RMIT International University Vietnam

Assignment Cover Page

Subject Code	ISYS2099
Subject Name	Database Applications
Location	SGS Campus
Title of Assignment	Database Project
Teacher Name	Dr. Tri Dang
Group Name	Group 10
Group Members	s3974735 – Tran Manh Cuong s3965528 – Truong Quang Bao Loc s3978598 – Phan Nhat Minh s3986287 – Nguyen Vinh Gia Bao
Assignment Due Date	September 9, 2024
Date of Submission	September 9, 2024
Declaration of Authorship	We declare that in submitting all work for this assessment, we have read, understood, and agreed to the content and expectations of the Assessment Declaration as specified in https://www.rmit.edu.au/students/my-course/assessment-results/assessment
Consent to Use	We give RMIT University permission to use our work as an exemplar and for showcase/exhibition display

Table of Contents

1. INTRODUCTION

1.1. Overview

In today's increasingly digital landscape, hospital management systems have become essential tools for streamlining operations and improving healthcare delivery. This project focuses on developing a Hospital Management System (HMS) tailored to the specific needs of a small hospital with a single location. The system is designed to manage various critical functions, including patient records, staff scheduling, appointment management, and treatment tracking, all within a secure and efficient framework.

To successfully develop this project, we were tasked with conceptualizing, designing, and implementing a comprehensive solution for handling structured and unstructured data in a hospital environment. The key objectives of the project included robust database design, ensuring data integrity, optimizing performance, and securing sensitive information.

The objectives were met by utilizing a variety of front-end and back-end technologies, including Node.js for the application layer, MySQL for managing structured data, and MongoDB for handling unstructured data such as medical records and images. The following sections of the report will carefully explain the design choices, implementation strategies, and technical details of the system, addressing how it achieves its goals in terms of efficiency, data consistency, and security.

1.2. Assumption

Here are some assumptions of our application:

- The application was designed for the hospital staff only, including receptionists, doctors, nurses, and the administrator. The receptionist will always stay at the reception area to help patients register, book an appointment, and add treatment with an associated doctor.
- The system was designed to be deployed and operated within the hospital's local network. Each computer in the hospital will be set up appropriately based on the staff's role and information by the hospital's IT technicians so that there is no login/sign-up functionality for this application.
- Each staff can only access appropriate resources based on their role's permission, which was defined on both the front-end side and back-end side of the application. For example, nurses are not allowed to view the patient's sensitive information, the hospital's report can be only viewed by the administrator, etc. More details of the role's permission can be found in section *5.1. Database Permissions*.

2. DATABASE DESIGN

2.1. Data Analysis

The data managed by the Hospital Management System can be broadly categorized into structured and unstructured data:

- **Structured Data**: Includes patient information (e.g., personal details, allergies), staff records (e.g., job type, department, salary), appointment schedules, and treatment histories. This data is highly organized and fits well within a relational database structure, allowing for efficient querying and reporting.
- **Unstructured Data**: Comprises documents such as doctor's notes, diagnostic images, and training certificates. This data is less structured and often does not fit neatly into a tabular format. Hence, a non-relational database like MongoDB is more suitable for storing and managing these documents.

