# Database Design and Implementation Report

ISYS2099 | Database Applications

Lecturer: Mr. Tri Dang Tran

Team 4

Ly Minh Khoi – s3989037

Luong Thanh Trung – s3679813

Pham Nhat Minh – s4019811

Eduardo Salcedo Fuentes – s4118015

Hoang Dinh Tri – s3877818

# Table of Contents

# Introduction

In today's rapidly advancing healthcare landscape, effective hospital management systems (HMS) are essential for ensuring that medical facilities operate efficiently and deliver high-quality patient care. This project aims to design and implement a robust HMS tailored to the specific needs of a small hospital with a single physical location. The system seeks to streamline the management of patient data, staff information, appointments, and reporting by integrating advanced technologies such as Node.js, MySQL, and MongoDB. It is designed to handle both structured data, like patient records and staff schedules, and unstructured data, including doctor's notes, diagnostic images, and test results.

To achieve these objectives, the project incorporates a combination of front-end and back-end technologies. Node.js will be used for the application layer, MySQL for managing relational data, and MongoDB for handling unstructured data. This report details the implementation strategy for the project, focusing on how these components will work together to create an efficient and effective hospital management system that meets the unique needs of the hospital.

# Project Description and Implementation Details

## 1. Database Design

The database design phase is a fundamental step in the development of this Hospital Management System (HMS), playing a pivotal role in creating a structured and efficient data architecture to meet the specific requirements of a modern healthcare facility. This phase is thoroughly documented, detailing the process of designing the database, which encompasses data analysis, the creation of entity-relationship diagrams (ERD), relational schema design, the establishment of constraints and relationships, as well as considerations for integrating non-relational data structures. By carefully addressing both structured and unstructured data, this phase ensures that the HMS will effectively support the hospital's operations, providing a solid foundation for subsequent development stages.

### a. Data Analysis

The scenario of this project required the development of a Hospital Management System (HMS) tailored to the needs of a small hospital. To achieve this, we conducted a thorough data analysis to understand the nature of the data that needs to be managed within the system. The primary entities interacting within the HMS include various user groups, such as Front Desk staff, HR staff, Business Officer staff, doctors, and nurses, each with distinct roles and data interactions. Additionally, the analysis focused on understanding patient management, appointment scheduling, and staff operations, with special attention to automating routine processes, such as patient record updates and test result management, to reduce manual intervention and enhance efficiency. This foundational analysis guided the overall design of the database, ensuring it meets the specific requirements of the hospital environment.

### b. ERD and Relational Schema

**Please refer to the Section 1 and Section 2 in the Appendix for more **

**Table Relationships:**

**Allergies Table:** Since a patient may have multiple allergies, and the same allergy can be present in more than one patient, the relationship between the *Allergies* and *Patients* tables is many-to-many. As is typical in such relationships, a bridging (or join) table connects these two entities. The primary keys in this bridging table are formed by the combination of primary keys from the *Patients* and *Allergies* tables. Users cannot directly write data to the *Allergies* table; instead, it is populated with data from a central medical repository. This ensures more accurate data aggregation and minimizes redundancy. Additionally, the *Patients* table has one-to-many relationships with the *Billings*, *Appointments*, *Prescriptions*, and *TreatmentHistory* tables. A single patient can have multiple appointments, undergo various treatments, and consequently generate multiple billing records. As for the *TreatmentHistory* table, it also shares a many-to-many relationship with the *Drugs* table. In the system, the TreatmentHistory table could be loosely interpreted as a list of prescriptions, and each prescription might contain multiple drugs, thus a many-to-many relationship. The TreatmentHistory table is also linked with the Diagnoses table, which in turns is also connected to the conditions table on a many-to-many relationship. Much Like the *Allergies* table, drug and condition data are imported from a central medical repository to minimize human errors and data redundancy. The followings are the benefits of this standardized storage approach:

- It supports better scalability since data will not grow as the hospital expands its operation as a result of redundancy reduction
- It supports complex and more accurate aggregation reports. Suppose that patient A and patient B have the same allergies. However, the doctor attending to them input the allergy name in different formats. While ‘nut allergy’ and ‘Nut Allergy’ are similar from our perspective, they represent completely different value for the database. As such, the database would yield incorrect data if the hospital would like to find out about the number of patients with a particular type of allergies. The accurate support for complex aggregation report and data visualization offers a major competitive advantage for the hospital as it could now, for example, find the most prescribed drugs for a particular condition.
- It allows for more effective and efficient indexing, as queries can use the index on standardized allergy codes (which is also the primary key) instead of having to define another index based on names.
- It allows for better integration and communication with third-party medical applications as all applications are now sharing one single data interface

The staff table is designed to store staff’s personal information and authentication credentials. It has a many to one relationship with the department and the job table, as one staff could only hold one job and be in one department at a point in time. There are 3 additional tables connecting with the staff table in a one-to-many relationship. These are the tables that monitor movement in jobs, departments and wage (Job_Change, Department_Change, and Wage_Change).

## c. Non-relational documents and their structure

Beside utilizing relational databases, we also implemented noSQL for data that demands a higher degree of flexibility. For instance, staff’s qualifications could encompass a great many things. While on one hand it could refer to official records testify for the holder’s competency, on the other hand it could also refer to the more unofficial sources such as experience. With this in mind, we capture the flexibility and multi-dimensional nature of qualification data by storing the specific details in MongoDB, while their unique identifiers are stored in the Qualification table in MySQL. Beside qualifications, highly multi-dimensional data such as lab-test results are also stored in MongoDB.

