Hospital Appointment & Billing System

Database Management Systems Project

INSTRUCTOR:
Miss Abeera Tariq
Class No. 99393

TEAM MEMBERS:

1. Mohammad Faraz Ansari : 29201
2. Ashhal Aamir : 29114

December 8, 2025

Contents

List of Figures

1 Business Scenario & Application Analysis

Hospitals today manage thousands of patients, appointments, and billing transactions daily. Many small and medium-sized hospitals still rely on manual or semi-digital processes, which often lead to overlapping appointments, lost records, and delays in billing. Our project, the **Hospital Appointment & Billing System**, aims to automate and streamline these processes through a centralized database and interactive interface.

The system allows patients to register, book appointments with doctors, and receive electronic bills once the appointment is completed. Doctors can view their daily schedules, record prescriptions, and mark appointments as completed, while administrators can manage departments, staff, and payment records. This approach reduces human error, improves scheduling efficiency, and enhances patient satisfaction.

Our system design is informed by targeted research conducted to identify and address genuine operational challenges. We began by analyzing existing online systems and forum discussions to map common features and gaps. Furthermore, we held exploratory discussions with several family doctors to understand practitioner-level pain points. To gain crucial, frontline administrative insight, we conducted a formal, structured interview with a medical trainee, whose daily role offers a direct view of scheduling and patient flow.

The research yielded consistent findings that directly shape our project's features. The interview specifically highlighted that manual appointment books cause frequent double bookings and scheduling errors, creating daily operational friction. A significant portion of staff time is consumed by the tedious manual entry of lengthy patient histories. Most tellingly, there was a strong expressed need for digitized solutions—specifically online appointment booking to give patients direct access and electronic prescription/test modules to eliminate paper-based delays. These insights validate that our system's core functionalities—real-time digital scheduling, integrated patient records, and automated digital workflows—are precisely targeted to solve the most pressing inefficiencies confirmed by our research.

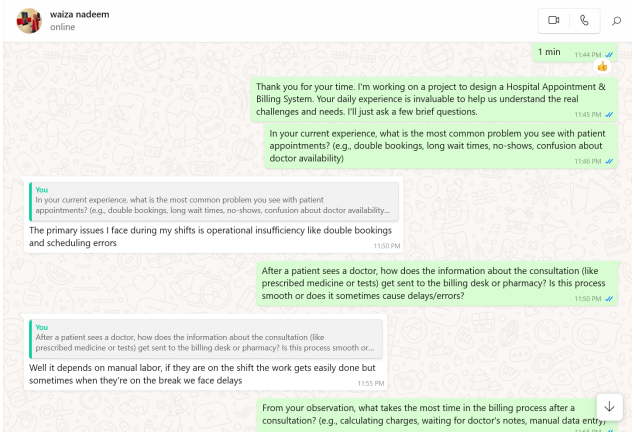


Figure 1: Research Discussion Screenshot 1

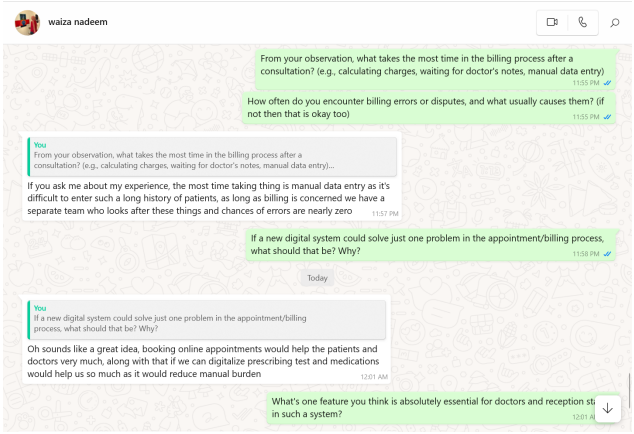


Figure 2: Research Discussion Screenshot 2

1.1 Applications Reviewed

Marham (`https://www.marham.pk/`)

- Offers online doctor booking, video consultations, and hospital listings.
- The website is user-friendly and allows filtering by specialization, city, and gender.

- However, many users mention in reviews that appointment confirmation is sometimes delayed, and customer service response can be slow.
- Good for: Large hospitals or specialized clinics with digital staff.
- Limitation: May be too feature-heavy for small hospitals needing a simple internal appointment + billing system.

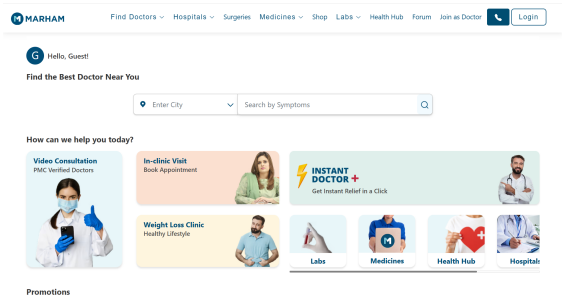


Figure 3: Marham Platform Interface

Healthwire (`https://www.healthwire.pk/`)

- Provides an integrated hospital management system with modules for billing, patient history, and staff records.
- Reviews highlight that it's powerful but complex, requiring training to use efficiently.
- The UI looks dense and technical, which can overwhelm smaller clinics.
- Good for: Multi-branch hospitals needing complete management.
- Limitation: Not ideal for local clinics or small hospitals due to its steep learning curve.

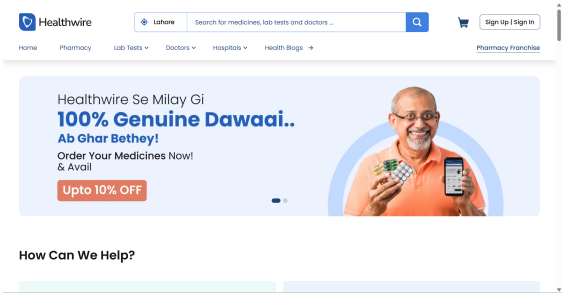


Figure 4: Healthwire Platform Interface

Oladoc (`https://oladoc.com/`)

- Focused primarily on doctor-patient appointment booking.
- Interface is clean and quick, with SMS reminders and patient reviews.
- However, there's no built-in billing or payment tracking, and limited backend management for staff or departments.

- Good for: Standalone doctors or teleconsultation.
- Limitation: Lacks billing and record management modules.

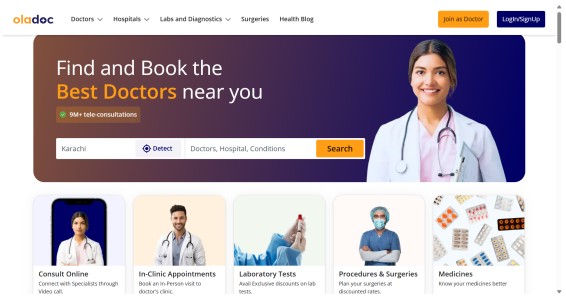


Figure 5: Oladoc Platform Interface

1.2 Summary

Based on the analysis, existing solutions like Marham, Healthwire, and Oladoc each specialize in certain aspects of healthcare management — either online bookings or full scale hospital systems. However, none provide a simple, integrated platform for both appointment scheduling and billing designed specifically for small to medium-sized hospitals.

Our proposed Hospital Appointment & Billing System fills this gap by offering an efficient yet lightweight solution with appointment handling, prescription management, and automated billing in one system.

2 Business Rules & Use Cases

2.1 Business Rules

1. **Doctor Availability Rule:** A doctor cannot have two appointments scheduled at the same date and time.
2. **Appointment Completion Rule:** When an appointment status is marked as "Completed," a bill is automatically generated for that appointment.
3. **Prescription Rule:** A prescription can only be created for appointments that are completed.
4. **Patient Registration Rule:** A patient must be registered in the system before booking any appointment.
5. **Department Association Rule:** Each doctor and staff member must be assigned to exactly one department.
6. **Payment Rule:** A bill must have a valid payment status — only "Paid" or "Unpaid" are acceptable values.
7. **Cancellation Rule:** If a patient cancels an appointment, the slot becomes available for other bookings.
8. **Doctor Fee Consistency Rule:** The consultation fee for each doctor is stored in the Doctor table and automatically fetched when generating a bill.

9. **Data Integrity Rule:** Deleting a patient record should also remove all related appointments, prescriptions, and bills to maintain database consistency.
10. **Signup/Registration Rule:** Only patients can sign up using the signup page, doctors' accounts are created by admins who then add those doctors to their respective departments.
11. **Booking Rule:** A patient may only book an appointment in the upcoming week, no appointments beyond 7 days.
12. **Booking Limit:** A patient can only have 2 scheduled appointment at any given time.

2.2 Use Cases

1. **Register Patient:** A new patient fills out a form with name, contact, age, and gender. The system saves the record in the Patient table.
2. **Book Appointment:** A patient selects a doctor, date, and time slot. The system checks for conflicts, then records the appointment if available.
3. **Complete Appointment:** The doctor marks the appointment as "Completed." A trigger automatically generates a bill and enables prescription creation.
4. **Generate Bill:** When an appointment is completed, the system creates a bill using the doctor's fee and sets payment_status to "Unpaid."
5. **Add Prescription:** The doctor enters medicines, dosage, and notes for the completed appointment. The record is stored in the Prescription table.
6. **Cancel Appointment:** A patient cancels an existing appointment, changing its status to "Cancelled" and freeing that time slot.
7. **View Appointments:** A patient or doctor can view all upcoming and completed appointments from their dashboard.
8. **Pay Bill:** The admin or receptionist updates the bill's payment status to "Paid" once the transaction is done.
9. **Manage Staff:** The admin adds, updates, or removes staff members within departments.
10. **View and edit department statistics:** The admin can view total departments, add departments or delete departments.
11. **View and edit doctor statistics:** The admin can view total doctors, add doctors and change their information.

3 Entities, Attributes, and Relationships

3.1 Overview

Our Hospital Appointment & Billing System consists of eight main entities that represent the hospital's core operations. These are users, patients, doctors, departments, appointments, bills, prescriptions and prescription_medicines. Each entity contains attributes relevant to its role in the system. The relationships among these entities define how hospital processes such as appointment booking, billing, and prescription generation are managed.

3.2 Entities and Attributes

Table 1: Entities, Attributes and Key Types

Entity	Attributes	Key Type / Notes
Users FK: -	user_id, email, password, user_type, is_active, created_at	PK: user_id
Departments FK: head_doctor_id → doctors.doctor_id	department_id, name, floor, head_doctor_id, max_capacity, created_at	PK: department_id
Doctors FK: user_id → users.user_id department_id → departments.department_id	doctor_id, user_id, department_id, name, specialization, license_number, phone, experience_years, consultation_fee, status, created_at	PK: doctor_id
Patients FK: user_id → users.user_id	patient_id, user_id, name, date_of_birth, gender, blood_type, phone, address, emergency_contact, medical_history, created_at	PK: patient_id
Appointments FK: patient_id → patients.patient_id doctor_id → doctors.doctor_id	appointment_id, patient_id, doctor_id, appointment_date, appointment_time, reason, status, created_at	PK: appointment_id
Bills FK: appointment_id → appointments.appointment_id	bill_id, appointment_id, amount, payment_status, generated_date, paid_date	PK: bill_id
Prescriptions FK: appointment_id → appointments.appointment_id	prescription_id, appointment_id, diagnosis, notes, created_at	PK: prescription_id
Prescription_Medicines FK: prescription_id → prescriptions.prescription_id	medicine_id, prescription_id, medicine_name, dosage, duration, instructions	PK: medicine_id

3.3 Relationships and Multiplicity Constraints

Table 2: Relationships and Multiplicity Constraints

Relationship	**Entities Involved**	**Type (Crow's Foot)**	**Description**
User-Doctor	User (1) → Doctor (1)	1:1	Each user account can be linked to exactly one doctor record.
User-Patient	User (1) → Patient (1)	1:1	Each user account can be linked to exactly one patient record.
Department-Doctor	Department (1) → Doctor (Many)	1:M	Each department can have multiple doctors, but each doctor belongs to one department.
Doctor-Department (Head)	Doctor (1) → Department (1)	1:1	A department is managed by one head doctor, and a doctor can manage at most one department.
Doctor-Appointment	Doctor (1) → Appointment (Many)	1:M	A doctor can have many appointments, but each appointment is assigned to one doctor.
Patient-Appointment	Patient (1) → Appointment (Many)	1:M	Each patient can book multiple appointments, but each appointment belongs to one patient.
Appointment-Bill	Appointment (1) → Bill (1)	1:1	Each appointment generates exactly one bill.
Appointment-Prescription	Appointment (1) → Prescription (1)	1:1	Each appointment results in exactly one prescription.
Prescription-Medicine	Prescription (1) → Medicine (Many)	1:M	Each prescription can contain multiple medicines, but each medicine record belongs to one prescription.