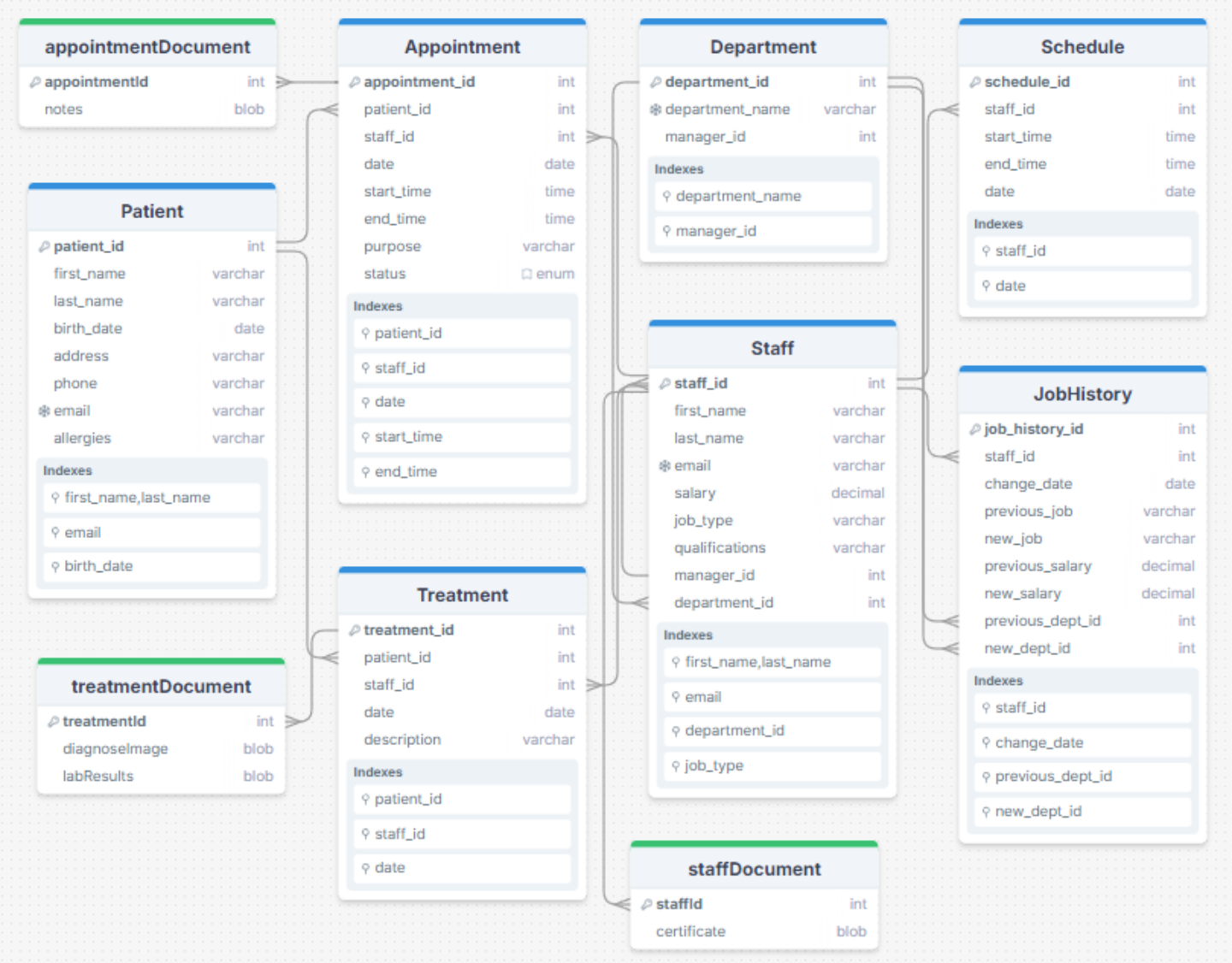

Figure 1: Hospital Management System Database Design

Figure 1 visualizes the database design of the Hospital Management System. While structured data is managed by MySQL, unstructured data like images, files, and notes are converted into binary and managed by MongoDB. The following sections will analyze and describe all the tables, keys, indexes, and constraints of our database design in detail.

2.2. Relational Database Design

2.2.1. Entities and Attributes

- **Patient**: patient_id (PK), first_name, last_name, birth_date, address, phone, email, allergies.
- **Department**: department_id (PK), department_name, manager_id (FK).
- **Staff**: staff_id (PK), first_name, last_name, email, salary, job_type, qualifications, manager_id (FK), department_id (FK).
- **JobHistory**: job_history_id (PK), staff_id (FK), change_date, previous_job, new_job, previous_salary, new_salary, previous_dept_id (FK), new_dept_id (FK).
- **Schedule**: schedule_id (PK), staff_id (FK), start_time, end_time, date.
- **Appointment**: appointment_id (PK), patient_id (FK), staff_id (FK), date, start_time, end_time, purpose, status.
- **Treatment**: treatment_id (PK), patient_id (FK), staff_id (FK), date, description.

