Hospital Management System

Manasi Jadhav
Computer Science and Engineering
University at Buffalo
Buffalo, USA
mjadhav@buffalo.edu

Sneha Singh
Computer Science and Engineering
University at Buffalo
Buffalo, USA
singh43@buffalo.edu

Sneha Yadav
Computer Science and Engineering
University at Buffalo
Buffalo, USA
snehayad@buffalo.edu

I. Problem Statement

Title:

Enhancing Healthcare Management by development of a Comprehensive Hospital Management System (HMS) Database.

Problem Statement:

The healthcare industry faces increasing challenges in managing vast amounts of patient data, medical records, staff information, appointment schedules, and pharmacy stocks efficiently. As medical facilities expand and the volume of patient interactions grows, traditional management methods, such as Excel spreadsheets, are proving inadequate. This project addresses the critical need for an advanced Hospital Management System (HMS) database, outlining why a transition away from Excel-based management is not just beneficial but imperative for modern healthcare institutions.

Background and Significance of the Problem:

The current healthcare landscape is characterized by a staggering growth of patient data, necessitating robust management solutions. Excel spreadsheets, the erstwhile staple of data management, fall short in several key areas:

- Data Integrity and Consistency: Excel cannot easily enforce data validation rules, leading to potential errors and inconsistencies that can compromise patient care and record-keeping.
- Real-time Data Access and Updates: In a hospital setting, where conditions change rapidly, the inability of Excel to support real-time data access can hinder responsive care and delay critical decision-making.
- Concurrent Access: Excel's limitations in supporting multiple concurrent users can lead to bottlenecks, where data is siloed or overwritten, and collaborative updates are hampered.
- Complex Queries Handling: Excel's capabilities are limited in executing complex queries essential for extracting insights from large datasets, which is necessary for informed decision-making.
- Data Sharing and Security: Excel lacks sophisticated controls for data sharing and access rights, posing risks to the sensitive nature of patient information and impeding compliance with healthcare data protection regulations.

These limitations are magnified in the context of growing regulatory demands for data privacy, such as HIPAA in the United States, and the increasing expectation for personalized patient care. The hospital's ability to provide high-quality care is intrinsically linked to its capability to manage data efficiently and securely.

Potential Contribution of the HMS Database Project:

The proposed HMS database stands to revolutionize healthcare management by introducing a system that is capable of:

- Ensuring Robust Data Integrity: By utilizing structured database schemas, the HMS can enforce data validation and integrity constraints, eliminating errors that can lead to patient risks.
- Enabling Real-time Data Synchronization: A database system can handle simultaneous transactions, ensuring that all users have access to up-to-date information.
- Facilitating Complex Data Analysis: With advanced query capabilities, healthcare providers can extract meaningful patterns and insights, informing both operational decisions and patient care strategies.
- Streamlining Operations: Automating scheduling, inventory management, and record-keeping reduces administrative burdens and redirects resources to patient care.
- Meeting Regulatory Compliance: A database system can be designed to comply with data privacy regulations, with built-in security measures for access control and data encryption.

Implementing the HMS database will directly contribute to the improved efficiency of hospital operations, significantly reduce the possibility of errors, and enhance patient outcomes. This project is particularly crucial in a landscape where digital health records are the cornerstone of medical informatics. By shifting to a dedicated HMS database, hospitals can ensure scalability, adaptability, and resilience in their data management systems, all of which are vital in an era where healthcare data is not only a record but also a tool for predictive analytics and personalized medicine.

The proposed HMS database is not just an incremental improvement over existing systems; it represents a transformative step towards integrating technology deeply into healthcare management. It will allow hospitals to leverage data science and machine learning algorithms for

predictive healthcare, enhance patient engagement through mobile health data accessibility, and facilitate seamless integration with emerging telemedicine platforms. Implementing a hospital management system database can significantly contribute to streamlining hospital operations, improving patient care, and enhancing decision-making through real-time data analytics. This project's contribution is crucial in promoting technological advancements in healthcare management, leading to increased operational efficiency, reduced errors, and improved patient outcomes.

Along with these perks, in future if the need for scalability arises, this system will be up for the challenge with minimal modifications and data preservation.

Ultimately, the contribution of this project transcends operational efficiencies—it positions healthcare providers to meet the future demands of an increasingly data-driven medical environment, where the quality of patient care and outcomes can be significantly uplifted through technological empowerment.

II. Target User

There are several categories of users who will have a vested interest in this hospital management system. Some of these are as given:

A. *Primary Users*

- Healthcare Providers (Doctors, Nurses, Lab Technicians): They will use the database for accessing and updating patient records, scheduling appointments, and recording lab results. They can also use it to view lab reports and monitor the health status of their patients. To check their own schedules and manage their workload.
- Hospital Administrators: They will utilize the system for managing staff schedules, overseeing hospital departments, and monitoring hospital performance.
- Pharmacists: They will use the database to manage inventory, track prescriptions, and update stock levels.
- Receptionist: They will use the database to register new patients and create initial patient records. They can use it to schedule and confirm appointments and send reminders.

B. *Secondary Users*

- Patients: Through a patient portal, they can access their medical history, upcoming appointments, and test results.

C. *Database Administrator (DBA)*

- Database Administrators (DBAs): They will be responsible to ensure the database is running efficiently, performing regular maintenance, applying updates, and managing backups. They will make sure that the database is enforcing data security measures, managing user access levels, and protecting sensitive health information in compliance with healthcare regulations like HIPAA. They will troubleshoot and resolve any issues that arise within the database.
- IT Staff/Database Professionals: They will be responsible for the database's technical administration, ensuring its integrity, performance, security, and availability. They will also handle database updates, backups, and user management.
- Data Analysts: They will analyze data from the database for reports on hospital efficiency, patient outcomes, and other metrics important for hospital administration and improvement.