# 2. Performance Analysis

## a. Query Optimization

Where appropriate the database prioritizes bulk operations over single-row operations. This could very well be illustrated by the operation to add allergies to patients. For this operation, the front end will send a single request containing the patient id and the list of allergies' ids to be added to them. The server will then process this array into a string and pass it as a parameter into a procedure called *AddAllergiesToPatients*, which will loop through the string. At this stage, we are presented with 2 options, either insert allergies individually, or build a single query as a String and insert them after the loop concludes. While the former is undeniably simpler, the second alternative is twice as fast, with the reason being the database only must parse and process the query once. Beside the observed speed advantage, there is also the fact that bulk operations allow related data to be arranged contiguously on disk. Specifically, the smallest writable unit in a flash drive is a page, which is usually about 4 KB. Since each individual patient-allergy row is smaller than 4 KB, the operating system will have to pad the empty page a 0s.

Lastly, one of the strengths of the database is the selective use of non-clustered indexes. Specifically, most critical operations of the applications have been engineered in such a way that they strictly depend on the primary keys of the table. For instance, instead of sending drugs' name for prescription, which would necessitate the need for an index on the *Drugs* table's *drug_name* column, the client sends only a list of *drug_codes,* which are the primary key of the *Drugs* table. This allows the application to make full use of the clustered indexes created at the creation of the primary key. The same could be applied to other core operations such as billings creation, diagnoses creation. While future expansion might see additional indexes being added, we are currently limiting the number of indexes to what the application needs. This allows us to limit storage used and avoid the write amplification issues that inevitably occurs when one of the indexes get updated and its B-tree must be re-structured. Having said that, the followings are the non-clustered indexes implemented:

***Patients' full-name*:** For this, a full text index has been implemented on the Patients table. The reason as to why the full-text index has been opted for over the traditional index is because the application is being developed for an organization operating in a multinational context, with which comes potentially confusion in naming convention. For instance, Vietnamese' naming convention usually places first name after last name, while it is the other way around for the Western naming convention. As a result, the flexible nature of the full-text index suits the situation better. The use of index, as shown by the following illustration, vastly improves query speed

***idx_doctor_appointment_date:*** One frequently used operation is the doctor availability checking operation where the database receives the queried date, start and end time and returns a list of doctors together with their availability statuses. For this, the database must read through the Appointments table for rows with a specific appointment_date value and doctor_id value. In this operation, the Appointments table is usually joined with the Staff table to fetch doctors' full names. At this stage, there are 3 indexing options, we can either index on the appointment_date column, the doctor_id column, or the start_time and the end_time column. Since the first alternative has higher selectivity than the other three, a composite index of appointment_date and doctor_id has been implemented

***idx_staff_schedule_date:*** An index has also been implemented on the Staff_Schedule table's schedule_date column. Checking for whether or not a staff already has a schedule on a particular day is essential to upholding the business logic of the organization. For scheduling, the database would first check if a schedule exists for a staff on a particular date. If it does, the old schedule entry will be updated with the new start_time and end_time. With weekends accounted for, a staff in this hospital would have at least 300 entries every year. A hospital with roughly 1000 staff would see up to 300.000 additional schedule entries every year. As such, this index would help speed up scheduling operation as the number of staff increases.

***idx_date_change_salary:*** This index is used to make it faster to filter through past salary changes by dates.

*idx_date_change_job:* This index makes it fast to filter through past job changes by dates.
***idx_date_change_department:*** This index makes it faster to filter through past department changes by date.
***idx_evaluation_date:*** This index improves the speed of filtering through staff evaluation records by date.
***idx_diagnosis_date:*** This index improves the speed of filtering through patient's historical diagnoses by date.
***idx_treatment_start_date:*** This index improves the speed of filtering through patients' historical prescription data by date.
***idx_administering_date:*** This index is used when doctors or nurses want to reach a particular set of test details on a particular time frame.
***idx_billing_date:*** This index is used to improve the speed of finding billing entries by date.
**idx_total_amount:** This index is used to improve the speed of finding billing entries that fall within a certain range of payment amount.
***idx_hire_date:*** This index makes it faster to filter through employee by their hire dates
**idx_wage:** This index makes it fast to reach a set of employees that have a particular range of wage

### b. Concurrent Access

Beside implementing data consistency via trigger, we have also implemented a series of locking mechanisms that would protect data consistency in a multithreading environment. Specifically, throughout the development process, 4 scenarios could be identified where the data consistency could be endangered.

**Medicine Prescribing**: As demonstrated by the following figures, 2 doctors prescribing the same drugs at roughly the same time could lead to the drug inventory being incorrectly updated. This is what we call data race conditions in a multithreading environment. To remedy this, we implement an exclusive lock at the initial stage where the inventory drug is checked. While this successfully updates the drug inventory, it could potentially lead to deadlock in scenarios where 2 doctors prescribing the same set of drugs but send to the database in different orders. The deadlock is fixed at the application level where we anticipate specifically for an exception with a particular code. When the deadlock has been caught, the application will continuously call the procedure until it succeeds.

**Appointment Reservation**: There are several ways this could arise, for example 2 front-desk officers trying to create an appointment with a single doctor on conflicting time slots, or a front-desk officer creating an appointment with a doctor who is being rescheduled at the same time. The final way this could arise, albeit quite rare, is when the job or the department of a doctor with whom an appointment is being set up changes. For these, we implement exclusive locks at various constraint checking methods to lock relevant row. As of now, we have yet to envision a situation where deadlock might arise.

## 3. Data Integrity

### a. Triggers

The database ensures data consistency using triggers and locks. The following tables detail all triggers used in the database and their respective purpose:

*Update_Job_Aspects*. This trigger is triggered before any job_related update (wage, department, and job) to the staff table. The effect of this trigger could be seen in the 2 following snippets which try to change the wage of a doctor, the min and max wage for whom is 8862 and 9863 respectively. The trigger prevents the incorrect wage to be entered into the database, thus protecting business logic. The following is a brief surmise of its effects:

- It verifies if the new wage falls within the acceptable wage range set in the Jobs table
- It verifies if any change in department and job is followed by a change to a matching manager.

