J119811 REPORT

by MD TANVIR SARDARR

Submission date: 07-May-2025 02:55AM (UTC+0100)
Submission ID: 258150098
File name: J119811_REPORT_1199577_1240353098.pdf (773.65K)
Word count: 3174
Character count: 23369

NoSQL Database Design and Build Assignment

Student Assessment Number: J119811

Module Code: CO7401

Module Title: Databases – SQL and NoSQL

Word count: 1318 (Excluding References, Cover Page, Table of Content & Code)

Date of Submission: 7 May 2025

Contents

1 Introduction

Healthcare institutions today produce massive heterogeneous data composed of patient documents alongside medicine prescriptions and medical appointments and financial records. Relational databases show limited capability when coping with dynamic healthcare data volumes because their strict schema structure and performance limitations create obstacles. This project develops MediCareDB as a NoSQL-based healthcare management system which utilizes MongoDB for database operation.

The document model in MongoDB ensures flexible medical record management since it adapts to diverse health data without necessitating labor-intensive schema reorganizations. The platform provides high performance along with real-time analytics and effortless scaling while meeting the essential requirements of healthcare establishments that require fast operation speed, permanent operational capabilities and adaptable solutions (Khan et al., 2022; Van Landuyt et al., 2023). The report includes detailed information regarding MediCareDB's data modeling along with its batch data generation and query design implementation along with performance assessment. The system design allows for upcoming patient volume expansion, medical services growth, and geographic expansion through its ability to scale and respond as a contemporary healthcare data management solution.

2 Data Modelling

2.1 Attributes of Each Collection

Patients: PatientID, FirstName, LastName, Age, DOB, Gender, BloodType, ContactNumber, Address, Allergies, MedicalHistory

Doctors: DoctorID, FirstName, LastName, Specialization, ContactNumber, Email, DepartmentID (Links to departments), HospitalID (Links to hospitals)

Appointments: PatientID, DoctorID, PatientName, DoctorName *(denormalized)*, Date, Status, Notes

billing_records: PatientID, PatientName, TotalAmount, PaymentStatus

prescriptions: PatientID, Medication, Dosage, Duration

lab_results: PatientID, TestName, Result, ResultDate

room_assignments: PatientID, RoomNumber, AdmissionDate, DischargeDate

pharmacy: DrugName, Manufacturer,ExpiryDate, Stock

departments: DepartmentID, DepartmentName

hospitals: HospitalID, HospitalName, Location, PhoneNumber, Email

Why is a Document-Oriented Database suitable?

MongoDB's document-based model is well-suited for this healthcare system because it supports schema flexibility, allowing each document to evolve independently as medical requirements change. This is essential in healthcare, where patient records and treatment formats vary widely (Khan et al., 2022; Alflahi et al., 2023). Document stores like MongoDB also perform better in read-heavy systems, especially when denormalized data is used to reduce cross-collection joins.

In MediCareDB, relationships between collections are managed using a hybrid approach:

- **References** (e.g., PatientID, DoctorID) maintain link integrity across collections like appointments, prescriptions, and billing.
- **Embedded Fields** (e.g., PatientName, DoctorName) are added directly to documents where quick access is crucial, improving reading efficiency.

This combination ensures performance and consistency, which is ideal for real-time hospital applications (Van Landuyt et al., 2023

3 Methodology

3.1 Database Setup

The system connects to MongoDB Atlas using a secure URI.

```python
from pymongo import MongoClient, ASCENDING
import random
import datetime

# MongoDB URI & client setup
uri = "mongodb+srv://tanvirxf:7T0YMWUdLEjVXWtl@cluster0.uakz7xn.mongodb.net/?retryWrites=true&w=majority&ssl=true&authSource=admin"
client = MongoClient(uri)
db = client["MediCareDB"]

try:
    client.admin.command('ping')
    print("Connected to MongoDB Atlas.")
except Exception as e:
    print("Connection failed:", e)
    exit()
```