Real-life Scenario: -

Imagine a hospital named "Buffalo Health Hospital," which uses this database to coordinate with patients and manage the hospital operations. When a patient visits for the first time, the receptionist enters their personal information into the Patient table and assigns them a unique PatientID. The patient's address and phone number are entered into the PatientAddress and PhoneNumber tables, respectively. If a patient is booking an appointment online via portal, then he will have to fill in these details by himself in a sign up page.

As the patient goes through the healthcare process, doctors, nurses, lab technicians and other related entities update the MedicalHistory, LabReports, and Prescription tables with relevant information from each visit. Patients engage with the database indirectly through a patient portal, where they can review their medical history, lab reports, and future appointments. This transparency allows patients to trust and encourages patient involvement in their healthcare.

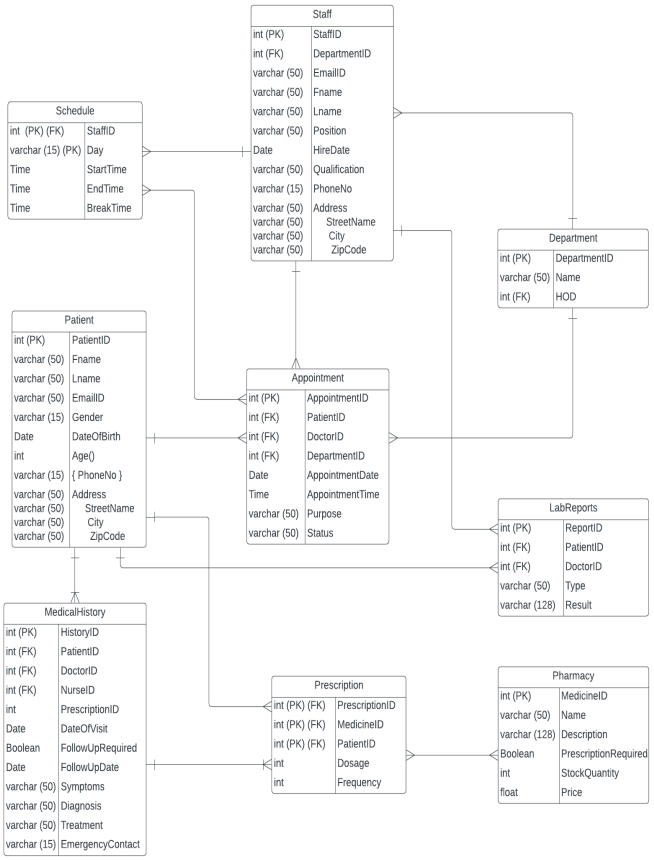
The pharmacy uses the Pharmacy table to track medication inventory and validate prescriptions. In the pharmacy, pharmacists verify prescriptions using the Pharmacy and Prescription tables. They maintain accurate records of medications dispensed to prevent errors and ensure patient safety. Stock levels are closely monitored, with automatic alerts for restocking set up based on data from the StockQuantity attribute.

DBAs at Buffalo Health Hospital monitor and maintain the database, ensuring its availability, integrity, and security. They work closely with IT support staff to address any technical issues and with data analysts who use the database information to improve hospital services and management.

Doctors and Nurses also can use schedules tables to check their schedules and accept the appointments accordingly. Department heads access the HOD attribute in the Department table to monitor their departments' performance, streamline operations, and address any issues raised by both staff and patients.

With this dependency on the database, the database becomes the backbone of Buffalo Health Hospital's operations, supporting every step of the healthcare process and enabling a high standard of patient care. It also provides many facilities to the primary user of the database.

III. E/R Diagram



Cardinalities: -

Patient to MedicalHistory: The cardinality here is one-to-many. A patient can have one or more medical history records, but each medical history record pertains to one patient.

MedicalHistory to Prescription: The cardinality here can be one-to-many. Each medical history entry might result in one or many prescriptions.

Prescription to Pharmacy: The cardinality is many-to-many. Many prescriptions can be written for one medication, similarly many medications can be written in a single prescription.

Prescription to Patient: The cardinality is many-to-one. Many prescriptions can be written for a single patient, but each prescription will be associated with a single patient.

Patient to LabReports: The cardinality is one-to-many. A patient can have multiple lab reports, but each lab report will be associated with a single patient.

Patient to Appointment: The cardinality is one-to-many. A patient can have multiple appointments, but each appointment will be associated with a single patient.

Staff (as Doctor) to LabReports: The cardinality is one-to-many. A doctor can be responsible for many lab reports, each potentially from different patients.

Staff to Department: The cardinality is many-to-one. Many staff members can work in one department.

Department to Appointment: The cardinality is one-to-many. A department can have many appointments scheduled within it.

Staff (as Doctor) to Appointment: The cardinality is one-to-many. A doctor (staff member) can have multiple appointments with different patients.

Staff to Schedule: The cardinality is one-to-many as each staff will have multiple schedules designed for different days.

Schedule to Appointment: The cardinality is many-to-many as multiple schedules can be on the same day as appointment and multiple appointments can be scheduled on the same day.

IV. Tasks

A. *Task 1*

We have decided to choose healthcare as our use case domain for our project. This has always been one of the major industries which will now and in future be needed by everyone. The design of a comprehensive hospital management system is extremely necessary to facilitate a smooth and efficient working environment with minimal errors and problems for patients as well as the staff and management. This system will help with the current data along with maintaining scope for scalability in the future.

The data for the patient table and appointment table will be brought in with the help of a user application where they can register as patients and book appointments. We plan to use queries to check with the doctors schedules during appointment booking. Along with this, lab reports can be checked using queries to extract data of a patient. Prescriptions can be searched from the patient's medical history and each medicine can be checked for availability in the pharmacy using queries.