- It verifies if the new department and job exist within the Departments and Jobs table

```
async function testingUpdateJobAspectTrigger(){
    const job = await poolAdmin.query(
        "SELECT job_id FROM Staff WHERE id = 1"
    )
    console.log("Job of Staff 1: " + JSON.stringify(job[0]))
    const doctor_wage = await poolAdmin.query(
        "SELECT min_wage, max_wage FROM Jobs WHERE id = 2"
    )
    console.log(JSON.stringify("min_wage and max_wage: " + JSON.stringify(doctor_wage[0])));
    await hrRepo.ChangeWage(1, 3999)

}
```

```
Job of Staff 1: [{"job_id":2}]
"min_wage and max_wage: [{\"min_wage\":\"8862.23\",\"max_wage\":\"9863.30\"}]"
file:///Users/trungluong/Desktop/ISYS2099-project-group-4/backend/Models/HrModel.js:26
      throw new Error(error.message);
            ^
Error: Wage does not fall within the correct range
    at Object.ChangeWage
(file:///Users/trungluong/Desktop/ISYS2099-project-group-4/backend/Models/HrModel.js:26:13)
    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)
    at async testingUpdateJobAspectTrigger
(file:///Users/trungluong/Desktop/ISYS2099-project-group-4/backend/demonstration.js:111:5)
```

*ValidatePatientsAllergies.* This trigger ensures that a doctor can only add allergies to a patient with whom they are having an appointment with.

*ValidateDiagnoses.* This trigger ensures that a doctor can only make diagnoses for a patient with whom they are having an appointment with.

*ValidateTreatment.* This trigger ensures that a doctor can only issue prescriptions for a patient with whom they are having an appointment with.

*ValidateTests.* This trigger ensures that a doctor can only order medical test for a patient with whom they are having an appointment with.

The effect of the 4 triggers above is illustrated by the following code snippets which try to add allergies to patient 33 using the account of doctor 103, even though patient 103 has no appointment with patient 33:

```
async function testDoctorPatientAppointmentMatch(){
    const appointments = await poolAdmin.query(
        "SELECT * FROM Appointments WHERE patient_id = 33 AND doctor_id = 103"
    )
    console.log("Appointments that patient 33 has with doctor 103: " +
JSON.stringify(appointments[0]))
    await doctorRepo.AddAllergyToPatient(103,33,'1,2,3');
}
```

```
Appointments that patient 33 has with doctor 103: []
file:///Users/trungluong/Desktop/ISYS2099-project-group-4/backend/Models/DoctorModel.js:55
      throw new Error(error.message);
            ^

Error: Either the patient id is incorrect or you do not have the privilege to perform this
action currently
```

```
    at Object.AddAllergyToPatient
 (file:///Users/trungluong/Desktop/ISYS2099-project-group-4/backend/Models/DoctorModel.js:55:13)
    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)
    at async testDoctorPatientAppointmentMatch
 (file:///Users/trungluong/Desktop/ISYS2099-project-group-4/backend/demonstration.js:141:5)
```

*VerifyingManagementRelationshipForScheduling*: This trigger ensures that a staff can only schedule for another staff when the latter is under the former's management.

*VerifyingManagementRelationshipForEvaluation:* This trigger ensures that a staff can only evaluate the performance of another staff when the latter is under the former's management.

*VerifyingManagementRelationshipForSchedulingDeleting:* This trigger ensures that a staff can only delete the schedule of another staff when the latter is under the former's management.

The effect of these three triggers is illustrated in the 2 following code snippets. Specifically, the function is trying to schedule for staff number 4 using the account of staff number 102. However, the manager of staff number 4 is staff number 2, this leads to an exception being raised preventing staff number 102 from adjusting the schedule if a staff they do not manager.

```
async function testingManagementCheckTrigger() {
    const manager_id = await poolAdmin.query(
        "SELECT manager_id FROM Staff WHERE id = 4", []
    )
    console.log("Manager: ", JSON.stringify(manager_id[0]))
    doctorRepo.Scheduling(102,4,'2024-09-20;12:00:00-17:00:00')
}
testingManagementCheckTrigger()
```

```
Manager:  [{"manager_id":2}]
file:///Users/trungluong/Desktop/ISYS2099-project-group-4/backend/Models/DoctorModel.js:224
      throw new Error(error.message);
            ^

Error: You do not have the authority to schedule for this staff
    at Object.Scheduling
(file:///Users/trungluong/Desktop/ISYS2099-project-group-4/backend/Models/DoctorModel.js:224:13)
    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)
```

In cases that necessitate cross-table constraint checking, a series of constraint checks are implemented within procedures to ensure that the business logic is preserved in cases. For instance, when a user performs a scheduling operation by calling the Scheduling procedure, the database will check if:

- The user puts in the correct time format. For the hospital, the correct time format is one with the minute as a multiplier of 30.
- There is already an existing schedule for that date. If there is, the database will perform an update operation in place of an insert to overwrite the old schedule
- If there are appointments associated with the existing schedule, no update will be performed.