2.2.2. Constraints

Primary Key Constraints: Ensure unique identifiers for each record in the respective tables.

- Patient Table: patient_id
- Department Table: department_id
- Staff Table: staff_id
- JobHistory Table: job_history_id
- Schedule Table: schedule_id
- Appointment Table: appointment_id
- Treatment Table: treatment_id

Unique Constraints: Ensure unique values across specific fields.

- Patient Table: email must be unique across patients.
- Department Table: department_name must be unique.
- Staff Table: email must be unique across staff members.
- Appointment Table: Combination of staff_id, date, start_time, and end_time must be unique to prevent overlapping appointments.

Foreign Key Constraints: Ensure referential integrity between related tables:

- Staff.manager_id references Staff.staff_id (Self-referencing with ON DELETE SET NULL).
- Staff.department_id references Department.department_id (ON DELETE SET NULL).
- JobHistory.staff_id references Staff.staff_id (ON DELETE CASCADE).

- JobHistory.previous_dept_id reference Department.department_id.
- JobHistory.new_dept_id reference Department.department_id.
- Schedule.staff_id references Staff.staff_id (ON DELETE CASCADE).
- Appointment.patient_id references Patient.patient_id (ON DELETE CASCADE).
- Appointment.staff_id references Staff.staff_id (ON DELETE CASCADE).
- Treatment.patient_id references Patient.patient_id (ON DELETE CASCADE).
- Treatment.staff_id references Staff.staff_id (ON DELETE CASCADE).

Check Constraints: Enforce specific conditions on data values.

- Staff Table: salary must be non-negative.
- JobHistory Table: previous_salary and new_salary must be non-negative.

Not Null Constraints: Ensure critical fields cannot be left blank.

- Fields such as first_name, last_name, birth_date, address, phone, email, department_name, salary, job_type, change_date, previous_job, new_job, start_time, end_time, date, purpose, and status are all required to have values where applicable.

2.2.3. Relationships

- **Patient to Appointment/Treatment**: The patient_id in the Appointment and Treatment tables references patient_id in the Patient table. If a patient is deleted, all related appointments and treatment records are automatically deleted (ON DELETE CASCADE).
- **Department to Staff**: The department_id in the Staff table references department_id in the Department table. If a department is deleted, the department_id in the Staff table is set to NULL to preserve staff records (ON DELETE SET NULL).
- **Department to JobHistory**: The previous_dept_id and new_dept_id in the JobHistory table reference department_id in the Department table. Deleting a department does not affect job history records.
- **Staff to JobHistory/Schedule/Appointment/Treatment**: The staff_id in the JobHistory, Schedule, Appointment, and Treatment tables references staff_id in the Staff table. If a staff member is deleted, all related records in these tables are also deleted (ON DELETE CASCADE).
- **Self-Referencing Staff Table (Manager Relationship)**: The manager_id in the Staff table references staff_id within the same table. If a manager is deleted, the manager_id for their staff is set to NULL (ON DELETE SET NULL).

In summary, the relational database of the application was designed strictly with several constraints and relationships between the entities. Here is the Entity Relationship Diagram, which visualizes the relational database design of our hospital management system:

