Database Design and Implementation Report

ISYS2099 – Database Application

Lecturer: Mr. Tri Dang Tran

Nguyen Ha Kieu Anh - s3818552

Phan Le Quynh Anh - s3927427

Nguyen Pham Tan Hau - s3978175

Ong Gia Man - s3938231

Vu Nguyet Minh – s3878520

Table of Content

I. Introduction

In the rapidly evolving landscape of healthcare, the convergence of medical advancements and cutting-edge technology has revolutionized hospital operations. As a result, robust management systems have become indispensable assets for modern healthcare institutions. As a result, our team has provided a group project focusing on designing a hospital management system, with a goal of assisting hospital stakeholders such as patients, doctors, managers, and administrators by addressing database design, performance analysis, data integrity, and data security.

For this project, MySQL for structured data management and MongoDB for handling flexible, document-oriented information will be chosen. For the system's architecture, we will employ NodeJS, a versatile JavaScript runtime, to develop both the front-end and back-end components of our website. Express and Mongoose are also used to develop the website's back-end. For front-end development, we utilized Material UI, React, and TailwindCSS. This report will explain further our group's system, focusing on database handling.

II. Project Description and Implementation Details

a) Database Design

As a database project, database design is the initial and most crucial stage to obtain a structured and competent system. In this section, we will explain clearly the process of designing our database, including data analysis, entity-relationship diagrams (**ERD**) and relational schema, relationships, constraints, as well as non-relational documents and their structures.

i. Data analysis

For our database, the primary entities interacting within the model are patients and staff members, including doctors, administrators, and managers. Initially, nurses are also one of the entities, but we decided to not include them since they do not serve any significant interaction.

All users can log in to the system using their email and password. Only patients can sign up for a new account and edit their personal information, including name, email address, phone number, and allergies. The patient_credentials and staff_credentials tables store only login credentials for users to ensure that no sensitive patient or staff information, such as personal details or medical data, is stored in these tables, maintaining privacy and security.

Booking appointments is a patient's unique feature, They can view a list of doctors with available future time slots, book an appointment with their desired doctor, and add notes before confirming the appointment. They can also view their treatment and appointment log, including past and future appointments, with the option to cancel future appointments. Appointments are assumed to be confirmed automatically after a patient has booked it. Past appointments will include past payments and treatment notes, with a maximum of one treatment note per appointment. A treatment note is automatically created when an appointment is booked, and the doctor's availability status for that slot changes from 'Available" to "Busy". If the appointment is canceled, the treatment note will be deleted, and the doctor's status will revert from "Busy" to "Available".

Doctors can view the list of appointments they have completed and the associated treatment notes. Furthermore, they can also view their upcoming appointments. For each appointment, they can update the treatment for the patient. Doctors can also view, add, or delete their upcoming work schedule, but not their past schedule.

Administrators and managers have similar features, with the main difference being that admins can view all staff members, while managers can only view the staff members they manage. If a manager adds a new doctor to the system, that doctor will automatically be assigned to the manager. However, if an admin adds a new doctor, they will need to assign a manager to the doctor. Both admins and managers can view staff members, including filtering by department and name (in ascending or descending order). They can also update a staff member's information, including their job, salary, and department, and view the staff member's job history. Additionally, they can view the working schedule and workload of all doctors within a given duration or of a specific doctor. Admins and managers can also search for patients by name or ID and view all patients' treatment histories within a given duration or for a specific patient.

ii. ERD and Relational Schema

Please refer to the Appendix section for images of the ERD and Relational Schema.

Entities and their attributes

- **Patient:** patient_id (PK), patient_name, allergies, contact_number, dob, gender, address
- **Patient Credential:** patient_id (PK), email, password, created_at, updated_at
- **Staff:** staff_id (PK), department_id (FK), manager_id (FK), qualification, staff_name, salary, job_type, start_date, updated_at
- **Appointment:** appointment_id (PK), patient_id (FK), staff_id (FK), purpose, status, appointment_datetime, end_time, payment_amount
- **Treatment Note:** treatment_id (PK), patient_id (FK), appointment_id (FK), diagnosis, patient_id, procedure, treatment_date, medication, instruction
- **Job History:** staff_history_id (PK), staff_id (FK), department_id (FK), manager_id (FK), staff_name, qualification, salary, job_type, updated_at
- **Department:** department_id (PK), department_name
- **Doctor Schedule:** schedule_id (PK), staff_id (FK), shift_start, shift_end, availability_status
- **Performance Rating:** performance_id (PK), doctor_id (FK), appointment_id (FK), performance_rating

