CADT IDT

Database Administation

Final Project Report on

Hospital Management System (HMS)

Lecturer: Thear Sophal

Lecturer: Ang Mengchhuong

Preapared By :

Chhun Nika

Kon Sotheara

Kim Solida

Ly Sreypich

SE Group 01

Table of Contents

I. Introduction

As the healthcare sector continues to grow and become more complex, managing hospital operations such as patient registration, appointment scheduling, medical records, billing, and staff coordination has become increasingly challenging. Many hospitals still rely on manual paperwork or separate, disconnected systems, which often lead to data loss, delays, human errors, and poor communication between departments.

To address these problems, our team developed a Hospital Management System (HMS) which is a software application designed to help hospitals manage their daily operations more efficiently. An HMS handles essential tasks such as patient registration, appointment scheduling, storing medical records, managing billing processes, and overseeing hospital staff including receptionists, doctors, and nurses. It improves efficiency, reduces paperwork, and ensures better coordination between departments by centralizing all key data and operations in one system.

The main objective of this project is to streamline hospital workflows, improve data accuracy, and enhance communication through an integrated digital system. This report covers the design of the HMS database, including the ER diagram and relational schema, the implementation using SQL, data population, and user management with role-based access control. It also discusses backup and recovery plans, as well as the development and optimization of complex SQL queries to support hospital operations.

II. Database Design

The Hospital Management System (HMS) is designed using a relational database model to ensure data integrity, reduce redundancy, and enable efficient data retrieval. The design focuses on core hospital operations such as managing patients, appointments, staff, medical records, and billing.

1. Entity Relational (ER) diagram

The Entity-Relationship (ER) diagram is a visual representation of the major entities in a system and the relationships between them. It helps in designing the database structure by clearly showing how data is organized and how different parts of the

system interact. In our Hospital Management System project, we have a total of 11 entities, which include both strong and weak entities. Among these, 9 are strong entities, meaning they have their own primary keys and can exist independently.

1.1. Strong Entity

Strong Entities are those that have their own primary key, ensuring their existence is independent of other entities. These entities typically represent core objects or concepts within the system, such as:

- **Patient**: Represents individuals receiving medical care at the hospital. Stores personal information such as name, gender, date of birth, contact details, emergency contacts, and registration date.
- **Doctor:** Contains details of medical professionals, including their department, specialization, contact information, and work schedule.
- **Nurse:** Holds information about nursing staff, including their department assignment, shift, contact details, and hire date.
- **Receptionist:** Represents hospital front-desk staff responsible for patient registration, appointment scheduling, and general administrative tasks. Stores contact info, department, and shift.
- **Department**: Represents the hospital's various clinical and administrative departments (e.g., Cardiology, Radiology), including the department name, type, location, and head of department.
- **Medication:** Stores details about drugs and medicines used in patient treatment, including name, description, price, and expiration date.
- **Room:** Tracks hospital rooms, including room number, type, availability status, and current patient assignment.
- **Appointment:** Represents scheduled visits between patients and doctors, including date/time, reason, and status (e.g., Scheduled, Completed).
- **Billing:** Manages financial transactions related to patient care, recording amounts due, payment status, billing date, and payment methods.

1.2. Weak Entity

On the other hand, weak entities are dependent on strong entities and do not have their own primary key. These entities rely on the existence of strong entities to define their identity and relationships.

- **Medical_Record:** Contains detailed medical information related to a specific appointment, such as diagnosis, treatment, visit date, and doctor's notes. Depends on the appointment entity to exist.
- **Prescription:** Represents medication prescriptions linked to a medical record, including the date issued and any additional notes. Cannot exist without a corresponding medical record.

1.3. Relationship

Our Hospital Management System includes various relationships that reflect real-world hospital operations. These relationships, such as one-to-many and many-to-many, ensure accurate links between entities like patients, doctors, appointments, and medications. Below are the key relationships defined in the system:

- **Department and Doctor (1:M Relationship) :** The Department entity has a one-to-many (1:M) relationship with the Doctor entity. Each department can have multiple doctors, but each doctor belongs to only one department. This is established through the dep_id foreign key in the Doctor table, referencing department_id in the Department table.
- **Department and Nurse (1 : M Relationship):** Each department can have many nurses, but each nurse is assigned to only one department. The relationship is created using the dep_id foreign key in the Nurse table.
- **Department and Receptionist (1 : M Relationship):** One department can employ multiple receptionists. The department_id foreign key in the Receptionist table links each receptionist to a specific department.
- **Department and Room (1:M Relationship):** Each department can have multiple rooms assigned to it. This is defined by the department_id foreign key in the Room table.
- **Department and Medication (M : N Relationship):** Departments can store many types of medication, and each medication can be stocked in multiple departments. This many-to-many relationship is implemented through the DepartmentMedication junction table, which also tracks stock_quantity.
- **Doctor and Appointment (1 : M Relationship):** A doctor can have many appointments with patients. This relationship is handled by the doctor_id foreign key in the Appointment table.

- **Patient and Appointment (1:M Relationship):** Each patient can have multiple appointments. The patient_id in the Appointment table establishes this link.
- **Appointment and Medical_Record (1 : 1 or 1 : M Relationship) :** Each appointment can have one or more medical records (depending on design). The appointment_id in the Medical_Record table represents this relationship.
- **Medical_Record and Prescription (1 : M Relationship):** Each medical record can have multiple prescriptions. This relationship is established by the medical_record_id in the Prescription table.
- **Prescription and Medication (M : N Relationship via):** Prescriptions can include multiple medications, and a medication can be part of many prescriptions. This many-to-many relationship is implemented via the PrescriptionMedication table.
- **Appointment and Billing (1 : 1 Relationship):** Each appointment results in exactly one bill. This one-to-one relationship is maintained by the appointment_id foreign key in the Billing table, ensuring that every billing record corresponds to a specific appointment.
- **Patient and Billing (1 : M Relationship):** A single patient can have multiple billing records due to multiple visits or services. This one-to-many relationship is managed using the patient_id foreign key in the Billing table, linking each bill to the appropriate patient.
- **Patient and Room (1 : M Relationship):** Each patient can be assigned to only one room at a time, and each room can accommodate many patients. The assigned_patient_id foreign key in the Room table captures this exclusive one-to-many relationship.

The finalized ER diagram for the Hospital Management System is shown in Figure 01

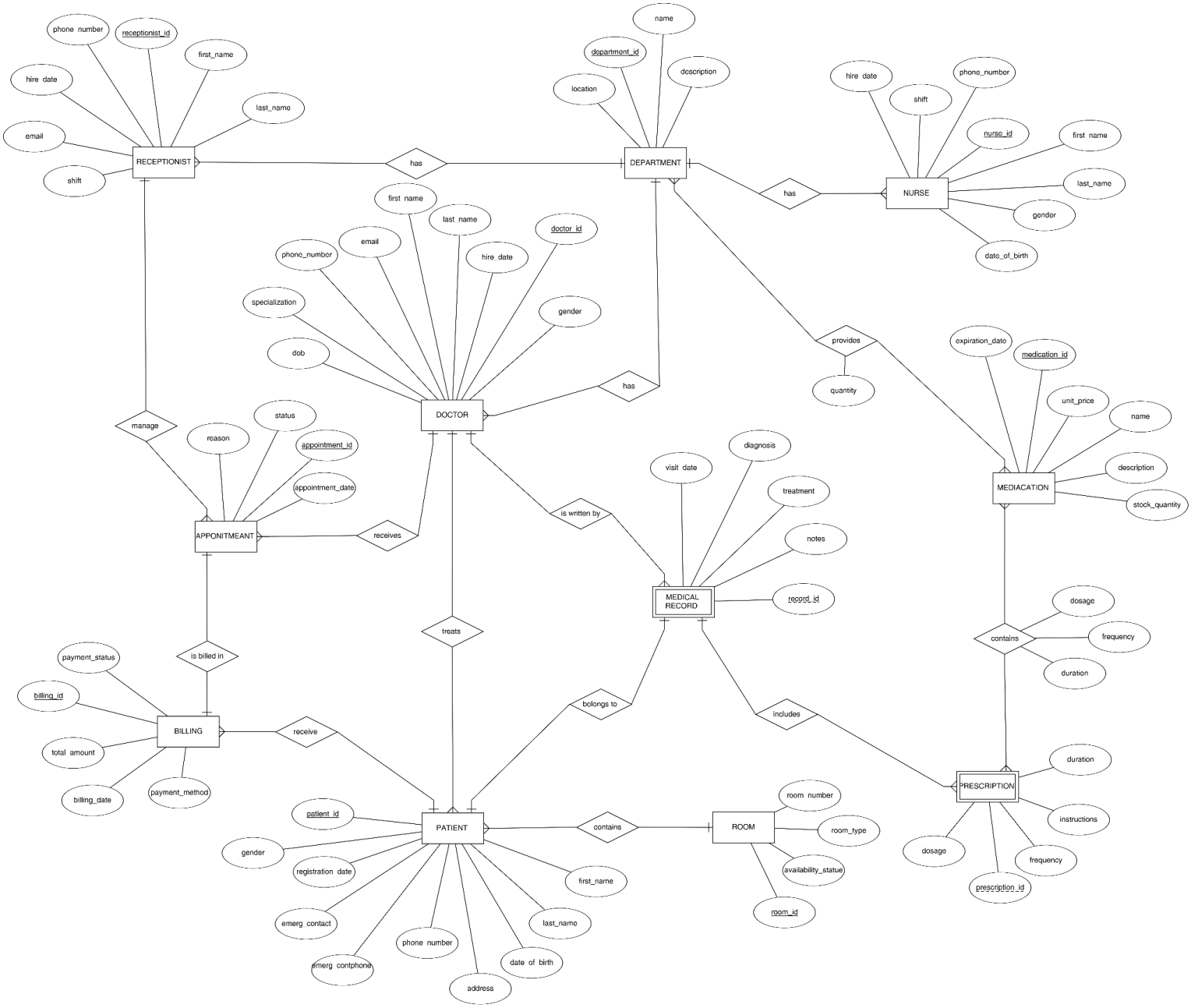


Figure 01

2. Relational Model (RM)

After designing the Entity-Relationship (ER) diagram for the Hospital Management System, we converted it into a relational model to implement the database using SQL. In this process, each entity in the ER diagram became a table, and each attribute of those entities was mapped as a column in their respective tables.

We defined primary keys (PK) to uniquely identify each record (e.g., patient_id in the Patient table, doctor_id in the Doctor table), and foreign keys (FK) to enforce

relationships between related entities (e.g., doctor_id in the Appointment table refers to the Doctor table).
This relational structure ensures **data consistency** and **referential integrity**, making it easier to track interactions among different components of the hospital system (e.g., which doctor treated which patient, or which appointment has a corresponding bill).

Additionally, to represent many-to-many (M : N) relationships, we created junction tables such as:

- **DepartmentMedication** (to track which medications are available in which departments along with stock quantity)
- **PrescriptionMedication** (to associate each prescription with multiple medications and include dosage, frequency, and duration).

These junction tables are crucial because relational databases do not natively support many-to-many relationships. They break down these relationships into two one-to-many (1:M) relationships using foreign keys, allowing us to store more contextual information and preserve the link between the entities in a normalized, scalable way.