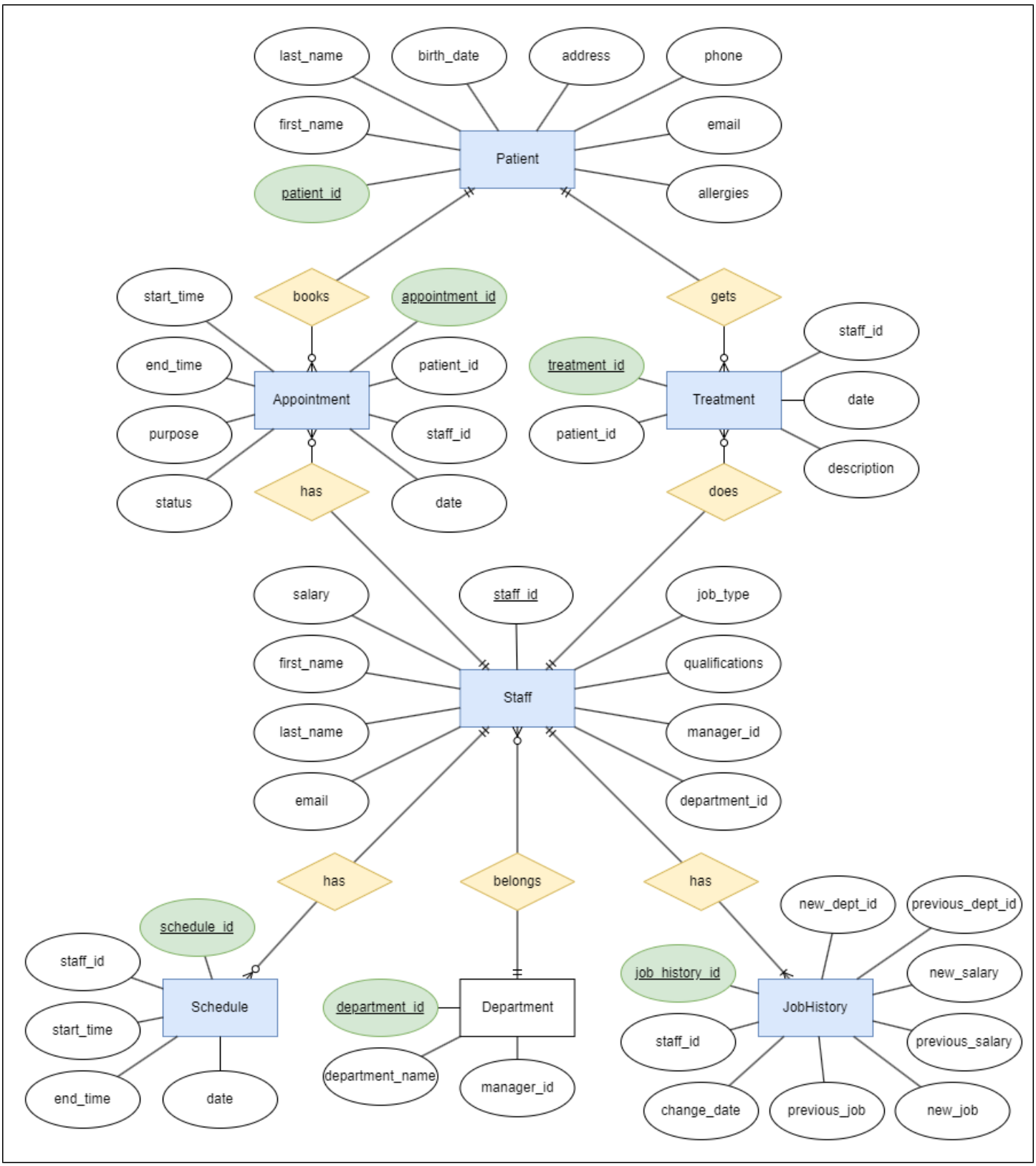

Figure 2: Entity Relationship Diagram (ERD)

2.3. Non-Relational Database Design

The MongoDB database in this Hospital Management System is used to store unstructured data, complementing the structured data stored in a relational database. The system connects to a MongoDB cluster, using the HospitalManagementSystem database.

Three main schemas are defined:

- **Staff Document Schema**: Stores staff-related documents, specifically certificates, with fields for staffId and certificate data in a base64 format.
- **Appointment Document Schema**: Stores appointment-related notes, with fields for appointmentId and the notes' content.
- **Treatment Document Schema**: Manages treatment-related data, including diagnostic images and lab results, storing them as base64-encoded strings.

These schemas correspond to collections in MongoDB, providing flexibility for storing varied, unstructured data like text, images, and documents. This setup allows the system to handle both structured and unstructured data efficiently, leveraging MongoDB's strengths in managing large, diverse datasets.

Additionally, the MongoDB collections are linked with the MySQL tables using corresponding IDs (staffId, appointmentId, and treatmentId). This integration allows the system to maintain relational data integrity while also accommodating the storage and retrieval of unstructured data, ensuring seamless connectivity between the structured and unstructured data layers.