B. *Task 2*

Using our problem statement, we have decided upon the tables shown in the ER diagram displayed earlier. This diagram displays the tables and their attributes along with the cardinality between relations. This ER model is then converted to relational schema for further decomposition to facilitate more efficiency and reduce data redundancy and inconsistency.

The converted relational schema is given as:

- Patient (PatientID,Fname,Lname,EmailID,Gender, DateOfBirth,Age,PhoneNumber,Address)
- Staff (StaffID,DepartmentID,EmailID,Fname,Lname, Position,HireDate,Qualification,PhoneNo,Address)
- Schedule (StaffID,Day,StartTime,EndTime,BreakTime)
- Department (DepartmentID,Name,HOD)
- Appointment (AppointmentID,PatientID,DoctorID, DepartmentID,AppointmentDate,AppointmentTime,Purpose,Status)
- LabReports (ReportID,PatientID,DoctorID,Type,Result)
- MedicalHistory (HistoryID,PatientID,DoctorID,NurseID ,PrescriptionID,DateOfVisit,FollowUpRequired,Follow

UpDate,Symptoms,Diagnosis,Treatment,EmergencyContact)

- Pharmacy (MedicineID,Name,Description, PrescriptionRequired,StockQuantity,Price)

Prescription (PrescriptionID,MedicineID,PatientID,Dosage, Frequency)

Actions on foreign key: -

The data for the patient table can be updated simply from the user portal, and staff can have a separate portal for updating their own data. Rest other data where foreign keys are used get updated automatically. Upon deletion of records from a referencing table, we use ON DELETE CASCADE for automatically removing the records from certain tables while using ON DELETE NO ACTION for other tables like medical history where we need to preserve the medical history of patients even though they are not there any more.

C. Task 3

For this project we have decided to go forward with generating synthetic data using python libraries due to unavailability of a proper complete dataset as per our requirements. This generated data will closely resemble our requirements and the data types needed for our relations.

We will be using Python script to generate the necessary data and export it accordingly for initial testing in necessary tables. Later more data can be added to the tables directly using either SQL statements or user applications.

D. Task 4

Patient Table: -

PatientID → Fname, Lname, EmailID, Gender, DateOfBirth, Age

PatientID is the primary key and uniquely identifies each patient. Assuming Age is calculated from DateOfBirth, Age does not need to be stored in the database. The table is in BCNF because all attributes depend only on the primary key.

PatientPhoneNumber

{PatientID, PhoneNumber}

If PatientID and PhoneNumber is the primary key, the Patient table would not be in BCNF because it suggests that a patient can have a phone number, and having them in the same table with PatientID as the key violates the rule that every determinant must be a candidate key. A BCNF design would separate the phone numbers into their own records. That is why we made a separate PhoneNumber table to store those records. This table has a phone number and a patient id as a composite key.

PatientAddress

PatientID → StreetName, City, ZipCode

As PatientID is the primary key, that is why this table is in BCNF.

Staff

StaffID → DepartmentID, EmailID, Fname, Lname, Position, HireDate, Qualification

StaffID is the primary key. This table is in BCNF because all attributes depend on the primary key.

StaffAddress

StaffID → StreetName, City, ZipCode

As StaffID is the primary key, that is why this table is in BCNF.

StaffPhoneNumber

{StaffID, PhoneNumber}

If StaffID and PhoneNumber is the primary key, the Staff table would not be in BCNF because it suggests that a staff can have a phone number, and having them in the same table with StaffID as the key violates the rule that every determinant must be a candidate key. A BCNF design would separate the phone numbers into their own records. That is why we made a separate PhoneNumber table to store those records. This table has a phone number and a staff id as a composite key.

Schedule

{StaffID, Day} → StartTime, EndTime, BreakTime

This table is in BCNF as the composite key {StaffID, Day} is the primary key and all attributes are dependent on it.

Department

DepartmentID → Name, HOD

This table is in BCNF asDepartmentID is the primary key.

Appointment

AppointmentID → PatientID, DoctorID, DepartmentID, AppointmentDate, AppointmentTime, Purpose, Status

This table is in BCNF as AppointmentID is the primary key.

LabReports

ReportID → PatientID, DoctorID, Type, Result

This table is in BCNF as ReportID is the primary key.

MedicalHistory

HistoryID → PatientID, DoctorID, NurseID, DateOfVisit, FollowUpRequired, FollowUpDate, Symptoms, Diagnosis, Treatment, EmergencyContact

This table is in BCNF as HistoryID is the primary key.

Pharmacy

MedicineID → Name, Description, PrescriptionRequired, StockQuantity, Price

This table is in BCNF as MedicineID is the primary key.

Prescription

PrescriptionID → MedicineID, PatientID, Dosage, Frequency

This table is in BCNF as PrescriptionID is the primary key.

From these functional dependencies, we can see that above all the tables are in BCNF.

Cardinalities of Decomposed Tables: -

PatientAddress to Patient: The cardinality is one-to-one. A single address will be associated with a single patient record.

StaffAddress to Staff: The cardinality is one-to-one. A single address will be associated with a single staff record.

PhoneNumber to Patient: The cardinality is many-to-one. A single patient can have multiple phone numbers.