The finalized relational model for the Hospital Management System is shown in Figure 02

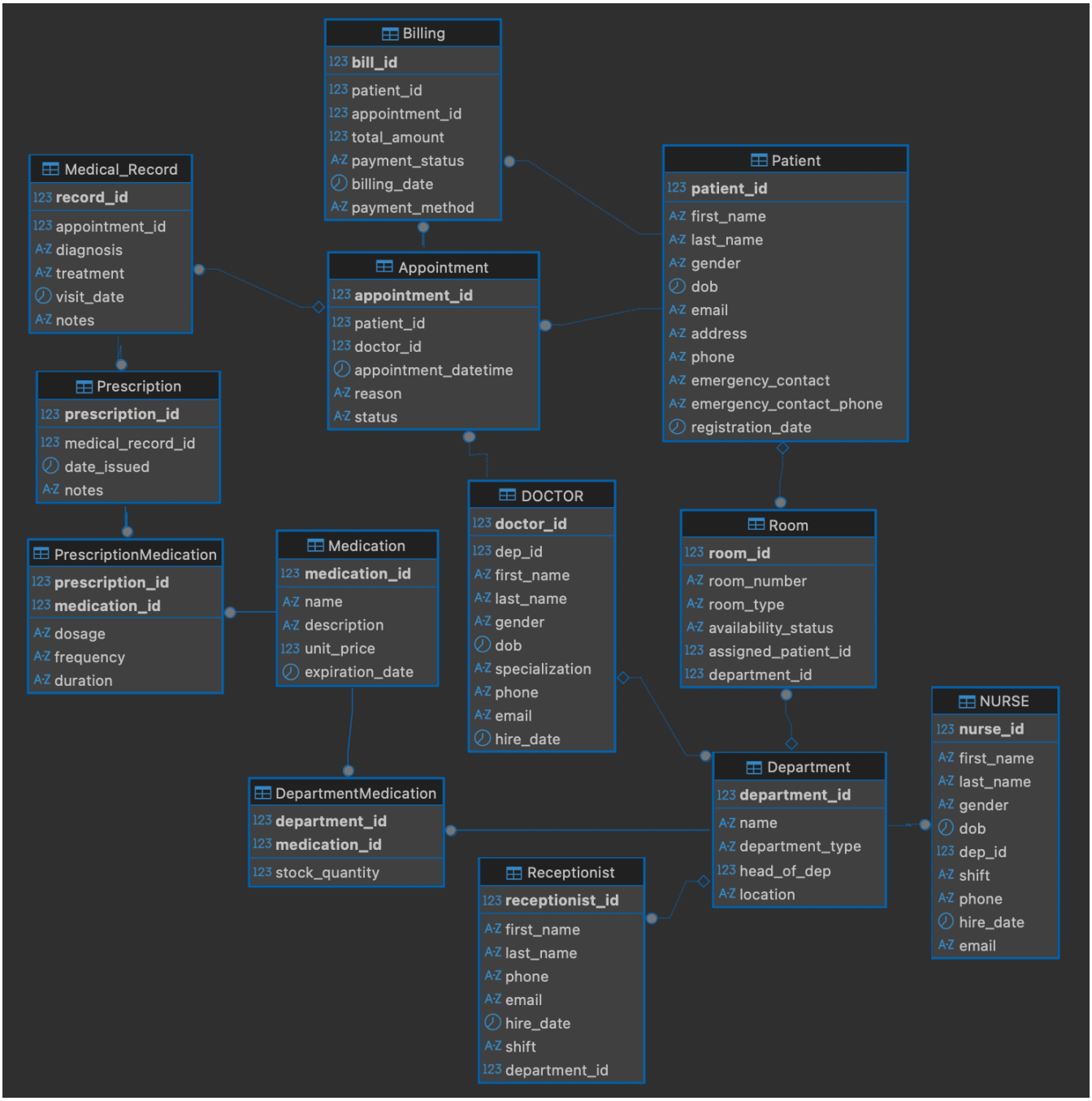


Figure 02

III. Database Implementation

After completing the ER diagram and relational model, we moved to the implementation phase. In this part, we translated the design into a working database using SQL and Python. This section presents the implementation of the Hospital Management System (HMS) database, including the creation of the schema using SQL Data Definition Language (DDL), the population of data using Python scripts with Data Manipulation Language (DML) to insert approximately the million records into each

table, and the development of 15 optimized SQL queries that include joins, subqueries, aggregations, functions, stored procedures, and triggers to support key hospital operations.

1. Data Definition Language (DDL)

The database schema was implemented using SQL Data Definition Language (DDL), enabling us to create tables, define primary keys, and establish constraints that accurately reflect the system's data requirements. Each entity from the ER diagram was translated into a corresponding table with columns representing its attributes, and primary keys such as patient_id and doctor_id were defined to uniquely identify records. To maintain data integrity and enforce relationships, foreign keys were set up between related tables.

To represent many-to-many relationships, we created junction tables like **DepartmentMedication** and **PrescriptionMedication**. These tables use composite primary keys and foreign keys to link related entities while storing additional details such as stock quantity and medication dosage.

We also applied various constraints, including **NOT NULL**, **UNIQUE**, and **DEFAULT** values, to ensure valid data entry. Enumerated types (ENUM) were also used for fields with fixed sets of values, such as gender, appointment status, and staff shifts.
This structured implementation approach results in a well-organized, normalized database design that reduces redundancy, supports efficient querying, and allows for future scalability.

The tables that we have created using Data Definition language is shown in Figure 3
*** **NOTE:** The DDL script is in the **MySQL_script** folder, which is accessible via the **GitHub** link provided in the submission.

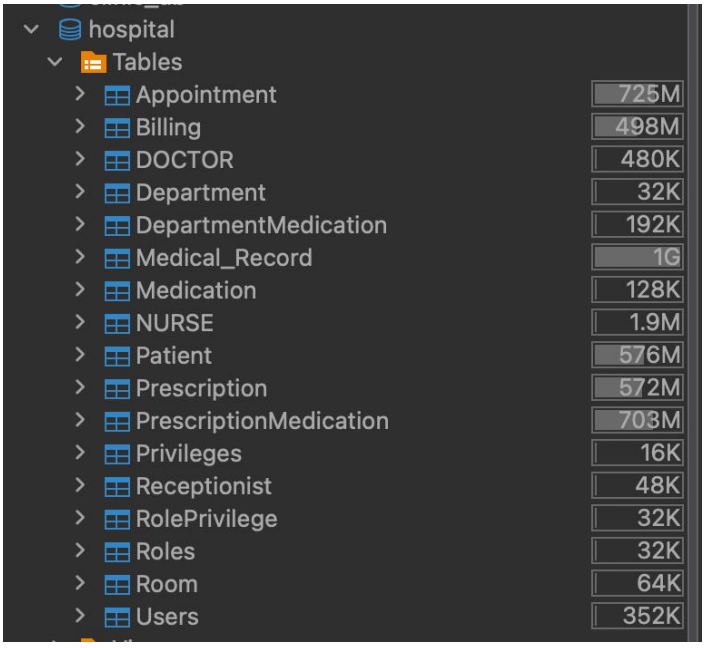


Figure 03

In total of 17 tables, where 13 tables are Hospital management system tables and the other 4 tables are for web-based of Role-Base Access Control (RBAC)

2. Data Population (DML)

Due to the large volume of data needed to simulate realistic hospital operations, our team decided to automate the data insertion process using Python instead of adding records manually. We used the Faker library to generate realistic fake data for all tables in the system, including patients, doctors, appointments, medical records, billing, and more.

Each table was populated with a different number of records based on its role in the system. For example, key tables like Patient, Appointment, and Medical_Record had millions of entries to reflect real-world usage, while reference tables such as Department and Room had fewer but meaningful values. We ensured that all generated data respected foreign key constraints, maintained logical consistency (e.g., valid dates of birth, unique emails), and matched the schema's validation rules.

This automated approach allowed us to efficiently populate the entire database, test system performance, and ensure the platform could handle realistic workloads during queries, functions, and triggers.

The Python script for generating Data files is shown in Figure 04

***** NOTE:** The python script for generating data is in the **pythonScript** folder, which is accessible via the **GitHub** link provided in the submission.

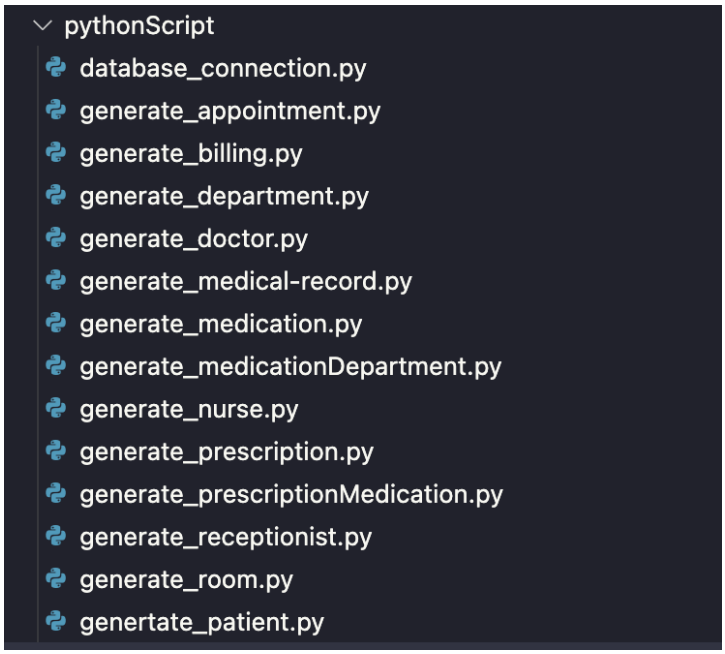


Figure 04

In total of 14 python files including the separation file of database connection function.

Additionally, since the system involves large-scale data insertion with over **10 million** records it became necessary to expose the MySQL database through **Tailscale** as shown in **Figure 05**. This enables secure remote access for team members, ensuring collaboration and data availability across different environments.