3.4 ER Diagram

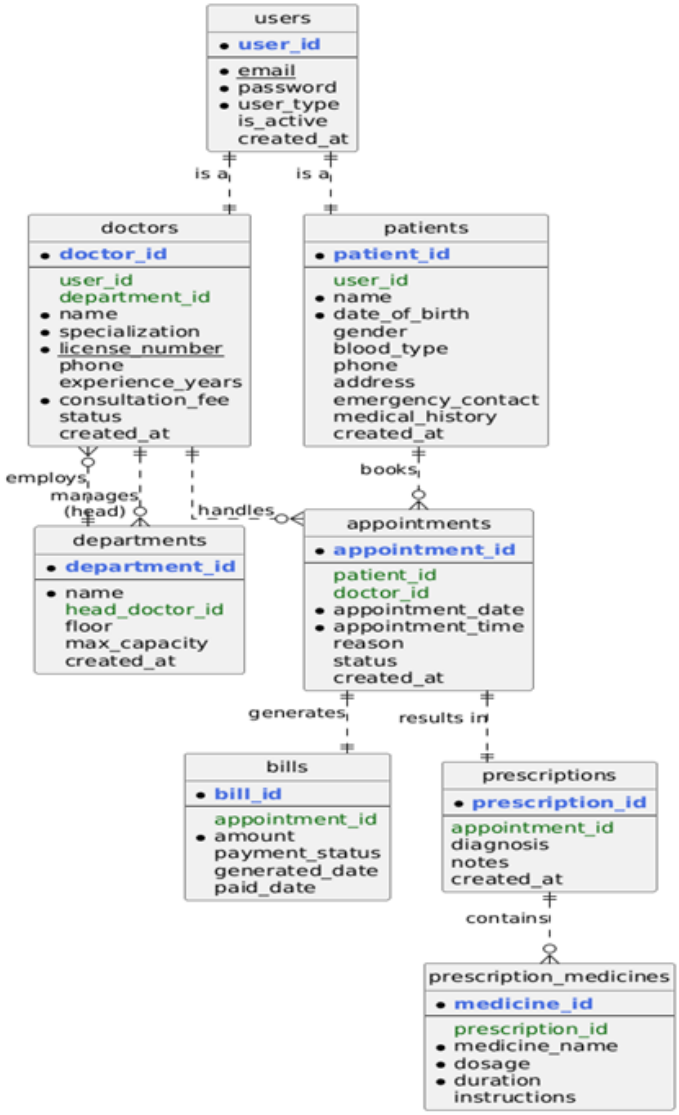


Figure 6: Crow's Foot ER Diagram

4 Relational Schema

4.1 Overview

The relational schema of the Hospital Appointment & Billing System defines how data is organized across multiple related tables in the database. It was designed based on the conceptual ERD and then normalized up to Third Normal Form (3NF) to eliminate redundancy and ensure data integrity.

The schema includes seven core entities — Departments, Doctors, Users, Patients, Appointments, Prescriptions, Prescription_Medicines and Bills — each representing a specific real-world object or process within the hospital system.

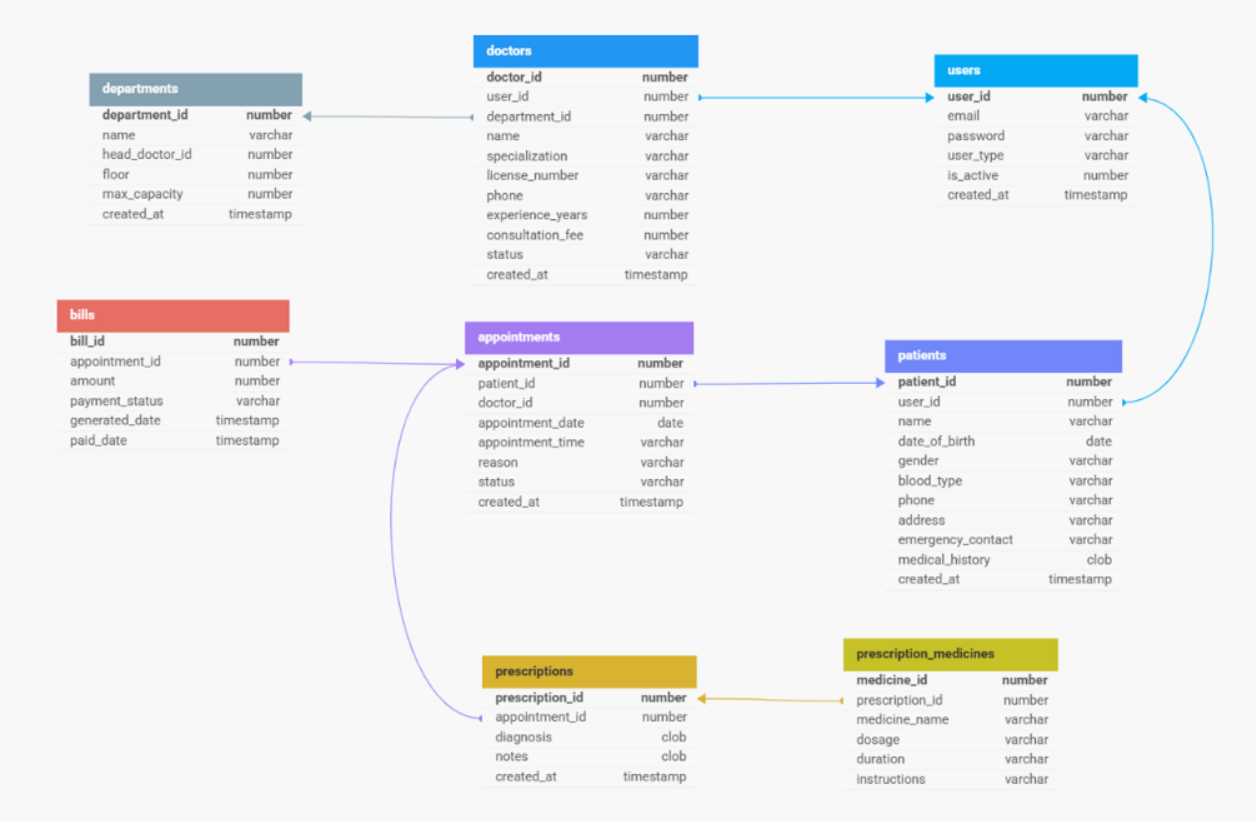


Figure 7: Visual representation of the schema using DBDesigner

4.2 Normalization and Validation Process

Our database schema has been designed following the principles of relational database normalization up to the Third Normal Form (3NF). The normalization process ensures data integrity, eliminates redundancy, and maintains consistency across the system.

4.2.1 Initial State: Unnormalized Form (UNF)

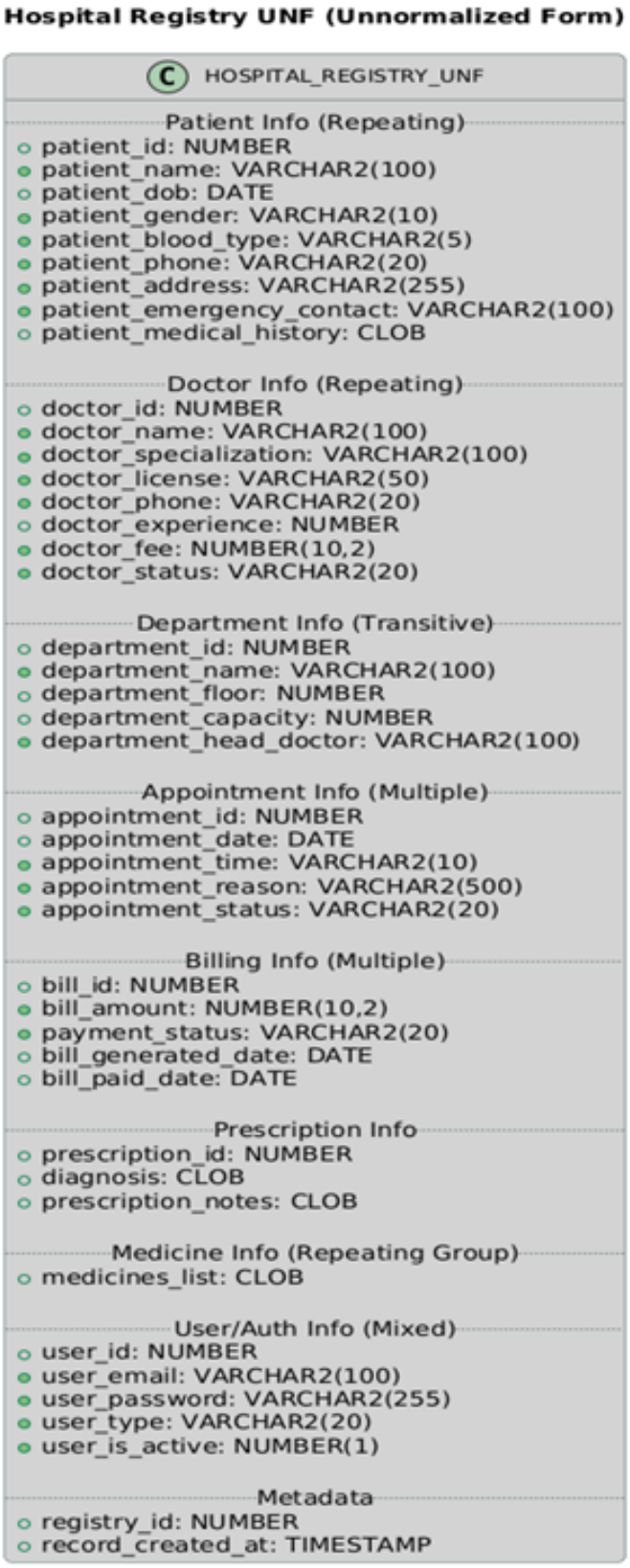


Figure 8: Unnormalized Form (UNF) Table Overview

The initial database design consisted of a single massive table named `HOSPITAL_REGISTRY_UNF` that included all hospital operations within a single data structure. This table contained attributes representing patients, doctors, departments, appointments, billing information, prescriptions, medications, and user authentication data—essentially putting every entity and relationship within the hospital management system into one massive repository.

Critical Problems and Anomalies in UNF Design:

- **Data Integrity and Update Anomalies:** Modifying a doctor's contact information necessitated updates across all rows containing that doctor's records, creating opportunities for inconsistent data states if updates failed partially. Changing department details required locating and updating every record associated with doctors from that department.
- **Insertion and Deletion Anomalies:** The design imposed artificial constraints on data creation and removal. Adding a new doctor to the system was impossible without simultaneously creating a patient appointment record, as the table structure demanded complete rows with values for all columns. Deleting a patient's final appointment would also erase all information about that patient from the system.

4.2.2 First Normal Form (1NF)

To convert our `HOSPITAL_REGISTRY_UNF` table to First Normal Form, we apply these fundamental rules:

1. Atomic Values: Ensure each column contains only indivisible values
2. Eliminate Repeating Groups: Extract repeating data into separate tables
3. Define Primary Keys: Establish unique identifiers for each table
4. Consistent Column Definitions: Ensure each column contains values of the same type

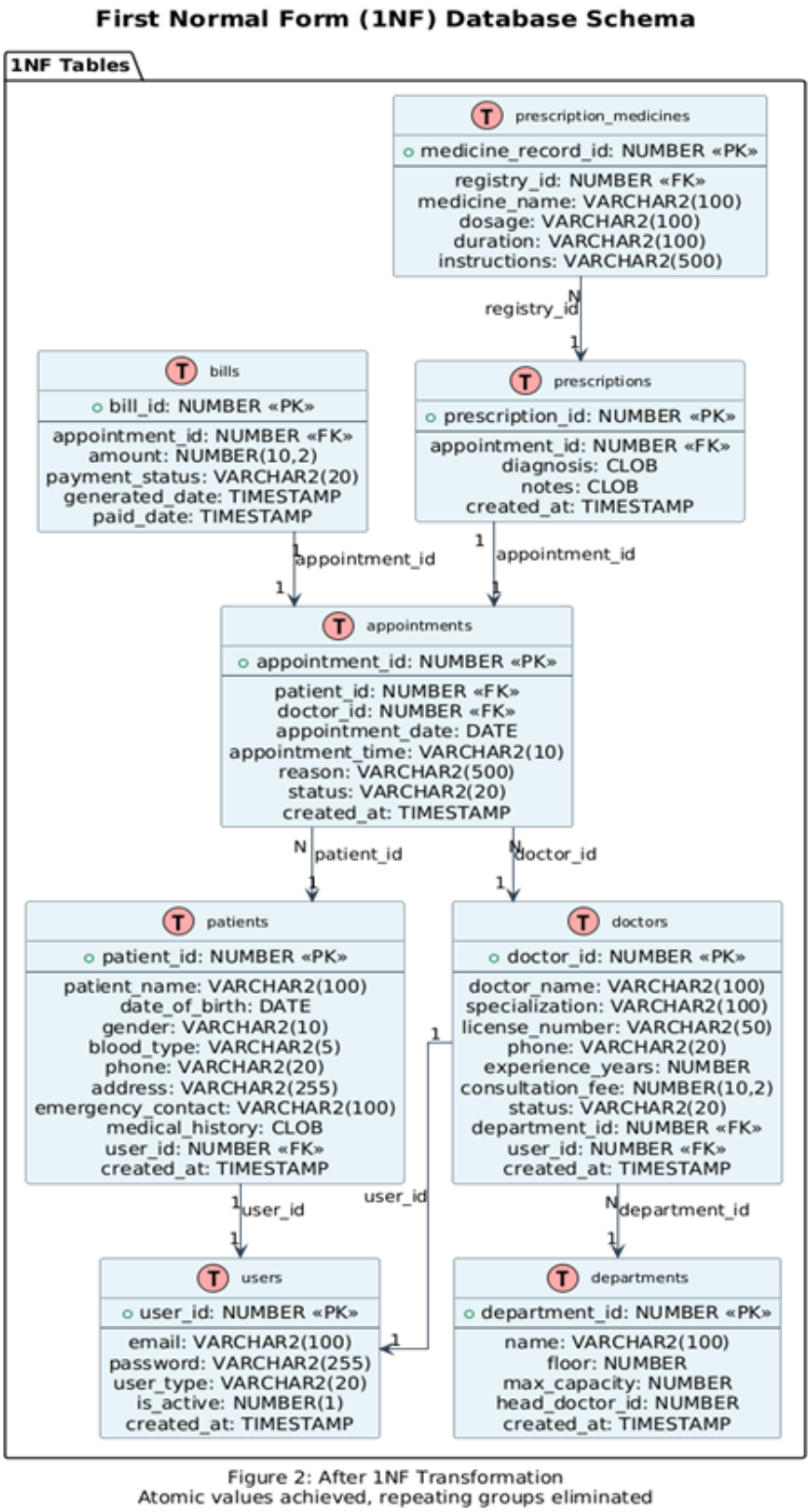


Figure 9: First Normal Form (1NF)

Following the systematic decomposition of the initial UNF table, the database schema has been transformed into First Normal Form. This transformation resulted in the creation of eight distinct tables: bills, prescriptions, doctors, users, patients, departments, prescription_medicines, appointments.