This is illustrated in the following code snippets that are trying to create a new appointment with doctor 103, although earlier availability check returned "Unavailable" for this doctor. As a result, an exception is raised indicating appointment conflict:

```
async function testAddNewAppointment(){
        const availability = await frontDeskRepo.CheckAvailability("2024-09-16", "11:30:00",
"12:30:00", 2);
    console.log(availability[0])
     await frontDeskRepo.AddNewAppointment(2, 103, 1, "Routine Checkup", "2024-09-16", "11:30:00",
"12:30:00", null)
}
```

```
  { id: 6, full_name: 'Robert Brown', Availability: 'Unavailable' },
  {
    id: 7,
    full_name: 'Kathleen Rodriguez',
    Availability: 'Unavailable'
  },
  { id: 8, full_name: 'Jose Perry', Availability: 'Unavailable' },
  {
    id: 10,
    full_name: 'Natalie Johnston',
    Availability: 'Unavailable'
  },
  { id: 93, full_name: 'Hoang Dinh Tri1', Availability: 'Unavailable' },
  { id: 101, full_name: 'Maria Ozawa', Availability: 'Unavailable' },
  { id: 102, full_name: 'Marai Ozu', Availability: 'Unavailable' },
  { id: 103, full_name: 'Maru Ozai', Availability: 'Unavailable' },
  { id: 104, full_name: 'Johny Weeb', Availability: 'Unavailable' },
  { id: 105, full_name: 'Johny Weeba', Availability: 'Unavailable' },
  { id: 106, full_name: 'Johny Boba', Availability: 'Unavailable' }
]
file:///Users/trungluong/Desktop/ISYS2099-project-group-4/backend/Models/FrontDeskModel.js:60
      throw new Error(error.message);
            ^

Error: Time slot has already been reserved. Please try again
                                                    at              Object.AddNewAppointment
(file:///Users/trungluong/Desktop/ISYS2099-project-group-4/backend/Models/FrontDeskModel.js:60:13
)
    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)
                                            at            async            testAddNewAppointment
(file:///Users/trungluong/Desktop/ISYS2099-project-group-4/backend/demonstration.js:130:5)
```

## b. Transactions

The application can achieve atomicity at various level of granularity. Specifically, interrelated write operations in MySQL are grouped together in one single transaction scope, as illustrated by the following code snippet.

```
START TRANSACTION;

    -- Insert a new record into the PerformanceEvaluation table
    INSERT INTO PerformanceEvaluation (evaluator_staff_id, evaluated_staff_id, evaluation_date)
    VALUES (para_manager_id, para_staff_id, CURDATE());
    SELECT LAST_INSERT_ID() INTO latest_evaluation_id;

    -- Initialize the base SQL INSERT statement for the EvaluationCriteria table
        SET @insert_query = 'INSERT INTO EvaluationCriteria (evaluation_id, criteria_id,
criteria_score) VALUES ';

    -- Loop through the evaluation_string to process each criteria and score
    WHILE current_index <= LENGTH(evaluation_string) DO
        -- Check if the current character is a comma, indicating the end of a criteria and score
pair
        IF SUBSTRING(evaluation_string, current_index, 1) = ',' THEN
            -- Process the current criteria and score
             SELECT ParsingCriteriaScoreString(latest_evaluation_id, current_string_code, 0) INTO
@single_value;
            -- Append the processed value to the INSERT query
            SET @insert_query = CONCAT(@insert_query, @single_value);
            -- Reset the accumulated criteria and score string for the next iteration
            SET current_string_code = '';
        ELSE
```

```
            -- Accumulate the current criteria and score character
              SET current_string_code = CONCAT(current_string_code, SUBSTRING(evaluation_string,
 current_index, 1));
        END IF;
        -- Move to the next character in the string
        SET current_index = current_index + 1;
    END WHILE;

    -- Process the last criteria and score in the string
         SELECT  ParsingCriteriaScoreString(latest_evaluation_id,  current_string_code,  1)  INTO
 @single_value;
    -- Append the final processed value to the INSERT query
    SET @insert_query = CONCAT(@insert_query, @single_value);

    -- Prepare and execute the final INSERT statement for EvaluationCriteria
    PREPARE statement FROM @insert_query;
    EXECUTE statement;
    DEALLOCATE PREPARE statement;
      SET @parent_proc = NULL;
    -- Commit the transaction to save all changes
    COMMIT;
```

Atomicity is also achieved at the inter-database level. Specifically, the creation of appointments, lab results, and qualifications involve write operations at both databases. When the write operation in a database fails, the application would roll back any changes done to the other to ensure data integrity, as seen in the following code snippet:

```
export async function addNewAppointment(req, res) {
  const transaction = await mongoose.startSession()
  try {
    transaction.startTransaction()
    console.log(transaction.toString())
    const user_info = req.user;
    if (user_info.role !== 'FrontDesk' && user_info.role !== 'Front Desk') {
      return res.status(403).json({ message: 'Unauthorized access' });
    }

    const {
      patient_id,
      doctor_id,
      department_id,
      purpose,
      appointment_date,
      appointment_start_time,
      appointment_end_time,
      pre_appointment_note
    } = req.body;

    // Create a new document in MongoDB using the method from Methods.js
    let newAppointmentNote;
    try {
              newAppointmentNote  =  await  createAppointmentNoteFromPreNote(pre_appointment_note,
{transaction});
    } catch (error) {
      console.error("Error creating appointment note in MongoDB:", error);
      return res.status(500).json({ message: error.message });
    }

    // Retrieve the document id from MongoDB
    const document_id = newAppointmentNote._id.toString();

    // Add the new appointment to the SQL database
    const result = await frontDeskRepo.AddNewAppointment(
```

```
      department_id,
      doctor_id,
      patient_id,
      purpose,
      appointment_date,
      appointment_start_time,
      appointment_end_time,
      document_id
    );

    await transaction.commitTransaction()
    res.status(201).json({
      message: 'Appointment created successfully',
    }

  );
  } catch (error) {
    await transaction.abortTransaction()
    console.log("Operation Aborted");
    res.status(500).json({ message: error.message });
  } finally {
    transaction.endSession()
  }
}
```

## c. Error Handlings

Error handling is a critical component of this project, especially within the context of SQL procedures and functions. To strengthen the system's security and reduce the risk of attacks, we have implemented custom error messages in place of raw MySQL error messages. This strategy effectively prevents the exposure of sensitive database information, which could otherwise be exploited by potential attackers. By using the `SIGNAL SQLSTATE '45000'` statement, we ensure that only meaningful error messages are presented to the application. These messages provide sufficient feedback for debugging and logging without revealing internal database details.

