Healthcare Management System in SQL and MongoDB

Fyrooz Anika Khan, Fred Liu, Fardin Hafiz, Suraj Ravindra Kapare
Department of Data Science, The George Washington University, Washington, D.C. 20052

***ABSTRACT*:** *Modern healthcare systems demand robust database solutions that can efficiently handle large-scale datasets, facilitate complex queries, and adapt to evolving requirements. This research explores the development of a healthcare management system using SQL and MongoDB, two widely utilized database technologies with distinct structural paradigms. This study allows for a greater understanding of SQL and MongoDB through the utilization of these database systems in managing tasks for a faker-generated Healthcare dataset. Some tasks were completed both in SQL and MongoDB in order to develop an understanding of how these separate systems compare in a similar objective. At the end of this project, we became more familiar with handling datasets with database systems like SQL and MongoDB.*

Keywords—SQL, MongoDB

I. INTRODUCTION

Effective data management is critical in the healthcare industry, where the accurate organization, storage, and retrieval of patient and appointment records directly impact the quality of care provided. The primary objective of this project is to create a dual database system capable of managing two core entities: patients and their appointments. These entities form the foundation of a typical healthcare management system, encompassing basic patient details, appointment scheduling, symptoms, treatment plans, and medication dosages. By employing SQL and MongoDB, the project aims to demonstrate the structural differences between relational and NoSQL databases while evaluating their performance and flexibility in managing large datasets.

This project's objective is to deepen our understanding of database systems such as SQL and MongoDB. This was done by emulating real-life usage of these systems with Faker generated Healthcare data.

II. SQL AND MONGODB

SQL (Structured Query Language) is a standard language for managing and manipulating relational databases, where data is stored in tables with a predefined schema. It is used to perform various operations such as querying, updating, and managing data, ensuring data integrity and relationships through primary and foreign keys. MongoDB, on the other hand, is a NoSQL database that uses a flexible, document-oriented model. It stores data in JSON-like documents with dynamic schemas, allowing for easy handling of unstructured or semi-structured data, and is known for its scalability and performance in handling large volumes of data [1], [2].

A. Comparison of SQL and MongoDB Data Structures

SQL uses a relational model with a strict schema. The data is structured into tables, and relationships are defined using foreign keys. MongoDB follows a document-oriented approach. Every document is self-contained and supports nested objects and arrays [1], [2].

Both databases handle missing values differently. SQL uses NULL to represent missing data. This requires careful handling of queries to avoid unexpected results. MongoDB skips fields with missing values, simplifying the data structure but may require additional checks during data retrieval [1], [2].

SQL is ideal for systems that require well-defined relationships and strong data integrity. It works effectively in scenarios requiring data consistency, such as patient billing or compliance reporting. MongoDB is better suited for systems that handle complex, dynamic, or semi-structured data. Its ability to embed data within documents makes it ideal for storing nested information in applications such as patient feedback systems [1], [2].

SQL ensures that data is consistent and optimized for relational queries, but it may struggle in cases where the schema must change regularly.

MongoDB thrives in these scenarios, allowing developers to change the layout of documents without affecting existing data. When there is a lot of data redundancy, MongoDB's flexibility can result in higher storage requirements and slower queries [1], [2].

B. Comparison of SQL and MongoDB Database Management Tools

PhpMyAdmin is used for SQL queries, which is a free, web-based tool that provides a user-friendly interface for managing MySQL databases. It allows users to perform various database operations such as creating, modifying, and deleting databases, tables, and records, as well as executing SQL queries and managing user permissions. MongoDB Compass is used for MongoDB queries which is a graphical user interface that provides an effortless way to interact with MongoDB databases, offering features such as schema visualization, query performance insights, and CRUD operations, making it easier to manage and optimize MongoDB data [1], [2].

III. METHODOLOGY

The methodology section covers how data were generated and imported to perform the tasks in SQL and MongoDB.

A. Data Generation

This project's data was generated using the Python Faker package, which produced realistic results. A predetermined seed value ensured that the datasets were reproducible. While both the SQL and MongoDB datasets contained the same underlying healthcare data, their structures differed dramatically due to the storage methods of relational and document-oriented databases.