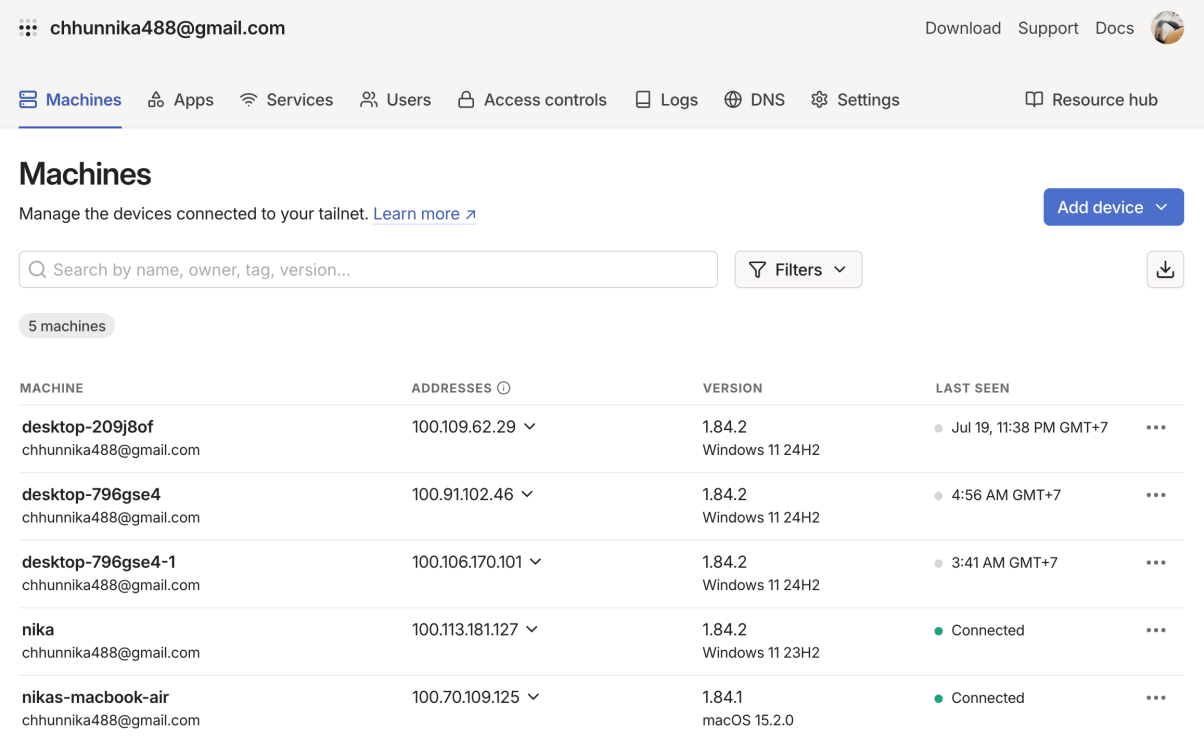


Figure 05

3. SQL Query

We developed a set of at least 15 SQL queries to support common hospital operations and meet system requirements. These queries include complex joins to retrieve and combine data from multiple tables, subqueries for precise filtering and data extraction, and aggregations to summarize information such as total billing amounts. Additionally, we implemented user-defined functions, triggers, and stored procedures to automate routine tasks and enforce business logic.

3.1. Join

Join is one of the query types we developed to combine data from multiple related tables in the Hospital Management System. Since many tables are connected through foreign keys, using JOIN allows us to bring that related information together in one result. This includes:

- **List all appointments with patient and doctor full names and their status**

*** Query ***

```
SELECT
  a.appointment_id,
```

```
    CONCAT(p.first_name, ' ', p.last_name) AS patient_name,
    CONCAT(d.first_name, ' ', d.last_name) AS doctor_name,
    a.appointment_datetime,
    a.status
FROM Appointment a
JOIN Patient p ON a.patient_id = p.patient_id
JOIN Doctor d ON a.doctor_id = d.doctor_id;
```

Description: This query retrieves detailed information about each appointment by combining data from the Appointment, Patient, and Doctor tables. It shows the appointment ID, patient name, doctor name, appointment date/time, and the current status of the appointment (e.g., Scheduled, Completed).

Importance: We created this query to give hospital staff a clear and organized view of all appointments. It helps them manage daily schedules, track patient visits, and ensure smooth coordination between patients and doctors. By combining relevant details in one result, it reduces the need to check multiple tables manually.

Result:

appointment_id	patient_name	doctor_name	appointment_datetime	status
679	Evan Harmon	Anne Guzman	2025-06-12 19:58:06	No-Show
3730	Christopher Smith	Anne Guzman	2023-11-27 22:00:50	No-Show
8196	Ryan Parker	Anne Guzman	2023-11-08 17:02:44	Scheduled
14882	Kenneth Price	Anne Guzman	2024-07-20 19:38:21	Completed

- **List doctors and their departments**

*** Query ***

```
SELECT
    CONCAT(d.first_name, ' ', d.last_name) AS doctor_name,
    dept.name AS department_name
FROM Doctor d
JOIN Department dept ON d.dep_id = dept.department_id;
```

Description: This query links doctors to their respective departments by joining the Doctor and Department tables. It shows each doctor's full name along with the department name they belong to.

Importance: This query is important for managing and organizing hospital staff. It helps administrators understand which doctor is assigned to which department, making it easier to coordinate internal operations, assign tasks, and allocate resources appropriately.

Result:

doctor_name	department_name
Anne Guzman	Neurology
Melanie Goodman	Nephrology
Kenneth Cook	Endocrinology
Jason Miller	Pediatrics
Angela West	Radiology

- **List rooms and the patient assigned**

*** Query ***

```
SELECT
   r.room_number,
   r.room_type,
   r.availability_status,
   CONCAT(p.first_name, ' ', p.last_name) AS patient_name
FROM Room r
LEFT JOIN Patient p ON r.assigned_patient_id = p.patient_id;
```

Description: This query retrieves information about hospital rooms, including room number, room type, availability status, and the full name of the assigned patient. It uses a LEFT JOIN to ensure that even rooms without any assigned patients are included.

Importance: We use this query to help hospital staff track room occupancy and availability in real time. This supports better room allocation, improves patient flow, and ensures no room is overlooked during patient assignment or discharge.

Result:

room_number	room_type	availability_status	patient_name
R001	Double	Occupied	Betty Gomez
R002	Deluxe	Occupied	Dana Garza
R003	Deluxe	Occupied	Amanda Trevino
R004	Double	Available	NULL
R005	Double	Occupied	Sandra Nguyen
R006	ICU	Occupied	Matthew Rogers
R007	Double	Available	NULL

3.2. Sub Query

Subquery is a query nested inside another query used to filter or retrieve data based on results from the inner query. In the Hospital Management System, subqueries help perform multi-step analysis and complex filtering across related tables. This include:

- **Get all patients who had appointment with a doctor specializing in 'Cardiology'**

*** Query ***

```
SELECT * FROM Patient
WHERE patient_id IN (
   SELECT a.patient_id
   FROM Appointment a
   JOIN Doctor d ON a.doctor_id = d.doctor_id
   WHERE d.specialization = 'Cardiology'
);
```

Description: This query retrieves patients who have visited doctors specializing in Cardiology by using a subquery to find patient IDs linked to such appointments.

Importance: It helps hospital staff identify and analyze patients receiving specialized care in Cardiology, which can assist in targeted treatments, resource planning, and reporting.

Result:

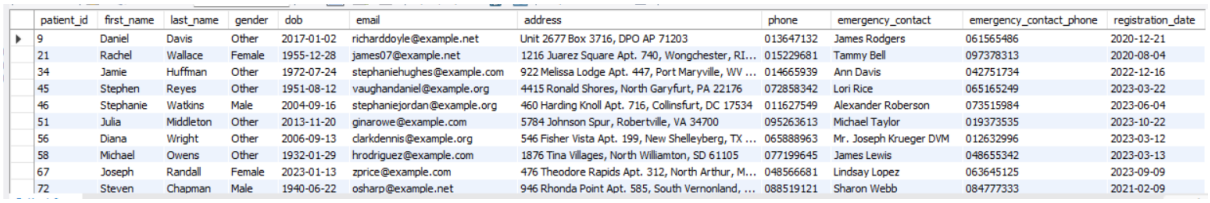
patient_id	first_name	last_name	gender	dob	email	address	phone	emergency_contact	emergency_contact_phone	registration_date
20	Nicole	Bennett	Male	2007-12-21	garneredward@example.net	969 Kim Row, West Thomasberg, OH 36331	032182861	Cassandra Bates MD	061647736	2022-08-27
21	Rachel	Wallace	Female	1955-12-28	james07@example.net	1216 Juarez Square Apt. 740, Wongchester, RI...	015229681	Tammy Bell	097378313	2020-08-04
28	Manuel	Miller	Male	1968-04-06	peggyross@example.org	96884 Walker Glen Apt. 074, Port Maryberg, IN...	059373163	Terrance Horton	065661815	2025-06-08
38	Matthew	Roberson	Male	1999-08-07	uwhite@example.com	1099 Ritter Drive, Wellsshire, LA 38498	019498681	Maria Franco	098537529	2024-09-22
45	Stephen	Reyes	Other	1951-08-12	vaughandaniel@example.org	4415 Ronald Shores, North Garyfurt, PA 22176	072858342	Lori Rice	065165249	2023-03-22
46	Stephanie	Watkins	Male	2004-09-16	stephaniejordan@example.org	460 Harding Knoll Apt. 716, Collinsfurt, DC 17534	011627549	Alexander Roberson	073515984	2023-06-04
51	Julia	Middleton	Other	2013-11-20	ginarowe@example.com	5784 Johnson Spur, Robertville, VA 34700	095263613	Michael Taylor	019373535	2023-10-22
53	Gwendolyn	Molina	Other	2014-09-26	acevedoamanda@example.net	242 Willis Vista Suite 366, Cunninghamstad, FL ...	087548173	Adrienne Davis	085783476	2023-01-01
70	Nicole	Shelton	Male	2000-10-08	brian28@example.net	51183 Valenzuela Greens Apt. 455, South Mark...	096322176	Anita Norman	097921578	2021-12-08
72	Steven	Chapman	Male	1940-06-22	osharp@example.net	946 Rhonda Point Apt. 585, South Vernonland, ...	088519121	Sharon Webb	084777333	2021-02-09
77	Billy	Payne	Female	2024-04-26	ysmith@example.com	0237 Stephenson Corner Apt. 650, Ramirezfurt...	067751168	Heather Lewis	088825567	2023-03-31
82	Anna	Castro	Female	2024-01-07	oconnellkara@example.com	121 Clark Trafficway Apt. 951, Alexismouth, GU...	078355372	Terry Sutton	082149157	2021-02-20
93	Kim	Martin	Male	1985-11-30	boonegabrielle@example.org	512 Mack Mission Apt. 293, Timothyberg, AZ 63...	039335938	Jordan Smith	026613841	2023-01-06
96	Sheena	Elliott	Female	1967-05-20	amber64@example.com	USNS Hansen, FPO AA 18006	063926254	Samantha Moore	052432854	2021-10-17

- **List patients who have more than 3 appointments**

*** Query ***

```
SELECT * FROM Patient
WHERE patient_id IN (
   SELECT patient_id FROM Appointment
   GROUP BY patient_id
   HAVING COUNT(*) > 3
);
```