3. PERFORMANCE ANALYSIS

3.1. Query Optimization

Query optimization in the HospitalManagementSystem ensures efficient database performance. The process involves evaluating and selecting the most cost-effective query plans to minimize disk I/O. Strategic indexing on key columns, such as those used in WHERE clauses and JOINs, enables faster index scans, reducing the need for full table scans and improving query speed.

JOIN operations are optimized through efficient methods like Hash Join and Sort-Merge Join, minimizing resource use. Reordering query operations, such as performing selections before joins, further reduces execution costs. Avoiding full table scans and optimizing sorting operations by leveraging indexes also enhance performance.

Tools like EXPLAIN and EXPLAIN ANALYZE provide insights into query execution plans, helping identify and resolve bottlenecks. Overall, these techniques ensure the database remains fast and efficient, even as data complexity increases.

3.2. Indexing

Indexing is a crucial aspect of database performance optimization. In this schema, indexes are created on various tables to improve the efficiency of queries, particularly those involving searching, filtering, and sorting. Here's a breakdown of the indexing strategy:

Patient Table Indexes:

- **idx_patient_name** (last_name, first_name): Indexing both last_name and first_name supports queries that search for patients by their full name or by either component, improving the performance of searches on these fields.
- **idx_patient_email** (email): Given that email is unique and often used to identify or contact patients, indexing this field speeds up lookups by email.
- **idx_patient_birth_date** (birth_date): Indexing birth_date optimizes queries filtering or sorting patients by their date of birth, which could be useful in age-based reports or validations.

Staff Table Indexes:

- **idx_staff_name** (last_name, first_name): Similar to the patient table, this index facilitates quick searches by staff names.
- **idx_staff_email** (email): Since emails are unique and frequently used for communication or authentication, indexing email improves the performance of these operations.
- **idx_staff_department** (department_id): Indexing department_id enhances the efficiency of queries that filter or group staff by department, which is common in reporting and department-specific operations.
- **idx_staff_job_type** (job_type): Indexing job_type helps in quickly retrieving or filtering staff based on their roles, which can be particularly useful in staff management and assignment tasks.

Department Table Indexes:

- **idx_department_name** (department_name): This index ensures fast lookups of departments by name, useful for administrative tasks and department management.
- **idx_department_manager** (manager_id): Indexing manager_id improves performance in queries that retrieve departments by their manager or manage staff assignments under a specific manager.

JobHistory Table Indexes:

- **idx_jobhistory_staff_id** (staff_id): Indexing staff_id in the JobHistory table accelerates queries that track or review an individual staff member's job history.
- **idx_jobhistory_change_date** (change_date): This index optimizes queries that focus on job changes over time, facilitating chronological reports or audits of job history.
- **idx_jobhistory_previous_dept_id** (previous_dept_id): Indexing previous_dept_id speeds up queries analyzing historical department changes for staff.
- **idx_jobhistory_new_dept_id** (new_dept_id): Similar to the previous index, this one supports efficient retrieval of records by the new department assignment, essential for analyzing department transitions.

Schedule Table Indexes:

- **idx_schedule_staff_id** (staff_id): Indexing staff_id enables quick lookups of staff schedules, which is crucial for managing work shifts and planning.
- **idx_schedule_date** (date): Indexing date helps in querying schedules for specific days, which is essential for day-to-day operations and schedule planning.
- **idx_schedule_staff_date_time** (staff_id, date, start_time, end_time): This composite index optimizes queries that require filtering or sorting of staff schedules based on the staff member, the date, and specific time intervals. It is particularly useful for ensuring efficient schedule management and conflict resolution by quickly retrieving relevant schedules within defined time periods.

Appointment Table Indexes:

- **idx_appointment_patient_id** (patient_id): This index improves the performance of queries that retrieve appointments for a specific patient.
- **idx_appointment_staff_id** (staff_id): Indexing staff_id allows for quick retrieval of appointments assigned to a particular staff member, facilitating efficient management of appointments.
- **idx_appointment_date** (date): Indexing date optimizes the scheduling and retrieval of appointments on a particular day, which is crucial for daily operations.