1) Patients Dataset

name	age	contact_number	email
Adam Bryan	54	453-749-6251x6482	schmidtjill@example.net
Brittany Ball	48	255.207.5086	
Shelby Townsend	56		
Brett Perry	56	+1-492-663-8950x31738	iholmes@example.org
Elizabeth Lin	24	2232585265	dawsonchristian@gmail.com
Tammie Mccormick PhD	53	(538)717-9007x596	
Melissa Hawkins	27	472-353-3224	

The patient dataset included each person's contact details and demographic data. This dataset was arranged in SQL as a flat table, with a row for each patient and columns for name, age, contact_number, and email. Patient_id, a unique identifier, was produced automatically using SQL's AUTO_INCREMENT feature. Missing values, such as contact numbers or email addresses, were explicitly marked as NULL.

_id	name	age	contact_number	email
101	Adam Bryan	54	453-749-6251x6482	schmidtjill@example.net
102	Brittany Ball	48	255.207.5086	
103	Shelby Townsend	56		
104	Brett Perry	56	+1-492-663-8950x31738	iholmes@example.org
105	Elizabeth Lin	24	2232585265	dawsonchristian@gmail.com
106	Tammie Mccormick PhD	53	(538)717-9007x596	
107	Melissa Hawkins	27	472-353-3224	

In the MongoDB dataset, each patient was represented as a document. Each document had a _id field, which acted as a unique identifier. By default, MongoDB creates _id as a large random number, while in this project, the _id was modified to match the patient_id values in the appointment's dataset. This option allowed for consistent mapping between the two databases and eased cross-database comparisons. Missing data in contact numbers or email addresses, were left empty in the MongoDB documents. This could avoid the unnecessary representation of missing data, but it requires extra caution when accessing such records.

2) Appointments Dataset

patient_id	date	is_follow_up	symptom1	symptom2	medication_name	dosage	frequency	dose_day1	dose_day2	dose_day3	dose_day4	dose_day5	dose_day6	dose_day7
101	1/6/2020	1	fatigue	dizziness	Diphenhydramine	9mg	3 times a day	1	1	3	0	3	1	3
102	6/10/2020	0			Ciprofloxacin	7mg	3 times a day	3	0	3	1	1	0	2
103	12/3/2021	0	dizziness		Acetaminophen	16mg	2 times a day	2	1	0	2	1	0	2
104	4/9/2023	0	fever	muscle pain	Diphenhydramine	1mg	2 times a day	2	1	0	2	1	2	1
105	7/20/2024	1	fatigue	sore throat	Aspirin	33mg	1 time a day	1	1	1	1	1	0	0
106	1/29/2023	1			Diphenhydramine	16mg	3 times a day	2	3	1	0	0	1	0

The appointments dataset includes information such as appointment dates, symptoms, and prescribed treatment plans. The SQL dataset saved this data in a flat table style with columns for patient identification (patient_id), appointment date (date), and a binary column (is_follow_up) to indicate if the appointment was a follow-up. A follow-up appointment was marked as 1, while a first-time appointment was marked as 0. The appointments table connects to the patients table via patient_id. The dataset also has two different columns for symptoms (symptom1 and symptom2), with missing symptoms denoted as NULL. Medication name, dosage, and frequency were stored in separate columns. In addition, seven columns (dose_day1 to dose_day7) were used to specify the week's daily medicine doses.

_id	patient_id	date	is_follow_up	symptoms	treatment_details
1	101	1/6/2020	TRUE	["fatigue", "dizziness"]	{"medication_name": "Diphenhydramine", "dosage": "9mg", "frequency": "3 times a day", "dose_schedule": [1, 1, 3, 0, 3, 1, 3]}
2	102	6/10/2020	FALSE	[]	{"medication_name": "Ciprofloxacin", "dosage": "7mg", "frequency": "3 times a day", "dose_schedule": [3, 0, 3, 1, 1, 0, 2]}
3	103	12/3/2021	FALSE	["dizziness"]	{"medication_name": "Acetaminophen", "dosage": "16mg", "frequency": "2 times a day", "dose_schedule": [2, 1, 0, 2, 1, 0, 2]}
4	104	4/9/2023	FALSE	["fever", "muscle pain"]	{"medication_name": "Diphenhydramine", "dosage": "1mg", "frequency": "2 times a day", "dose_schedule": [2, 1, 0, 2, 1, 2, 1]}
5	105	7/20/2024	TRUE	["fatigue", "sore throat"]	{"medication_name": "Aspirin", "dosage": "33mg", "frequency": "1 time a day", "dose_schedule": [1, 1, 1, 1, 1, 0, 0]}
6	106	1/29/2023	TRUE	[]	{"medication_name": "Diphenhydramine", "dosage": "16mg", "frequency": "3 times a day", "dose_schedule": [2, 3, 1, 0, 0, 1, 0]}
7	107	7/18/2024	TRUE	["sore throat"]	{"medication_name": "Azithromycin", "dosage": "69mg", "frequency": "3 times a day", "dose_schedule": [0, 1, 2, 0, 2, 0, 3]}
8	108	6/21/2021	FALSE	["muscle pain", "dizziness"]	{"medication_name": "Azithromycin", "dosage": "37mg", "frequency": "2 times a day", "dose_schedule": [1, 1, 2, 2, 0, 2, 2]}
9	109	7/1/2024	TRUE	["headache"]	{"medication_name": "Azithromycin", "dosage": "85mg", "frequency": "1 time a day", "dose_schedule": [1, 0, 1, 1, 1, 1, 0]}