3.2 Data Generation

Synthetic data is generated using Python functions designed to simulate realistic hospital data across collections:

```python
def generate_patient_info(id):
    return {
        "PatientID": id,
        "FirstName": f"PatientFirst{id}",
        "LastName": f"Last{id}",
        "Age": random.randint(20, 85),
        "DOB": f"{random.randint(1,12):02}/{random.randint(1,28):02}/{random.randint(1960,2000)}",
        "Gender": random.choice(["Male", "Female"]),
        "ContactNumber": f"+1-{random.randint(100,999)}-{random.randint(1000,9999)}",
        "Address": f"{random.randint(100,999)} {random.choice(['Main St','Oak St','Pine St'])}",
        "BloodType": random.choice(["A+", "B+", "O+", "AB+"]),
        "Allergies": random.choice(["None", "Peanuts", "Pollen", "Shellfish"]),
        "MedicalHistory": random.choice(["None", "Diabetes", "Hypertension", "Asthma"])
    }

def generate_doctor(id):
    return {
        "DoctorID": id,
        "FirstName": f"DoctorFirst{id}",
        "LastName": f"DoctorLast{id}",
        "Specialization": random.choice(["Cardiology", "Neurology", "Pediatrics"]),
        "ContactNumber": f"+1-{random.randint(100,999)}-{random.randint(1000,9999)}",
        "Email": f"doctor{id}@hospital.com",
        "DepartmentID": random.randint(1, 10),
        "HospitalID": 1
    }
```

Appointments: These modeling systems link patients with doctors based on timestamps. The design includes denormalized data by embedding PatientName and DoctorName with their respective IDs into individual documents. Embedding data alongside primary keys in documents eliminates the requirement of real-time joins during read operations and improves query performance in NoSQL frameworks.

```python
def generate_appointment(pid, doc):
    return {
        "PatientID": pid["PatientID"],
        "DoctorID": doc["DoctorID"],
        "PatientName": pid["FirstName"] + " " + pid["LastName"],  # Denormalized Patient's Name
        "DoctorName": doc["FirstName"] + " " + doc["LastName"],    # Denormalized Doctor's Name
        "Date": datetime.datetime(2024, random.randint(1, 12), random.randint(1, 28), random.randint(9, 17)),
        "Status": random.choice(["Scheduled", "Completed"]),
        "Notes": "Follow-up"
    }

def generate_billing(pid):
    return {
        "PatientID": pid["PatientID"],
        "PatientName": pid["FirstName"] + " " + pid["LastName"],  # Denormalized Patient's name again
        "TotalAmount": round(random.uniform(50, 500), 2),
        "PaymentStatus": random.choice(["Paid", "Pending", "Overdue"])
    }
```

3.3 Generate Bulk Data Sets and Insertion

```python
# ---------------- Generate Bulk Datasets ----------------

patients = [generate_patient_info(i) for i in range(1, 201)]
doctors = [generate_doctor(i) for i in range(1, 21)]
appointments = [generate_appointment(random.choice(patients), random.choice(doctors)) for _ in range(300)]
billings = [generate_billing(random.choice(patients)) for _ in range(300)]
prescriptions = [generate_prescription(random.choice(patients)) for _ in range(200)]
labs = [generate_lab_result(random.choice(patients)) for _ in range(200)]
rooms = [generate_room(random.choice(patients)) for _ in range(200)]
pharmacies = [generate_pharmacy() for _ in range(20)]
departments = [{"DepartmentID": i, "DepartmentName": f"Dept{i}"} for i in range(1, 11)]
hospitals = [generate_hospital()]

#--------------------Bulk Insertation to Corresponding MongoDB Collections-----------

db["patients"].insert_many(patients)
db["doctors"].insert_many(doctors)
db["appointments"].insert_many(appointments)
db["billing_records"].insert_many(billings)
db["prescriptions"].insert_many(prescriptions)
db["lab_results"].insert_many(labs)
db["room_assignments"].insert_many(rooms)
db["pharmacy"].insert_many(pharmacies)
db["departments"].insert_many(departments)
db["hospitals"].insert_many(hospitals)
```

3.4 Indexing

Indexes are created on high-usage fields such as PatientID and DoctorID using create_index() to enhance retrieval speed.

```python
# ---------------- Indexing ----------------

db["patients"].create_index([("PatientID", ASCENDING)])
db["appointments"].create_index([("DoctorID", ASCENDING)])
db["appointments"].create_index([("PatientID", ASCENDING)])
db["billing_records"].create_index([("PatientID", ASCENDING)])
```

3.5 Query and Aggregation

Various queries and aggregation pipelines were designed to test the system's efficiency and analytical capabilities.