Table Relationships:

- **Patient and Patient Credential:** One and Only One relationship, where one patient can only have one set of credentials available at once, and vice versa.
- **Patient and Appointment:** One-to-Many relationship, as one patient can book multiple appointments, but each appointment can only be booked by one patient.
- **Performance Rating and Appointment:** One and Only One relationship, illustrating that an appointment can only have one performance rating attached to it, and vice versa.
- **Treatment Note and Appointment:** One and Only One relationship, where one appointment can only lead to one treatment history log, and vice versa.

- **Billing Report and Appointment:** One and Only One relationship, showing that one appointment can only have one billing report, and vice versa.
- **Staff and Appointment:** One-to-Many relationship, illustrating that one staff member (doctor in this project) can be assigned to multiple appointments, but each appointment can only be assigned to one staff member.
- **Staff and Performance Rating:** One-to-Many relationship, where one staff member can receive multiple performance ratings, but one rating can only refer to one staff member.
- **Staff and Job History:** Zero-to-Many relationship, showing that one staff member can have their job history changed multiple times or none, but each job history log in unique to one staff member.
- **Staff and Department:** One-to-Many relationship, showing that one department can have multiple staff members, but one staff can only belong to one department.
- **Staff and Doctor Schedule:** One-to-Many relationship, illustrating that one staff member can have and edit their unique schedule, but each schedule only applies to one staff member.

Unique Constraints

- **patient_id** and **staff_id** are primary keys in patient, patient_credentials, staff and staff_credentials tables, ensuring unique identification for each record.
- **uq_patient_credentials_email:** Ensures that patient email addresses are unique within the patient_credentials table.
- **uq_staff_credentials_email:** Ensures that staff email addresses are unique within the staff_credentials table.
- **appointment_id** is the primary key in the appointments table, ensuring that each appointment ID is unique within the table.
- **treatment_id** is the primary key in the treatments table, ensuring each treatment note's ID is unique to the Treatment Note table.
- **performance_id** is the primary key in the performance_ratings table, ensuring that each performance rating ID is unique within the table.
- **staff_history_id** is the primary key in the job_history table, ensuring that each staff member's history log ID is unique within the table.
- **schedule_id** is the primary key in the doctor_schedule table, ensuring that each doctor's schedule ID is unique within the table.
- **department_id** is the primary key in the department table, ensuring that each department ID is unique within the table.

Foreign Key Constraints

- **fk_patient_credentials_patient**: Connects patient_credentials to patients, ensuring that each credential is associated with a valid patient.
- **fk_department_id:** Connects staff to departments, ensuring that each staff member belongs to a valid department.
- **fk_manager_id:** Connects staff to itself, linking each staff member to their manager who is also a staff member.

- **fk_staff_credentials_staff**: Connects staff_credentials to staff, ensuring that each staff credential is associated with a valid staff member.
- **fk_appointment_patient**: Connects appointments to patients, ensuring that each appointment is associated with a valid patient.
- **fk_appointment_staff:** Connects appointments to staff, ensuring that each appointment is associated with a valid staff member.
- **fk_treatment_appointment:** Connects treatments to appointments, ensuring that each treatment is associated with a specific appointment.
- **fk_treatment_patient_id**: Connects treatments to patients, ensuring that each treatment is associated with a valid patient.
- **fk_staff_id:** Connects job_history to staff, ensuring that each job history record is associated with a valid staff member.
- **fk_history_department_id**: Connects job_history to departments, ensuring that each job history record is associated with a valid department.
- **fk_history_manager_id**: Connects job_history to staff, ensuring that each job history record is associated with a valid manager.
- **fk_performance_doctor:** Connects performance_rating to staff, ensuring that each performance rating is linked to a valid doctor.
- **fk_performance_appointment:** Connects performance_rating to appointments, ensuring that each performance rating is associated with a specific appointment.
- **fk_schedule_staff:** Connects doctor_schedules to staff, ensuring that each schedule is associated with a valid staff member.

iii. Non-relational Documents and their Structure

To handle non-relational documents, our group has implemented Mongoose, a MongoDB object modeling tool, since MongoDB stores data in Binary JSON (BSON), allowing the structure of each document to vary, which is specifically convenient when handling unstructured data, such as patient notes, treatment logs, or job history logs.