In contrast, the MongoDB appointments dataset followed a document-oriented format. Each appointment was stored as a separate document with fields and nested objects. Symptoms were displayed as an array under the symptoms field, so the information is more compact, and additional fields are not needed. If no symptoms were recorded, the array stayed empty. Treatment details were grouped into a nested object called treatment_details, which included fields like medication name, dosage, and frequency. The weekly dose schedule was also stored as an array within this object, which eliminated the need for numerous fields to store daily doses.

B. *Connecting to SQL Server and MongoDB, Database creation and Data Import*

Connections to both the MySQL server and MongoDB instance were pre-established. After creating the tables in MySQL, the relevant CSV files were imported into their respective tables using phpMyAdmin. For MongoDB, the database HealthcareMS_db and two collections: Patients and Appointments were created. The commands are to be executed in the MongoDB shell. Following this, the CSV files were imported into their respective collections using MongoDB Compass.

C. *Performing queries and execution time calculations*

Queries in both SQL and MongoDB were carried out for all 10 tasks, and the execution times were calculated. MongoDB queries were only used for the last 6 tasks.

IV. QUERIES AND THEIR EXPLANATION

This section contains the SQL and MongoDB queries used and what they mean.

Task 1: *Update the First Element in an Array Inside a Nested Object*

In this task, the goal was to update a specific treatment detail (the first element of the dose_schedule array) for an appointment identified by its ID.

SQL Query:

```
UPDATE appointmentdetails
SET dose_day1 = 10
WHERE appointment_id = 5;
```

The query updates the dose_day1 column in the Appointments table to a value of 10 for the row where appointment_id equals 5. This demonstrates SQL's ability to modify specific records in a table using UPDATE, with the SET clause specifying the new value and the WHERE clause filtering the target row.

MongoDB Shell Script:

```
db.Appointments.updateOne(
  { _id: 5 },
  { $set: { "treatment_details.dose_schedule.0": 10 } }
);
```

This script updates a single document in the Appointments collection. The filter {_id: 5} targets the document with an ID of 5. The $set operator modifies the first element (index 0) of the dose_schedule array inside the treatment_details field, setting its value to 10. MongoDB's flexibility with nested structures makes this operation seamless.

Task 2: *Retrieve Patients with Specific Criteria*

This task was to retrieve patients under 20 or over 50 years old with a valid contact number or email.

SQL Query:

```
SELECT *
FROM patientdetails
WHERE (age < 20 OR age > 50)
  AND ((contact_number IS NOT NULL AND
contact_number <> '')
      OR (email IS NOT NULL AND email <> ''));
```

This SQL query uses SELECT to fetch all columns from patients where patients meet specific criteria: either their age is less than 20 or greater than 50, and they have a non-null, non-empty contact_number or email. Logical operators AND and OR filter rows efficiently.

MongoDB Shell Script:

```
db.Patients.find({
  $and: [
    { $or: [ { age: { $lt: 20 } }, { age: { $gt: 50 } } ] },
    { $or: [ { contact_number: { $ne: null } }, { email: {
$ne: null } } ] }
  ]
});
```

The MongoDB query uses $and to combine conditions. The first $or ensures patients are either younger than 20 or older than 50. The second $or verifies that contact_number or email fields are not

null. MongoDB's find query enables sophisticated filtering within a flexible schema.

Task 3: *Increase Age by 1 Year for All Patients Over 60*

This task was to increment the age of patients older than 60 by 1 year.