Treatment Table Indexes:

- **idx_treatment_patient_id** (patient_id): This index enhances the performance of queries that retrieve treatments associated with a specific patient, supporting detailed patient care records.
- **idx_treatment_staff_id** (staff_id): Indexing staff_id in the Treatment table speeds up queries that track treatments provided by a particular staff member.
- **idx_treatment_date** (date): Indexing date facilitates the retrieval of treatments based on the date they were administered, supporting chronological treatment records and reporting.
- **idx_appointment_start_time** (start_time): This index enhances the performance of queries that need to filter or sort appointments based on their start time, which is particularly useful for managing appointment schedules throughout the day.
- **idx_appointment_end_time** (end_time): Indexing end_time improves the efficiency of queries that focus on the conclusion of appointments, supporting operations that require an understanding of when appointments end.
- **idx_appointment_staff_date_time** (staff_id, date, start_time, end_time): This composite index is designed to optimize complex queries that involve filtering or sorting appointments based on the staff member, the appointment date, and the specific time intervals. It is particularly useful for scenarios where appointments are managed based on staff availability within specific timeframes.

3.3. Partitioning

Partitioning is a technique used to improve database performance and manageability by dividing large tables into smaller, more manageable pieces. However, in this database schema, we cannot implement partitioning because

the database includes foreign key constraints, which are not supported with partitioned tables in MySQL. As a result, partitioning is not a viable option for optimizing this database.

3.4. Concurrent Access

Concurrent access in database systems can lead to data inconsistencies and conflicts if not properly managed. In the HospitalManagementSystem database, several mechanisms are implemented to handle concurrent access effectively.

To manage this, we use transactions (START TRANSACTION) to ensure that operations are completed as a single unit, which either commits fully or rolls back entirely if an error occurs. This prevents partial updates and maintains data integrity.

Furthermore, FOR SHARE locks are employed in read operations to allow multiple transactions to read the data simultaneously while preventing any modifications. For updates, FOR UPDATE locks are used to ensure that once a row is selected for modification, no other transaction can alter it until the current transaction is complete. This prevents conflicts such as race conditions.

We also include conditional checks in procedures to verify data before proceeding, preventing issues like double-booking appointments. Error handling through EXIT HANDLER blocks ensures that any SQL exceptions result in a rollback, avoiding corrupt data from partial transactions.

Additionally, temporary tables are used to manage intermediate results without affecting other transactions, with cleanup processes ensuring no leftover data interferes with subsequent operations. These combined strategies ensure that the HospitalManagementSystem maintains consistent and accurate data, even under concurrent access conditions.

4. DATA INTEGRITY

4.1. Triggers and Stored Procedures

4.1.1. Triggers

- **trg_after_staff_update**: Logs changes in job type, salary, or department to JobHistory. It also cancels future appointments if a staff member's role or department changes.
- **trg_before_schedule_insert** and **trg_before_schedule_update**: Prevents overlapping schedules for staff by checking for conflicts before inserting or updating schedules.
- **trg_before_staff_delete** and **trg_before_patient_delete**: Blocks deletion of staff or patients if they have future appointments, ensuring no disruption in scheduled services.

4.1.2. Stored Procedures

- **Patient and Appointment Management**: Procedures like createPatient, updatePatientInformation, and createAppointment manage patient and appointment data, ensuring all actions are performed within transactions to maintain consistency.
- **Job History and Schedule Management**: Procedures like getAllJobHistories and addSchedule handle job changes and staff schedules, working alongside triggers to prevent conflicts and ensure accurate record-keeping.
- **Department and Staff Management**: Procedures such as createStaff and updateStaff manage staff and department data, ensuring updates are done reliably within transactions.
- **Availability and Workload Management**: Custom procedures like getAvailableStaffsInDuration assess staff availability and workload, aiding in efficient scheduling and resource management.