4.2.3 Second Normal Form (2NF)

Steps to Achieve Second Normal Form (2NF):

1. Identify Partial Dependencies - Find attributes depending on only part of primary key
2. Separate Mixed Concerns - Remove embedded data from entities
3. Formalize Foreign Keys - Replace vague references with explicit FK relationships
4. Implement Business Constraints - Add CHECK, UNIQUE, and NOT NULL constraints
5. Eliminate Partial Dependencies - Ensure all non-key attributes depend on entire PK
6. Verify 2NF Compliance - Confirm no partial dependencies remain

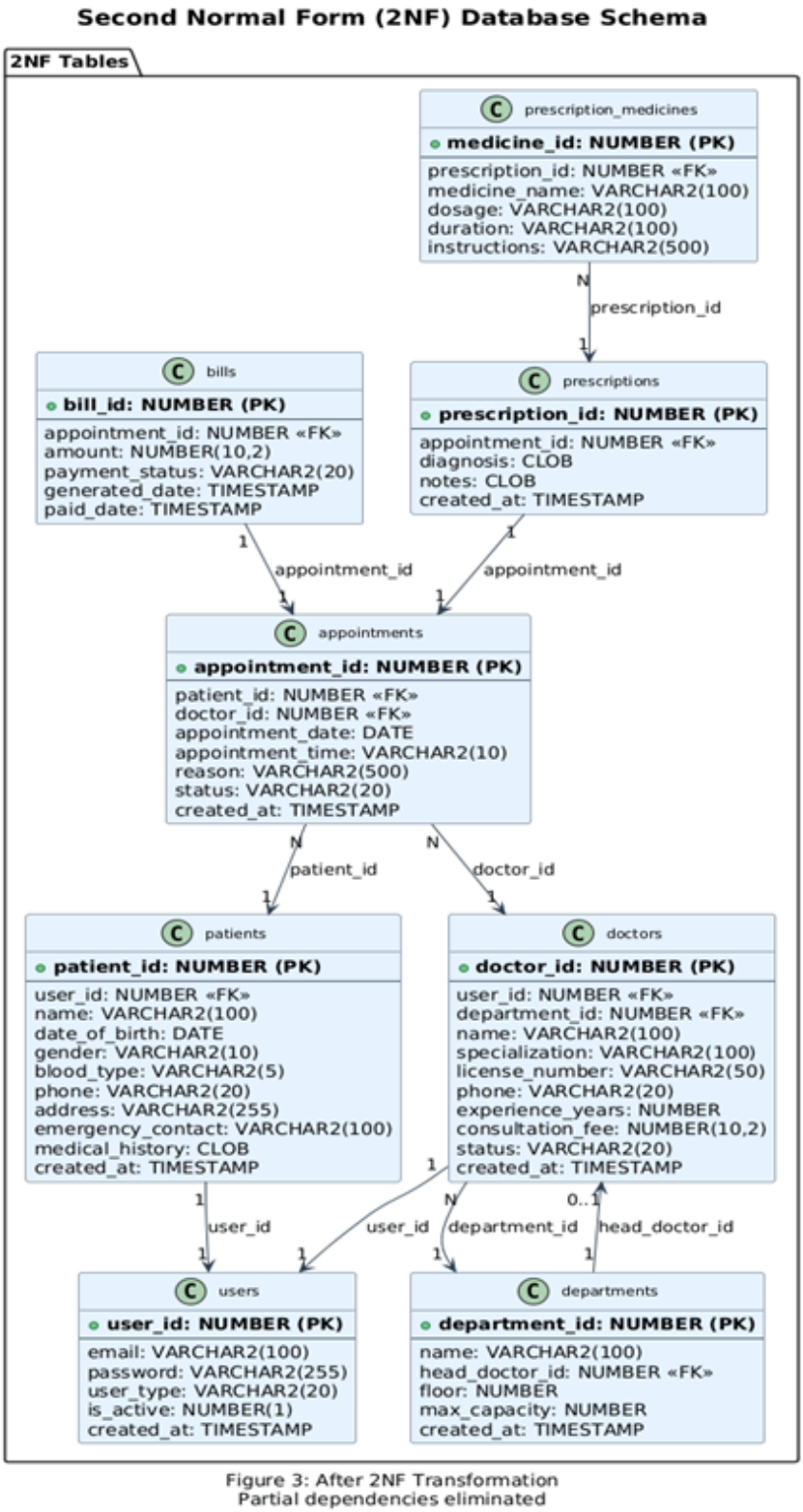


Figure 10: Second Normal Form (2NF)

Second Normal Form (2NF) Design Description: The Second Normal Form transformation successfully eliminated all partial dependencies from our database schema, establishing a structure where every non-key attribute fully depends on the entire primary key of its respective table. The 2NF design features eight clearly defined tables with properly formalized relationships: users, patients,

doctors, departments, appointments, bills, prescriptions, and prescription_medicines. Critical improvements include the separation of user authentication data from entity information, explicit foreign key definitions with referential integrity constraints, and the implementation of business rule validations through CHECK constraints on status fields.

Persistent Problems in 2NF Design: Despite achieving partial dependency elimination, the 2NF schema continues to exhibit transitive dependencies that violate Third Normal Form requirements. The `departments` table maintains a problematic `head_doctor_id` attribute that creates a transitive chain where department information depends on a doctor, who in turn has various attributes of their own. These transitive dependencies introduce update anomalies where changes to a doctor's information might require updates to department records, and modifications to appointment time formats would need sweeping changes across all appointment records.

4.2.4 Third Normal Form (3NF)

Steps to Achieve Third Normal Form (3NF):

1. Identify Transitive Dependencies - Find attributes depending on other non-key attributes
2. Evaluate Practical Normalization - Determine which transitive dependencies should be eliminated vs. accepted
3. Eliminate Unnecessary Transitive Dependencies - Create new tables or restructure relationships where practical
4. Accept Controlled Denormalization - Keep intentional 2NF structures in departments and appointments tables
5. Implement Final Constraints - Add deferred foreign keys and remaining business rules
6. Verify 3NF Compliance - Confirm 6 tables achieve 3NF, 2 tables remain in 2NF intentionally

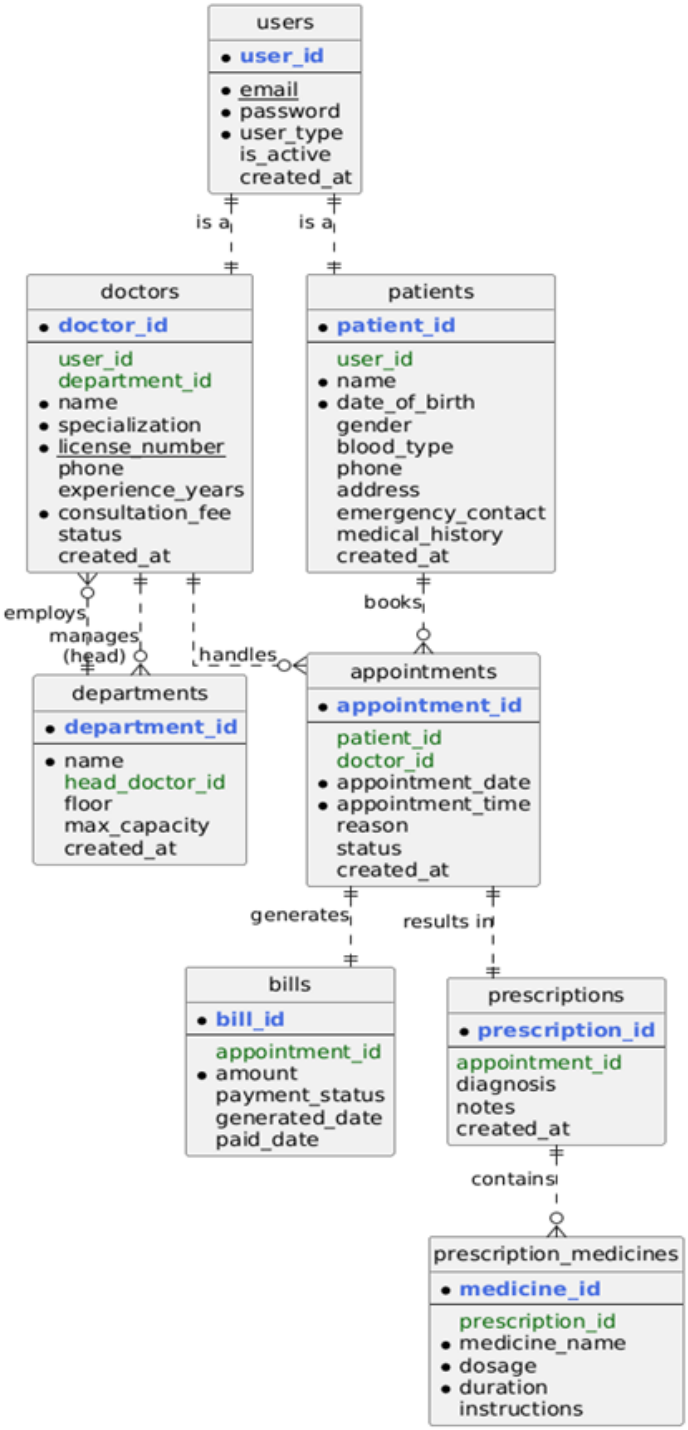


Figure 11: Third Normal Form (3NF) Schema

Third Normal Form (3NF) Design Description: The Third Normal Form transformation has produced our final optimized database schema, representing the culmination of the normalization journey from a monolithic UNF structure. The 3NF design comprises the original eight tables: `users`, `patients`, `doctors`, `departments`, `appointments`, `bills`, `prescriptions`, and `prescription_medicines`. The tables achieve full Third Normal Form compliance, eliminating all transitive dependencies and ensuring that no non-key attribute depends on another non-key attribute within each table. This level of normalization minimizes data redundancy, prevents update anomalies, and establishes a theoretically sound foundation for data integrity. Each table now serves a single, well-defined purpose with clear

boundaries and optimal data organization.

4.3 DDL Script Screenshots & Snippets

Provided below are the screenshots for the Data Definition Language scripts for the eight tables in our schema and the constraints that are applied to each table.

4.3.1 Users Table

```
CREATE TABLE users (
    user_id NUMBER PRIMARY KEY,
    email VARCHAR2(100) UNIQUE NOT NULL,
    password VARCHAR2(255) NOT NULL,
    user_type VARCHAR2(20) NOT NULL CHECK (user_type IN ('admin', 'doctor', 'patient')),
    is_active NUMBER(1) DEFAULT 1,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Figure 12: Users Table DDL

Users Table: This table serves as the central authentication hub, enforcing critical constraints to maintain system integrity. The `user_id` is the primary key, ensuring each user is uniquely identifiable. The `email` field has a UNIQUE constraint to prevent duplicate registrations, while `NOT NULL` guarantees mandatory email and password entries. The `user_type` column uses a CHECK constraint to restrict values to predefined roles ('admin', 'doctor', 'patient'), ensuring valid user categorization. The `is_active` flag defaults to 1 for active accounts, and `created_at` automatically timestamps each record.

4.3.2 Departments Table

```
CREATE TABLE departments (
    department_id NUMBER PRIMARY KEY,
    name VARCHAR2(100) NOT NULL,
    head_doctor_id NUMBER,
    floor NUMBER,
    max_capacity NUMBER DEFAULT 10,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Figure 13: Departments Table DDL

Departments Table: The Departments table organizes hospital units with structured constraints. `department_id` acts as the primary key for unique identification. The `name` field is mandatory (NOT NULL), while `head_doctor_id` references a doctor's record (though foreign key enforcement is deferred until Doctors table creation). The `max_capacity` defaults to 10 (Note: we changed the default max capacity later during debugging to 100), and `created_at` automatically records creation time. Logical grouping is supported via `floor` numbering and department naming.

4.3.3 Doctors Table

```
CREATE TABLE doctors (
    doctor_id NUMBER PRIMARY KEY,
    user_id NUMBER NOT NULL,
    department_id NUMBER NOT NULL,
    name VARCHAR2(100) NOT NULL,
    specialization VARCHAR2(100) NOT NULL,
    license_number VARCHAR2(50) UNIQUE NOT NULL,
    phone VARCHAR2(20),
    experience_years NUMBER,
    consultation_fee NUMBER(10,2) NOT NULL,
    status VARCHAR2(20) DEFAULT 'Active' CHECK (status IN ('Active', 'On Leave')),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id),
    FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

Figure 14: Doctors Table DDL

Doctors Table: This table manages doctor profiles with relational and domain constraints. `doctor_id` is the primary key, while `user_id` and `department_id` are foreign keys linking to Users and Departments respectively, ensuring referential integrity. The `license_number` is UNIQUE to prevent duplicate licenses, and `specialization` and `consultation_fee` are mandatory. A CHECK constraint on `status` limits values to 'Active' or 'On Leave', and `created_at` auto-timestamps entries.

4.3.4 Patients Table

```
CREATE TABLE patients (
    patient_id NUMBER PRIMARY KEY,
    user_id NUMBER NOT NULL,
    name VARCHAR2(100) NOT NULL,
    date_of_birth DATE NOT NULL,
    gender VARCHAR2(10) CHECK (gender IN ('Male', 'Female', 'Other')),
    blood_type VARCHAR2(5),
    phone VARCHAR2(20),
    address VARCHAR2(255),
    emergency_contact VARCHAR2(100),
    medical_history CLOB,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id)
);
```

Figure 15: Patients Table DDL

Patients Table: Patient records are controlled via constraints to ensure data reliability. `patient_id` serves as the primary key, and `user_id` is a foreign key to the Users table. Mandatory fields (NOT NULL) include `name` and `date_of_birth`. The `gender` column uses a CHECK constraint to allow only 'Male', 'Female', or 'Other'. `blood_type`, `phone`, and `address` provide optional details, while `medical_history` uses a CLOB for extensive text. `created_at` automatically captures record creation.

4.3.5 Appointments Table