Description: This query finds patients with more than three appointments by grouping appointments by patient and filtering those with counts above three.
Importance: Identifying frequent patients supports better management of chronic or ongoing treatments and helps allocate hospital resources effectively.
Result:



patient_id	first_name	last_name	gender	dob	email	address	phone	emergency_contact	emergency_contact_phone	registration_date
9	Daniel	Davis	Other	2017-01-02	richarddoyle@example.net	Unit 2677 Box 3716, DPO AP 71203	013647132	James Rodgers	061565486	2020-12-21
21	Rachel	Wallace	Female	1955-12-28	james07@example.net	1216 Juarez Square Apt. 740, Wongchester, RI...	015229681	Tammy Bell	097378313	2020-08-04
34	Jamie	Huffman	Other	1972-07-24	stephaniehughes@example.com	922 Melissa Lodge Apt. 447, Port Maryville, WV ...	014665939	Ann Davis	042751734	2022-12-16
45	Stephen	Reyes	Other	1951-08-12	vaughandaniel@example.org	4415 Ronald Shores, North Garyfurt, PA 22176	072858342	Lori Rice	065165249	2023-03-22
46	Stephanie	Watkins	Male	2004-09-16	stephaniejordan@example.org	460 Harding Knoll Apt. 716, Collinsfurt, DC 17534	011627549	Alexander Roberson	073515984	2023-06-04
51	Julia	Middleton	Other	2013-11-20	ginarowe@example.com	5784 Johnson Spur, Robertville, VA 34700	095263613	Michael Taylor	019373535	2023-10-22
56	Diana	Wright	Other	2006-09-13	clarkdennis@example.org	546 Fisher Vista Apt. 199, New Shelleyberg, TX ...	065888963	Mr. Joseph Krueger DVM	012632996	2023-03-12
58	Michael	Owens	Other	1932-01-29	hrodriguez@example.com	1876 Tina Villages, North Williamton, SD 61105	077199645	James Lewis	048655342	2023-03-13
67	Joseph	Randall	Female	2023-01-13	zprice@example.com	476 Theodore Rapids Apt. 312, North Arthur, M...	048566681	Lindsay Lopez	063645125	2023-09-09
72	Steven	Chapman	Male	1940-06-22	osharp@example.net	946 Rhonda Point Apt. 585, South Vernonland, ...	088519121	Sharon Webb	084777333	2021-02-09

- **Get departments with more than 1 doctor**

*** Query ***

```
SELECT * FROM Department
WHERE department_id IN (
   SELECT dep_id FROM Doctor
   GROUP BY dep_id
   HAVING COUNT(*) > 1
);
```

Description: This query lists departments that have more than one doctor by grouping doctors by their department and selecting those departments that meet the condition.
Importance: It assists in understanding departmental staffing levels and workload distribution, facilitating better departmental planning and management.

Result:

department_id	name	department_type	head_of_dep	location
1	Cardiology	Medical	505	Building D, Floor 5
2	Neurology	Medical	1084	Building B, Floor 2
3	Pediatrics	Medical	2068	Building E, Floor 4
4	Oncology	Medical	1998	Building B, Floor 1
5	Orthopedics	Surgical	631	Building A, Floor 4
6	Dermatology	Medical	848	Building A, Floor 5
7	Radiology	Diagnostic	2966	Building C, Floor 5
8	Psychiatry	Mental Health	2175	Building E, Floor 3
9	Anesthesiology	Critical Care	2283	Building C, Floor 3

3.3. Aggregation

Aggregation is one of the query types we developed to summarize and analyze data in the Hospital Management System. Aggregation functions like COUNT, SUM, AVG, MIN, and MAX help us calculate totals, averages, and other summary statistics across multiple records. These queries are important for generating reports, monitoring hospital performance, and supporting decision-making. This includes:

- **Total billing amount per patient**

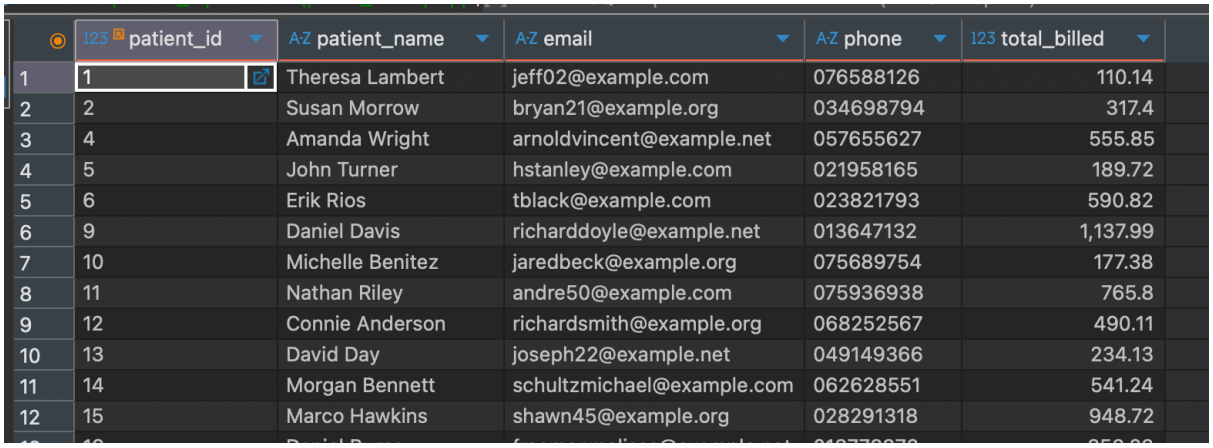
*** Query ***

```
SELECT
   b.patient_id,
   CONCAT(p.first_name, ' ', p.last_name) AS patient_name,
   p.email,
   p.phone,
   SUM(b.total_amount) AS total_billed
FROM Billing b
JOIN Patient p ON b.patient_id = p.patient_id
GROUP BY b.patient_id, p.first_name, p.last_name, p.email, p.phone;
```

Description: This query calculates the total billing amount for each patient by summing all their bills, showing patient details alongside the total charges.

Importance: It is essential for the hospital's finance team to track the overall amount billed to each patient, helping with billing management, payment follow-ups, and financial reporting.

Result:



	patient_id	patient_name	email	phone	total_billed
1	1	Theresa Lambert	jeff02@example.com	076588126	110.14
2	2	Susan Morrow	bryan21@example.org	034698794	317.4
3	4	Amanda Wright	arnoldvincent@example.net	057655627	555.85
4	5	John Turner	hstanley@example.com	021958165	189.72
5	6	Erik Rios	tblack@example.com	023821793	590.82
6	9	Daniel Davis	richarddoyle@example.net	013647132	1,137.99
7	10	Michelle Benitez	jaredbeck@example.org	075689754	177.38
8	11	Nathan Riley	andre50@example.com	075936938	765.8
9	12	Connie Anderson	richardsmith@example.org	068252567	490.11
10	13	David Day	joseph22@example.net	049149366	234.13
11	14	Morgan Bennett	schultzmichael@example.com	062628551	541.24
12	15	Marco Hawkins	shawn45@example.org	028291318	948.72

- **Count of medication types per department**

*** Query ***

```
SELECT
   d.department_id,
   d.name AS department_name,
   COUNT(dm.medication_id) AS total_medication_types
FROM DepartmentMedication dm
JOIN Department d ON dm.department_id = d.department_id
GROUP BY d.department_id, d.name;
```

Description: This query counts the number of different medication types available in each hospital department by grouping and summarizing data from the DepartmentMedication table.

Importance: This helps department managers monitor inventory diversity, ensuring departments are stocked with a sufficient variety of medications and improving supply chain decisions.

Result:

patient_id	patient_name	email	phone	total_billed
1	Theresa Lambert	jeff02@example.com	076588126	110.14
2	Susan Morrow	bryan21@example.org	034698794	317.40
4	Amanda Wright	arnoldvincent@example.net	057655627	555.85
5	John Turner	hstanley@example.com	021958165	189.72
6	Erik Rios	tblack@example.com	023821793	590.82
9	Daniel Davis	richarddoyle@example.net	013647132	1137.99

- **Count Total Appointments per Doctor**

*** Query ***

```
SELECT
   d.doctor_id,
   CONCAT(d.first_name, ' ', d.last_name) AS doctor_name,
   COUNT(a.appointment_id) AS total_appointments
FROM Doctor d
JOIN Appointment a ON d.doctor_id = a.doctor_id
GROUP BY d.doctor_id;
```

Description: This query calculates the total number of appointments for each doctor by grouping records in the Appointment table based on doctor_id. The result shows each doctor's ID along with how many appointments they have in total.

Importance: This query is useful for evaluating a doctor's workload and understanding appointment distribution across the hospital. It helps administrators manage staff schedules more effectively, identify overbooked doctors, and ensure balanced patient care.

Result:

doctor_id	doctor_name	total_appointments
1	Anne Guzman	1901
2	Melanie Goodman	1992
3	Kenneth Cook	2034
4	Jason Miller	2049
5	Angela West	2006
6	Alison Briggs	1958
7	Theresa Knapp	2039
8	Charlene Johnson	1985
9	Derrick Zuniga	1990

3.4. Function

Function is one of the query types we used to perform reusable calculations in the Hospital Management System. It helps simplify complex logic, reduce repetition, and ensure consistent results. This includes:

- **Calculate Age Function**

*** Query ***

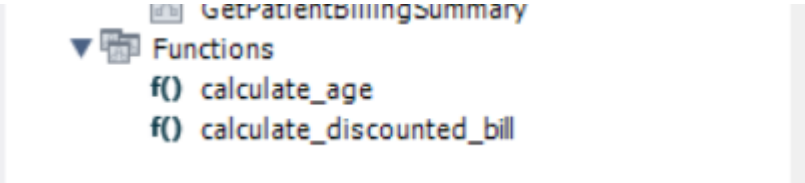
```
DELIMITER //

CREATE FUNCTION calculate_age(dob DATE) RETURNS INT
```

```
DETERMINISTIC
BEGIN
   RETURN TIMESTAMPDIFF(YEAR, dob, CURDATE());
END //

DELIMITER ;
```

Description: This function takes a patient's date of birth as input and returns their current age in years by calculating the difference between the birth date and today's date.

Importance: Age is a critical factor in medical decision-making and patient management. This function allows consistent and accurate age calculation across the system without repeating code, improving data reliability and simplifying queries that depend on age.

Result:



Call function to calculate age of patient and here is result:

patient_id	name	age
1	Theresa Lambert	25
2	Susan Morrow	95
3	Andrew Myers	45
4	Amanda Wright	1
5	John Turner	19
6	Erik Rios	88
7	David Fitzpatrick	52
8	Heidi Johnson	56

- **Calculate Discounted Bill Function**

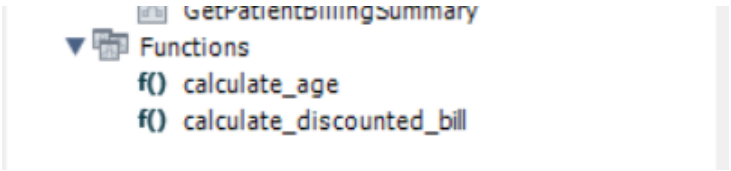
*** Query ***

```
DELIMITER //

CREATE FUNCTION calculate_discounted_bill(original_amount DECIMAL(10,2),
discount_percent INT)
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
   DECLARE discounted_amount DECIMAL(10,2);
   SET discounted_amount = original_amount - (original_amount * discount_percent /
100);
   RETURN discounted_amount;
END //

DELIMITER ;
```

Description: This function calculates the discounted billing amount for a patient based on the original bill and a given discount percentage. It returns the final amount after applying the discount.

Importance: This function supports billing adjustments such as discounts or promotions, ensuring accurate and automated billing calculations. It reduces errors and streamlines billing workflows, improving patient satisfaction and financial management.