The final relational schema after normalization looks like:

- Patient (PatientID,Fname,Lname,EmailID,Gender, DateOfBirth,Age)
- PhoneNumber (PatientID,PhoneNumber)
- PatientAddress (PatientID,StreetName,City,ZipCode)
- Staff (StaffID,DepartmentID,EmailID,Fname,Lname, Position,HireDate,Qualification,PhoneNo)
- StaffAddress (StaffID,StreetName,City,ZipCode)
- Schedule (StaffID,Day,StartTime,EndTime,BreakTime)
- Department (DepartmentID,Name,HOD)
- Appointment (AppointmentID,PatientID,DoctorID, DepartmentID,AppointmentDate,AppointmentTime,Purpose,Status)
- LabReports (ReportID,PatientID,DoctorID,Type,Result)
- MedicalHistory (HistoryID,PatientID,DoctorID,NurseID ,PrescriptionID,DateOfVisit,FollowUpRequired,FollowUpDate,Symptoms,Diagnosis,Treatment,EmergencyContact)
- Pharmacy (MedicineID,Name,Description, PrescriptionRequired,StockQuantity,Price)
- Prescription (PrescriptionID,MedicineID,PatientID,Dosage, Frequency)

Table Descriptions: -

PatientAddress Table: -
Attributes: -
1. PatientID (int) - A foreign key linking to a Patient record. Primary Key, Non-null, as every patient should have an address on file.
2. StreetName (varchar(50)) - Street part of the patient's address. Non-null.
3. City (varchar(50)) - City part of the patient's address. Non-null.
4. ZipCode (varchar(10)) - Postal or ZIP code part of the patient's address. Non-null.

StaffAddress Table: -
Attributes: -
1. StaffID (int) - A foreign key linking to a Staff record. Non-null, as every staff member should have an address on file.
2. StreetName (varchar(50)) - Street part of the patient's address. Non-null.
3. City (varchar(50)) - City part of the patient's address. Non-null.
4. ZipCode (varchar(10)) - Postal or ZIP code part of the patient's address. Non-null.

PatientPhoneNumber Table: -
Attributes: -
1. PatientID (int) - A foreign key linking to a Patient record. Primary key, Non-null, as every patient should have a phone number on file.
2. PhoneNumber (varchar (15)) - PhoneNumber of the person. Primary key, Non-null

StaffPhoneNumber Table: -
Attributes: -
1. StaffID (int) - A foreign key linking to a Staff record. Primary key, Non-null, as every staff member should have a phone number on file.
2. PhoneNumber (varchar (15)) - PhoneNumber of the staff. Primary key, Non-null

Patient Table: -
Attributes: -
1. PatientID (int) - A unique identifier for each patient. Primary key, auto-incremental, non-null.
2. Fname (varchar(50)) - The first name of a patient. Non-null.
3. Lname (varchar(50)) - The last name of a patient. Non-null.
4. EmailID (varchar(50)) - The email address of a patient. Null allowed, as not all patients may provide an email.
5. Gender (varchar(15)) - The gender of the patient. Non-null, as it is often a required field.
6. DateOfBirth (date) - The birth date of a patient. Non-null, as it is crucial for patient records.
7. Age (int) - The age of the patient. It is a derived attribute from the Date of Birth column.

Department Table: -
Attributes: -
1. DepartmentID (int) - Unique identifier for each department. Primary key, auto-incremental, non-null.
2. Name (varchar(50)) - Name of the department. Non-null.
3. HOD (varchar(50)) - Every department will have an HOD who will be a doctor whose information will be stored in the Staff Table. Non-null.

Staff Table: -
Attributes: -
1. StaffID (int) - Unique identifier for each staff member. Primary key, auto-incremental, non-null.
2. DepartmentID (int) - A foreign key that links to a Department record. Non-null.
3. EmailID (varchar(50)) - Email address of the staff member. Null allowed.
4. Fname (varchar(50)) - First name of the staff member. Non-null.
5. Lname (varchar(50)) - Last name of the staff member. Non-null.
6. Position (varchar(100)) - Job title or position of the staff member. Non-null.
7. HireDate (date) - Date the staff member was hired. Non-null.
8. Qualification (varchar(50)) - Qualifications of the staff member. Non-null.

Schedule Table: -
Attributes: -
1. StaffID (int) - A foreign key linking to a Staff record. A primary key, Non-null, every staff member will have a schedule planned.
2. Day (varchar (15)) - Day of the week for which the schedule is planned. A primary key, Non-null.
3. StartTime (Time) - Starting time of the staff member. Non-null.
4. EndTime (Time) - End time of the staff member. Non-null
5. BreakTime (Time) - Break time of the staff member. Non-null.

Appointment Table: -
Attributes: -
1. AppointmentID (int) - Unique identifier for each appointment. Primary key, auto-incremental, non-null.

2. PatientID (int) - A foreign key linking to a Patient record. Non-null.
3. DoctorID (int) - A foreign key linking to a Staff record. Non-null.
4. DepartmentID (int) - A foreign key linking to a Department record. Non-null.
5. AppointmentDate (date) - Date of the appointment. Non-null.
6. AppointmentTime (Time) - Time of the appointment. Non-null.
7. Purpose (varchar(500)) - Reason or purpose for the appointment. Non-null.
8. Status (varchar(50)) - Current status of the appointment (e.g., scheduled, completed, canceled). Non-null.

LabReports Table: -

Attributes: -

1. ReportID (int) - Unique identifier for each laboratory report. Primary key, auto-incremental, non-null.
2. PatientID (int) - A foreign key linking to a Patient record. Non-null.
3. DoctorID (int) - A foreign key linking to an Address record. Non-null, as every lab test should be prescribed by the doctor.
4. Type (varchar(50)) - Type of the laboratory test. Non-null.
5. Result (varchar(128)) - Result of the test. Non-null.