```
CREATE TABLE appointments (
    appointment_id NUMBER PRIMARY KEY,
    patient_id NUMBER NOT NULL,
    doctor_id NUMBER NOT NULL,
    appointment_date DATE NOT NULL,
    appointment_time VARCHAR2(10) NOT NULL,
    reason VARCHAR2(500),
    status VARCHAR2(20) DEFAULT 'Scheduled' CHECK (status IN ('Scheduled', 'Completed', 'Cancelled', 'No Show')),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (patient_id) REFERENCES patients(patient_id),
    FOREIGN KEY (doctor_id) REFERENCES doctors(doctor_id)
);
```

Figure 16: Appointments Table DDL

Appointments Table: Appointments are regulated with constraints to maintain scheduling accuracy. The `appointment_id` is the primary key, while `patient_id` and `doctor_id` are foreign keys linking to

Patients and Doctors. `appointment_date` and `appointment_time` are mandatory. A CHECK constraint on `status` restricts values to 'Scheduled', 'Completed', 'Cancelled', or 'No Show', with a default of 'Scheduled'. `created_at` records the booking timestamp automatically.

4.3.6 Bills Table

```
CREATE TABLE bills (
    bill_id NUMBER PRIMARY KEY,
    appointment_id NUMBER NOT NULL,
    amount NUMBER(10,2) NOT NULL,
    payment_status VARCHAR2(20) DEFAULT 'Unpaid' CHECK (payment_status IN ('Paid', 'Unpaid')),
    generated_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    paid_date TIMESTAMP,
    FOREIGN KEY (appointment_id) REFERENCES appointments(appointment_id)
);
```

Figure 17: Bills Table DDL

Bills Table: Billing data integrity is enforced through specific constraints. `bill_id` is the primary key, and `appointment_id` is a foreign key to Appointments, linking each bill to a specific visit. The `amount` is mandatory (NOT NULL). A CHECK constraint on `payment_status` allows only 'Paid' or 'Unpaid' (defaulting to 'Unpaid'). `generated_date` auto-timestamps bill creation, while `paid_date` records payment completion separately.

4.3.7 Prescriptions Table

```
CREATE TABLE prescriptions (
    prescription_id NUMBER PRIMARY KEY,
    appointment_id NUMBER NOT NULL,
    diagnosis CLOB,
    notes CLOB,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (appointment_id) REFERENCES appointments(appointment_id)
);
```

Figure 18: Prescriptions Table DDL

Prescriptions Table: This table stores prescription metadata with controlled constraints. `prescription_id` is the primary key, and `appointment_id` is a foreign key to Appointments, ensuring each prescription ties to a documented visit. `diagnosis` and `notes` use CLOB data types for detailed text entries without length restrictions. `created_at` automatically records the prescription issuance time. No explicit NOT NULL constraints are applied to diagnosis/notes, allowing optional entries.

4.3.8 Prescription_Medicines Table

```
CREATE TABLE prescription_medicines (
    medicine_id NUMBER PRIMARY KEY,
    prescription_id NUMBER NOT NULL,
    medicine_name VARCHAR2(100) NOT NULL,
    dosage VARCHAR2(100) NOT NULL,
    duration VARCHAR2(100) NOT NULL,
    instructions VARCHAR2(500),
    FOREIGN KEY (prescription_id) REFERENCES prescriptions(prescription_id) ON DELETE CASCADE
);
```

Figure 19: Prescription_Medicines Table DDL

Prescription_Medicines Table: Individual medicine details within prescriptions are managed with strict constraints. `medicine_id` is the primary key, and `prescription_id` is a foreign key to Prescriptions with ON DELETE CASCADE, automatically removing medicines if a prescription is deleted. `medicine_name`, `dosage`, and `duration` are mandatory (NOT NULL). `instructions` provides optional guidance. This structure ensures each medicine is uniquely recorded while maintaining referential integrity.

4.4 Sequences

```
-- Create Sequences
CREATE SEQUENCE seq_users START WITH 1001 INCREMENT BY 1;
CREATE SEQUENCE seq_departments START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE seq_doctors START WITH 101 INCREMENT BY 1;
CREATE SEQUENCE seq_patients START WITH 10001 INCREMENT BY 1;
CREATE SEQUENCE seq_appointments START WITH 100001 INCREMENT BY 1;
CREATE SEQUENCE seq_bills START WITH 1000001 INCREMENT BY 1;
CREATE SEQUENCE seq_prescriptions START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE seq_medicines START WITH 1 INCREMENT BY 1;
```

Figure 20: Database Sequences

The sequences serve as automated counter systems for primary keys, ensuring each new record gets a unique identifier without manual intervention.

The `seq_users` sequence starts at 1001, creating user IDs beginning with a four-digit format that distinguishes user accounts. `seq_departments` begins at 1 with simple incremental numbering for hospital departments. `seq_doctors` starts at 101, reserving lower numbers potentially for system use while establishing doctor IDs in the hundreds range.

For patient identification, `seq_patients` begins at 10001, using a five-digit starting point that clearly distinguishes patient records from other entities. `seq_appointments` starts at 100001 with six digits, accommodating potentially high volumes of appointment records over time.

The billing system uses `seq_bills` starting at 1000001 (seven digits), because there could be numerous financial transactions. Finally, `seq_prescriptions` and `seq_medicines` both start at 1 with simple incremental patterns, as these are detail tables where numbering can be more straightforward.

All sequences increment by 1, ensuring consecutive numbering. This structured approach prevents ID collisions, maintains referential integrity across related tables, and provides human-readable ID patterns where higher starting values indicate specific entity types. The sequences optimize performance by eliminating the need for manual ID tracking and MAX()+1 queries that could cause locking issues in multi-user environments.

4.5 Triggers: Code and Explanations

4.5.1 Double-Booking Prevention Trigger

```
-- 1. Trigger: Prevent double booking for doctors
CREATE OR REPLACE TRIGGER trg_prevent_double_booking
BEFORE INSERT ON appointments
FOR EACH ROW
DECLARE
    v_count NUMBER;
BEGIN
    SELECT COUNT(*) INTO v_count
    FROM appointments
    WHERE doctor_id = :NEW.doctor_id
    AND appointment_date = :NEW.appointment_date
    AND appointment_time = :NEW.appointment_time
    AND status != 'Cancelled';

    IF v_count > 0 THEN
        RAISE_APPLICATION_ERROR(-20001, 'Doctor already has an appointment at this time');
    END IF;
END;
/
```

Figure 21: Double-Booking Prevention Trigger Code

Double-Booking Prevention Trigger: This `BEFORE INSERT` trigger on the appointments table enforces a core scheduling constraint by preventing double-booking of doctors. For each new appointment attempt, it queries existing records to count any non-cancelled appointments (status != 'Cancelled') for the same doctor at the identical date and time slot. If a conflict is detected (count ¿ 0), it immediately raises application error -20001 with the message "Doctor already has an appointment at this time," blocking the insertion. This database-level validation ensures that even direct SQL operations or multiple application instances cannot create scheduling conflicts, guaranteeing reliable doctor availability management.

4.5.2 Auto-Bill Generation Trigger

```
-- 2. Trigger: Auto-generate bill when appointment is completed
CREATE OR REPLACE TRIGGER trg_auto_generate_bill
AFTER UPDATE ON appointments
FOR EACH ROW
WHEN (OLD.status != 'Completed' AND NEW.status = 'Completed')
DECLARE
    v_consultation_fee NUMBER;
BEGIN
    SELECT consultation_fee INTO v_consultation_fee
    FROM doctors
    WHERE doctor_id = :NEW.doctor_id;

    INSERT INTO bills (bill_id, appointment_id, amount, payment_status)
    VALUES (seq_bills.NEXTVAL, :NEW.appointment_id, v_consultation_fee, 'Unpaid');

    DBMS_OUTPUT.PUT_LINE('Bill generated for appointment ' || :NEW.appointment_id);
END;
/
```

Figure 22: Auto-Bill Generation Trigger Code

Auto-Bill Generation Trigger: This `AFTER UPDATE` trigger on the appointments table automates billing processes by generating a bill immediately when an appointment's status changes to 'Completed'. Using a conditional clause (WHEN), it activates only when the status transitions specifically from non-completed to completed, ensuring bills are created exactly once per appointment. Upon activation, it retrieves the doctor's consultation fee from the doctors table and inserts a corresponding record into the bills table with a unique bill ID from the sequence, linking it to the appointment, setting the amount to the consultation fee, and defaulting the payment status to 'Unpaid'. The trigger includes a diagnostic output message confirming bill generation.

4.5.3 Prescription Validation Trigger

```
-- 4. Trigger: Prevent prescription for incomplete appointments
CREATE OR REPLACE TRIGGER trg_prescription_appointment_check
BEFORE INSERT ON prescriptions
FOR EACH ROW
DECLARE
    v_appointment_status VARCHAR2(20);
BEGIN
    SELECT status INTO v_appointment_status
    FROM appointments
    WHERE appointment_id = :NEW.appointment_id;

    IF v_appointment_status != 'Completed' THEN
        RAISE_APPLICATION_ERROR(-20003, 'Prescriptions can only be created for completed appointments');
    END IF;
END;
/
```

Figure 23: Prescription Validation Trigger Code

Prescription Validation Trigger: This `BEFORE INSERT` trigger on the prescriptions table enforces a critical workflow rule by ensuring prescriptions can only be created for appointments with a 'Completed' status. When a new prescription is attempted, it queries the associated appointment's status from the appointments table. If the appointment status is anything other than 'Completed', the trigger immediately raises application error -20003 with a clear message blocking the insertion. This validation prevents premature or erroneous prescription creation for appointments that are still scheduled, cancelled, or marked as no-shows, ensuring that medical documentation aligns with actual patient consultations. By enforcing this business logic at the database level, the trigger maintains data integrity and supports proper clinical workflow, where prescriptions logically follow completed medical examinations.

4.5.4 Appointment Booking Validation Trigger

```
CREATE OR REPLACE TRIGGER validate_appointment_booking
BEFORE INSERT ON appointments
FOR EACH ROW
DECLARE
    scheduled_count NUMBER;
    max_appointments NUMBER := 2;
    max_booking_date DATE;
BEGIN
    -- Rule 1: Check booking date is within 7 days
    max_booking_date := TRUNC(SYSDATE) + 7;

    IF :NEW.appointment_date > max_booking_date THEN
        RAISE_APPLICATION_ERROR(-20002,
            'Appointments can only be booked within the next 7 days.');
    END IF;

    -- Rule 2: Check maximum scheduled appointments
    SELECT COUNT(*)
    INTO scheduled_count
    FROM appointments
    WHERE patient_id = :NEW.patient_id
    AND status = 'Scheduled'
    AND appointment_id != NVL(:NEW.appointment_id, -1);

    IF scheduled_count >= max_appointments THEN
        RAISE_APPLICATION_ERROR(-20003,
            'Maximum of ' || max_appointments || ' scheduled appointments allowed per patient.');
    END IF;
END;
/
```

Figure 24: Appointment Booking Validation Trigger Code

Appointment Booking Validation Trigger: This `BEFORE INSERT` trigger enforces two key business rules for appointment scheduling. First, it restricts bookings to within a 7-day window by comparing the requested appointment date against the current date plus seven days, raising an error if exceeded. Second, it prevents patients from having more than two scheduled appointments simultaneously by counting their existing 'Scheduled' appointments (excluding the current one being inserted) and enforcing a maximum limit. These validations ensure manageable scheduling horizons and prevent appointment hoarding, maintaining system efficiency and fair access.

4.6 Procedures and Functions

4.6.1 Complete Appointment Procedure

```
-- 1. Procedure: Complete appointment and create prescription
CREATE OR REPLACE PROCEDURE complete_appointment(
    p_appointment_id IN NUMBER,
    p_diagnosis IN CLOB,
    p_notes IN CLOB
)
AS
    v_appointment_count NUMBER;
BEGIN
    -- Check if appointment exists and is scheduled
    SELECT COUNT(*) INTO v_appointment_count
    FROM appointments
    WHERE appointment_id = p_appointment_id AND status = 'Scheduled';

    IF v_appointment_count = 0 THEN
        RAISE_APPLICATION_ERROR(-20004, 'Appointment not found or not scheduled');
    END IF;

    -- Update appointment status
    UPDATE appointments
    SET status = 'Completed'
    WHERE appointment_id = p_appointment_id;

    -- Create prescription
    INSERT INTO prescriptions (prescription_id, appointment_id, diagnosis, notes)
    VALUES (seq_prescriptions.NEXTVAL, p_appointment_id, p_diagnosis, p_notes);

    COMMIT;
END;
/
```

Figure 25: Complete Appointment Procedure Code

Complete Appointment Procedure: This stored procedure streamlines the workflow by finalizing appointments and creating prescriptions in a single, atomic operation. It accepts an appointment ID, diagnosis, and notes as parameters, first validating that the appointment exists and has a 'Scheduled' status. Upon successful validation, it updates the appointment status to 'Completed', which automatically triggers the `trg_auto_generate_bill` to create a corresponding bill. Then, it inserts a new prescription record using the provided medical details. The procedure concludes with an explicit COMMIT, saving everything.

4.6.2 Doctor Availability Procedure

```
-- 2. Procedure: Get doctor availability
CREATE OR REPLACE PROCEDURE get_doctor_availability(
    p_doctor_id IN NUMBER,
    p_date IN DATE,
    p_available_slots OUT SYS_REFCURSOR
)
AS
BEGIN
    OPEN p_available_slots FOR
    SELECT time_slot
    FROM (
        SELECT '09:00 AM' as time_slot FROM DUAL UNION
        SELECT '09:30 AM' FROM DUAL UNION
        SELECT '10:00 AM' FROM DUAL UNION
        SELECT '10:30 AM' FROM DUAL UNION
        SELECT '11:00 AM' FROM DUAL UNION
        SELECT '11:30 AM' FROM DUAL UNION
        SELECT '02:00 PM' FROM DUAL UNION
        SELECT '02:30 PM' FROM DUAL UNION
        SELECT '03:00 PM' FROM DUAL UNION
        SELECT '03:30 PM' FROM DUAL UNION
        SELECT '04:00 PM' FROM DUAL UNION
        SELECT '04:30 PM' FROM DUAL
    ) all_slots
    WHERE time_slot NOT IN (
        SELECT appointment_time
        FROM appointments
        WHERE doctor_id = p_doctor_id
        AND appointment_date = p_date
        AND status != 'Cancelled'
    );
END;
/
```

Figure 26: Doctor Availability Procedure Code

Doctor Availability Procedure: This procedure generates available time slots for a specific doctor on a given date by comparing a fixed schedule of predefined consultation times against already booked appointments. It accepts a doctor ID and date as input parameters and returns a cursor (SYS_REFCURSOR) containing available slots. The procedure defines a standard set of 12 time slots

(morning and afternoon) using a derived table, then filters out any slots where the doctor already has non-cancelled appointments on that date. This provides real-time availability information for scheduling purposes, ensuring patients are only offered slots that are genuinely free. The use of a ref cursor allows efficient handling of the result set by calling applications, supporting dynamic scheduling interfaces.

4.6.3 Department Statistics Function