Result:



The result when we call function to calculate 10 % discount:

	bill_id	total_amount	discounted_total
▶	1	110.14	99.13
	2	317.40	285.66
	3	205.56	185.00
	4	350.29	315.26
	5	189.72	170.75
	6	126.23	113.61
	7	464.59	418.13

3.5. Triggers

Trigger is one of the SQL features we used in the Hospital Management System to automatically execute predefined actions when certain events like INSERT, UPDATE, or DELETE occur on a table. This helps enforce business logic, maintain consistency, and automate tasks such as logging or updating related records. This includes:

- **Prevent Negative Stock in DepartmentMedication**

*** Query ***

```
DELIMITER //

CREATE TRIGGER prevent_negative_stock
BEFORE INSERT ON DepartmentMedication
FOR EACH ROW
BEGIN
   IF NEW.stock_quantity < 0 THEN
      SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Stock quantity cannot be negative';
   END IF;
END //

DELIMITER ;
```

Description: This trigger is activated before a new record is inserted into the DepartmentMedication table. It checks whether the stock_quantity is negative, and if so, it blocks the operation by raising an error with a custom message.

Importance: This trigger is essential for **data integrity**. It ensures that medication stock levels remain logical and valid, preventing human error or system faults from

introducing invalid negative quantities. This protects inventory accuracy and avoids disruptions in medical services due to incorrect stock data.

- **Auto-Update Medication Stock After Prescription**

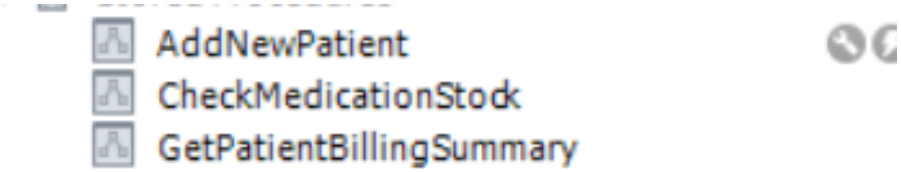
*** Query ***

```
DELIMITER //

CREATE TRIGGER update_stock_after_prescription
AFTER INSERT ON PrescriptionMedication
FOR EACH ROW
BEGIN
   DECLARE dep_id INT;

   SELECT d.dep_id
   INTO dep_id
   FROM Prescription p
   JOIN Medical_Record mr ON p.medical_record_id = mr.record_id
   JOIN Appointment a ON mr.appointment_id = a.appointment_id
   JOIN Doctor d ON a.doctor_id = d.doctor_id
   WHERE p.prescription_id = NEW.prescription_id;

   UPDATE DepartmentMedication
   SET stock_quantity = stock_quantity - 1
   WHERE medication_id = NEW.medication_id
     AND department_id = dep_id;
END;
//

DELIMITER ;
```

Description: This trigger is executed after a new record is inserted into the PrescriptionMedication table. It identifies the department associated with the prescription and then automatically reduces the stock quantity of the prescribed medication by 1 in the corresponding department.

Importance: This trigger **automates inventory management**. It ensures that every time a medication is prescribed, the available stock is immediately updated, which supports real-time inventory tracking and reduces manual work. It helps maintain accurate stock levels and ensures medications are restocked as needed.

Result: This is the result after we run trigger script in the database workbench



3.6. Procedures

Stored Procedure is one of the SQL features we used in the Hospital Management System to encapsulate and reuse a sequence of SQL statements under a single callable name. Stored procedures are useful for performing common operations like retrieving data, calculating totals, or updating multiple tables. They help reduce redundancy, improve performance, and ensure consistent business logic. This includes:

- **Get doctor's appointments**

*** Query ***

```
DELIMITER //
CREATE PROCEDURE get_doctor_appointments(IN doc_id INT)
BEGIN
   SELECT * FROM Appointment
   WHERE doctor_id = doc_id
   ORDER BY appointment_datetime DESC;
END;
//
DELIMITER ;
```

Description: This procedure retrieves all appointments associated with a specific doctor based on the given doctor ID (doc_id). It sorts the results in descending order by appointment date and time, showing the most recent appointments first.
Importance: This procedure is helpful for doctors and administrative staff to quickly view a doctor's schedule, monitor past and upcoming appointments, and manage time efficiently.

- **Add medication and Stock**

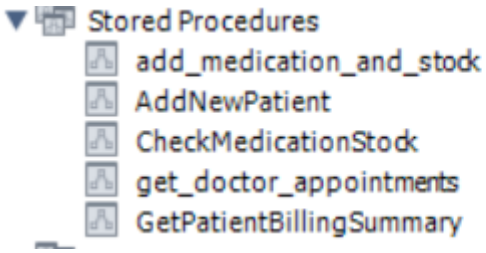
*** Query ***

```
DELIMITER //
CREATE PROCEDURE add_medication_and_stock(
   IN med_name VARCHAR(100),
   IN dep_id INT,
   IN stock INT,
   IN price DECIMAL(10,2),
   IN expiry DATE
)
BEGIN
   DECLARE med_id INT;

   INSERT INTO Medication(name, unit_price, expiration_date)
   VALUES (med_name, price, expiry);

   SET med_id = LAST_INSERT_ID();

   INSERT INTO DepartmentMedication(department_id, medication_id, stock_quantity)
   VALUES (dep_id, med_id, stock);
END;
//
DELIMITER ;
```

Description: This procedure adds a new medication to the system and initializes its stock for a specific department. It first inserts the medication details (name, price, expiration date) into the Medication table, retrieves the auto-generated medication_id, and then inserts stock information into the DepartmentMedication table.

Importance: This procedure simplifies the process of managing pharmacy inventory by combining two related actions (adding a medication and setting its stock) into one step. It reduces human error and ensures proper linkage between medications and departments.

Result: This is the result of procedure script after we run in workbench



IV. User and Role Schema

To support Role-Based Access Control (RBAC) in our Hospital Management System, we designed and implemented a user access model consisting of three core components: Users, Roles, and Privileges. Rather than creating these tables manually, our team used Sequelize, a Node.js ORM, to define the schema and relationships in index.js. Sequelize then automatically generated the necessary tables and managed their associations.

We also designed an Entity Relationship (ER) diagram to visualize how users are linked to roles and how roles are granted specific privileges. This helped ensure a clear structure and maintainable access control logic.

- **Role:** The entity defines the different access levels or responsibilities assigned to users within the Hospital Management System. Each role groups a set of privileges that reflect a specific job function or level of authority (e.g., Admin, Developer, Data Analyst).
- **Privilege:** The entity defines specific actions that can be performed on resources within the system. These are assigned to roles to control what users with that role are allowed to do (e.g., read patient data, update billing records).
- **User:** The entity stores account-level information for individuals who interact with the system. Each user is associated with one or more roles that define their permissions.

The ED diagram is shown in Figure 06

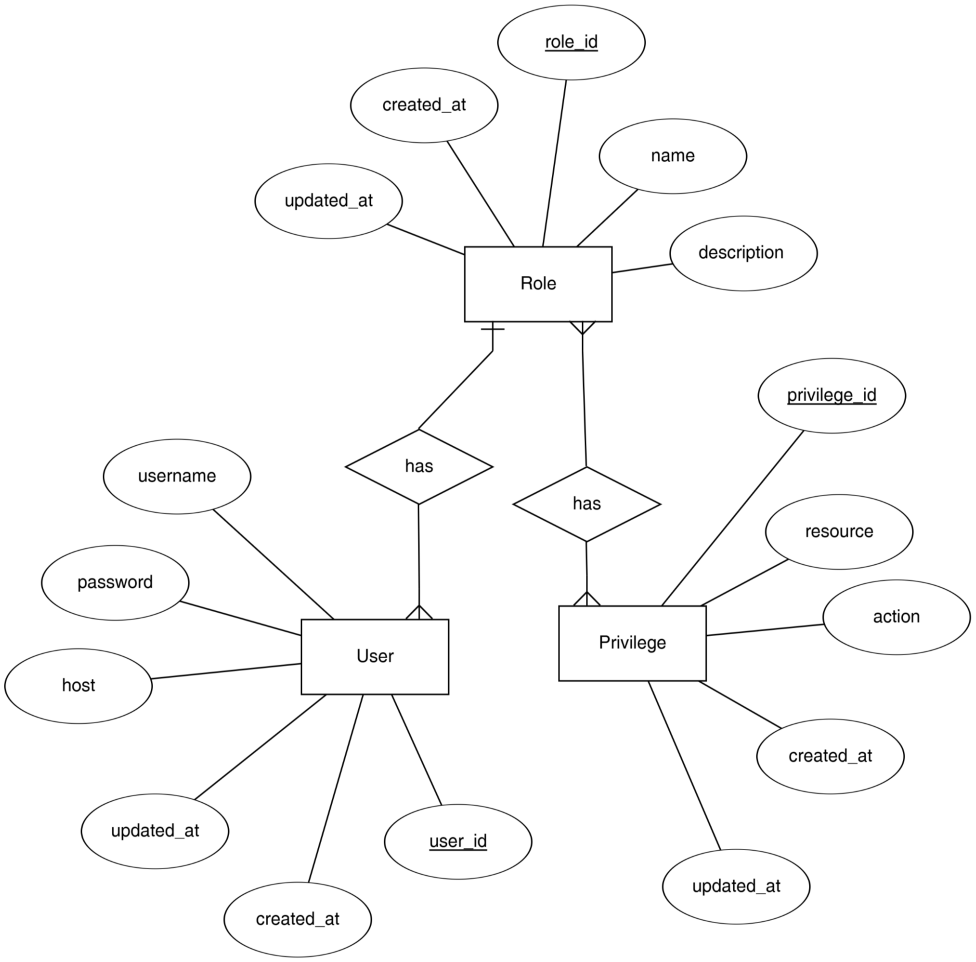


Figure 06

DDL is not used to create the **User**, **Role**, and **Privilege** tables. Instead, Sequelize is used to explicitly generate these tables from the server side of the web-based application.

The relationships are also implemented using Sequelize. Since there is a **many-to-many** relationship between **Role** and **Privilege**, a junction table is automatically created. As a result, the relational model includes four tables.

Figure 07 below is the relational model of User and Role used in the web-based system

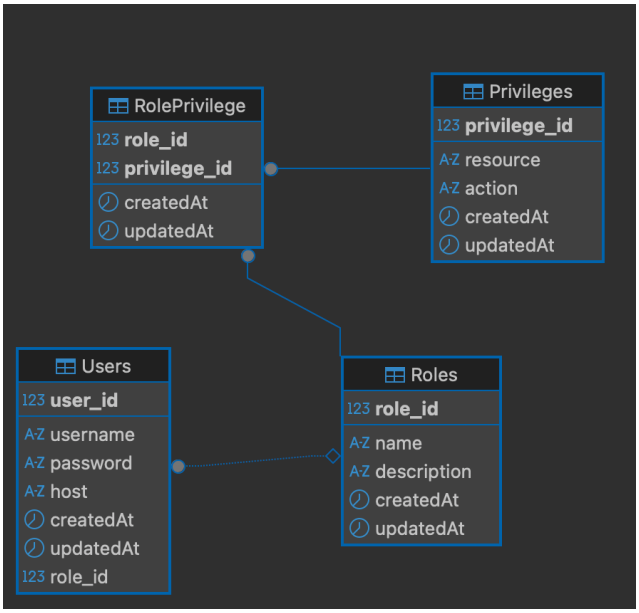


Figure 07

To make the system manageable for administrators, we developed a **web-based** admin interface that offers a user-friendly and secure environment for managing access control. Through this interface, administrators can:

- **Create and manage user accounts** with ease, avoiding the need to directly manipulate the database.
- **Assign roles to specific user** like Admin, Developer, or Data Analyst based on real responsibilities.
- **Assign and adjust privileges** for each role, determining exactly what users can view, insert, update, or delete, which ensures only authorized users can interact with sensitive data.