To further manage errors effectively, our implementation makes extensive use of the `DECLARE EXIT HANDLER FOR SQLEXCEPTION` construct. This handler captures SQL exceptions, rolls back transactions to maintain data integrity, and returns generic error messages to the caller. By avoiding the display of detailed error messages, we mitigate the risk of inadvertently exposing system vulnerabilities to users or attackers. The use of this method ensures that the application remains secure, even in the event of unexpected errors.

In addition to custom error messages, we have implemented various validation functions to enhance security. Functions such as `CheckTestOrderExists` and `CheckTestTypeExists` are used to verify the existence of records before operations are performed, ensuring that only valid data is processed. This approach not only prevents unauthorized access but also guards against the execution of potentially harmful SQL statements. Furthermore, parameterized queries are used throughout the stored procedures and functions. By treating input values strictly as data rather than executable code, parameterized queries prevent SQL injection attacks, making the application more secure against such threats.

Overall, these strategies contribute significantly to the prevention of SQL injection attacks and enhance the system’s overall security. Transaction management, using commands like `START TRANSACTION`, `COMMIT`, and `ROLLBACK`, ensures that operations are executed automatically. If an error occurs, the

entire transaction is rolled back, which prevents partial updates and maintains data integrity. Additionally, we have implemented role-based access control, allowing only authorized users to execute specific procedures and functions. This further restricts access to sensitive operations and reduces the potential for unauthorized manipulation of data.

Lastly, all error messages are captured and logged, providing valuable insights into potential security issues and enabling timely responses to any detected threats. By focusing on custom error handling, robust input validation, parameterized queries, and role-based access control, we have significantly enhanced the security of the application. These measures collectively ensure that the system is resilient against unauthorized access and data breaches, maintaining the confidentiality, integrity, and availability of the application.

# 4. Data Security

## a. Database Permission

- Access Control

To ensure maximum security, access to the database is restricted at the row level by exclusively using stored procedures with *SQL SECURITY DEFINER* set as their security context. Execution rights for these procedures are then granted to the five user roles, based on their specific responsibilities:

- **HR**

***Staff Management*:** This, firstly, entails adding new staff into the *Staff* table. The *HR* are allowed view on all columns and all rows of the *Staff* table. Moreover, updates to this table's columns are also granted exclusively to this user type. For instance, HR could update the personal information of Staff (password, email, phone number, home address). The *HR* is also allowed to insert into the Qualifications table

***Wage Change Management*:** This involves performing update operations on the wage column of the staff table and insert operations on the Salary Change table. View of the *Salary_Change* table is granted exclusively to *HR*.

***Job Change Management*.** This involves performing update operations on the *job_id* column of the *Staff* table and insert operations on the *Job_Change* table. In case the staff in question is a doctor, this responsibility also extends to updating the *appointment_status* column of the *Appointments* table (Appointment Cancellation), and the right for doing so is only valid so long as the HR do it within the procedure. View of the *Job_Change* table is granted exclusively to *HR*.

***Department Change Management*.** This involves performing update operations on the *department_id* column of the *Staff* table and insert operations on the *Deparment_Change* table. This also extends to updating the *appointment_status* column of the *Appointments* table in case the staff in question is a doctor. View of the *Department_Change* table is granted exclusively to *HR*.

- **Front Desk Officers**

***Patient Management:*** No restriction is placed on this role when it comes to inserts, updates, and select operation to the *Patients* table. The permission to insert and update patients' non-medical details are granted exclusively to this role

***Appointment Management:*** procedures for inserts, updates, and deletes operations to the *Appointments* table are granted primarily to the *Front_Desk* role. A front desk officer is allowed access to all columns of all entries in the Appointments table. The procedure to create new appointments also allow this role temporary access to all entries in

the *Staff* and the *Staff_Schedule.* This is because for a valid appointment to be created, the doctor must have been scheduled to be on service on that date.

- **Business Officers**

***Billing Management.*** Execute permission to procedures for insert, update, and delete operations to the *Billing* table are granted exclusively to BusinessOfficers role. As parts of the billing creation process, a business officer must input the ids of the prescription, test, and appointments associated with the patient. To preserve data integrity, the database will have to check if the input ids exist in their respective tables, and if they are associated with the input patient's id. As a result, this role is also implicitly granted temporary access to the id columns of the *Appointments*, *TreatmentHistory*, and *Test_Order* tables.

- **Doctors**

***Patient Medical Information Management.*** View of the Patients table is restricted only to the *full_name, birth_date, and gender* column. Moreover, doctors could only get access to patients with whom they have an appointment with on that date. For this, the database must perform a read operation on the Appointments table to fetch *patient_id* from rows with *doctor_id* equal to the *id* of the accessing doctor and *appointment_date* equal to the access date. Since it is the doctors who will record the patients' allergies, doctors are granted permission to insert and update rows in the *PatientAllergy* table.

***Diagnosis Management.*** Doctors are allowed to insert into the *Diagnosis_Details* table for patients with whom they currently have an appointment with. Thanks to the triggers mentioned above, we are able to restrict this privilege to a particular period of the date only. This aside, doctors have access to all columns and all rows in the Diagnoses and Diagnosis_Details table so long as those rows represent the patients who they will be having an appointment with on that date.

***Prescription Management.*** Similar to diagnoses, doctors are granted privilege to perform read and write on the TreatmentHistory and Prescription_Details table. Like above, this privilege is time locked, meaning that doctors could only view prescription information of the patients they will be attending to on that date, and the privilege to write to the TreatmentHistory and Prescription Detail table is restricted to the temporal bound of their appointments.