4.2. Transaction Management

The transaction management strategy implemented in this code is designed to ensure data consistency, integrity, and robustness, particularly in the event of errors or conflicts during database operations. The following key aspects of transaction management are employed:

- **Transaction Control with START TRANSACTION, COMMIT, and ROLLBACK**:
  - Each procedure that modifies data begins with a START TRANSACTION statement, which initiates a transaction. This ensures that all subsequent operations within the transaction block are treated as a single unit of work.
  - If all operations within the transaction block complete successfully, the COMMIT statement is executed, making all changes permanent in the database.
  - If any operation fails or an error is encountered, a ROLLBACK statement is executed, undoing all changes made during the transaction. This prevents partial updates or inconsistent data from being committed.
- **Error Handling with EXIT HANDLER FOR SQLEXCEPTION**:
  - An EXIT HANDLER FOR SQLEXCEPTION is defined at the beginning of critical procedures to handle any SQL exceptions that may occur. If an error is encountered, the handler ensures that a ROLLBACK is performed, preventing any erroneous data from being committed.
  - After the rollback, a result code is set (p_result = -1) to indicate that an error occurred, which can be used by calling applications to handle the error appropriately.
- **Validation Checks Before Data Modification**:
  - Before performing any insert or update operations, the procedures include various validation checks to ensure that the data being processed meets the necessary criteria. For example, checks are performed to ensure that required fields are not null, that email addresses are unique, and that there are no overlapping appointments or schedules.
  - If any validation fails, the transaction is rolled back, and an appropriate result code is set (p_result = 1 for data not existing or illegal operations, p_result = 2 for illegal argument values).
- **Locking Mechanisms (FOR SHARE and FOR UPDATE)**:

- Select queries in some procedures are executed with FOR SHARE or FOR UPDATE clauses to lock the relevant rows. This prevents other transactions from modifying the data until the current transaction is complete, ensuring data consistency and preventing conflicts in concurrent environments.

- **Composite Operations as a Single Transaction**:
  - Procedures that involve multiple steps or operations, such as inserting a new patient or updating appointment details, are wrapped in a single transaction. This ensures that either all operations succeed together, or none are applied, maintaining the integrity of the database.
  - By implementing these transaction management techniques, the code ensures that database operations are reliable, and that the system can recover gracefully from errors or unexpected conditions, maintaining the consistency and integrity of the hospital management system's data.

5. DATA SECURITY

In the Hospital Management System, data security is an essential aspect, particularly given the sensitive nature of medical records and patient information. However, due to the scope of this project and the specific deployment environment, the primary security measure implemented at the database level is the use of role-based access control (RBAC). Below is a detailed description of the data security measures:

5.1. Database Permissions

5.1.1. Role-based Permissions

The system employs role-based permissions to control access to different parts of the database. This ensures that users only have access to the data and functionalities necessary for their specific roles.

The roles implemented in the system are:

- **Admin Role**: Admins have unrestricted access to the database. They can manage user accounts, assign roles, and perform all CRUD (Create, Read, Update, Delete) operations across all tables. Admins also have the ability to manage system settings and generate reports.
- **Doctor Role**: Doctors have access to patient profiles, medical history, and treatment records. They can update patient treatments, manage their schedules, and view the schedules of other staff members. Doctors do not have access to modify staff data or system configurations.
- **Nurse Role**: Nurses can view patient profiles and treatment histories. They assist doctors by updating patient treatments and managing patient appointments. Nurses can also view staff schedules but have no permission to alter them.
- **Receptionist Role**: Receptionists manage patient records and appointments. They can view and update patient details, book appointments, and manage the schedules of doctors and other staff. Receptionists have no access to treatment records or other sensitive patient data.

These roles are enforced within the database, ensuring that users can only access the data and perform actions that are relevant to their responsibilities within the hospital.

5.1.2. Views for Sensitive Information

To further secure sensitive data, views are created to restrict access to specific fields based on user roles:

- **PatientOverview**: Displays basic patient information without sensitive data like email and phone numbers.
- **StaffOverview**: Shows staff information excluding sensitive details like salary.
- **StaffWithDepartment**: Displays staff information with department names instead of IDs and includes the manager's name.
- **AppointmentDetails**: Shows appointment details with patient and staff names, but only necessary fields are exposed.
- **DoctorScheduleWithStatus**: Provides a view of doctor schedules with their availability status, limited to the next 30 days.