SQL Query:

```
UPDATE patientdetails
SET age = age + 1
WHERE age > 60;
```

The UPDATE statement modifies the Patients table by increasing the age column by 1 for rows where age is greater than 60. SQL's arithmetic operations within the SET clause simplify numerical updates across filtered records.

MongoDB Shell Script:

```
db.Patients.updateMany(
  { age: { $gt: 60 } },
  { $inc: { age: 1 } }
);
```

The MongoDB updateMany method applies the $inc operator to increment the age field by 1 for all documents where the age is greater than 60. MongoDB's ability to batch update multiple documents in a single command enhances scalability for large datasets.

Task 4: *Perform Aggregation to Count Appointments by Patient*

The task was to count how many appointments each patient had.

SQL Query:

```
SELECT patient_id, COUNT(*) AS appointment_count
FROM appointmentdetails
GROUP BY patient_id;
```

This SQL query groups rows in the Appointments table by patient_id, then calculates the number of rows in each group using COUNT(*). The GROUP BY clause is essential for aggregation, and the AS keyword renames the result column for clarity.

MongoDB Shell Script:

```
db.Appointments.aggregate([
  {
    $group: {
      _id: "$patient_id",
      appointment_count: { $sum: 1 }
    }
  }
]);
```

The aggregate method in MongoDB groups documents by patient_id (_id: "$patient_id") and calculates the total number of documents in each group using $sum: 1. MongoDB's pipeline approach allows complex aggregations over large collections, demonstrating its power and flexibility in analytics.

Task 5: Find Appointments with Symptoms Containing 'fever' but Not 'cough'

This task was to find all appointments that contained the symptom ‘fever’ but not ‘cough’.

SQL Query:

```
SELECT *FROM appointments_sql WHERE ('fever' IN
(symptom1, symptom2))AND NOT ('cough' IN
(symptom1, symptom2));
```

The SELECT statement in this code determines where the data is being drawn from. The WHERE statement determines the conditions that must be met for the desired output. This code asks for appointments where there is the term ‘fever’ in. symptom1 or symptom2 AND NOT the term ‘cough’ in symptom1 or symptom2.

M*ongoDB Shell Script:*

```
db.Appointments.find({symptoms: { $in: ['fever']
},symptoms: { $nin: ['cough'] }});
```

The db.Appointments.find allows for the searching of Appointments that meet certain given conditions. These conditions are $in the term ‘fever’ and $nin the term ‘cough’. This means that this script is searching for appointments that contain fever symptoms but not cough.

Task 6: Retrieve Patients with Email Addresses *from Gmail*

This task was to retrieve all the patients that have gmail.com.

SQL Query:

```
SELECT *FROM patients_sql WHERE email LIKE
'%@gmail.com';
```

The SELECT statement in this code determines from which dataset the data is being drawn from. The WHERE statement determines the conditions that must be met for the desired output. In this query the command LIKE finds all the patients that

have the phrase '@gmail.com' since this phrase would be present in all patients that used gmail.
MongoDB Shell Script:

```
db.Patients.find({email: { $regex: /@gmail\.com$/i }});
```

The db.Patients.find allows for the searching of Patients that meet certain given conditions.ons. The conditions searched in the email column are by the $regex command. This command searches for all the emails with the phrase @gmail.com.

Task 7: *List Appointments Where More ThanOne Symptom Was Reported*

This task was to retrieve all of the appointments where more than one symptom was reported. There were varying numbers of symptoms reported between 0 to 2.
SQL Script:

```
SELECT *
FROM appointments_sql
WHERE symptom1 <> '' AND symptom2 <> '';
```

The SELECT statement in this query determines which dataset the query is being searched from. In the Appointments dataset, there are two symptoms columns, symptom1 and symptom2. This code uses the WHERE statement to meet the conditions of the search. This search is asking for the conditions where symptom1 and symptom2 are not empty strings, meaning that there is information in these strings. If both contain information, then the appointment has 2 symptoms.
MongoDB Shell Script:

```
db.Appointments.find({$expr: { $eq: [{ $size:
"$symptoms" }, 2] }});
```

This script uses the find() command to search in the database Appointments to find appointments that match the provided conditions. These conditions are the symptoms array having 2 items, which means the patients have 2 symptoms.