```
-- 3. Function: Calculate department statistics
CREATE OR REPLACE FUNCTION get_department_stats(
    p_department_id IN NUMBER
) RETURN SYS_REFCURSOR
AS
    v_stats SYS_REFCURSOR;
BEGIN
    OPEN v_stats FOR
    SELECT
        d.name as department_name,
        COUNT(DISTINCT doc.doctor_id) as doctor_count,
        COUNT(DISTINCT p.patient_id) as patient_count,
        COUNT(a.appointment_id) as appointment_count,
        SUM(CASE WHEN b.payment_status = 'Paid' THEN b.amount ELSE 0 END) as total_revenue
    FROM departments d
    LEFT JOIN doctors doc ON d.department_id = doc.department_id
    LEFT JOIN appointments a ON doc.doctor_id = a.doctor_id
    LEFT JOIN patients p ON a.patient_id = p.patient_id
    LEFT JOIN bills b ON a.appointment_id = b.appointment_id
    WHERE d.department_id = p_department_id
    GROUP BY d.name;

    RETURN v_stats;
END;
/
```

Figure 27: Department Statistics Function Code

Department Statistics Function: This function returns comprehensive performance metrics for a specified department by aggregating data across multiple related tables. Accepting a department ID as input, it returns a `SYS_REFCURSOR` containing a single-row result with key statistics: department name, count of assigned doctors, count of unique patients served, total appointments conducted, and total revenue generated from paid bills. Using `LEFT JOIN` operations, it safely combines data from departments, doctors, appointments, patients, and bills tables, ensuring departments with no activity still return baseline information. This function enables efficient reporting and departmental analytics without requiring multiple individual queries, supporting data-driven management decisions.

4.7 Data Insertion Samples

For this part of the report we used an application known as DBeaver, to show our tables and the values inserted in them. We inserted values using the command line interface initially to start us off and then we used the application/website's front-end functionalities to test that it all works.

4.7.1 Initial CLI Insertions

```
-- Insert Users
INSERT INTO users (user_id, email, password, user_type) VALUES (seq_users.NEXTVAL, 'admin@hospital.com', 'admin123', 'admin');
INSERT INTO users (user_id, email, password, user_type) VALUES (seq_users.NEXTVAL, 'sarah.johnson', 'doctor123', 'doctor');
INSERT INTO users (user_id, email, password, user_type) VALUES (seq_users.NEXTVAL, 'michael.chen', 'doctor123', 'doctor');
INSERT INTO users (user_id, email, password, user_type) VALUES (seq_users.NEXTVAL, 'john.doe', 'patient123', 'patient');
INSERT INTO users (user_id, email, password, user_type) VALUES (seq_users.NEXTVAL, 'mary.johnson', 'patient123', 'patient');

-- Insert Departments
INSERT INTO departments (department_id, name, floor, max_capacity) VALUES (seq_departments.NEXTVAL, 'Cardiology', 3, 8);
INSERT INTO departments (department_id, name, floor, max_capacity) VALUES (seq_departments.NEXTVAL, 'Neurology', 4, 6);
INSERT INTO departments (department_id, name, floor, max_capacity) VALUES (seq_departments.NEXTVAL, 'Pediatrics', 2, 10);
INSERT INTO departments (department_id, name, floor, max_capacity) VALUES (seq_departments.NEXTVAL, 'Orthopedics', 5, 7);
INSERT INTO departments (department_id, name, floor, max_capacity) VALUES (seq_departments.NEXTVAL, 'Emergency', 1, 12);

-- Insert Doctors
INSERT INTO doctors (doctor_id, user_id, department_id, name, specialization, license_number, phone, experience_years, consultation_fee)
VALUES (seq_doctors.NEXTVAL, 1002, 1, 'Dr. Sarah Johnson', 'Cardiologist', 'CARD12345', '555-0101', 15, 200.00);

INSERT INTO doctors (doctor_id, user_id, department_id, name, specialization, license_number, phone, experience_years, consultation_fee)
VALUES (seq_doctors.NEXTVAL, 1003, 2, 'Dr. Michael Chen', 'Neurologist', 'NEURO67890', '555-0102', 12, 180.00);

-- Insert Patients
INSERT INTO patients (patient_id, user_id, name, date_of_birth, gender, blood_type, phone, address, emergency_contact)
VALUES (seq_patients.NEXTVAL, 1004, 'John Doe', DATE '1990-05-15', 'Male', 'O+', '555-0123', '123 Main St, City', 'Jane Doe - 555-0124');

INSERT INTO patients (patient_id, user_id, name, date_of_birth, gender, blood_type, phone, address, emergency_contact)
VALUES (seq_patients.NEXTVAL, 1005, 'Mary Johnson', DATE '1985-08-22', 'Female', 'A+', '555-0125', '456 Oak St, City', 'Robert Johnson - 555-0126');

COMMIT;
```

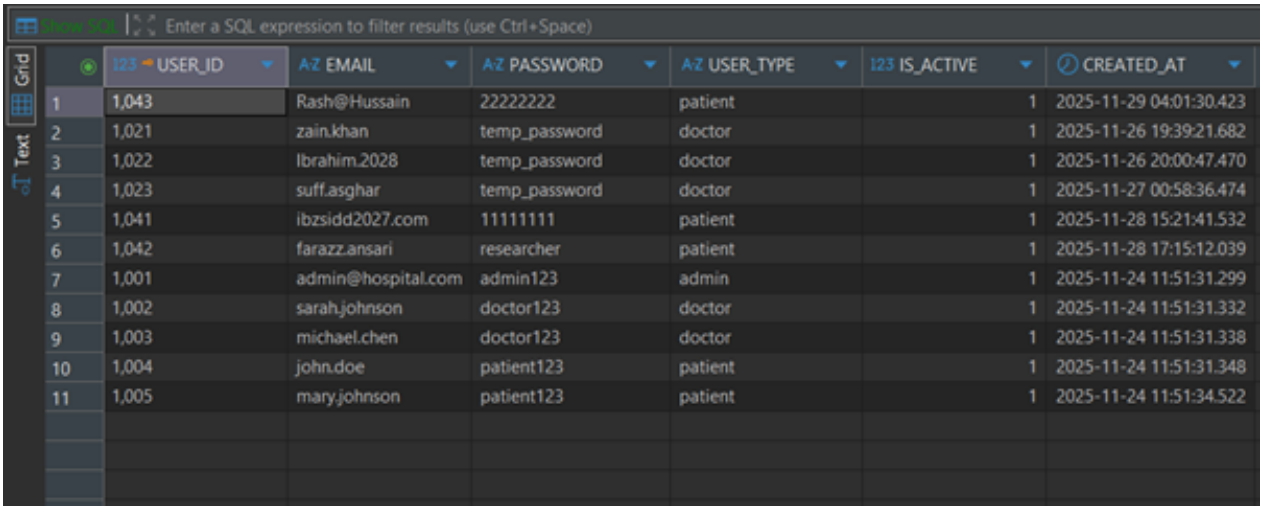
Figure 28: Initial CLI Data Insertion Screenshot

```
-- Add appointments
INSERT INTO appointments (appointment_id, patient_id, doctor_id, appointment_date, appointment_time, reason, status)
VALUES (100001, 10001, 101, DATE '2025-11-25', '10:00 AM', 'Heart palpitations', 'Scheduled');

INSERT INTO appointments (appointment_id, patient_id, doctor_id, appointment_date, appointment_time, reason, status)
VALUES (100002, 10002, 102, DATE '2025-11-26', '02:30 PM', 'Migraine consultation', 'Scheduled');

INSERT INTO appointments (appointment_id, patient_id, doctor_id, appointment_date, appointment_time, reason, status)
VALUES (100003, 10001, 103, DATE '2025-11-27', '11:00 AM', 'Child vaccination', 'Completed');

INSERT INTO appointments (appointment_id, patient_id, doctor_id, appointment_date, appointment_time, reason, status)
VALUES (100004, 10002, 104, DATE '2025-11-28', '03:00 PM', 'Knee pain evaluation', 'Scheduled');