***Medical Test Management.*** Like above, doctors are granted privileges to perform read and write on the *Test_Orders* and *Test_Details* table, which are time locked. Doctors insert into the *Test_Orders* when they want to conduct medical tests on a patient. Since the *Test_Details* table contains highly confidential information, only doctors with whom the patients will be seeing are allowed to read from that table.

- **Nurses**

For tables that contain medical information such as TreatmentHistory, Diagnoses nurses are given the privileges to perform select operations on all of them. Unlike doctors, these privileges are not time locked. However, the only write privilege they have is updating the *Test_Details* table. This is because medical tests in hospitals are typically caried out by nurses.

Beside the role-specific privileges mentioned above, the followings are some privileges that all roles share:

***Fetching staff information*:** All staff are allowed to fetch the information of their subordinates. This information includes full name, date of birth, phone number, email, and qualifications. However, apart from HR this privilege does not grant access to the highly sensitive *home_address, wage, hire_date* column for the other roles. Moreover, a staff is granted privilege to view, add, update, and delete rows from the Staff_Schedule. While view of certain

columns are restricted when it comes to the subordinates, there is no such restriction when the staff is viewing their own information

***Staff Evaluation:*** A staff is allowed to insert into the PerformanceEvaluation and EvaluationCriteria table so long as the rows they are operating on pertain to a staff under their management

To conclude this section, it is noteworthy that all privileges defined above are only valid within the context of procedures. A doctor, for example, could not insert a new row into the TreatmentHistory table using a plain Insert statement. This approach of exclusively using procedures allows us to implement stringent constraint checking and achieve more granular and time-based access control. This helps mitigate impact on the database should the server gets compromised

## b. SQL Injection prevention

The single most important defensive mechanism for SQL injection attacks is stringent exception handling. These attacks usually happen when attackers have some form of knowledge regarding the structure of the database, knowledge that could potentially be gained by looking at the exceptions returned by the database. For example, by looking at the following exception the attacker could gain preliminary knowledge on the database structure such as what columns a table have, what are the primary and foreign keys:

ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails (`database_name`.`Staff`, CONSTRAINT `Staff_ibfk_1` FOREIGN KEY (`department_id`) REFERENCES `Departments` (`id`) ON DELETE SET NULL)

With this in mind, our database has been designed in such a way that it only propagates custom exception messages to both the client and the server, as seen in the following code snippet.