MedicalHistory Table: -

Attributes: -

1. HistoryID (int) - A unique identifier for each medical history record. Primary key, auto-incremental, non-null.
2. PatientID (int) - A foreign key linking to a Patient record. Non-null, as each medical history entry is associated with a specific patient.
3. DoctorID (int) - A foreign key linking to a Doctor record. Non-null, as each medical history record is associated with a specific doctor.
4. NurseID (int) - A foreign key linking to a Nurse record. Non-null, as nurses are typically involved in the patient's care.
5. DateOfVisit (Date) - The date when the medical visit took place. Non-null.
6. FollowUpRequired (Boolean) - Indicates whether a follow-up visit is necessary. Non-null.
7. FollowUpDate (Date) - The date for the scheduled follow-up visit, if needed. Null allowed, as follow-ups may not be required for every visit.
8. Symptoms (varchar(50)) - A description of symptoms reported by the patient. Null allowed, as not every visit may involve symptoms.
9. Diagnosis (varchar(50)) - The doctor's diagnosis of the patient's condition. Null allowed, as not every visit may result in a definitive diagnosis.
10. Treatment (varchar(50)) - The treatment provided or recommended to the patient. Null allowed, as treatment may not be necessary for every visit.
11. EmergencyContact (varchar(15)) - Contact information for the patient's designated emergency contact. Null allowed, as not all patients may provide this information.

Pharmacy Table: -

Attributes: -

1. MedicineID (int) - Unique identifier for each medicine stocked in the pharmacy. Primary key, auto-incremental, non-null.
2. Name (varchar(50)) - The name of the medicine. Non-null, as each medicine must be identifiable.
3. Description (varchar(128)) - Additional information about the medicine, such as its use or side effects. Null allowed, as some basic medications may not require extensive descriptions.
4. PrescriptionRequired (Boolean) - Indicates whether the medicine can only be sold with a doctor's prescription. Non-null, as this is a regulatory requirement.
5. StockQuantity (int) - The current stock level of the medicine. Non-null, for inventory management.
6. Price (float) - The price at which the medicine is sold. Non-null, as every medicine must have a price for sale.

Prescription Table: -

Attributes: -

1. PrescriptionID (int) - Unique identifier for each prescription issued. A foreign key linking to a Prescription record in the MedicalHistory table. Primary key, non-null.
2. MedicineID (int) - A foreign key linking to a Medicine record in the Pharmacy table. Non-null, as a prescription is for a specific medicine. Primary key, non-null.
3. PatientID (int) - A foreign key linking to a Patient record, indicating who the prescription is for. Non-null. Primary key, non-null.
4. Dosage (int) - The prescribed amount of medication to be taken each time. Non-null.
5. Frequency (int) - How often the medication should be taken. Non-null.

Phase 2

E. Task 5

Indexing -

The dataset that we have with us has around 5000 entries. We came across some issues while performing some operations. These are:

- Running subqueries or join operations consume a lot of time. In our case, when we wanted to extract staff IDs whose appointments are not scheduled, the database took almost 600-700 msec to run which is little-bit much for a dataset with 5000 entries. Dataset can become larger and this could cause an issue.
- We used indexing specifically for the WHERE clauses. we found that after implementing indexing, our queries with WHERE clauses were executing a bit faster. This could be because of data being easily searched and returned with the help of indexes.
- For indexing in our database, we have used B-tree-based indexing (Postgres uses this by default) as it is efficient for running range queries (something that our database might be used for), ordering operations and other equality-based checks.
- Along with this, we made sure that all the primary keys of all our tables were set to autogenerate. Meaning no manual increment of these keys was required. If a value is not included in the insert query, it will autoincrement itself.

This issue can be resolved using indexing. Indexes are nothing but data structures used for quickly locating tuples that meet a specific type of condition.

F. Task 6

Standard Query Language -

SQL is a standard language for accessing and manipulating databases.

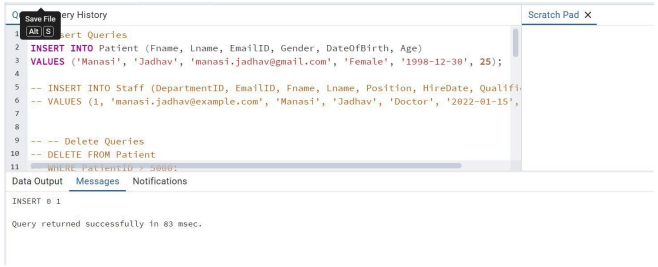
Insert Queries -

The INSERT INTO statement is used to insert new records in a table.

Insert Query 1 -

We inserted a new entry with first name ‘Manasi’, last name ‘Jadhav’, email ID as ‘manasi.jadhav@gmail.com’, gender ‘Female’, date of birth ‘1998-12-30’ and age as ‘25’.

INSERT INTO Patient (Fname, Lname, EmailID, Gender, DateOfBirth, Age) VALUES ('Manasi', 'Jadhav', 'manasi.jadhav@gmail.com', 'Female', '1998-12-30', 25);



Then we ran a select query with Patient ID more than 5000 to check if the record has been inserted into the table or not. The output is as follows -

SELECT * FROM Patient where patientid >= 5000

Insert Query 2 -

Then we inserted the staff entry as Department ID ‘1’, email ID ‘manasi.jadhav@example.com’, first name ‘Manasi’, last name ‘Jadhav’, position as ‘Doctor’, hire date ‘2022-01-15’ and lastly qualification as ‘MD’.

INSERT INTO Staff (DepartmentID, EmailID, Fname, Lname, Position, HireDate, Qualification) VALUES (1, 'manasi.jadhav@example.com', 'Manasi', 'Jadhav', 'Doctor', '2022-01-15', 'MD');

Then we ran a select query with the given first name and hire date to check if the record has been inserted into the table or not. The output is as follows -

SELECT * FROM Staff WHERE EmailID = 'manasi.jadhav@example.com' AND HireDate = '2022-01-15';

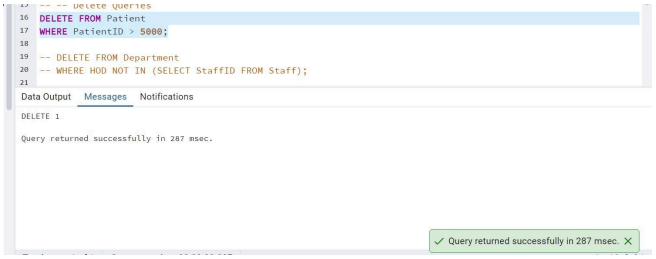
Delete Queries -

The DELETE statement is used to delete existing records in a table.