COMMIT;
```

Figure 29: Additional CLI Data Insertion Screenshot

All the other insertions in our eight tables were done through the front end, showing a successful backend-frontend integration, which was our main goal with this project.

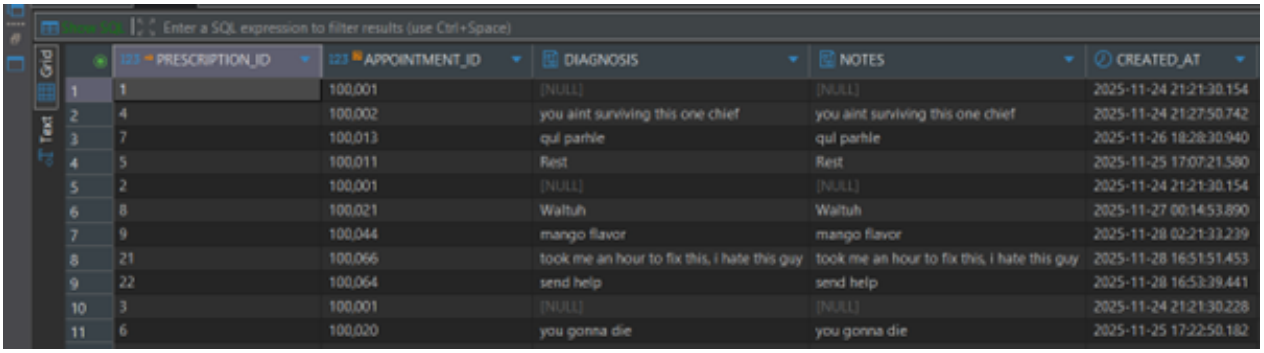
4.7.2 Users Table Data



	USER_ID	EMAIL	PASSWORD	USER_TYPE	IS_ACTIVE	CREATED_AT
1	1,043	Rash@Hussain	22222222	patient	1	2025-11-29 04:01:30.423
2	1,021	zain.khan	temp_password	doctor	1	2025-11-26 19:39:21.682
3	1,022	Ibrahim.2028	temp_password	doctor	1	2025-11-26 20:00:47.470
4	1,023	suff.asghar	temp_password	doctor	1	2025-11-27 00:58:36.474
5	1,041	ibzsidd2027.com	11111111	patient	1	2025-11-28 15:21:41.532
6	1,042	farazz.ansari	researcher	patient	1	2025-11-28 17:15:12.039
7	1,001	admin@hospital.com	admin123	admin	1	2025-11-24 11:51:31.299
8	1,002	sarah.johnson	doctor123	doctor	1	2025-11-24 11:51:31.332
9	1,003	michael.chen	doctor123	doctor	1	2025-11-24 11:51:31.338
10	1,004	john.doe	patient123	patient	1	2025-11-24 11:51:31.348
11	1,005	mary.johnson	patient123	patient	1	2025-11-24 11:51:34.522

Figure 30: Users Table Data

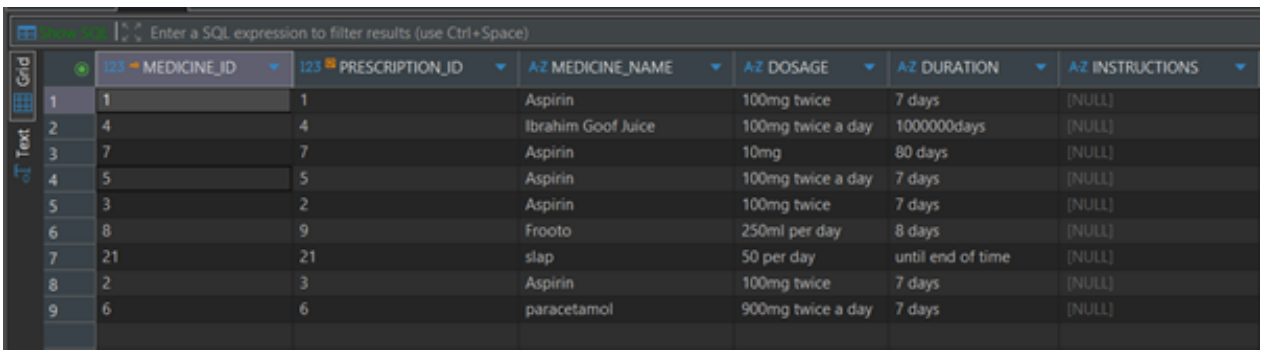
4.7.3 Prescriptions Table Data



	PRESCRIPTION_ID	APPOINTMENT_ID	DIAGNOSIS	NOTES	CREATED_AT
1	1	100,001	[NULL]	[NULL]	2025-11-24 21:21:30.154
2	4	100,002	you aint surviving this one chief	you aint surviving this one chief	2025-11-24 21:27:50.742
3	7	100,013	qui parhle	qui parhle	2025-11-26 18:28:30.940
4	5	100,011	Rest	Rest	2025-11-25 17:07:21.580
5	2	100,001	[NULL]	[NULL]	2025-11-24 21:21:30.154
6	8	100,021	Waltuh	Waltuh	2025-11-27 00:14:53.890
7	9	100,044	mango flavor	mango flavor	2025-11-28 02:21:33.239
8	21	100,066	took me an hour to fix this, i hate this guy	took me an hour to fix this, i hate this guy	2025-11-28 16:51:51.453
9	22	100,064	send help	send help	2025-11-28 16:53:39.441
10	3	100,001	[NULL]	[NULL]	2025-11-24 21:21:30.228
11	6	100,020	you gonna die	you gonna die	2025-11-25 17:22:50.182

Figure 31: Prescriptions Table Data

4.7.4 Prescription_Medicines Table Data



	MEDICINE_ID	PRESCRIPTION_ID	MEDICINE_NAME	DOSAGE	DURATION	INSTRUCTIONS
1	1	1	Aspirin	100mg twice	7 days	[NULL]
2	4	4	Ibrahim Goof Juice	100mg twice a day	1000000days	[NULL]
3	7	7	Aspirin	10mg	80 days	[NULL]
4	5	5	Aspirin	100mg twice a day	7 days	[NULL]
5	3	2	Aspirin	100mg twice	7 days	[NULL]
6	8	9	Frooto	250ml per day	8 days	[NULL]
7	21	21	slap	50 per day	until end of time	[NULL]
8	2	3	Aspirin	100mg twice	7 days	[NULL]
9	6	6	paracetamol	900mg twice a day	7 days	[NULL]

Figure 32: Prescription_Medicines Table Data

4.7.5 Doctors Table Data

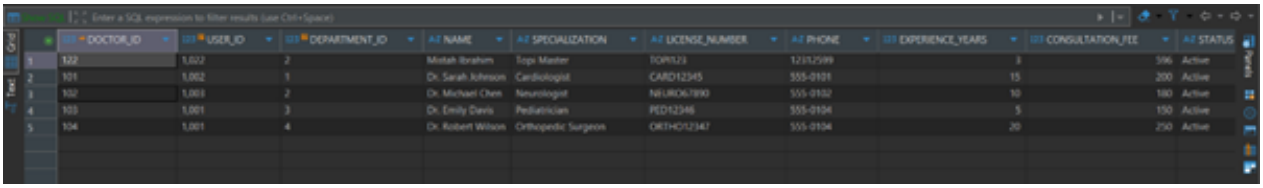
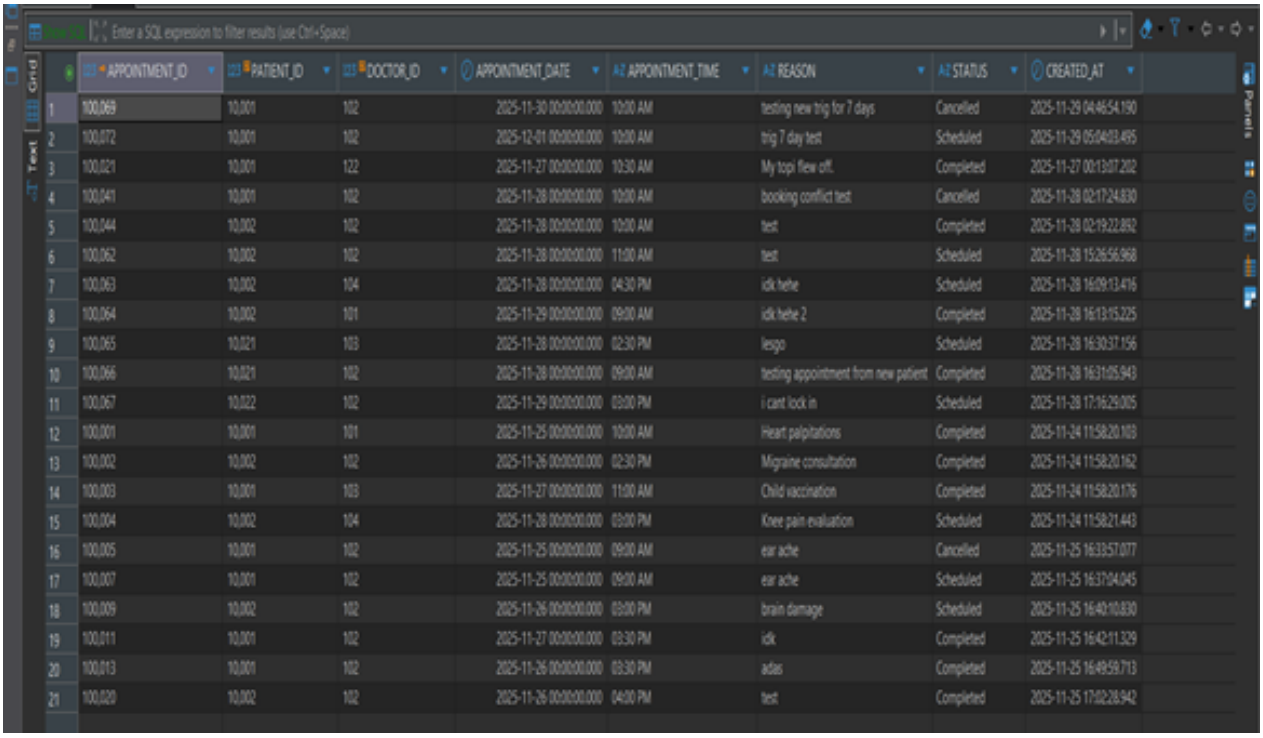


Figure 33: Doctors Table Data

4.7.6 Appointments Table Data



	APPOINTMENT_ID	PATIENT_ID	DOCTOR_ID	APPOINTMENT_DATE	APPOINTMENT_TIME	REASON	STATUS	CREATED_AT
1	100,069	10,001	102	2025-11-30 00:00:00.000	10:00 AM	testing new trig for 7 days	Cancelled	2025-11-29 04:46:54.190
2	100,072	10,001	102	2025-12-01 00:00:00.000	10:00 AM	trig 7 day test	Scheduled	2025-11-29 05:04:03.495
3	100,021	10,001	122	2025-11-27 00:00:00.000	10:30 AM	My topi flew off.	Completed	2025-11-27 00:12:07.202
4	100,041	10,001	102	2025-11-28 00:00:00.000	10:00 AM	booking conflict test	Cancelled	2025-11-28 02:17:24.830
5	100,044	10,002	102	2025-11-28 00:00:00.000	10:00 AM	test	Completed	2025-11-28 02:19:22.892
6	100,062	10,002	102	2025-11-28 00:00:00.000	11:00 AM	test	Scheduled	2025-11-28 15:26:56.968
7	100,063	10,002	104	2025-11-28 00:00:00.000	04:30 PM	idk hehe	Scheduled	2025-11-28 16:09:13.416
8	100,064	10,002	101	2025-11-29 00:00:00.000	09:00 AM	idk hehe 2	Completed	2025-11-28 16:13:15.225
9	100,065	10,021	103	2025-11-28 00:00:00.000	02:30 PM	lesgo	Scheduled	2025-11-28 16:30:37.156
10	100,066	10,021	102	2025-11-28 00:00:00.000	09:00 AM	testing appointment from new patient	Completed	2025-11-28 16:31:05.943
11	100,067	10,022	102	2025-11-29 00:00:00.000	03:00 PM	i cant lock in	Scheduled	2025-11-28 17:16:29.005
12	100,001	10,001	101	2025-11-25 00:00:00.000	10:00 AM	Heart palpitations	Completed	2025-11-24 11:58:20.103
13	100,002	10,002	102	2025-11-26 00:00:00.000	02:30 PM	Migraine consultation	Completed	2025-11-24 11:58:20.162
14	100,003	10,001	103	2025-11-27 00:00:00.000	11:00 AM	Child vaccination	Completed	2025-11-24 11:58:20.176
15	100,004	10,002	104	2025-11-28 00:00:00.000	03:00 PM	Knee pain evaluation	Scheduled	2025-11-24 11:58:21.443
16	100,005	10,001	102	2025-11-25 00:00:00.000	09:00 AM	ear ache	Cancelled	2025-11-25 16:33:57.077
17	100,007	10,001	102	2025-11-25 00:00:00.000	09:00 AM	ear ache	Scheduled	2025-11-25 16:37:04.045
18	100,009	10,002	102	2025-11-26 00:00:00.000	03:00 PM	brain damage	Scheduled	2025-11-25 16:40:10.830
19	100,011	10,001	102	2025-11-27 00:00:00.000	03:30 PM	idk	Completed	2025-11-25 16:42:11.329
20	100,013	10,001	102	2025-11-26 00:00:00.000	03:30 PM	adas	Completed	2025-11-25 16:49:59.713
21	100,020	10,002	102	2025-11-26 00:00:00.000	04:00 PM	test	Completed	2025-11-25 17:02:28.942

Figure 34: Appointments Table Data

4.7.7 Patients Table Data

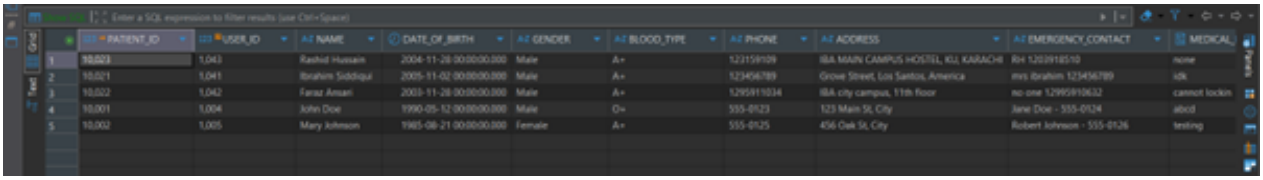


Figure 35: Patients Table Data

4.7.8 Departments Table Data

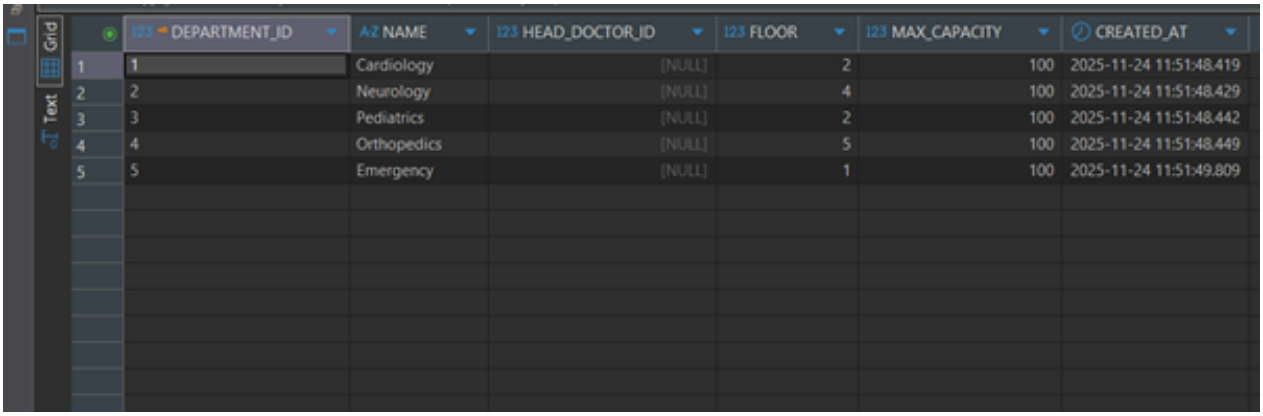


Figure 36: Departments Table Data

All tables were thoroughly checked to ensure there were no mismatches and data was correctly linked through primary and foreign key connections.

5 Application Flow

Below is a flow diagram that shows how users interact with the various components and functionalities in our application.

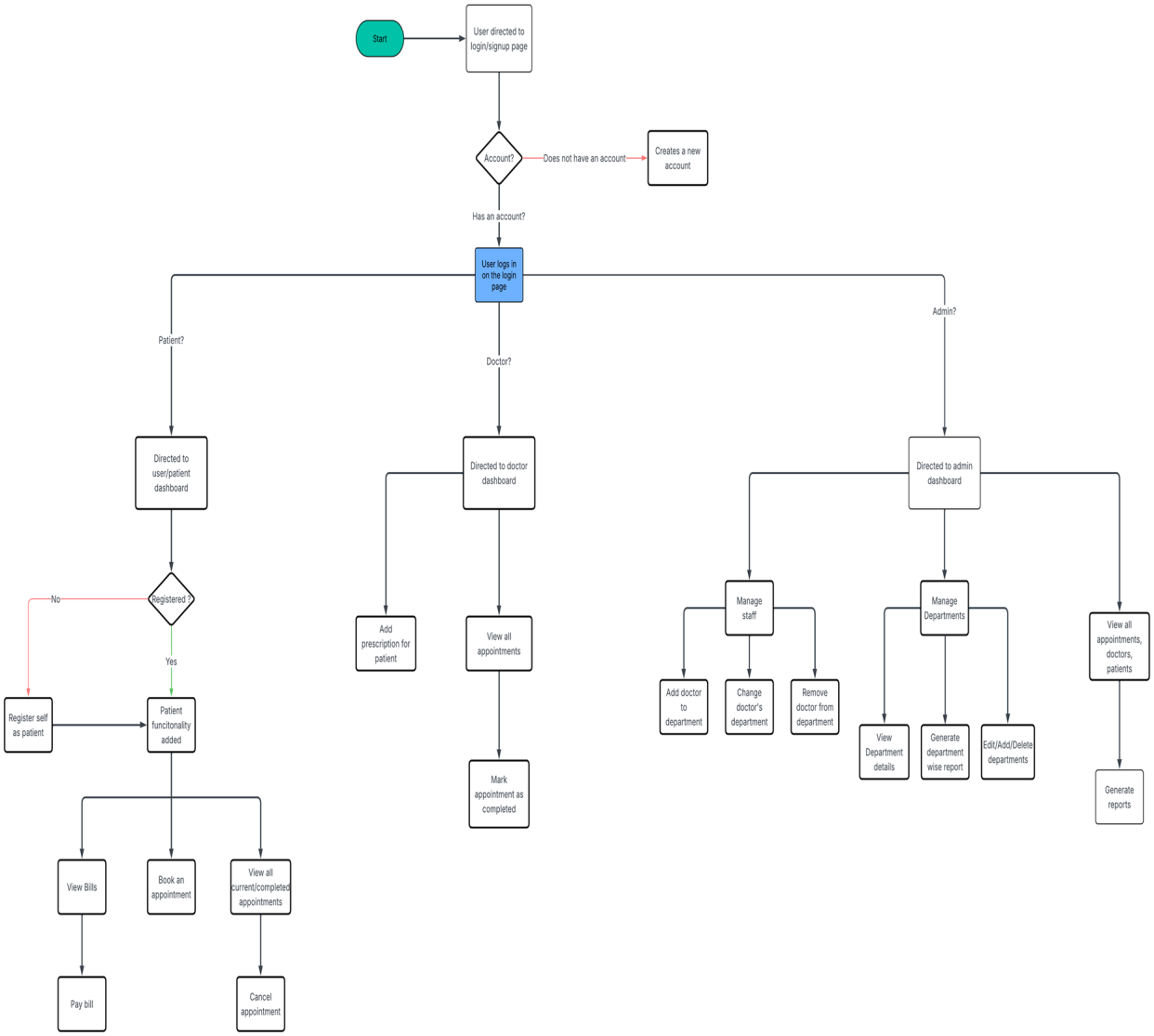


Figure 37: Application Flow Diagram

6 Page-by-Page Navigation & SQL Queries

6.1 Overview

Given below are the API calls, SQL queries, components and navigation for each of the 6 pages in our hospital DBMS project.

6.2 Page 1: Landing Page

6.2.1 Purpose

The landing page serves as the entry point and marketing page for the Medicare DBMS application. It showcases features, benefits, and buttons for user registration/login.

6.2.2 API Calls

None - This is a static informational page with no API calls.

6.2.3 SQL Queries

None - This page doesn't fetch or manipulate any database data.

6.2.4 Connection to System

This page serves as the entry point that directs users to:

- `/login` - For existing users
- `/signup` - For new user registration

6.3 Page 2: Login Page

6.3.1 Purpose

Authentication gateway that validates user credentials against the database and routes users to role-specific dashboards. User type dropdown allows the user to select if they are signing in as doctor, patient or admin.

6.3.2 API Endpoint

POST `/api/auth/login` (from `authAPI.login()` in frontend)

6.3.3 SQL Queries Executed