Roles in Hospital Management System Database

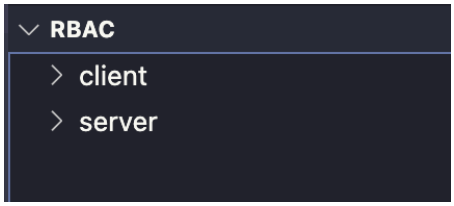
Roles	Description	Privileges
Admin	A general system administrator for the database application.	**ALL PRIVILEGES**
Data Analyst	A person who analyzes data to create reports, charts, or insights.	**SELECT**

Developer	Someone who writes code, builds features, and works on the application that uses the database.	**CREATE, READ, UPDATE, DELETE**
DBA (database admin)	The highest-level authority over the database system. They manage security, performance, backups, users.	**ALL PRIVILEGES**
Auditor	A person (internal or external) who reviews the system for compliance, security, or performance purposes.	**SELECT**
QA Support	Quality Assurance testers and customer support teams.	**SELECT, INSERT**

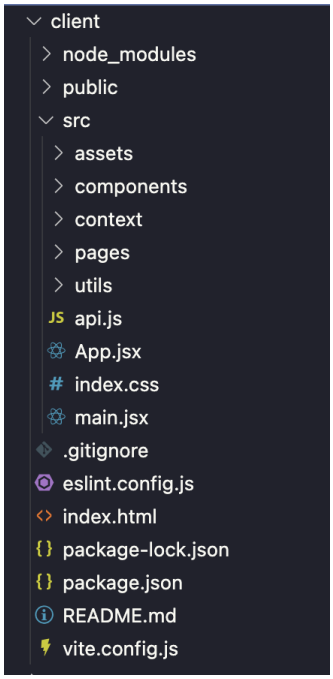
Web-based Implementation

The **frontend** of the web-based system is developed using **React.js** and styled with **Tailwind CSS**, while the **backend** is built with **Express.js**. Communication between the frontend and backend is handled through **RESTful APIs**.

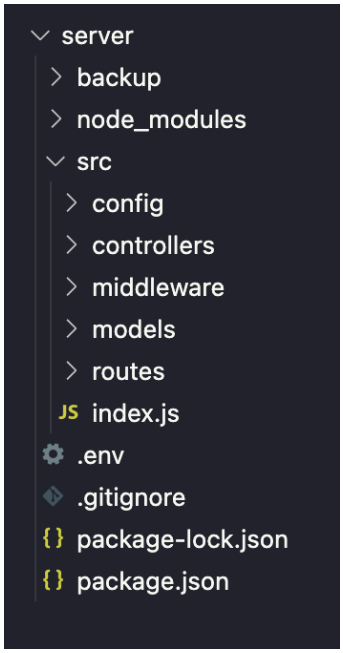
Below is the folder structure of the entire web-based application:
The project is organized into two main directories:



- The **client** folder manages the frontend interface, including UI components, data rendering, and user interactions using React.js and Tailwind CSS.



- The **server** folder is responsible for handling the backend logic, including API routes, database interactions, and authentication.



*** **NOTE**: The backup folder is used to stored when the backup button is clicked on the web based.

In this web-based application, only administrators are permitted to log in. If a non-admin user attempts to access the system, an “Access Denied: Admins Only” message is displayed. Once logged in, the admin is directed to the dashboard, which provides a summary of the total number of users, roles, and privileges in the database.

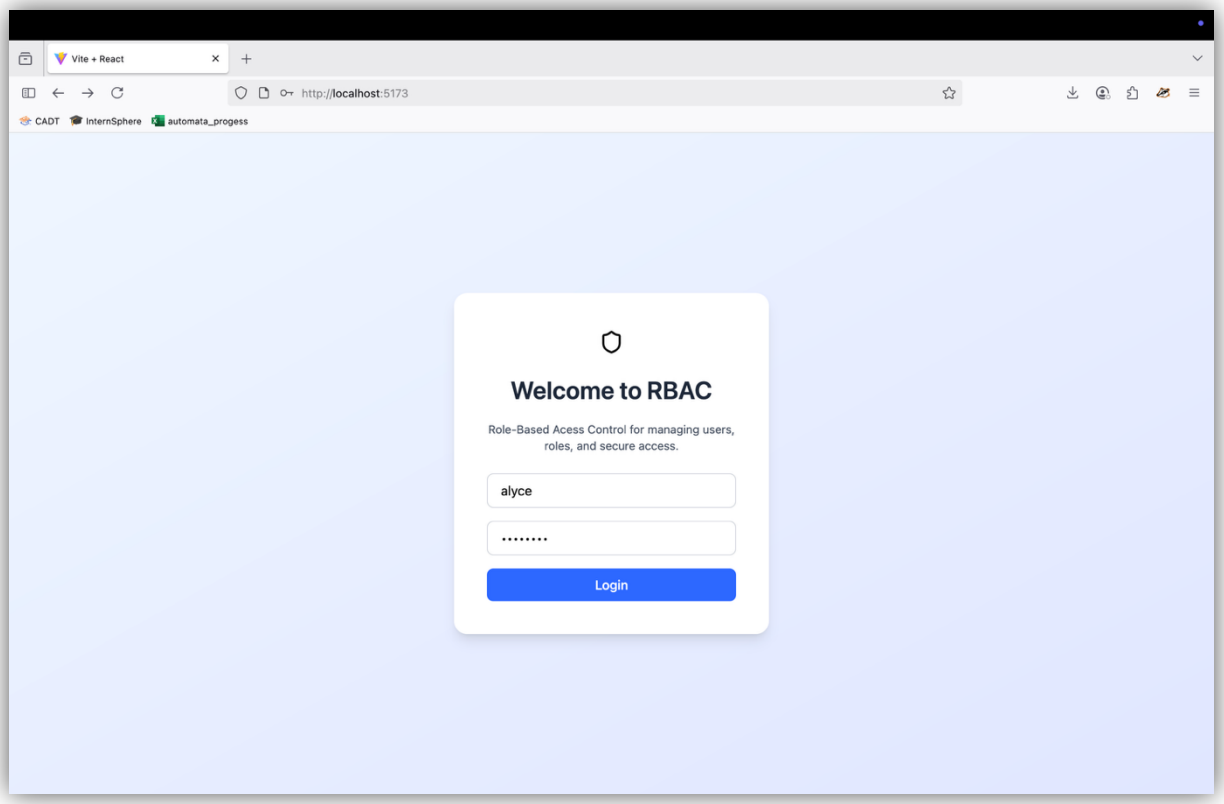
Additionally, the admin can view a detailed list of all users along with their associated information, each accompanied by Edit and Delete buttons for management purposes. The summary cards on the dashboard are interactive. When clicked, they allow the admin to view more detailed information about the corresponding roles and privileges, providing greater insight and control over access management. Moreover, the dashboard includes buttons for backup and recovery operations. These buttons trigger backend functions that allow the admin to back up the database or restore it from a previous state, ensuring data safety and system reliability.
When a new user is created or assigned a specific role, the backend automatically triggers functions that perform several tasks:

- Insert data into the User, Role, Privilege, and junction tables to reflect the relationships.
- Execute SQL queries to create a corresponding MySQL user and grant the appropriate role-based privileges.

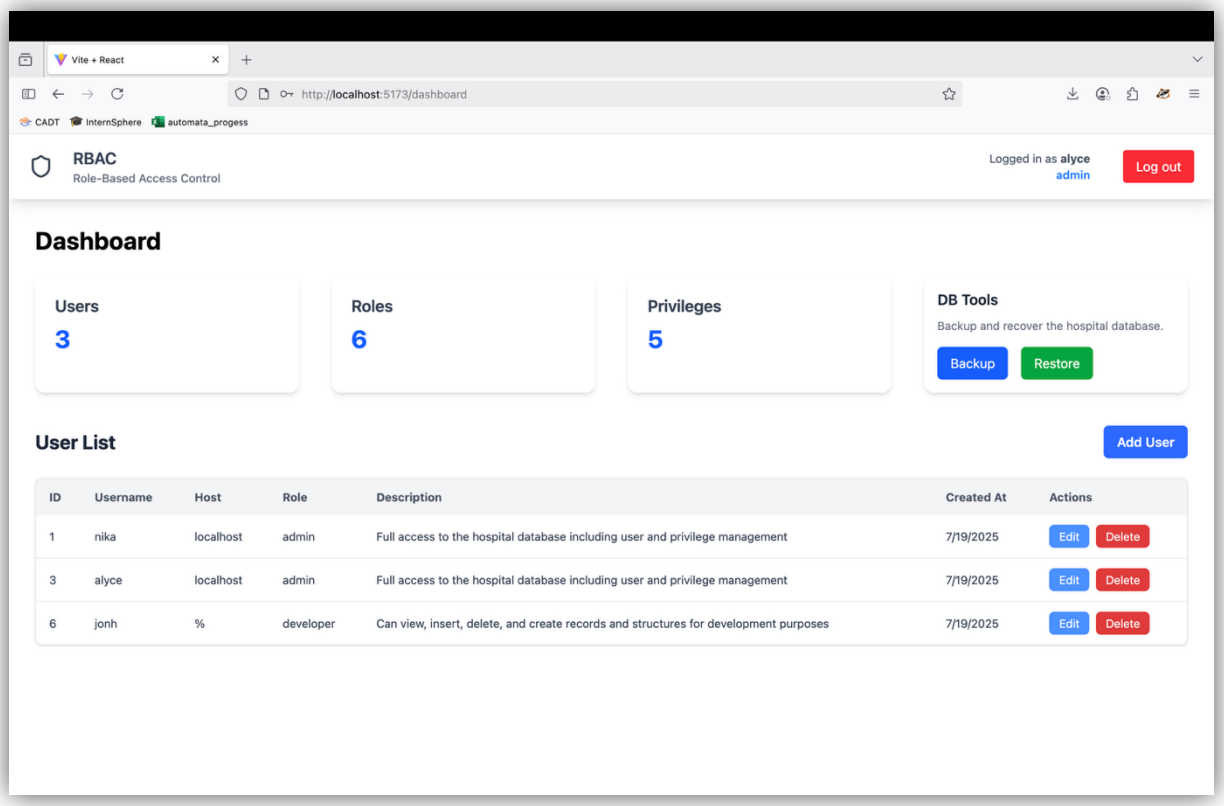
As a result, the system not only maintains logical data within the application tables but also ensures that real MySQL users with proper access rights are created and managed accordingly.