Delete Query 1 -

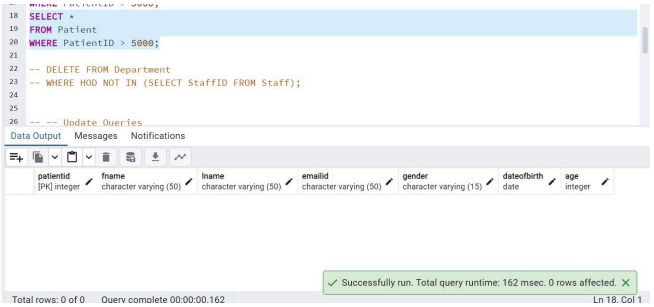
Now, we deleted all the patients whose IDs are greater than 5000. From the previous output we can see there is only 1 such patient.

DELETE FROM Patient WHERE PatientID > 5000;



Now, if we again run the select query with the same condition, we will not see any record with ID greater than 5000.

SELECT * FROM Patient WHERE PatientID > 5000;



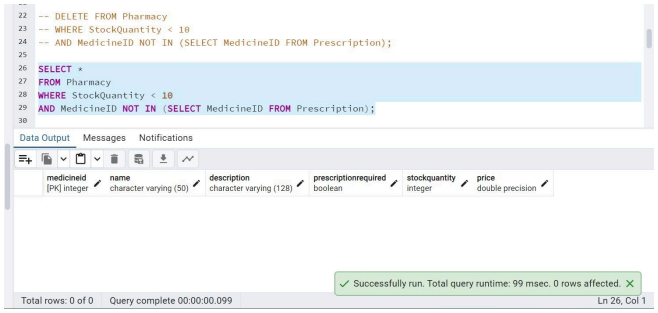
Delete Query 2 -

Now, we wanted to delete the entries which are low in stock from the Pharmacy table and also which are not prescribed by any doctor. So, we used not in operator along with delete query operation.

DELETE FROM Pharmacy WHERE StockQuantity < 10 AND MedicineID NOT IN (SELECT MedicineID FROM Prescription);

Now, if we again run the select query with the same condition, we will not see any record with stock quantity less than 10 and any medicines which are not present in the prescription.

SELECT * FROM Pharmacy WHERE StockQuantity < 10 AND MedicineID NOT IN (SELECT MedicineID FROM Prescription);



Update Queries -

The UPDATE statement is used to modify the existing records in a table.

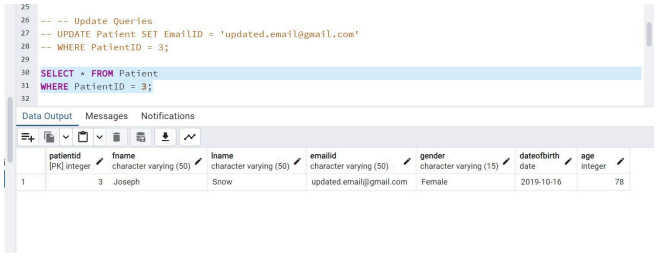
Update Query 1 -

Now, we are changing the email address of a patient based on his/her patient ID.

UPDATE Patient SET EmailID = 'updated.email@gmail.com' WHERE PatientID = 3;

Now to check if the email address has been updated or not we can use the same conditions in the select query.

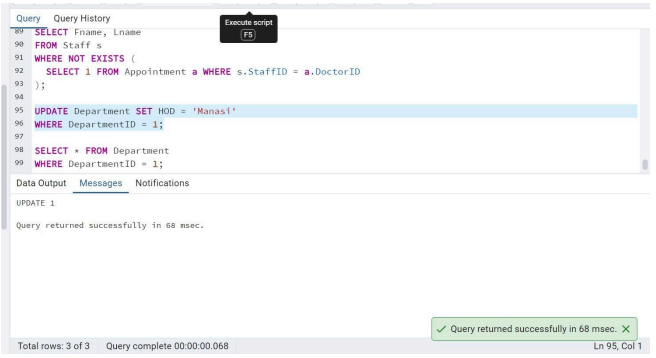
SELECT * FROM Patient WHERE PatientID = 3;



Update Query 2 -

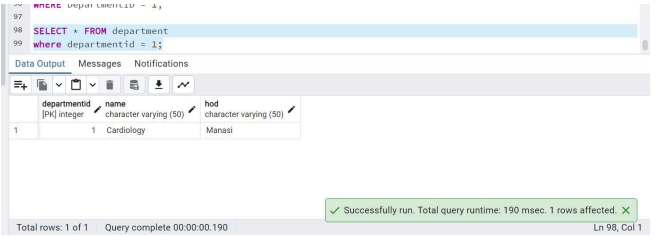
Now, suppose new staff has been assigned as a new HOD of the department number 1. So, we are now going to update the name of HOD to ‘Manasi’.

UPDATE Department SET HOD = ‘Manasi' WHERE DepartmentID = 1;



To check if the name has been updated or not we can use the same conditions in the select query.