Task 8: *Remove Appointments with No Reported Symptoms*

The task was to delete all appointments that have no symptoms listed.
SQL Query:

```
DELETE FROM appointments_sql
WHERE symptom1 = '' AND symptom2 = '' ;
```

This query uses the command DELETE FROM in order to delete from the dataset appointments where certain conditions are met. The WHERE command is used to show which conditions. These conditions are when both symptom1 and symptom2 are empty, which means there were no symptoms.
MongoDB Script:

```
db.Appointments.deleteMany({symptoms: { $size: 0 }});
```

This query uses the deleteMany command to the db.Appointments to delete all of the conditions that meet the condtions stated, the size of the symptoms array being 0.

Task 9: *Retrieve Distinct Appointment Dates*

The task was to collect and display a list of unique appointment dates from the appointments collection.
SQL Script:

```
SELECT DISTINCT date
FROM appointments_sql;
```

from the appointments table. The DISTINCT keyword ensures that duplicate dates are excluded from the result set. This is useful for identifying all the unique appointment dates without repetition.
Unlike a GROUP BY clause, which is used for aggregation, DISTINCT works directly on the selected column to filter out duplicate values. No renaming of columns or additional calculations are performed in this query.
MongoDB Shell Script:

```
db.Appointments.distinct("date");
```

This MongoDB query retrieves a list of unique dates from the appointments collection using the distinct method. The distinct method focuses on the "date" field and returns an array of unique values found in that field across all documents in the collection. It eliminates duplicates, ensuring that each date appears only once in the result set. This query is particularly useful for identifying all distinct appointment dates stored in the database without fetching other fields or full documents. MongoDB's distinct operation is efficient for extracting unique field values from large datasets.

Task 10: *Delete Patients Over Age 60 with No Email*

The task was to remove patients from the database who are over 60 years old and do not have an email address.

SQL Script:

```
DELETE FROM patients_sql
WHERE age > 60 AND (email IS NULL OR email = '');
```

This SQL query deletes records from the patients table where the age is greater than 60 and the email is either NULL or an empty string (' '). It uses the WHERE clause to specify the conditions for deletion, ensuring only patients meeting these criteria are removed from the database. This operation helps clean up records that lack critical contact information.

MongoDB Shell Script:

```
db.patients_csv.delete_many({
   "age": { "$gt": 60 },
   "$or": [
      { "email": { "$exists": False } },
      { "email": "" }
   ]
})
```

This MongoDB query removes all documents from the patients collection where the age is greater than 60 and the email field either does not exist or is an empty string. The $gt operator checks for age greater than 60, while the $oroperator combines the conditions for missing or empty email fields. This query ensures that incomplete records are effectively cleaned from the database.

Task 11: *Retrieve Patients Aged Between 30 and 50 with Contact Information*

The task was to get patient details, including contact information, for individuals aged between 30 and 50.

MongoDB Shell Script:

```
db.patients_csv.find({
   "age": { "$gte": 30, "$lte": 50 },
   "contact_number": { "$exists": True, "$ne": "" }
}, { "name": 1, "age": 1, "contact_number": 1, "email": 1 })
```

This MongoDB query retrieves documents from the patients collection for individuals aged between 30 and 50 ($gteand $lte specify the range). It ensures that the contact_number field exists ($exists: True) and is not empty ($ne: ""). The query projects only the name, age, contact_number, and email fields in the result for clarity and relevance.

Task 12: *List Appointments Marked as Follow-Up*

The task was to identify and list all appointments that have been marked as follow-up for easy tracking.

MongoDB Shell Script:

```
db.Appointments.find({ "is_follow_up": true })
```

This query is used to retrieve all documents from the appointments collection where the is_follow_up field has a value of true. It specifically filters and returns only those appointments that have been marked as follow-ups. This is useful for identifying and tracking appointments requiring further patient follow-up care.

Task 13: *Retrieve All Patients' Names and Ages*

The task was to extract and display the names and ages of all patients.

MongoDB Shell Script:

```
db.patients.find({}, { name: 1, age: 1, _id: 0 });
```

The query chooses the name and age fields but excludes the default _id field. Fields are included by assigning a value of 1, and _id is excluded by assigning a value of 0. With 100,000 patient data, we demonstrate MongoDB's flexible document structure, enabling us to focus only on the required fields. This query can effectively lower output size and enhance readability. The result is a simple list of patient names and ages.

Task 14: *Find Appointments with Symptoms in an Array*

The task was to identify appointments where symptoms are stored in an array format.