In this project, we utilized MongoDB to store unstructured data for patient's treatment note. This comprises of various fields such as patient_note, doctor_notes, diagnostic_images, lab_results, and prescriptions. Since the data for these fields may be lengthy, they are designed to allow for detailed and flexible storage of treatment-related information. Additionally, diagnostic images might consist of an array of multiple images, each with a different URL. Storing this type of array data in a relational database is challenging due to its complexity, as MySQL does not natively support arrays [10], which increases the potential for errors. By making use of NPM package "Multer", the process of storing and retrieving images is much easier. The treatment_id field is used to link with the MySQL Treatments table, ensuring consistency between the relational and NoSQL databases.

Moreover, the treatment process may involve additional data not stored in the relational database. The flexible schema of MongoDB, as a NoSQL database, allows for easier storage and management of such additional data without having to declare rigid table structures or predefined fields.

With this approach, we gain flexibility to manage dynamic data structures while using the schema, ensuring data integrity and consistency, as well as increasing data efficiency.

b) Performance Analysis

i. Query Optimization (Index & Partition)

In addition to the default primary indexes created when declaring the primary key for each table, we add additional secondary indexes to enhance query performance and optimize data retrieval. The choice of columns to be indexed is based on the join and WHERE conditions of the queries in the project.

However, it is worth noting that partitions cannot be used in tables that have foreign keys. Due to how partitions work, spreading referencing values across multiple partitions, enforcing them in tables with foreign keys can lead to performance issues or failure in foreign key check consistency.

Appointments Table:

- **idx_appointments_patient_id:** To speed up retrieval of appointment records based on the patient_id column.
- **idx_appointments_staff_id:** To speed up retrieval of appointment records based on the staff_id column.
- **idx_appointments_start_end_time:** To speed up retrieval of appointment records using start_time and end_time, or shift_start only.
- **idx_appointments_status_start_end:** To speed up retrieval of appointment records based on status, start_time, and end_time, or status only.

Doctor Schedules Table:

- **idx_doctor_schedules_staff_id**: To speed up retrieval of doctor schedules based on staff_id.
- **idx_doctor_schedules_shift_start_end**: To speed up retrieval of doctor schedules using shift_start and shift_end, or shift_start only.
- **idx_doctor_schedule_status_shift_start_end**: To speed up retrieval of doctor schedules based on availability_status, shift_start, and shift_end, or availability_status only.
- **idx_doctor_schedule_shift_end**: To speed up retrieval of doctor schedules based on shift_end.

Job History Table

- **idx_job_history_staff_id**: To speed up retrieval of job history records based on staff_id.
- **idx_job_history_manager_id**: To speed up retrieval of job history records based on manager_id.
- **idx_job_history_department_id**: To speed up retrieval of job history records based on department_id.

Staff Credentials Table:

idx_staff_credentials_email_password: To speed up retrieval of staff credentials based on email and password, or email only.

Patient Credentials Table:

idx_patient_credentials_email_password: To speed up retrieval of patient credentials based on email and password, or email only.

Patients Table:

idx_patients_patient_name: To speed up retrieval of patient records based on patient_name by leveraging full-text index.

Performance Rating Table:
- **idx_performance_rating_doctor_id**: To speed up retrieval of performance ratings based on doctor_id.
- **idx_performance_rating_appointment_id**: To speed up retrieval of performance ratings based on performance_id.

Staff Table:
- **idx_staff_manager_id**: To speed up retrieval of staff records based on manager_id.
- **idx_staff_department_id**: To speed up retrieval of staff records based on department_id.

Treatments Table:
- **idx_treatments_patient_id**: To speed up retrieval of treatments based on patient_id.
- **idx_treatments_appointment_id**: To speed up retrieval of treatments based on appointment_id.

ii. Concurrent Access

Concurrent database access refers to a situation where multiple users access the database consistently. **START TRANSACTIONS**, **COMMIT**, and **ROLLBACK** statements were implemented in our SQL transactions to manage concurrent access and establish optimal performance. These statements ensure all operations in a transaction either all succeed or all fail, which is crucial in a concurrent operation. [1]

Additionally, isolation levels are also a considerable factor to manage concurrent access. Specifically, we use **FOR UPDATE** in our **SELECT** statements to control data visibility and locking behavior. For example, applying the **FOR SHARE** clause indicates to MySQL that the selecting rows are only for reading, forbidding any modification until the transaction is complete. This ensures no exclusive locks (from **FOR UPDATE**) are acquired until the shared locks are deactivated. Furthermore, the clause **FOR UPDATE** indicates that the selected rows can be modified during the transaction, placing other transactions in a wait state until the current one either commits or rollbacks.