```
`SELECT u.user_id, u.email, u.user_type, u.password,
        CASE u.user_type
          WHEN 'doctor' THEN d.name
          WHEN 'patient' THEN p.name
          ELSE 'Admin'
        END as name,
        CASE u.user_type
          WHEN 'doctor' THEN d.doctor_id
          WHEN 'patient' THEN p.patient_id
          ELSE NULL
        END as profile_id
 FROM users u
 LEFT JOIN doctors d ON u.user_id = d.user_id
 LEFT JOIN patients p ON u.user_id = p.user_id
 WHERE u.email = :email AND u.password = :password AND u.user_type = :userType AND u.is_active = 1`,
```

Figure 38: Login Authentication Query

Purpose: This primary authentication query validates user credentials while simultaneously retrieving role-specific information. It joins the users table with both doctors and patients tables using LEFT JOINs to handle different user types. The CASE statements dynamically extract the appropriate name and profile ID based on the user's role. The WHERE clause verifies email, password, user type, and ensures the account is active, providing comprehensive authentication in a single database operation.

```
`SELECT email, user_type FROM users WHERE email = :email`,
```

Figure 39: User Existence Check Query

Purpose: This secondary query executes only when authentication fails. Its purpose is to help diagnose login issues by checking if a user exists with the provided email address. This allows the system to differentiate between "invalid password" and "user not found" scenarios, aiding in debugging while maintaining security by not revealing specific error details to end users.

```
`SELECT u.user_id, u.email, u.password, u.user_type,
        d.name as doctor_name, p.name as patient_name
 FROM users u
 LEFT JOIN doctors d ON u.user_id = d.user_id
 LEFT JOIN patients p ON u.user_id = p.user_id
 ORDER BY u.user_id`,
```

Figure 40: Debug Users Query

Purpose: This query supports the debug endpoint that displays all users in the system. It provides a comprehensive view of user data across all tables, showing email credentials, user types, and associated doctor or patient names. This is used to verify database state and troubleshoot authentication issues during system testing.

6.4 Page 3: Signup Page

6.4.1 Purpose

Registration portal for new patients to create accounts in the hospital management system.

6.4.2 API

POST `/api/auth/signup` - Called when patient submits registration form

```
`BEGIN
    INSERT INTO users (user_id, email, password, user_type)
    VALUES (seq_users.NEXTVAL, :email, :password, :userType);
 END;`,
```

Figure 41: User Account Creation Query

Purpose: This PL/SQL block creates the base user account by inserting a new record into the users table. It uses the sequence generator `seq_users.NEXTVAL` to automatically generate a unique user ID. The query stores the patient's email, password, and sets their user_type to 'patient', establishing their authentication credentials in the system.

```
`BEGIN
    INSERT INTO patients (patient_id, user_id, name, date_of_birth, gender, phone)
    VALUES (seq_patients.NEXTVAL, seq_users.CURRVAL, :name, :dob, :gender, :phone);
 END;`,
```

Figure 42: Patient Profile Creation Query

Purpose: This second PL/SQL block creates the detailed patient profile record. It utilizes `seq_patients.N` for a unique patient ID and references `seq_users.CURRVAL` to link to the newly created user account. The query stores personal information including name, calculated date of birth (derived from age), gender, and contact phone number. This separates demographic data from authentication data, following database normalization principles.

6.5 Page 4: Patient Dashboard

6.5.1 Purpose

Central hub where patients can view their medical information, manage appointments, pay bills, and update personal details.

6.5.2 API

GET `/api/patients/:id/debug` - Loads patient profile data

There are four total tabs in this dashboard which are the Overview, Appointments, Book Appointments and Bills tabs. Each tabs have their own buttons, and corresponding SQL queries.

6.5.3 Query 1: Patient Profile Retrieval

```
`SELECT p.patient_id, p.name, p.date_of_birth, p.gender, p.blood_type,
        p.phone, p.address, p.emergency_contact,
        NVL(DBMS_LOB.SUBSTR(p.medical_history, 4000, 1), '') as medical_history, -- CLOB FIX
        u.email
FROM patients p
JOIN users u ON p.user_id = u.user_id
WHERE p.patient_id = :id`,
```

Figure 43: Patient Profile Retrieval Query

Purpose: This query retrieves the complete patient profile by joining the patients table with the users table. It uses the `DBMS_LOB.SUBSTR` function to safely convert CLOB data (medical_history) to a readable string format. The query returns all essential patient information including personal details, contact information, and medical history for display on the dashboard overview.

6.5.4 Query 2: Appointment Statistics

```
let query = `
  SELECT
    a.appointment_id,
    a.appointment_date,
    a.appointment_time,
    a.reason,
    a.status,
    p.name as patient_name,
    p.patient_id,
    p.date_of_birth,
    p.gender,
    NVL(DBMS_LOB.SUBSTR(p.medical_history, 4000, 1), '') as medical_history,
    d.name as doctor_name,
    d.specialization,
    dep.name as department,
    FLOOR(MONTHS_BETWEEN(SYSDATE, p.date_of_birth) / 12) as age
  FROM appointments a
  JOIN patients p ON a.patient_id = p.patient_id
  JOIN doctors d ON a.doctor_id = d.doctor_id
  JOIN departments dep ON d.department_id = dep.department_id
  WHERE 1=1
`;

const binds = {};

if (doctorId) {
  query += ` AND d.doctor_id = :doctorId`;
  binds.doctorId = parseInt(doctorId);
}

if (patientId) {
  query += ` AND p.patient_id = :patientId`;
  binds.patientId = parseInt(patientId);
}

query += ` ORDER BY a.appointment_date DESC, a.appointment_time DESC`;
```

Figure 44: Appointment Statistics Query

Purpose: This comprehensive query fetches all appointments for the patient while calculating their age dynamically. It joins four tables (appointments, patients, doctors, departments) to provide complete appointment details including doctor information, department, and status. The results are used to populate the statistics cards showing current appointments, pending bills, and total visits on the dashboard.

6.5.5 Query 3: Bill Retrieval

```
let query = `
  SELECT b.bill_id, b.amount, b.payment_status, b.generated_date, b.paid_date,
         a.appointment_date, a.appointment_time,
         pat.name as patient_name, pat.patient_id,
         doc.name as doctor_name, doc.specialization
  FROM bills b
  JOIN appointments a ON b.appointment_id = a.appointment_id
  JOIN patients pat ON a.patient_id = pat.patient_id
  JOIN doctors doc ON a.doctor_id = doc.doctor_id
`;

const binds = {};

if (patientId) {
  query += ` WHERE pat.patient_id = :patientId`;
  binds.patientId = parseInt(patientId);
}

query += ` ORDER BY b.generated_date DESC`;
```

Figure 45: Bill Retrieval Query

Purpose: This query retrieves all bills associated with a patient by joining the bills, appointments, patients, and doctors tables. It provides billing information along with appointment details and doctor information. The payment_status field determines whether a bill is shown as "Pending" or "Paid" on the dashboard, and the amount is used for calculating the total amount due in the payment summary section.

6.5.6 Query 4: Patient Info Update

```
`UPDATE patients
SET name = :name, date_of_birth = TO_DATE(:dateOfBirth, 'YYYY-MM-DD'), gender = :gender,
    blood_type = :bloodType, phone = :phone, address = :address,
    emergency_contact = :emergencyContact, medical_history = :medicalHistory
WHERE patient_id = :id`,
```

Figure 46: Patient Info Update Query

Purpose: This UPDATE query allows patients to modify their personal information through the edit modal. It updates all editable fields in the patients table including personal details, contact information, and medical history. The TO_DATE function ensures proper date formatting for the date_of_birth field, maintaining data consistency in the database.

6.5.7 Tab 2: Appointments Management

```
`UPDATE appointments SET status = 'Completed' WHERE appointment_id = :id`,
```

Figure 47: Appointment Cancellation Query

Query 5: Appointment Cancellation Purpose: This simple UPDATE query changes the status of an appointment from "Scheduled" to "Cancelled". It enables patients to cancel upcoming appointments through the dashboard interface, maintaining a record of the cancellation in the database for audit purposes while freeing up the time slot.

```
`SELECT d.doctor_id, d.name, d.specialization, d.phone, d.license_number,
        d.experience_years, d.consultation_fee, d.status, d.user_id,
        dep.name as department_name, dep.department_id,
        u.email
 FROM doctors d
 JOIN departments dep ON d.department_id = dep.department_id
 LEFT JOIN users u ON d.user_id = u.user_id
 ORDER BY d.name`,
```

Figure 48: Doctor List Retrieval Query

Query 6: Doctor List Retrieval Purpose: This query fetches all available doctors for the appointment booking form. It joins doctors with their departments and user accounts to provide complete information including specialization, department, contact details, and consultation fees. The results populate the dropdown selection allowing patients to choose their preferred doctor when booking new appointments.

6.5.8 Tab 3: Book Appointments

```
`INSERT INTO appointments (appointment_id, patient_id, doctor_id, appointment_date, appointment_time, reason, status)
 VALUES (SEQ_APPOINTMENTS.NEXTVAL, :patientId, :doctorId, TO_DATE(:apptDate, 'YYYY-MM-DD'), :time, :reason, 'Scheduled')
 RETURNING appointment_id INTO :appointmentId`,
```

Figure 49: Appointment Booking Query

Query 7: Appointment Booking Purpose: This INSERT query creates a new appointment record using the sequence generator for unique IDs. It stores the patient ID, doctor ID, date, time, and reason for the appointment, setting the initial status to "Scheduled". The RETURNING clause captures the newly created appointment ID to confirm successful booking and provide reference for any follow-up operations.

```
`SELECT
    p.prescription_id,
    NVL(DBMS_LOB.SUBSTR(p.diagnosis, 4000, 1), '') as diagnosis,
    NVL(DBMS_LOB.SUBSTR(p.notes, 4000, 1), '') as notes,
    TO_CHAR(p.created_at, 'YYYY-MM-DD HH24:MI:SS') as created_at,
    TO_CHAR(a.appointment_date, 'YYYY-MM-DD') as appointment_date,
    a.appointment_time,
    pat.name as patient_name,
    pat.patient_id,
    TO_CHAR(pat.date_of_birth, 'YYYY-MM-DD') as date_of_birth,
    pat.gender,
    doc.name as doctor_name,
    doc.specialization
FROM prescriptions p
JOIN appointments a ON p.appointment_id = a.appointment_id
JOIN patients pat ON a.patient_id = pat.patient_id
JOIN doctors doc ON a.doctor_id = doc.doctor_id
WHERE p.appointment_id = :appointmentId`,
```

Figure 50: Prescription Retrieval Query

Query 8: Prescription Retrieval Purpose: This query retrieves prescription details by joining the prescriptions table with appointments, patients, and doctors. It converts CLOB fields (diagnosis, notes) to readable strings using DBMS_LOB.SUBSTR and formats dates appropriately. Patients can view their prescribed medications, dosage instructions, and doctor's notes for completed appointments through this functionality.

```
`SELECT medicine_name, dosage, duration, instructions
FROM prescription_medicines
WHERE prescription_id = :prescriptionId
ORDER BY medicine_id`,
```

Figure 51: Prescription Medicines Retrieval Query

Query 9: Prescription Medicines Retrieval Purpose: This complementary query fetches the individual medicines prescribed for a particular prescription. It retrieves medicine names, dosages, durations, and special instructions, providing detailed medication information that appears in the prescription viewing modal when patients click "View Prescription" on past appointments.

6.5.9 Tab 4: Bills & Payments

```
`UPDATE bills
 SET payment_status = 'Paid', paid_date = CURRENT_TIMESTAMP
 WHERE bill_id = :id`,
```

Figure 52: Bill Payment Processing Query

Query 10: Bill Payment Processing **Purpose**: This UPDATE query marks a bill as paid by updating the payment_status to 'Paid' and setting the paid_date to the current timestamp. It enables patients to pay their medical bills directly through the dashboard, providing immediate confirmation of payment and updating the payment status visible in the bills table.

6.6 Page 5: Admin Dashboard

6.6.1 Purpose

Administrative interface for managing hospital departments, doctors, and viewing patient records with comprehensive CRUD operations.

6.6.2 Tab 1: Overview Dashboard

```
`SELECT
    d.name as department_name,
    COUNT(DISTINCT doc.doctor_id) as doctor_count,
    COUNT(DISTINCT p.patient_id) as patient_count,
    COUNT(a.appointment_id) as appointment_count,
    NVL(SUM(CASE WHEN b.payment_status = 'Paid' THEN b.amount ELSE 0 END), 0) as total_revenue
  FROM departments d
  LEFT JOIN doctors doc ON d.department_id = doc.department_id
  LEFT JOIN appointments a ON doc.doctor_id = a.doctor_id
  LEFT JOIN patients p ON a.patient_id = p.patient_id
  LEFT JOIN bills b ON a.appointment_id = b.appointment_id
  WHERE d.department_id = :id
  GROUP BY d.name`,
```

Figure 53: Department Statistics Retrieval Query

Query 1: Department Statistics Retrieval **Purpose**: This comprehensive query calculates department statistics including staff count (doctors), patient count, and identifies department heads. It uses LEFT JOINs to connect departments with doctors and patients through appointments, then aggregates the data with GROUP BY to provide summary statistics for the admin dashboard overview. The results populate the department statistics section and contribute to the dashboard's key metrics.