SELECT * FROM Department WHERE DepartmentID = 1;



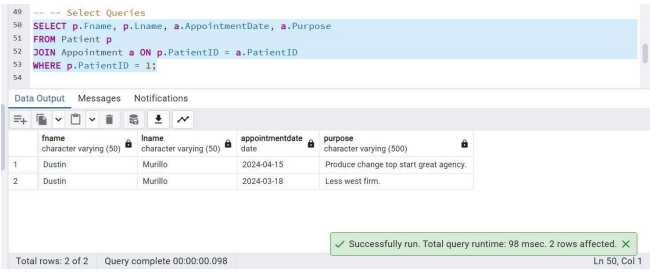
Select Queries -

The SELECT statement is used to select data from a database. In case of select statements we use operations such as join, group by, subqueries to get better results.

Select Query 1 -

A JOIN clause is used to combine rows from two or more tables, based on a related column between them. If we want to see the appointment date and purpose of the appointment of the given patient, we can use the join query as follows.

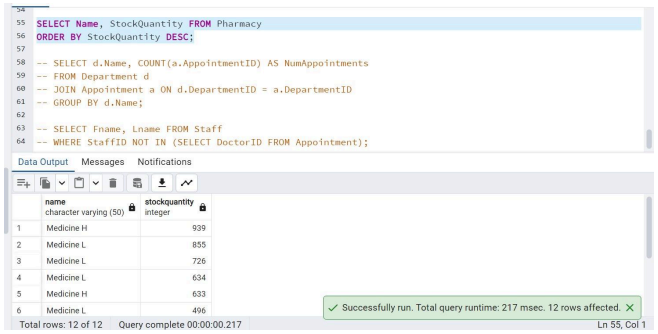
SELECT p.Fname, p.Lname, a.AppointmentDate, a.Purpose FROM Patient p JOIN Appointment a ON p.PatientID = a.PatientID WHERE p.PatientID = 1;



Select Query 2 -

The ORDER BY keyword is used to sort the result-set in ascending or descending order. Suppose, we want to see medicines available in the pharmacy in descending order of their stock quantity, we can use the following query.

SELECT Name, StockQuantity FROM Pharmacy ORDER BY StockQuantity DESC;

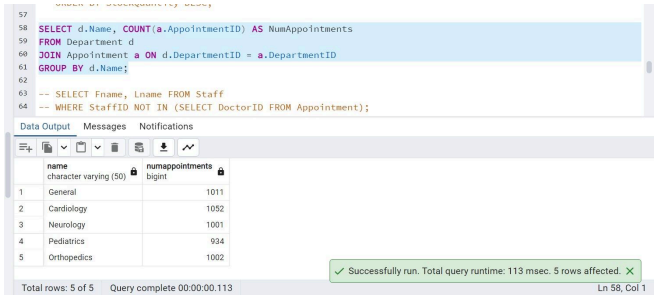


Select Query 3 -

The GROUP BY statement groups rows that have the same values into summary rows. The GROUP BY statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

Now suppose we want to check the number of appointments each department has. We can use count along with group by to check.

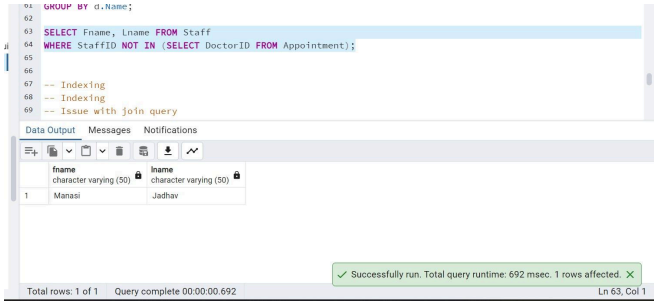
SELECT d.Name, COUNT(a.AppointmentID) AS NumAppointments FROM Department d JOIN Appointment a ON d.DepartmentID = a.DepartmentID GROUP BY d.Name;



Select Query 4 -

An SQL Subquery, is a SELECT query within another query. Now suppose if we want to see the first name and last name of the staff who does not have any appointment, we need at least 2 nested select statements. The query can be as follows.

SELECT Fname, Lname FROM Staff WHERE StaffID NOT IN (SELECT DoctorID FROM Appointment);



This is how we can alter and view the data in the database using insert, update, delete and insert queries.

G. Task 7

Query Execution Analysis -

3 potential problematic queries can be - join operation, group by operation and sub-query operation.

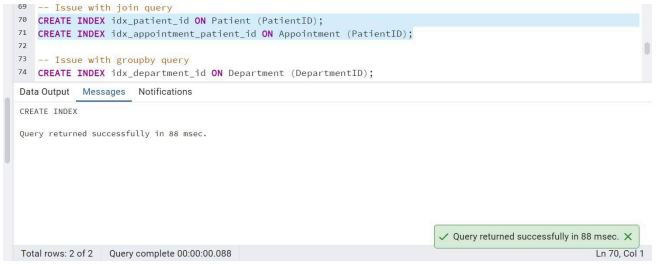
1. Join Operation Optimization (Select Query 1) -

For better performance we can introduce indexing before performing join operation. Indexes on the 'PatientID' columns in both 'Patient' and 'Appointment' tables can help the database engine quickly to locate the entries needed for the join. Specially, when we filter the entries by 'PatientID' it will show optimized results. Without these indexing, the database would perform a full table scan to find the required rows. This is very much inefficient for larger databases.

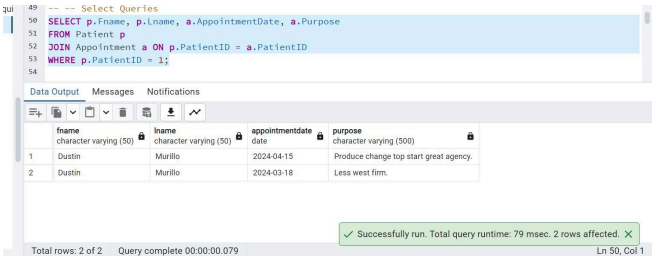
We created these two indexes using the following query.