By leveraging the isolation level, we use the FOR UPDATE SELECT statement to ensure that only one patient can book an appointment for a specific doctor slot at a time, preventing multiple patients from booking the same slot concurrently.

c) Data Integrity

i. Transaction

Transactions, which consist of multiple database duties grouped into a single unit of work, ensure that the task is completed successfully, or that the database reaches a stable state in case of a failure. [2] In our **hospital_rules.sql** file, transactions are initiated with the **START TRANSACTION** statement, followed by either **COMMIT** or **ROLLBACK**. Procedures such as

sp_cancel_appointment rely on transactions to maintain atomicity. In the case of an error, these procedures' integrity is still preserved by rolling back the transaction.

For instance, in procedures such as **DeleteStaffByAdmin** and **sp_cancel_appointment**, the system ensures that all related operations, including booking or canceling an appointment and managing the corresponding treatment records, are encapsulated within a transaction. If a conflict occurs—such as a timeslot already being booked during the appointment process, or an issue arises while canceling the appointment—the transaction is rolled back. This rollback prevents any incomplete or inconsistent data from being saved, ensuring that both the appointment details and the doctor's schedule remain accurate and consistent.

ii. Stored Procedure

Special procedures known as triggers are automatically executed in response to specific database changes, such as **INSERT**, **UPDATE**, or **DELETE** operations on tables. In our system, the stored procedure automatically checks all conditions before executing any operation. The operations can only be executed if all conditions passed, ensuring data integrity and consistency.

To effectively handle exceptions, our group has integrated error handling mechanisms within the triggers and stored procedures. An example of this is the **DECLARE CONTINUE HANDLER FOR SQL EXCEPTION** and **SIGNAL SQLSTATE** statement is used to control how stored procedures or triggers should respond when an exception is caught. This allows for appropriate actions to be taken, such as signaling a failed transaction by setting a rollback flag or providing feedback through custom error messages. This mechanism will ensure smooth execution and enhance overall system stability by properly managing errors. [4]

iii. Error Handling

The system's error-handling mechanisms are critical for maintaining security and enforcing role-based access. In the **AddStaff** procedure, managers are only permitted to add doctors. If they attempt to add other staff roles, such as administrators, the system generates an error and immediately stops the operation, preventing unauthorized actions. Similarly, the **UpdateStaffInfo** procedure ensures that managers can only modify the details of staff members under their supervision. If a manager attempts to update information outside their authority, an error is raised, and the update is blocked. These built-in error-handling processes not only prevent unauthorized changes but also preserve data integrity and ensure that all actions within the system adhere to the defined rules and permissions structure. Additionally, SIGNAL SQLSTATE is used whenever input fails to meet certain conditions in the procedures, terminating the process and returning error messages to the user. This ensures robust error handling and safeguards against invalid operations.

Summary

This section has shown how we have implemented a combination of transactions, stored procedures, and error handling to ensure data changes adhere to doctor-patient confidentiality and maintain consistency within the database. By enforcing this approach, scenarios like partial updates or data inconsistency due to errors or unexpected events can be prevented effectively and addressed before committing changes, preserving the integrity of the system.

d) Data Security

i. Database User Permission

Patient User can:

- SELECT, UPDATE their own information: patients table
- SELECT, INSERT, DELETE their own appointments: appointments table
- SELECT, INSERT, DELETE the treatment with the associated appointment: treatments table
- SELECT to view staff information, restricted to column staff_id, staff_name, department: staff table
- SELECT department name: department table
- SELECT to view doctor's schedule: doctor_schedules table

Doctor User can:

- SELECT their own appointments: appointments table
- SELECT, UPDATE patient's treatment note: treatments table
- SELECT, INSERT, DELETE their own schedules: doctor_schedules

Manager User can:

Note: The Manager role has almost the same features as the Admin role. However, the Admin role is more powerful as it has full control over the entire database, including the ability to modify and configure all records, whereas the Manager can only manage the related data of the staff they supervise. Additionally, the Manager does not have the privilege to delete records from the database.