Here is the Role-Based Access Control website

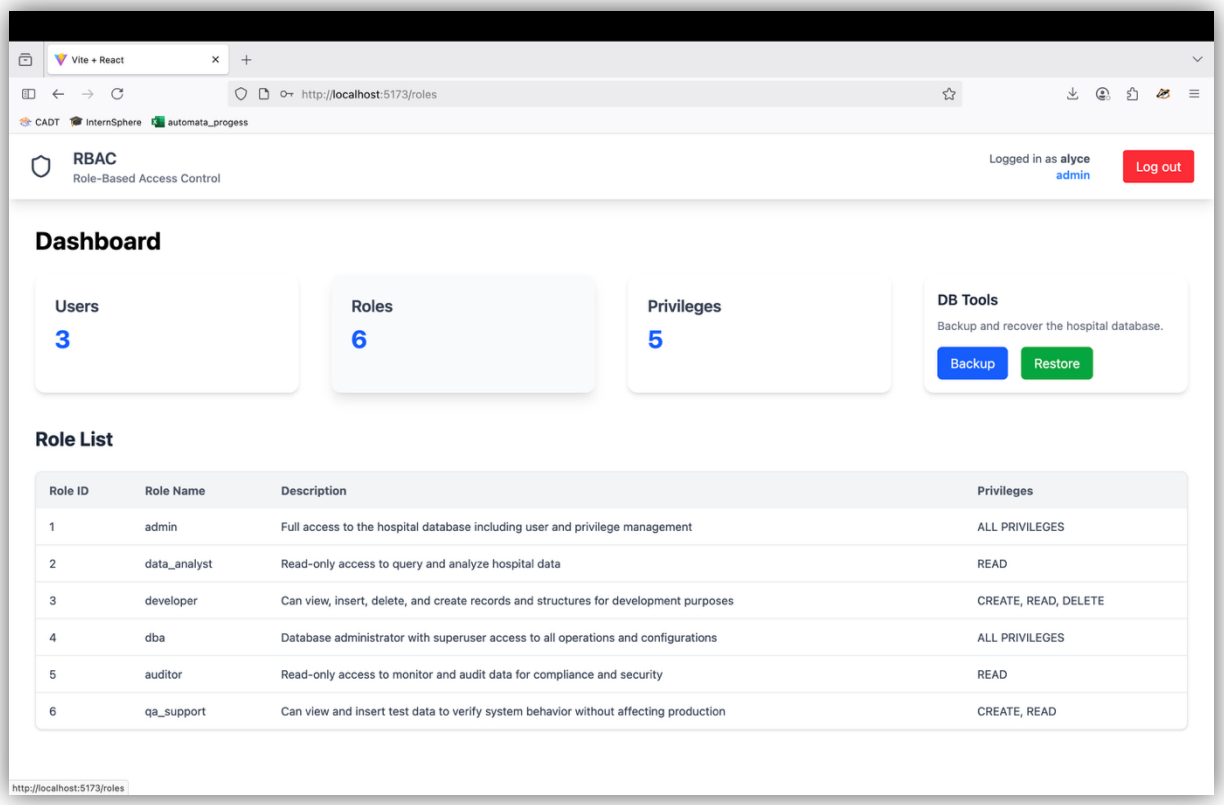
- **Login page**


Vite + React
http://localhost:5173
CADT
InternSphere
automata_progess
Welcome to RBAC
Role-Based Acess Control for managing users, roles, and secure access.
alyce
Login

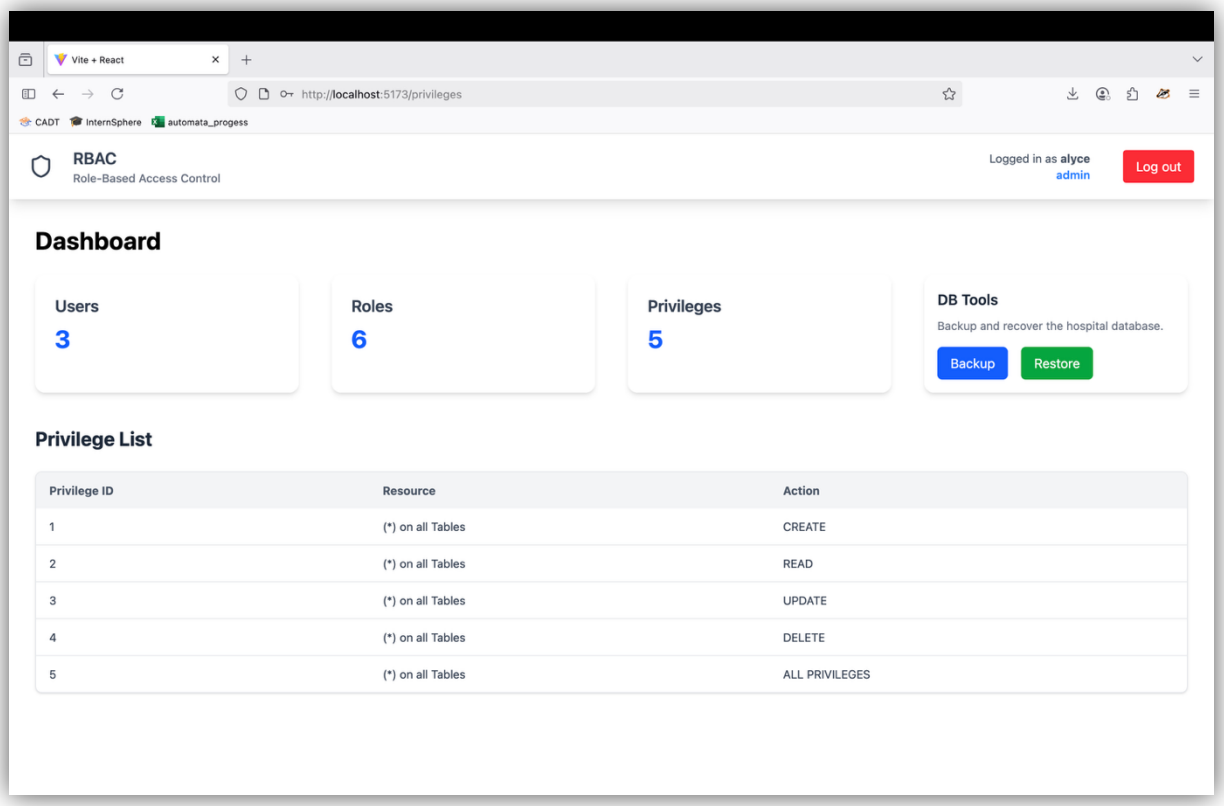
- **Dashboard**


Vite + React
http://localhost:5173/dashboard
CADT
InternSphere
automata_progess
RBAC
Role-Based Access Control
Logged in as alyce
admin
Log out
Dashboard
Users
3
Roles
6
Privileges
5
DB Tools
Backup and recover the hospital database.
Backup
Restore
User List
Add User
ID
Username
Host
Role
Description
Created At
Actions
1
nika
localhost
admin
Full access to the hospital database including user and privilege management
7/19/2025
Edit
Delete
3
alyce
localhost
admin
Full access to the hospital database including user and privilege management
7/19/2025
Edit
Delete
6
jonh
%
developer
Can view, insert, delete, and create records and structures for development purposes
7/19/2025
Edit
Delete