CREATE INDEX idx_patient_id ON Patient (PatientID);

CREATE INDEX idx_appointment_patient_id ON Appointment (PatientID);



Now, after indexing if we run this query again we can see that earlier it took 98 msec to run but now it requires just 79 msec to run. Even though this difference seems very minor for now, it can be very useful for larger datasets.



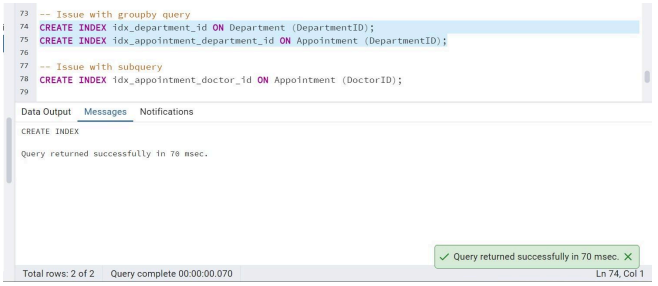
2. Group by Operation Optimization (Select Query 3) -

Similarly, the indexes ensure that data is quickly accessible and join operations are streamlined, reducing the amount of data that needs to be processed in memory and potentially lowering the costs of sorting and grouping operations. Now, here we can perform indexing on column 'DepartmentID' to quickly align rows from 'Department' with rows from 'Appointment' based on 'DepartmentID'. This is crucial for efficiently aggregating data grouped by department.

We created these 2 indexes using the following query.

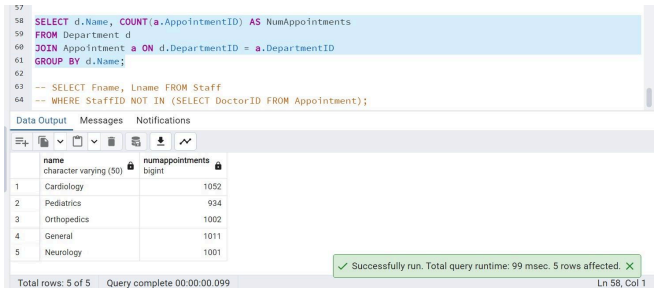
CREATE INDEX idx_department_id ON Department (DepartmentID);

CREATE INDEX idx_appointment_department_id ON Appointment (DepartmentID);



Now, after indexing if we run this query again we can see that earlier it took 113 msec to run but now it requires just

99 msec to run. Even though this difference seems very minor for now, it can be very useful for larger datasets.



3. Sub-query Operation Optimization (Select Query 4) -

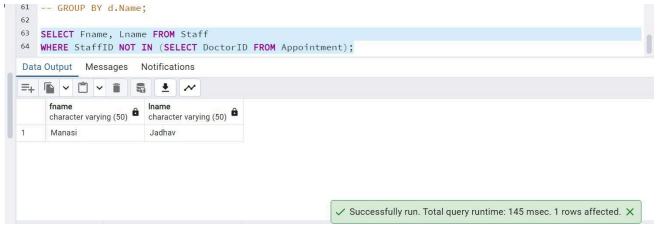
There are 2 techniques using which this query can be optimized.

- **Using Join Operation with Indexing -**

Firstly, same as above we will create an index of 'DoctorID' in 'Appointment' for better results. The query for that would be as follows.

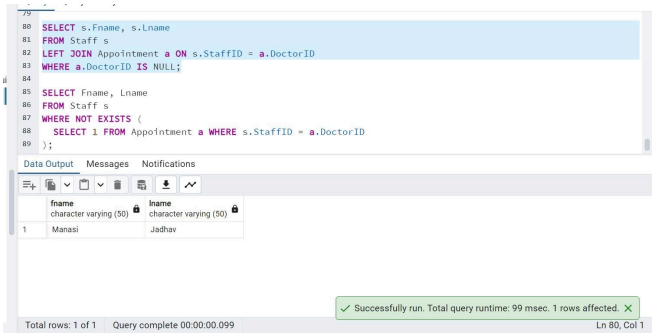
CREATE INDEX idx_appointment_doctor_id ON Appointment (DoctorID);

Now if we run the same query without any join operation we can see a significant decrease in the time. Earlier it required around 700 msec and now it's running in just 145 msec.



Now, if we perform a join operation on the table to get the same results but with a different query we can see that the time has been further reduced to 99 msec.

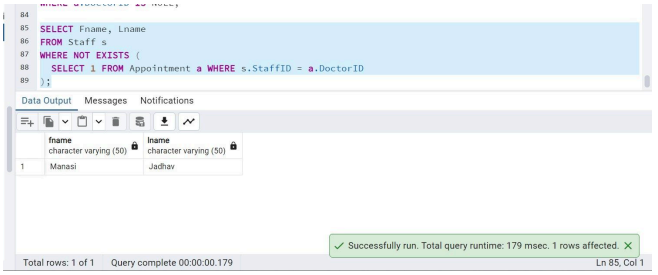
SELECT s.Fname, s.Lname FROM Staff s LEFT JOIN Appointment a ON s.StaffID = a.DoctorID WHERE a.DoctorID IS NULL



- **Using NOT EXISTS operation with Indexing -**

This solution does not work as great as join operation but still its running in less than the original time. With NOT EXISTS the query runs in 179 msec.

SELECT Fname, Lname FROM Staff s WHERE NOT EXISTS (SELECT 1 FROM Appointment a WHERE s.StaffID = a.DoctorID);



Demo Video and Bonus Application

Timestamps to follow the video:

- Project Overview - 00:05
- ER Diagram - 01:30
- Creating Datasets - 02:15
- Database Tables - 04:20
- Running various Queries - 06:05
- Bonus Task App overview - 08:40