- SELECT, INSERT new staff credentials: staff_credentials table
- SELECT, INSERT, UPDATE doctor information of that manager: staff table
- SELECT department name: departments table
- SELECT doctor's schedules of that manager: doctor_schedules table
- SELECT department: department table
- SELECT doctor's workloads, all columns except for payment_amounts: appointments table
- SELECT doctor's job history of doctor under manager's supervision: job_history table
- SELECT patient's information: patients table
- SELECT patient's treatment history of the doctor that the manager supervises: treatments table
- SELECT doctor's performance rating: performance_rating table

Admin User can:
Note: The Admin role has the most privileges in the system. However, due to the scope of this course, we have only implemented functions that utilize these privileges. While we could grant all privileges to the admin, just as the default root user in MySQL, we prefer to adhere to the principle

of least privilege. As a result, if the project expands and the Admin role requires additional functions, more privileges will be granted accordingly. For now, the Admin role differs from the Manager role in that it includes the ability to delete staff records and view the hospital's payment reports.

- SELECT, INSERT, DELETE staff credentials: staff_credentials table
- SELECT, INSERT, UPDATE, DELETE doctor: staff table
- SELECT department name: departments table
- SELECT doctor's schedules: doctor_schedules table
- SELECT department: department table
- SELECT doctor's workloads: appointments table
- SELECT doctor's job history of doctor: job_history table
- SELECT patient's information: patients table
- SELECT patient's treatment history of the doctor that the manager supervises: treatments table
- SELECT doctor's performance rating: performance_rating table
- DELETE staff information for any staff: staff_credentials and staff table
- SELECT payment report

ii. SQL injection prevention

Regarding SQL injection prevention, we utilized the least privileged principle, which as mentioned above, is restricting the database user's permission. Each user is granted limited authorization to perform operations based on their role, minimizing the risk of exploitation and the potential damage of an injection attack.

In addition, we rely on parameterized queries to further prevent SQL injection attacks. Since input values are treated as data instead of executable code when using parameterized queries, any user-supplied input, form fields for instance, is ensured to escape correctly and cannot alter the SQL query's structure. This approach prevents malicious users from injecting harmful SQL code into the query, since their input will be confined to specific parameters. [5]

Finally, we apply stored procedures to provide another layer of security. Stored procedures let us define specific sets of operations that can be executed in the database, allowing limited direct access to the underlying SQL queries. Implementing this approach enhances control over the database interactions.

iii. Password hashing

For our system, we implemented node.bcrypt.js, a **bcrypt** library for **NodeJS**, for password hashing. Since **bcrypt** applies a hashing algorithm that alters the original password into a fixed-length string of characters, along with the extra computational demand, making it considerably more difficult to reverse-engineer and slows down the attack.

Another crucial advantage of using **bcrypt** is the ability to handle password salting, which is the process of adding a random string to the password before hashing it, ensuring that users who have the same password will have different values after hashing. [7]

Due to the nature of a hospital management system, where patient data is extremely confidential, **bcrypt** will ensure strong password protection. By implementing **bcrypt**, we believe that every user's account will be safeguarded, while also strengthening overall data security, and guaranteeing minimized unauthorized access.

iv. Additional security mechanisms

For this project, we store configuration and sensitive information, like database credentials, API keys, and passwords, in a **.env** file separate from the actual code. Doing this prevents sensitive data from the codebase and ensures it will not be leaked accidentally. [8]

JSON Web Tokens (JWT) was also implemented for user authentication and session management, since JWT tokens are a secure way to verify a user's identity without requiring the server to store session information by sending the user a token generated every time that user logs in, verifying the user's authorization. JWT secret key is stored in the **.env** file, further ensuring the token's integrity. Furthermore, JWTs include an expiration time, limiting the session that they can be used, enhancing the system's overall security. [9]

III. Conclusion

This report has discussed the four main objectives, including database design, performance analysis, data integrity, and data security. They are all pivotal portions to create a basic hospital management system, specifically highlighting the database handling part.

The database design stage is the first and foremost step in our project, as it helped us form a basic, yet clear vision of our database models, along with discovering issues we would not have spotted without careful consideration. Performance analysis aids us in important query optimizations, indices implementation, and reaching settlement regarding concurrency access for optimal performance. Efficient transactions, triggers, and errors handling are the significant parts of our data integrity section, enforcing patient-doctor confidentiality and preserving data consistency throughout our database operations. Regarding data security, stored procedures were used to restrict a user's authorizations to their intended role, SQL injection prevention is strengthened by limiting authorization for each user role, enforcing parameterized queries and stored procedures, hashed passwords secured with **bcrypt**, and .env file and **JWTs** were implemented for extra security.