These views ensure that users can access the necessary data without exposing sensitive information, enhancing data security and privacy.

5.1.3. Local Deployment and Security Context

The system is designed to be deployed and operated within the hospital's local network. This means that access to the system is physically restricted to the hospital premises, where it is less vulnerable to external threats. Given this context, the primary security concern is internal access control, which is effectively managed through role-based permissions.

By restricting the system to local deployment, we mitigate the risks associated with external attacks, such as unauthorized remote access or data breaches from external sources. The assumption is that the hospital's internal network is secure, and that the role-based access control sufficiently protects sensitive data within this environment.

5.1.4. No Login/Sign-up Functionality

The system does not include login/signup functionality for the web application. Instead, user accounts and roles are managed directly by the admin through the database. This approach provides several security advantages:

- **Reduced Attack Surface**: By not implementing a web-based login/signup system, the potential entry points for external attackers are minimized. This reduces the overall risk of unauthorized access to the system.
- **Controlled User Management**: User creation, role assignment, and account management are performed exclusively by the admin, ensuring that only authorized personnel are granted access to the system. This control helps maintain a secure environment where access to sensitive data is strictly regulated.
- **Simplified Security Management**: Without the complexity of a web-based authentication system, the focus remains on managing access through well-defined roles within the database. This simplicity reduces the chances of security misconfigurations that could lead to vulnerabilities.

5.2. SQL Injection Prevention

To protect against SQL injection attacks, input validation is implemented at both the frontend and backend levels:

Front-end Validation:

- The frontend validates all user inputs before they are sent to the backend. This reduces the likelihood of malicious data being passed into the system.

Backend Validation:

- In addition to frontend validation, stored procedures in the database include additional validation checks. For instance, before inserting or updating data, the procedures check for null or illegal values and confirm the existence of referenced records. The procedures are designed to handle any errors gracefully, rolling back transactions if necessary to maintain data integrity.
- Example procedures include createPatient, updatePatientInformation, createAppointment, and updateAppointment, where input parameters are thoroughly checked to prevent SQL injection and other forms of data corruption.

These layers of validation help ensure that only legitimate and sanitized data is processed, significantly reducing the risk of SQL injection attacks.

5.3. Password Hashing and Encryption

In the current implementation of the Hospital Management System, there is no login or registration functionality, and therefore, password hashing has not been implemented. This approach simplifies the system by avoiding the complexities associated with managing user credentials.

However, for future development, if a user authentication system is introduced, it is recommended to implement password hashing using bcrypt, a library for NodeJS. Bcrypt is a robust algorithm that provides a high level of security by hashing passwords in a way that is computationally expensive to crack, thus protecting user credentials from potential breaches.

5.4. Additional Security Measures

Additional security measures include:

- **Environment Variables (.env file)**: Sensitive information such as database connection strings (e.g., mongo_uri), secret keys, and role definitions are stored in a .env file in the backend. This practice prevents these values from being hardcoded into the application code, reducing the risk of exposure in case of a code breach.
- **Local Deployment and Network Security**: The system is designed for deployment within the hospital's local network, limiting access to authorized internal users only. This configuration minimizes the risk of external attacks, such as unauthorized remote access.

These security practices collectively ensure that sensitive data is protected from unauthorized access and potential breaches, maintaining the confidentiality and integrity of the Hospital Management System's data.

6. CONCLUSION

This project successfully developed a Hospital Management System (HMS) tailored for a small hospital, focusing on efficient data management, security, and scalability.

- **Database Design**: MySQL handles structured data, while MongoDB manages unstructured data like medical images, ensuring comprehensive data storage.
- **Performance**: Effective indexing and transaction management optimize query efficiency and maintain data integrity during concurrent access.
- **Security**: Role-based access control safeguards sensitive data within the hospital's local network, with a focus on internal security.

The system meets the hospital's needs, offering a strong foundation for future enhancements and scalability.