MongoDB Shell Script:

```
db.appointments.find({ symptoms: { $type: "array" } });
```

The $type operator checks the data type of the symptoms field in each document. In this case, it was looking for fields of the array type. This query is very handy in MongoDB because fields can include a variety of data kinds. Identifying

documents that follow a given format is critical for data validation and consistency. The outcome confirmed that the symptoms field was properly recorded as arrays. This task demonstrates MongoDB's ability to handle and query heterogeneous data types rapidly, making it a good choice for datasets with variable schemas.

Task 15: *Retrieve Patients with Names Starting with Specific Letters*

The task was to find patients whose names start with certain letters using regex matching.

MongoDB Shell Script:

```
db.patients.find({ name: { $regex: "^[A-C]" } });
```

The $regex operator allows us to match patterns in strings, making it extremely useful for text-based queries. The ^[A-C] pattern specifies that the name field must start with a letter from A to C. Regular expressions are excellent tools for analyzing large datasets. The results only displayed patient names that began with the given letters. This exercise demonstrates MongoDB's extensive text processing capabilities as well as its ability to handle big text-based datasets seamlessly.

Task 16: *Update Medication Dosage for Specific Conditions*

The task was to change the dosage of a specific medication for all appointments that contain certain symptoms.

MongoDB Shell Script:

```
db.appointments.updateMany(
  { symptoms: { $in: ["fever", "cough"] },
"treatment_details.medication_name": "Ibuprofen" },
  { $set: { "treatment_details.dosage": "300mg" } }
);
```

The query used the $in operator to determine whether the symptoms array contained "fever" or "cough." It also identified documents where the medication_name was "Ibuprofen." The $set operator updated the dose field in the nested treatment_details object to "300mg." This exercise demonstrates MongoDB's ability to conduct bulk changes on nested fields, which is much easier than SQL. After running the query, the relevant documents would display the updated dosage.

V. DISCUSSION ON THE EXECUTION TIME OF THE QUERIES

A. Average Execution Times:

We first calculate the average execution time for SQL and MongoDB based on the results:

--*SQL Average Execution Time:*

Total SQL execution times for 10 queries:

```
0.023+0.680+0.105+0.718+0.0007+0.0008+0.
0011+0.0063+0.056+0.132=2.7239 seconds0.0
23 + 0.680 + 0.105 + 0.718 + 0.0007 + 0.0008
+ 0.0011 + 0.0063 + 0.056 + 0.132 = 2.7239
```

Average SQL Execution Time:

2.7239/10 = 0.27239 seconds

--*MongoDB Average Execution Time:*

Total MongoDB execution times for 10 queries:

```
0.024+0.179+0.478+0.366+0.174+0.207+0.41
2+0.704+0.100+0.191=3.835 seconds0.024 +
0.179 + 0.478 + 0.366 + 0.174 + 0.207 + 0.412
+ 0.704 + 0.100 + 0.191 = 3.835
```

Average MongoDB Execution Time:

3.835/10 = 0.3835 seconds

Based on the average execution times of SQL and MongoDB for the given queries, it can be said that SQL has a slightly faster average execution time (0.27239 seconds) compared to MongoDB (0.3835 seconds).

B. Suitability Analysis:

- SQL is more suitable for queries that involve complex filtering, aggregations, or joins, such as retrieving data with multiple conditions (e.g., Task 2) or counting appointments (e.g., Task 4), since its execution time is generally better for such tasks.
- MongoDB excels in queries that involve direct matching, searching for specific values, and simple aggregations (e.g., Task 5 and Task 9), where the average execution time is lower than SQL. Based on the provided queries, SQL appears to be more efficient for complex queries, while MongoDB is faster for simpler queries.

VI. CONCLUSION

In this project, we developed a comprehensive healthcare management system using both SQL and MongoDB. By creating structured SQL tables and flexible MongoDB collections, we were able to explore the strengths and limitations of both

database systems. Using the Faker library, we generated realistic data for patients and appointments, populating both databases with 100,000 records. We performed various advanced queries to understand the performance and capabilities of SQL and MongoDB.

While the project successfully demonstrated the practical applications of relational and NoSQL databases, some limitations were noted. The dataset could be improved with more diverse and comprehensive records to better simulate real-world scenarios. Additionally, the performance metrics might vary with different hardware and larger datasets. Despite these limitations, the project provided valuable insights into database management and query optimization.

REFERENCES

[1] Joel Murach, *Murach's MySQL (3rd Edition)*, 3rd ed. 2019.

[2] K. Chodorow, *MongoDB: the definitive guide*, Second edition. Beijing: O'Reilly, 2013