To conclude, this project has given us a chance to work on a basic database and its application in real life. Applying the principal objectives in database design, performance, data integrity and data security has been an important learning opportunity for our group. In the future, we wish to further improve our system, creating a more significant and complex, as well as a more user-friendly website, making it more applicable than it currently is.

IV. References

[1] GeeksForGeeks, “Concurrency Control in DBMS", geeksforgeeks.org, Mar. 12, 2024. [Online]. Available: https://www.geeksforgeeks.org/concurrency-control-in-dbms [Accessed Aug. 26, 2024]

[2] Javatpoint, “Consistency in DBMS", javatpoint.com, [Online]. Available: https://www.javatpoint.com/consistency-in-dbms [Accessed Aug. 29, 2024]

[3] MongoDB, “What Are Database Triggers?”, mongodb.com, [Online]. Available: https://www.mongodb.com/resources/products/capabilities/database-triggers [Accessed Aug. 30, 2024]

[4] MySQL, “15.6.7.2 DECLARE ... HANDLER Statement”, dev.mysql.com, [Online]. Available: https://dev.mysql.com/doc/refman/8.4/en/declare-handler.html [Accessed Aug. 30, 2024]

[5] H. Fadlallah, “Using parameterized queries to avoid SQL injection”, sqlshack.com, Nov. 18, 2022. [Online]. Available: https://www.sqlshack.com/using-parameterized-queries-to-avoid-sql-injection [Accessed Sept. 1, 2024]

[6] W3Schools, “SQL Stored Procedures”, w3schools.com, [Online]. Available: https://www.w3schools.com/sql/sql_stored_procedures.asp [Accessed Sept. 2, 2024]

[7] Auth0 by Okta, “Hashing in Action: Understanding bcrypt”, auth0.com, Feb. 25, 2021. [Online]. Available: https://auth0.com/blog/hashing-in-action-understanding-bcrypt [Accessed Sept. 3, 2024]

[8] A. Schimelpfening, “How to use dotenv in your CSE341 projects", gist.github.com, [Online]. Available: https://gist.github.com/727021/5e17e739bd0a286cd5e30b2c8f69ed2d#using-env [Accessed Sept. 5, 2024]

[9] npm, “jsonwebtoken", npmjs.com, 2023. [Online]. Available: https://www.npmjs.com/package/jsonwebtoken [Accessed: Sept. 5, 2024]

[10] MySQL, “13.5 The JSON Data Type”, dev.mysql.com, [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/json.html [Accessed: Sept. 9, 2024]

