requested quantity is less than or equal to available stock
IF stock_available >= quantitynew THEN
     UPDATE `Orders` SET quantity = quantitynew WHERE order_id = orderid;
END IF;

 END
```

Figure 12: up_update_order Stored Procedure

Also, whenever the order is updated, a trigger is being called which will update the stock in SupplierEquipmentList table based on the quantity that user inputs.

```
BEGIN

UPDATE SupplierEquipmentList
set stock = stock + old.quantity - new.quantity
where supplier_id = new.supplier_id
and equipment_id = new.equipment_id;

END
```

Figure 13: After update trigger on Orders Table

Deleting Orders: The page includes a functionality to delete orders. The user can delete their order that they have placed. It asks the user to input their order_id of the order they want to delete. Once the user presses the delete button the order_id is passed to a stored procedure "up_delete_order". The stored procedure then deletes the

order from the Orders table where order_id is equal to the order_id that customer gave as an input.

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `up_delete_order`(
IN orderid int)
BEGIN

Delete from orders where order_id=orderid;

END
```

Figure 14: up_delete_order Stored Procedure

Also, whenever a order is deleted, a trigger is being called which will increase the stock of the supplier in the SupplierEquipmentList table.

```
BEGIN

UPDATE SupplierEquipmentList
set stock = stock + old.quantity
where supplier_id = old.supplier_id
and equipment_id = old.equipment_id;

END
```

Figure 15: After Delete Trigger on Orders Table

4. Analytics Dashboard-

Figure 16: Dashboard 1

We have used user defined function to populate above three fields in the dashboard, these are listing distinct suppliers, states and equipment which are active in the orders like count of equipments for which order has been placed.

```
CREATE DEFINER=`root`@`localhost` FUNCTION `CountDistinctSuppliers`() RETURNS int
    DETERMINISTIC
BEGIN

DECLARE distinct_supplier_count INT;

SELECT COUNT(DISTINCT supplier_id) INTO distinct_supplier_count
FROM Orders;

RETURN distinct_supplier_count;

END
```

Figure 17: CountDistinctSuppliers Function

```
CREATE DEFINER=`root`@`localhost` FUNCTION `CountDistinctStates`() RETURNS int
    DETERMINISTIC
BEGIN

DECLARE distinct_state_count INT;

SELECT COUNT(DISTINCT State.state_id) Into distinct_state_count
FROM State inner join City on City.state_id=State.state_id
inner join Supplier on City.city_id=Supplier.city_id
inner join Orders on Orders.supplier_id=Supplier.supplier_id;

RETURN distinct_state_count;

END
```

Figure 18: CountDistinctStates Function

```
CREATE DEFINER=`root`@`localhost` FUNCTION `CountDistinctEquipments`() RETURNS int
    DETERMINISTIC
BEGIN

DECLARE distinct_equipment_count INT;

SELECT COUNT(DISTINCT equipment_id) INTO distinct_equipment_count
FROM Orders;

RETURN distinct_equipment_count;

END
```

Figure 19: CountDistinctEquipments Function

Then we have utilized views to get analytics on the orders like top 10 best selling equipments , average sales state wise and average sales day wise.

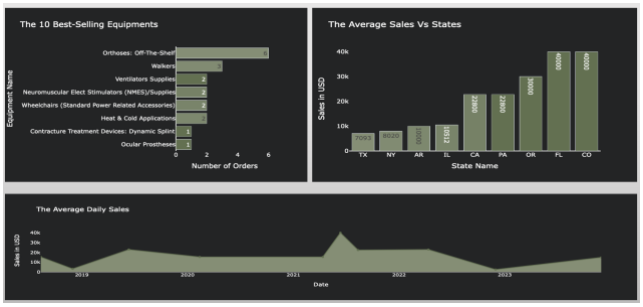

Figure 20: Dashboard 2

```
CREATE
    ALGORITHM = UNDEFINED
    DEFINER = `root`@`localhost`
    SQL SECURITY DEFINER
VIEW `average_sales_order_date` AS
    SELECT
        `orders`.`order_date` AS `order_date`,
        CAST(AVG((`orders`.`quantity` * `equipment`.`price`))
            AS SIGNED) AS `average_sales`
    FROM
        (`orders`
        JOIN `equipment` ON ((`orders`.`equipment_id` = `equipment`.`equipment_id`)))
    GROUP BY `orders`.`order_date`
    ORDER BY `orders`.`order_date` , AVG((`orders`.`quantity` * `equipment`.`price`))
```

Figure 21: Average_sales_order_date view

```
CREATE
    ALGORITHM = UNDEFINED
    DEFINER = `root`@`localhost`
    SQL SECURITY DEFINER
VIEW `average_sales_state_wise` AS
    SELECT
        `state`.`state_name` AS `state_name`,
        CAST(AVG((`orders`.`quantity` * `equipment`.`price`))
            AS SIGNED) AS `average_sales`
    FROM
        ((((`orders`
        JOIN `supplier` ON ((`orders`.`supplier_id` = `supplier`.`supplier_id`)))
        JOIN `city` ON ((`city`.`city_id` = `supplier`.`city_id`)))
        JOIN `state` ON ((`state`.`state_id` = `city`.`state_id`)))
        JOIN `equipment` ON ((`orders`.`equipment_id` = `equipment`.`equipment_id`)))
    GROUP BY `state`.`state_name`
    ORDER BY AVG((`orders`.`quantity` * `equipment`.`price`))
```

Figure 22: Average_sales_state_wise view

```
CREATE
    ALGORITHM = UNDEFINED
    DEFINER = `root`@`localhost`
    SQL SECURITY DEFINER
VIEW `best_selling_product_overall` AS
    SELECT
        `equipment`.`equipment_name` AS `equipment_name`,
        COUNT(`orders`.`order_id`) AS `total_orders`
    FROM
        (`orders`
        JOIN `equipment` ON ((`orders`.`equipment_id` = `equipment`.`equipment_id`)))
    GROUP BY `orders`.`equipment_id`
    ORDER BY COUNT(`orders`.`order_id`) DESC
    LIMIT 10
```

Figure 23: Best_selling_product_overall view

CONCLUSION

Throughout the development of the Medical Equipment Ordering System, our team has acquired a multitude of skills and insights. The project was a deep dive into the intricacies of database design, user interface creation, and the integration of front-end and back-end technologies. During the course of this project we have learned:

- **Complexity of Database Systems**: We have gained a greater appreciation for the complex nature of database systems, especially regarding normalization and the importance of maintaining data integrity.
- **User Experience Design:** Crafting user-friendly interfaces taught us that the user experience is paramount. Simplifying complex processes into intuitive steps was both challenging and rewarding.
- **Security Considerations:** Implementing secure login and session management highlighted the critical role of cybersecurity in safeguarding user data.
- **Agile Development:** This project reinforced the value of iterative development and testing, which enables continuous improvement and responsiveness to user feedback.

Future Scope:

We would enhance our analytics capabilities to include more predictive models using machine learning, providing users with actionable insights for decision-making. Although our system is responsive, further optimization for mobile platforms would ensure accessibility on all devices. We aspire to incorporate more personalized features, such as custom alerts for new equipment availability or price changes. Expanding the system's capability to integrate with external healthcare systems would streamline operations for users even further.

In conclusion, the project was a significant learning experience that went beyond the technical aspects of building a database system. It was an exercise in problem-solving, teamwork, and the application of theoretical knowledge in a practical setting. Future teams embarking on such projects should view challenges as opportunities for growth and be prepared to adapt to the evolving landscape of technology and user needs.