```
`SELECT d.doctor_id, d.name, d.specialization, d.phone, d.license_number,
        d.experience_years, d.consultation_fee, d.status, d.user_id,
        dep.name as department_name, dep.department_id,
        u.email
 FROM doctors d
 JOIN departments dep ON d.department_id = dep.department_id
 LEFT JOIN users u ON d.user_id = u.user_id
 ORDER BY d.name`,
```

Figure 54: Doctor List Retrieval Query

Query 2: Doctor List Retrieval Purpose: This query retrieves all doctors along with their department information and user account details. It joins the doctors table with departments to get department names and with users to get email addresses. The results are used to display the "Recent Doctors" section, calculate active doctor counts for statistics, and provide data for the doctor management tab.

```
`SELECT p.patient_id, p.name, p.date_of_birth, p.gender, p.blood_type,
        p.phone, p.address, p.emergency_contact,
        NVL(DBMS_LOB.SUBSTR(p.medical_history, 4000, 1), '') as medical_history, -- CLOB FIX
        u.email
 FROM patients p
 JOIN users u ON p.user_id = u.user_id
 ORDER BY p.name`,
```

Figure 55: Patient List Retrieval Query

Query 3: Patient List Retrieval Purpose: This query retrieves all patient records from the database, joining the patients table with the users table to get email addresses. It uses DBMS_LOB.SUBSTR to safely handle CLOB data in the medical_history field. The results provide the total patient count for dashboard statistics and populate the patient records tab with complete patient information.

6.6.3 Tab 2: Department Management

```
`INSERT INTO departments (department_id, name, floor, max_capacity, head_doctor_id, created_at)
 VALUES (seq_departments.NEXTVAL, :name, :floor, :maxCapacity, :headDoctorId, CURRENT_TIMESTAMP)`,
```

Figure 56: Department Creation Query

Query 4: Department Creation Purpose: This INSERT query creates a new department record using the sequence generator for unique IDs. It stores the department name, floor location, maximum capacity, and optionally assigns a head doctor. This enables administrators to expand hospital operations by adding new departments through the admin interface.

```
`UPDATE departments
SET name = :name,
    floor = :floor,
    max_capacity = :maxCapacity,
    head_doctor_id = :headDoctorId
WHERE department_id = :id`,
```

Figure 57: Department Update Query

Query 5: Department Update **Purpose**: This UPDATE query modifies existing department information. It allows administrators to update department details such as name, floor location, capacity, and head doctor assignment. This flexibility supports organizational changes and department restructuring as the hospital evolves.

```
"DELETE FROM departments WHERE department_id = :id",
```

Figure 58: Department Deletion Query

Query 6: Department Deletion **Purpose**: This DELETE query removes a department from the system. It enables administrators to reorganize hospital structure by eliminating departments that are no longer needed. The frontend includes confirmation prompts to prevent accidental deletions, and foreign key constraints would ensure referential integrity is maintained.

```
`INSERT INTO users (user_id, email, password, user_type, is_active, created_at)
VALUES (seq_users.NEXTVAL, :email, :password, :userType, :isActive, CURRENT_TIMESTAMP)`,
```

Figure 59: User Account Creation for Doctor Query

Query 7: User Account Creation for Doctor **Purpose**: This INSERT query creates a user account for the new doctor, storing authentication credentials and setting the user_type to 'doctor'. This establishes the foundation for the doctor's login access and integrates them into the authentication system alongside patients and other users.

```
`INSERT INTO doctors (doctor_id, user_id, department_id, name, specialization, license_number, phone, experience_years, consultation_fee, status, created_at)
VALUES (seq_doctors.NEXTVAL, :userId, :departmentId, :name, :specialization, :licenseNumber, :phone, :experience, :consultationFee, :status, CURRENT_TIMESTAMP)`,
```

Figure 60: Doctor Profile Creation Query

Query 8: Doctor Profile Creation **Purpose**: This INSERT query creates the doctor's professional profile after their user account is established. It stores all professional details including specialization, license number (which has a unique constraint), department assignment, experience, consultation fees, and status. The license_number validation prevents duplicate licenses in the system.

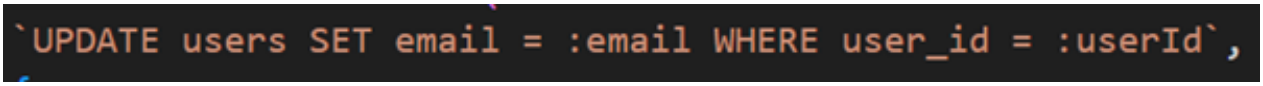


Figure 61: User Email Update for Doctor Query

Query 9: User Email Update for Doctor **Purpose**: This UPDATE query modifies the doctor's email address in the users table when their contact information is updated. Since email is part of the authentication system stored in the users table, this ensures login credentials remain synchronized with profile information.

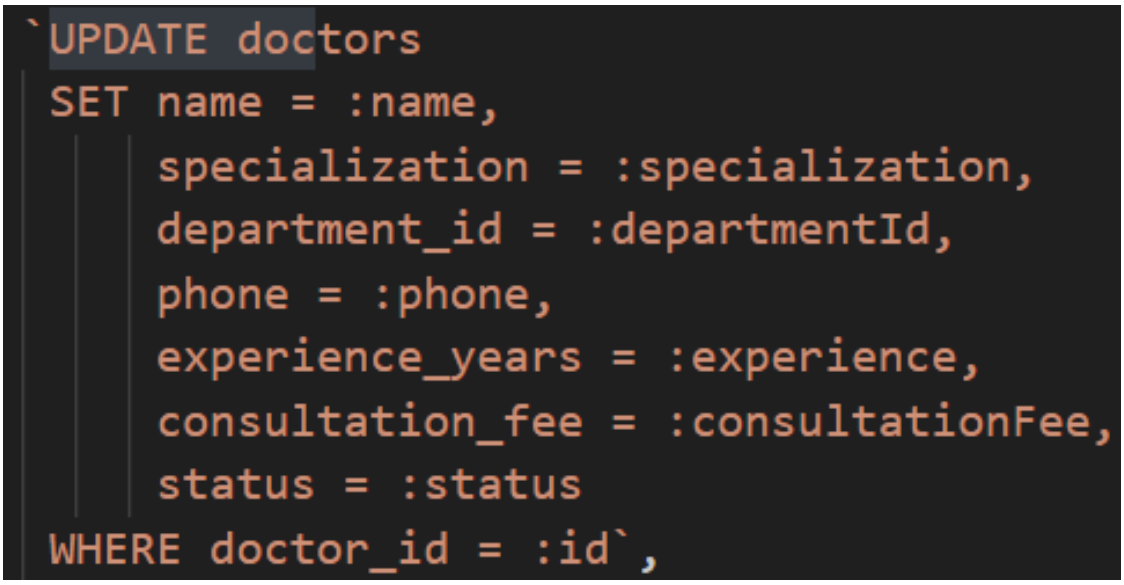


Figure 62: Doctor Profile Update Query

Query 10: Doctor Profile Update **Purpose**: This UPDATE query modifies the doctor's professional information including name, specialization, department assignment, contact details, experience, fees, and status. It allows administrators to update doctor profiles as their roles, departments, or statuses change within the hospital.

Figure 63: Doctor Deletion Query

Query 11: Doctor Deletion **Purpose**: This DELETE query removes a doctor's professional record from the system. The frontend includes confirmation prompts to prevent accidental deletions. Note that this only deletes the doctor record, not their user account, maintaining referential integrity with other tables that might reference the doctor.

Query 12: Patient Lookup **Purpose**: Same as query 3, provides administrators with a comprehensive view of all registered patients, displaying key information including name, age, gender, contact details, and blood type in a sortable, filterable table format.

6.7 Page 6: Doctor Dashboard

6.7.1 Purpose

An interface for doctors to view appointments, complete consultations, and create prescriptions for patients.

6.7.2 Query 1: Appointment Retrieval for Doctor

```
  SELECT
    a.appointment_id,
    a.appointment_date,
    a.appointment_time,
    a.reason,
    a.status,
    p.name as patient_name,
    p.patient_id,
    p.date_of_birth,
    p.gender,
    NVL(DBMS_LOB.SUBSTR(p.medical_history, 4000, 1), '') as medical_history,
    d.name as doctor_name,
    d.specialization,
    dep.name as department,
    FLOOR(MONTHS_BETWEEN(SYSDATE, p.date_of_birth) / 12) as age
  FROM appointments a
  JOIN patients p ON a.patient_id = p.patient_id
  JOIN doctors d ON a.doctor_id = d.doctor_id
  JOIN departments dep ON d.department_id = dep.department_id
  WHERE 1=1
`;

const binds = {};

if (doctorId) {
  query += ` AND d.doctor_id = :doctorId`;
  binds.doctorId = parseInt(doctorId);
}

if (patientId) {
  query += ` AND p.patient_id = :patientId`;
  binds.patientId = parseInt(patientId);
}

query += ` ORDER BY a.appointment_date DESC, a.appointment_time DESC`;
```

Figure 64: Appointment Retrieval for Doctor Query

Purpose: This comprehensive query retrieves all appointments for a specific doctor by joining four tables: appointments, patients, doctors, and departments. It calculates the patient's age dynamically using MONTHS_BETWEEN and includes the patient's medical history converted from CLOB format. The results provide doctors with complete patient information including personal details, appointment specifics, and medical history to prepare for consultations.

6.7.3 Query 2: Doctor Profile Retrieval

```
`UPDATE appointments SET status = 'Completed' WHERE appointment_id = :id`,
```

Figure 65: Appointment Completion Query

Purpose: This UPDATE query marks an appointment as "Completed" when a doctor finishes a consultation. It changes the status from "Scheduled" to "Completed", triggering the prescription creation workflow. This status change also affects the appointment categorization in both doctor and patient dashboards, moving appointments from upcoming to completed sections.

6.7.4 Query 3: Prescription Creation

```
`INSERT INTO prescriptions (prescription_id, appointment_id, diagnosis, notes)
VALUES (seq_prescriptions.NEXTVAL, :appointmentId, :diagnosis, :notes)
RETURNING prescription_id INTO :prescriptionId`,
```

Figure 66: Prescription Creation Query

Purpose: This INSERT query creates a new prescription record in the database after an appointment is completed. It uses the sequence generator for unique prescription IDs and stores the diagnosis and notes provided by the doctor. The RETURNING clause captures the newly created prescription ID, which is essential for linking the prescription to specific medicines in subsequent queries.

6.7.5 Query 4: Prescription Medicines Insertion

```
`INSERT INTO prescription_medicines (medicine_id, prescription_id, medicine_name, dosage, duration, instructions)
VALUES (seq_medicines.NEXTVAL, :prescriptionId, :name, :dosage, :duration, :instructions)`,
```

Figure 67: Prescription Medicines Insertion Query

Purpose: This INSERT query adds individual medicines to a prescription, storing details like medicine name, dosage instructions, duration of treatment, and special instructions. Each medicine gets its own unique ID via the sequence generator, and all medicines are linked to the parent prescription through the prescription_id foreign key, creating a one-to-many relationship.

6.7.6 Tab 2: Completed Appointments

```
`SELECT
    p.prescription_id,
    NVL(DBMS_LOB.SUBSTR(p.diagnosis, 4000, 1), '') as diagnosis,
    NVL(DBMS_LOB.SUBSTR(p.notes, 4000, 1), '') as notes,
    TO_CHAR(p.created_at, 'YYYY-MM-DD HH24:MI:SS') as created_at,
    TO_CHAR(a.appointment_date, 'YYYY-MM-DD') as appointment_date,
    a.appointment_time,
    pat.name as patient_name,
    pat.patient_id,
    TO_CHAR(pat.date_of_birth, 'YYYY-MM-DD') as date_of_birth,
    pat.gender,
    doc.name as doctor_name,
    doc.specialization
FROM prescriptions p
JOIN appointments a ON p.appointment_id = a.appointment_id
JOIN patients pat ON a.patient_id = pat.patient_id
JOIN doctors doc ON a.doctor_id = doc.doctor_id
WHERE p.appointment_id = :appointmentId`,
```

Figure 68: Prescription Details Retrieval Query

Query 5: Prescription Details Retrieval **Purpose**: This query retrieves complete prescription details by joining the prescriptions table with appointments, patients, and doctors. It converts CLOB fields (diagnosis, notes) to readable strings and formats dates appropriately. The result provides doctors with a comprehensive view of past prescriptions including patient information, appointment details, and prescription content for reference or follow-up consultations.

```
`SELECT medicine_name, dosage, duration, instructions
FROM prescription_medicines
WHERE prescription_id = :prescriptionId
ORDER BY medicine_id`,
```

Figure 69: Prescription Medicines Retrieval Query

Query 6: Prescription Medicines Retrieval **Purpose**: This supporting query fetches all medicines associated with a specific prescription. It retrieves the medication details including name, dosage, duration, and any special instructions. This data populates the medicines section in the prescription viewing modal, allowing doctors to review previously prescribed treatments and maintain continuity of care.

7 Work Contribution and Team Collaboration

Table 3: Team Work Distribution and Contributions

Task	**Member**
Planning, design, ERDs	Both
Backend + Database	Faraz Ansari
Frontend + wireframes/design	Ashhal Aamir
Integrations + Debugging + final	Both