V. Appendix

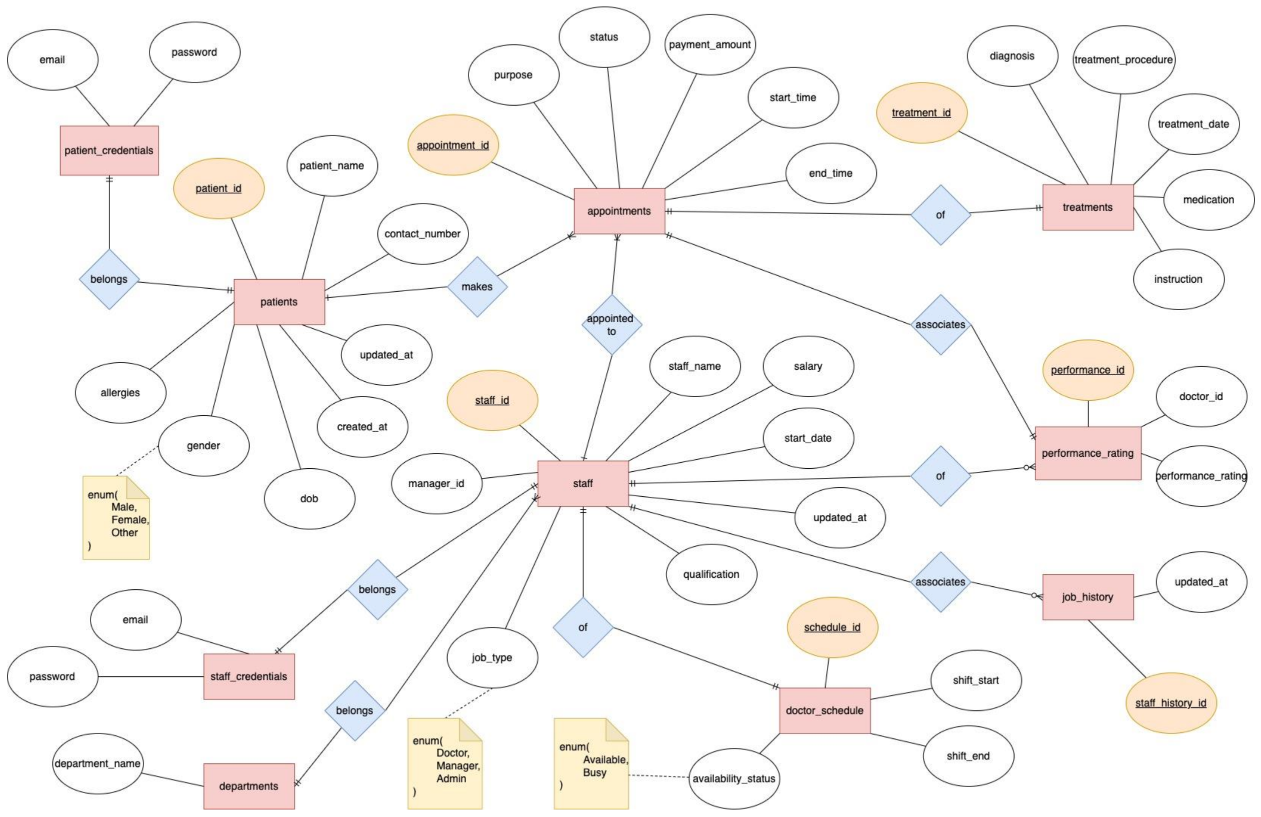

Figure 1: Entity-Relationship Diagram (ERD)