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
DECLARE returned_sqlstate CHAR(5) DEFAULT '';
        DECLARE returned_message TEXT;
-- Retrieve the SQLSTATE of the current exception
GET STACKED DIAGNOSTICS CONDITION 1
returned_sqlstate = RETURNED_SQLSTATE;
-- Check if the SQLSTATE is '45000'
IF returned_sqlstate = '45000' THEN
-- Resignal with the original message
RESIGNAL;
ELSE
-- Set a custom error message and resignal with SQLSTATE '45000'
SIGNAL SQLSTATE '45000'
SET MESSAGE_TEXT = 'Something is wrong. Please try again.';
END IF;
END;
```

In addition to this, the fact that all queries sent to the database are solely parameterized procedure calls whose parameters are scrutinized by constraint checkers makes it difficult, if not impossible, to launch a sql injection attack

## c. Password Hashing

For this hospital management application, we utilized the node.bcrypt.js library, a bcrypt implementation for NodeJS, installed via NPM, to hash the password of a staff member when their account is stored inside MySQL database. Bcrypt library is widely regarded as the most secure method for password hashing, specifically designed to enhance backend security against dictionary attacks (Batubara, 2021). Therefore, we believe that bcrypt would be a perfect implementation to enhance the security aspects for our application.

In our backend directory, we used Middleware/auth.js to store the middleware function for the authentication process. We have 3 distinct functions for authentication. The generateTokens function generates two types of tokens: an access token and a refresh token. The access token is created using the user's id, email, and role as payload data, along with a secret key **(process.env.ACCESS_TOKEN_SECRET)** and is set to expire in 15 minutes. The refresh token, also created with user-specific data and a different secret key **(process.env.REFRESH_TOKEN_SECRET)**, is designed to expire in 7 days. The function returns an object containing both tokens. To securely store these tokens, the setTokenCookie function sets them as HTTP-only cookies in the user's browser, using the res.cookie() method. The access token cookie is configured to expire after 15 minutes, while the refresh token cookie has a lifespan of 7 days. Both cookies are marked as HTTP-only to prevent access from client-side scripts and are only sent over HTTPS connections in production environments, enhancing security against attacks like Cross-Site Scripting (XSS).

The verifyToken function serves as middleware to protect routes by verifying the access token included in the user's cookies. It first retrieves the accessToken from the cookies (req.cookies.accessToken) and checks if it is present. If the token is missing, the function returns a 403 status (Forbidden) with a relevant message. If the token is provided, the function uses jwt.verify() to validate it against the secret key. If the token is valid, the user's information is decoded and attached to the request object (req.user), allowing authorized access to the route. If the token is invalid or expired, a 401 status (Unauthorized) is returned. Additionally, the function includes error handling to manage unexpected errors, responding with a 500 status (Internal Server Error) when necessary. This implementation ensures secure management of user authentication and session handling within the application.

## d. Additional Security Mechanism

Another additional security mechanism that we integrated in our project is the leverage of `. env` files to keep critical information hidden. This file allows us to separate sensitive data, such as passwords and API keys, from the actual codebase when sharing our code with others.

Integrating JWTs enhances the protection of sensitive information and user data, which is essential for secure authentication and authorization processes. By reducing the need for frequent database queries, JWTs help safeguard sensitive data, as noted by Shingala (2019). When a user is logged in, JWTs containing unique claim sets or ID Tokens help maintain data integrity.