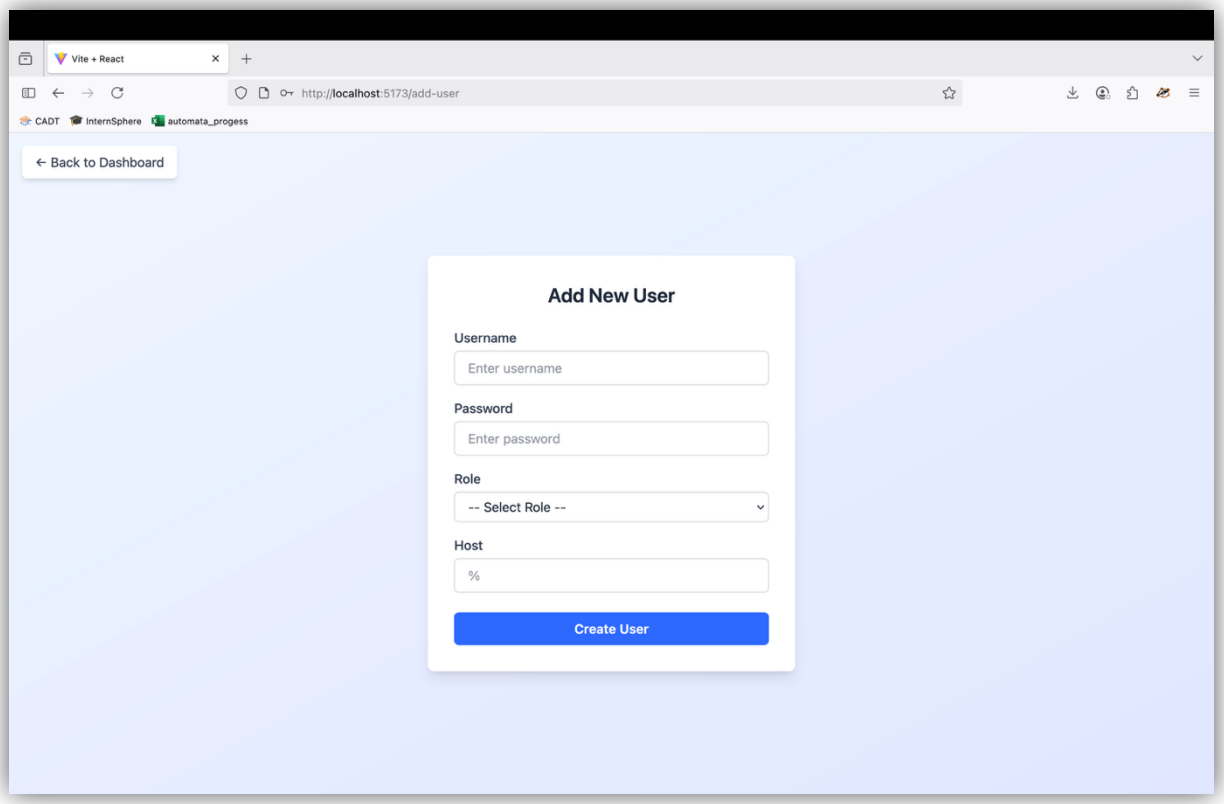
- **All roles in hospital database**



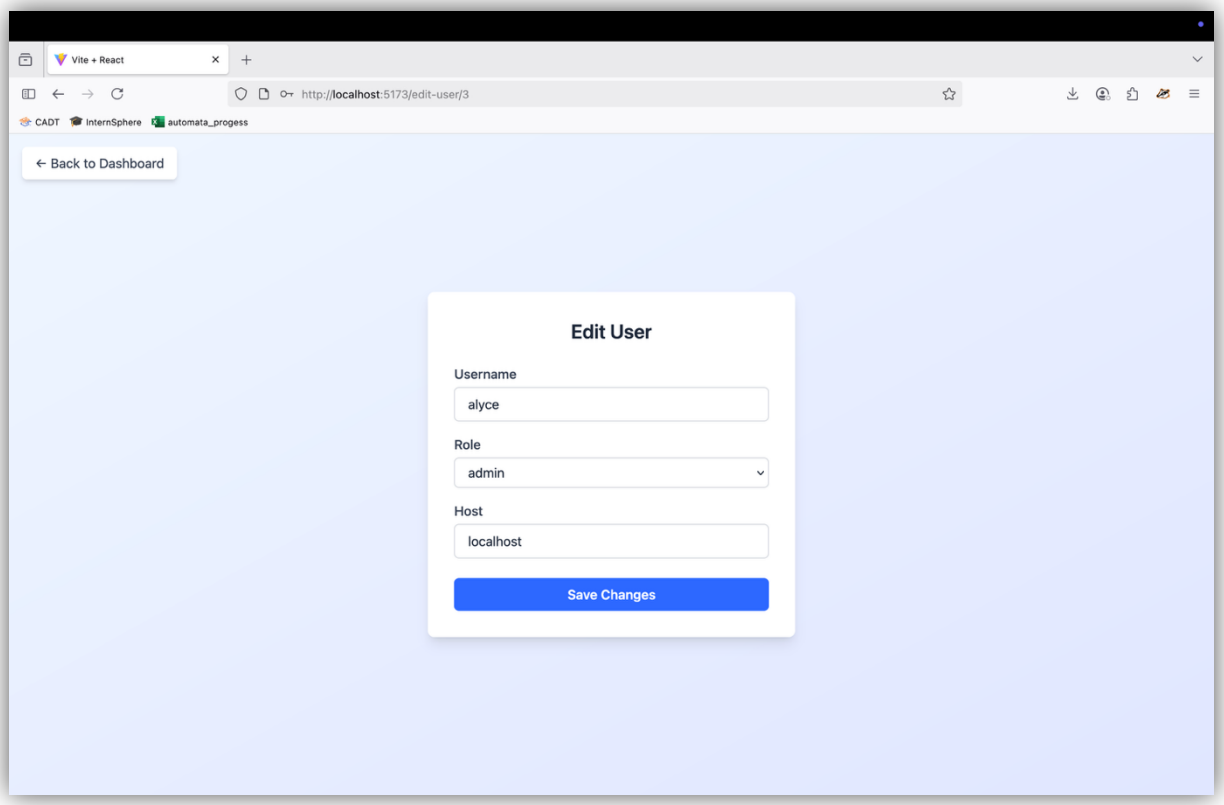
- **All Privileges in hospital database**



- **Add new user**



- **Edit a specific user**



V. Backup and Recovery Plan

To ensure data integrity and protect against data loss in the Hospital Management System, our team implemented a backup and recovery strategy. This is especially important in a healthcare environment where sensitive data such as patient records, appointments, billing, and medical history must be securely preserved and readily available.

The reason we implemented this feature is because in real-world hospital operations, unexpected events such as system crashes, accidental deletions, or hardware failures can occur. Without proper data protection mechanisms, these incidents could lead to serious consequences, including the loss of critical medical information.

To further enhance data protection and system reliability, we integrated backup and recovery functionality into the web-based admin interface. This allows administrators to:

- Manually trigger backups of the database through the interface, without needing to use command-line tools.
- Download backup files for secure offline storage.
- Restore data from backup files in the event of data corruption or accidental deletion.

By integrating backup and recovery into the system, we enabled non-technical hospital staff to easily perform critical data protection tasks, reducing downtime and safeguarding against permanent data loss.

Backup and Recovery Implementation

The system supports web-based backup and recovery of the MySQL database. This functionality is handled on the backend using Node.js's child_process.exec() to run native MySQL commands.

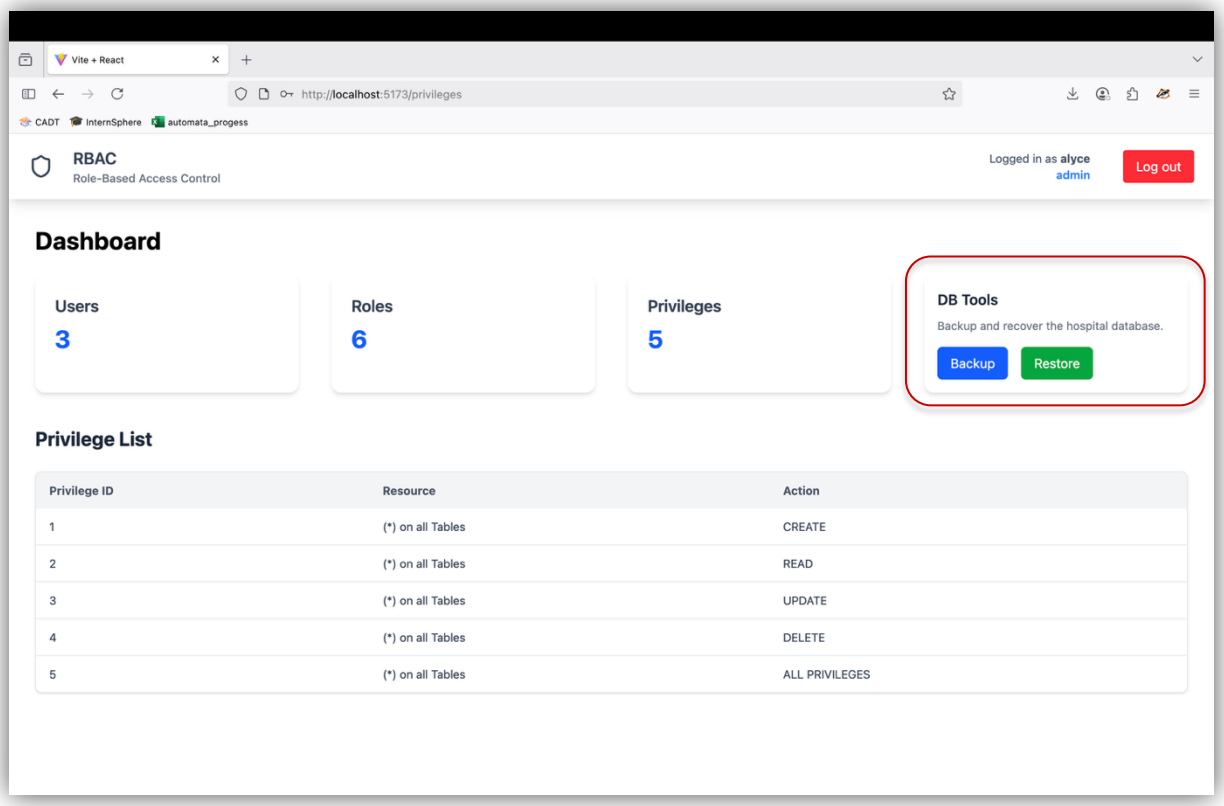
- **Backup Process:**

When the admin clicks the Backup button, the backend executes the mysqldump command to export the current database to a .sql file. This file is stored in the backup folder within the server directory. The backup file is named using the database name followed by _backup.sql.

- **Recovery Process:**

For recovery, the admin must first manually create a new database in MySQL Workbench. After that, clicking the Restore button sends a request to the backend, which executes a mysql command to import the backup file into the newly created database.

All commands are executed programmatically through JavaScript using the exec method, eliminating the need for manual command-line interaction. This design provides a user-friendly way to manage critical database operations securely and efficiently from the browser.



VI. Performance Evaluation

Our project is evaluated with large amounts of data, ensuring efficient query performance became a critical concern. To improve responsiveness and scalability, especially during frequent read operations, we implemented indexes on selected columns such as patient_id, doctor_id, and appointment_datetime.

We used this approach because indexes significantly reduce query time by allowing the database engine to quickly locate rows instead of scanning the entire table. This is especially beneficial when working with millions of records, as in our case.

To demonstrate the effectiveness of indexing, we compared the performance of queries with and without indexes. Below is a comparison showing the execution time and efficiency for a sample query retrieving appointment data:

Table of comparison

Query	Without indexing	With indexing
SELECT b.patient_id, CONCAT(p.first_name, ' ', p.last_name) AS patient_name, p.email, p.phone, SUM(b.total_amount) AS total_billed FROM Billing b JOIN Patient p ON b.patient_id = p.patient_id GROUP BY b.patient_id, p.first_name, p.last_name, p.email, p.phone;	✓ 3m 6.2s Time taken: 186.0992615222931 seconds - 2593896 rows **3m 6.2s**	CREATE INDEX idx_billing_patient_total ON Billing(patient_id, total_amount); ✓ 2m 33.1s Time taken: 153.0302860736847 seconds - 2593896 rows **2m 33.1s**
SELECT a.appointment_id, CONCAT(p.first_name, ' ', p.last_name) AS patient_name, CONCAT(d.first_name, ' ', d.last_name) AS doctor_name, a.appointment_datetime, a.status FROM Appointment a JOIN Patient p ON a.patient_id = p.patient_id JOIN Doctor d ON a.doctor_id = d.doctor_id;	✓ 18m 9.6s Time taken: 1089.3569416999817 seconds - 6000000 rows **18m 9.6s**	CREATE INDEX idx_appointment_patient_doctor ON Appointment(patient_id, doctor_id); ✓ 8m 55.2s Time taken: 535.0610563755035 seconds - 6000000 rows **8m 55.2s**
SELECT bill_id, total_amount, calculate_discounted_bill(total_amount, 10) AS discounted_total FROM Billing;	✓ 4m 42.7s Time taken: 282.5639090538025 seconds - 6000000 rows **4m 42.7s**	CREATE INDEX idx_billing_total_amount ON Billing(total_amount); ✓ 3m 37.5s Time taken: 217.52963399887085 seconds - 6000000 rows **3m 37.5s**

This evaluation proved that indexing is a simple yet powerful optimization strategy to ensure our Hospital Management System performs well under realistic, large-scale workloads.

VII. Challenges

Throughout the development of the Hospital Management System, our team encountered several challenges, including:

- **Data Generation:** Populating millions of realistic records using Faker while maintaining referential integrity and avoiding data duplication.
- **Performance Optimization:** Ensuring fast query response times under a large dataset, which required the implementation of indexing and query analysis.
- **Access Control:** Designing a secure and flexible Role-Based Access Control (RBAC) system that clearly defines what each user role can access or modify.
- **Backup and Recovery Integration:** Embedding backup and restore features into the web interface in a way that is simple enough for non-technical users to operate.
- **Web Interface Usability:** Creating an intuitive admin panel for managing users, roles, privileges, and backups without requiring direct database access.

But despite these challenges, we always tried our best collaborating as a team, learning from each obstacle, and continuously improving our design and implementation to build a reliable and user-friendly system.

VIII. Conclusion

Developing the Hospital Management System was a comprehensive and rewarding experience for our team. From designing the database structure to implementing advanced features like Role-Based Access Control, backup and recovery, and performance optimization, we tackled real-world challenges and applied practical solutions.

By using tools like Sequelize, Faker, and indexing, we were able to simulate large-scale hospital operations, ensure data integrity, and improve system efficiency. The web-based admin interface further enhanced usability, making the system manageable even for non-technical users.

Overall, this project helped us deepen our understanding of database design, system security, and full-stack development while delivering a functional and scalable solution for hospital data management.