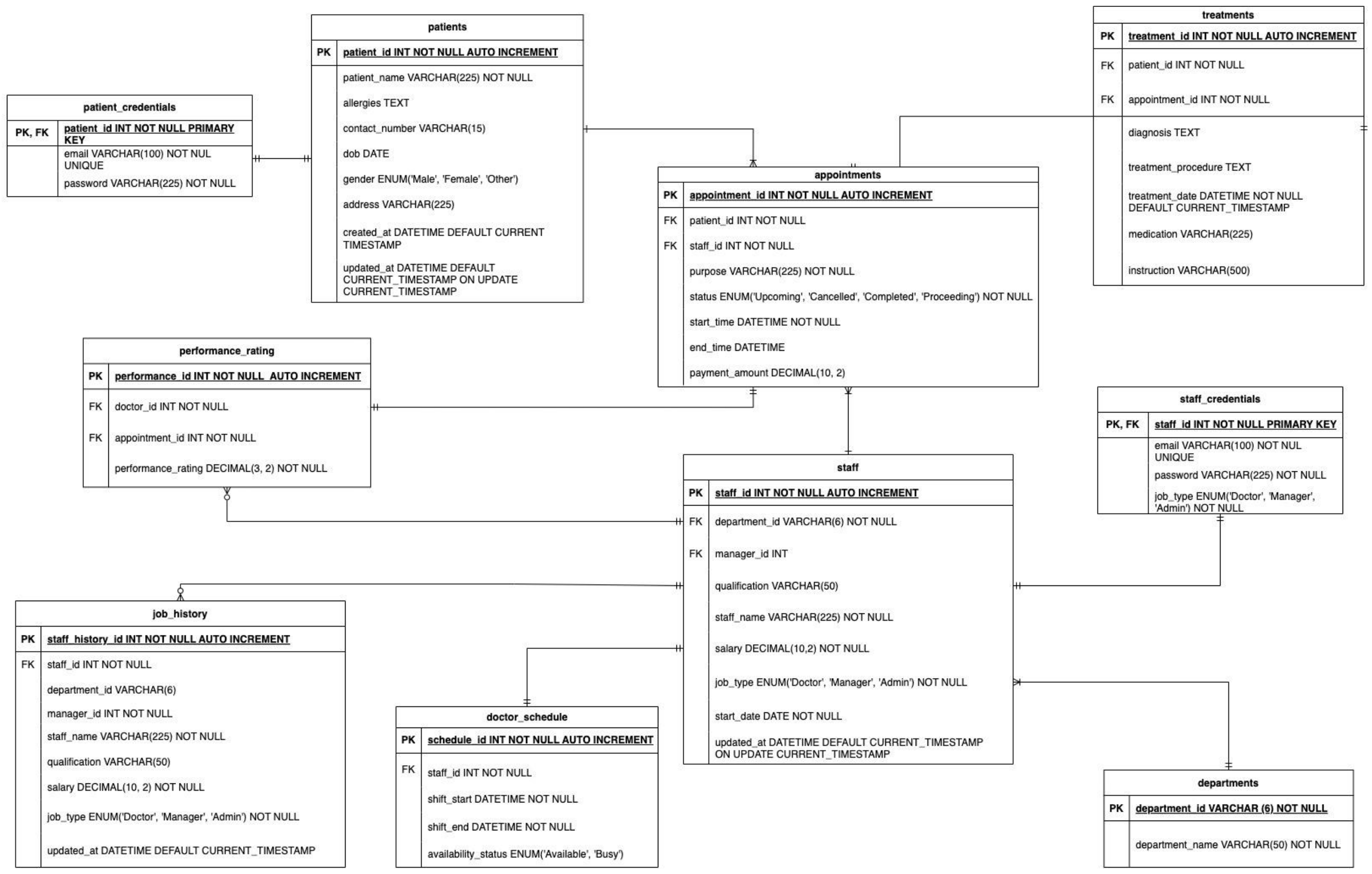
patient_credentials
PK, FK
patient_id INT NOT NULL PRIMARY KEY
email VARCHAR(100) NOT NUL UNIQUE
password VARCHAR(225) NOT NULL
patients
PK
patient_id INT NOT NULL AUTO INCREMENT
patient_name VARCHAR(225) NOT NULL
allergies TEXT
contact_number VARCHAR(15)
dob DATE
gender ENUM('Male', 'Female', 'Other')
address VARCHAR(225)
created_at DATETIME DEFAULT CURRENT TIMESTAMP
updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
treatments
PK
treatment_id INT NOT NULL AUTO INCREMENT
FK
patient_id INT NOT NULL
FK
appointment_id INT NOT NULL
diagnosis TEXT
treatment_procedure TEXT
treatment_date DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
medication VARCHAR(225)
instruction VARCHAR(500)
appointments
PK
appointment_id INT NOT NULL AUTO INCREMENT
FK
patient_id INT NOT NULL
FK
staff_id INT NOT NULL
purpose VARCHAR(225) NOT NULL
status ENUM('Upcoming', 'Cancelled', 'Completed', 'Proceeding') NOT NULL
start_time DATETIME NOT NULL
end_time DATETIME
payment_amount DECIMAL(10, 2)
performance_rating
PK
performance_id INT NOT NULL AUTO INCREMENT
FK
doctor_id INT NOT NULL
FK
appointment_id INT NOT NULL
performance_rating DECIMAL(3, 2) NOT NULL
staff_credentials
PK, FK
staff_id INT NOT NULL PRIMARY KEY
email VARCHAR(100) NOT NUL UNIQUE
password VARCHAR(225) NOT NULL
job_type ENUM('Doctor', 'Manager', 'Admin') NOT NULL
staff
PK
staff_id INT NOT NULL AUTO INCREMENT
FK
department_id VARCHAR(6) NOT NULL
FK
manager_id INT
qualification VARCHAR(50)
staff_name VARCHAR(225) NOT NULL
salary DECIMAL(10,2) NOT NULL
job_type ENUM('Doctor', 'Manager', 'Admin') NOT NULL
start_date DATE NOT NULL
updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
job_history
PK
staff_history_id INT NOT NULL AUTO INCREMENT
FK
staff_id INT NOT NULL
department_id VARCHAR(6)
manager_id INT NOT NULL
staff_name VARCHAR(225) NOT NULL
qualification VARCHAR(50)
salary DECIMAL(10, 2) NOT NULL
job_type ENUM('Doctor', 'Manager', 'Admin') NOT NULL
updated_at DATETIME DEFAULT CURRENT_TIMESTAMP
doctor_schedule
PK
schedule_id INT NOT NULL AUTO INCREMENT
FK
staff_id INT NOT NULL
shift_start DATETIME NOT NULL
shift_end DATETIME NOT NULL
availability_status ENUM('Available', 'Busy')
departments
PK
department_id VARCHAR (6) NOT NULL
department_name VARCHAR(50) NOT NULL

Figure 2: Relational Schema