Additionally, our implementation avoids storing JWTs in Local Storage, opting instead to use cookies to mitigate the risk of Cross-Site Scripting (XSS) attacks. Storing tokens, which might serve as credentials for API calls, in Local Storage can expose them to theft during an XSS attack (Malik, 2024). Such vulnerabilities can result in account hijacking, credential theft, data leakage, and other security breaches that could compromise users' sensitive information.

# Conclusion

In conclusion, this report has outlined the design and implementation of our Hospital Management System (HMS), focusing on the key components that contributed to its development: database design, performance analysis, data consistency, and data security. An optimal database design was essential to the project's success, involving multiple iterations to ensure that the final structure met all project goals while maintaining efficiency. This solid foundation enabled effective performance optimization, including the prioritization of bulk operations and the selective use of non-clustered indexes for optimal query performance.

Data consistency was maintained through the use of triggers, transactions, and error-handling mechanisms, ensuring accurate data storage and transfers. Additionally, a combination of database permissions and security measures established a high level of data security within the HMS, making it reliable and secure.

While the current system meets the hospital's needs, it is important to acknowledge that, with more time, we could have further refined and enhanced the system, incorporating additional features and improvements to deliver an even better product. Future development could involve scaling the system to handle larger volumes of data and more complex requirements.

Overall, this project carefully balanced technical design with practical business needs, resulting in a secure, fast, and effective Hospital Management System. With guidance from our lectures and tutorials, we have created a product that successfully fulfills its intended purpose, while recognizing that there is always room for further enhancement.

# References

https://www.auhd.edu.ye/upfiles/elibrary/Azal2020-01-22-12-28-11-76901.pdf

Batubara, TP, Efendi, S & Nababan, EB 2021, 'Analysis performance BCRYPT algorithm to improve password security from brute force'. Available at: https://iopscience.iop.org/article/10.1088/1742-6596/1811/1/012129/pdf

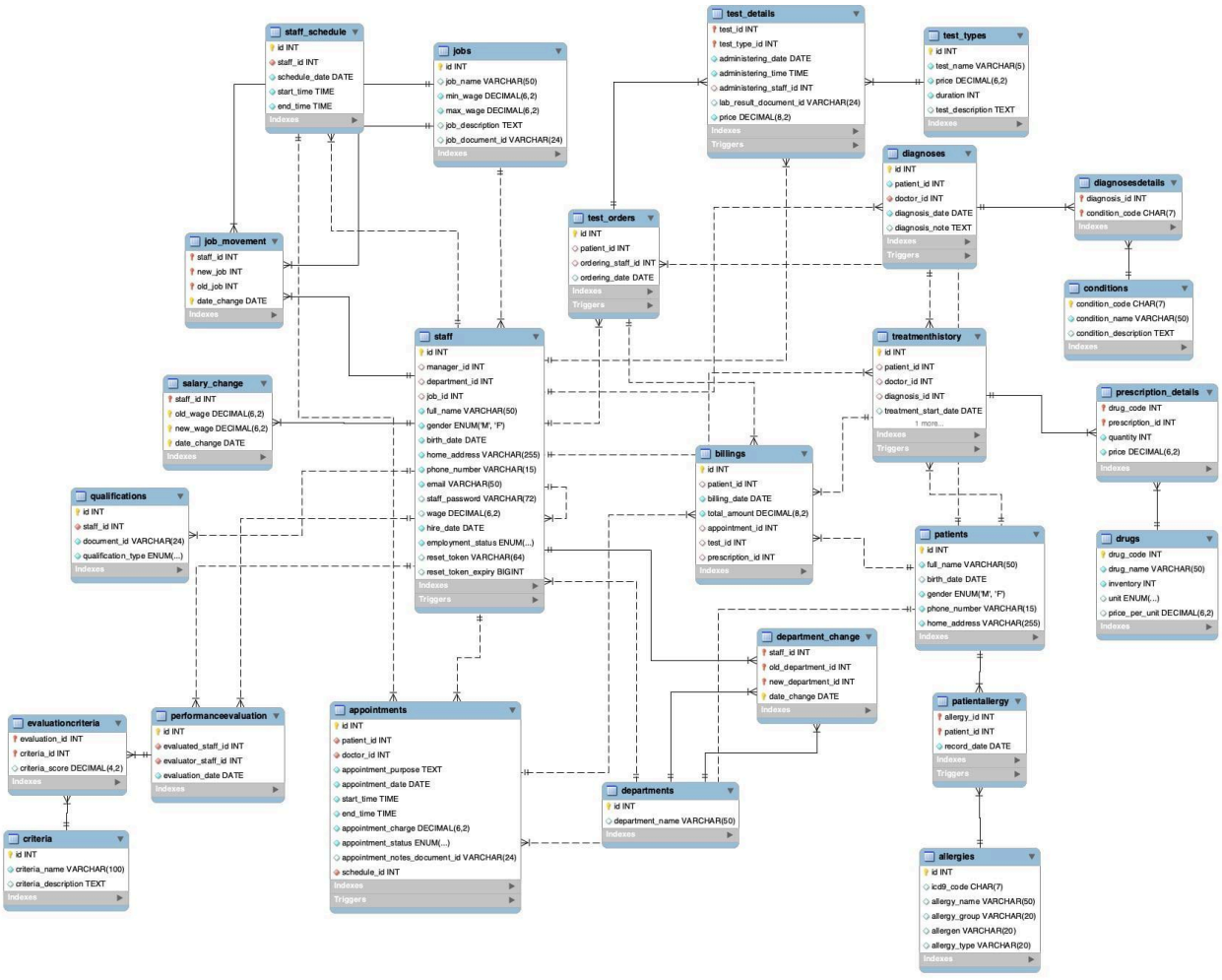
Shingala, K 2019, 'Json web token (JWT) based client authentication in message queuing telemetry transport (MQTT)', Available at: https://arxiv.org/pdf/1903.02895

Malik, M.S., 2024. Mitigation of XSS Vulnerabilities in Add-ons for Cloud based Applications and a Forensic Framework for Investigating Incidents. Available at: https://www.theseus.fi/bitstream/handle/10024/865288/Saad_Malik.pdf?sequence=2&isAllowed=y

# Appendix

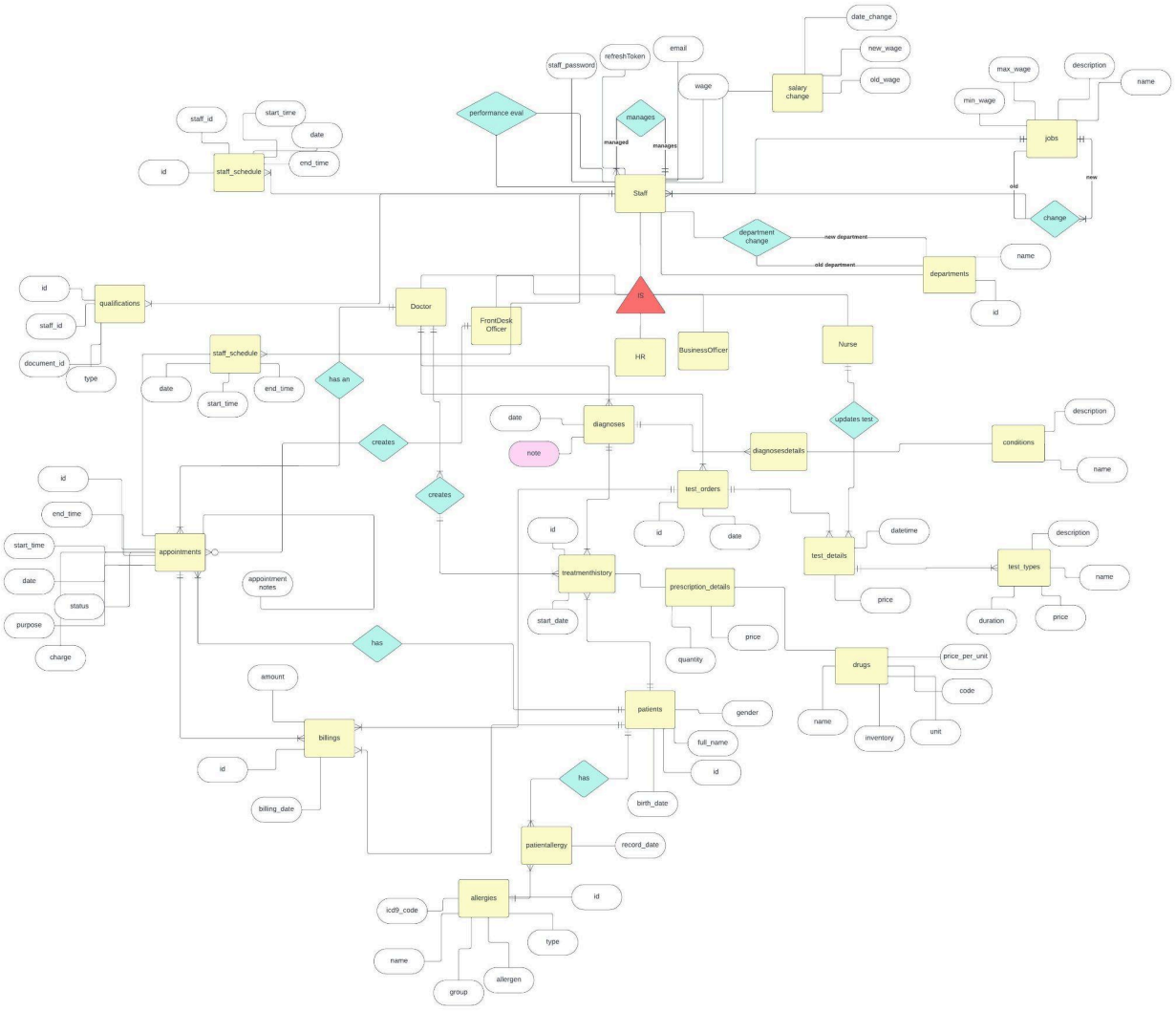
**Figma Design:**
https://www.figma.com/design/36ju4Eqkd8KWnqxZscgdMm/database_application_main_design?node-id=3-2&node-type=CANVAS&t=Fa9RwjvwkYwwiZWY-0

## A. Relational Schema




Higher quality image of the relational Schema could be found in the github

ERD


date_change
new_wage
old_wage
salary change
email
refreshToken
staff_password
wage
performance eval
manages
managed
manages
max_wage
description
name
min_wage
jobs
staff_id
start_time
date
id
staff_schedule
end_time
Staff
old
new
change
department change
new department
old department
name
departments
id
id
qualifications
staff_id
document_id
type
Doctor
FrontDesk Officer
IS
HR
BusinessOfficer
Nurse
staff_schedule
date
start_time
end_time
has an
date
diagnoses
note
creates
creates
updates test
description
conditions
name
diagnosesdetails
id
end_time
start_time
date
status
purpose
charge
appointments
appointment notes
has
test_orders
id
date
id
treatmenthistory
start_date
prescription_details
price
quantity
test_details
datetime
price
description
test_types
name
duration
price
drugs
price_per_unit
code
name
inventory
unit
amount
billings
id
billing_date
patients
gender
full_name
id
birth_date
has
patientallergy
record_date
allergies
id
icd9_code
type
name
allergen
group