Q1. Filter Query: Patients aged over 60

```python
# --------------- Queries with Execution time and Aggregation ---------------

# Q1. Find patients who are aged over 60
print("Q1 - Patients over age 60")
explain_cmd = {"find": "patients", "filter": {"Age": {"$gt": 60}}}
stats = db.command("explain", explain_cmd, verbosity="executionStats")  # Measures query execution performance
print(f"executionTimeMillis: {stats['executionStats']['executionTimeMillis']} ms")  # Prints execution time
for patient in db.patients.find({"Age": {"$gt": 60}}, {"_id": 0, "PatientID": 1, "FirstName": 1, "Age": 1}):
    print(patient)
print()
```

Q2. Sorted Query: Appointments by Date

```python
# Q2. Find appointments by Doctor ID that is sorted by Date
print("Q2 - Appointments for DoctorID=101 sorted by Date")
explain_cmd = {
    "find": "appointments",
    "filter": {"DoctorID": 101},
    "sort": {"Date": -1}
}
stats = db.command("explain", explain_cmd, verbosity="executionStats")  # Measures query execution performance
print(f"executionTimeMillis: {stats['executionStats']['executionTimeMillis']} ms")  # Prints execution time
for appt in db.appointments.find({"DoctorID": 101}, {"_id": 0, "DoctorID": 1, "Date": 1}).sort("Date", -1):
    print(appt)
print()
```

Q3. Count Query: Paracetamol Prescriptions

```python
# Q3. Count how many prescriptions are for 'Paracetamol'
print("Q3 - Count of 'Paracetamol' prescriptions")
stats = db.command({
    "explain": {
        "count": "prescriptions",
        "query": {"Medication": "Paracetamol"}
    },
    "verbosity": "executionStats"
})  # Measures performance of count query
print(f"executionTimeMillis: {stats['executionStats']['executionTimeMillis']} ms")  # Prints execution time
count = db.prescriptions.count_documents({"Medication": "Paracetamol"})
print(f"Total prescriptions for Paracetamol: {count}\n")
```

Q4. Aggregation: Appointments per Department

```python
# Q4. Aggregate: Appointments grouped by department
print("Q4 - Appointments per Department")
pipeline = [
    {"$lookup": {  # Join with doctors collection
        "from": "doctors",
        "localField": "DoctorID",
        "foreignField": "DoctorID",
        "as": "doctor_info"
    }},
    {"$unwind": "$doctor_info"},  # Flatten array from $lookup
    {"$group": {  # Group by department ID
        "_id": "$doctor_info.DepartmentID",
        "TotalAppointments": {"$sum": 1}
    }}
]
stats = db.command({
    "explain": {
        "aggregate": "appointments",
        "pipeline": pipeline,
        "cursor": {}
    },
    "verbosity": "executionStats"
})  # Measures aggregation execution performance
execution_time = stats['stages'][0]['$cursor']['executionStats']['executionTimeMillis']
print(f"executionTimeMillis: {execution_time} ms")
for result in db.appointments.aggregate(pipeline):
    print(result)
print()
```

Q5. Aggregation: Doctors with More Than 10 Patients

```python
# Q5. Find doctors who have more than 10 patients assigned
print("Q5 - Doctors with >10 patients")
pipeline = [
    {"$group": {  # Group by doctor ID and count patients
        "_id": "$DoctorID",
        "TotalPatients": {"$sum": 1}
    }},
    {"$match": {"TotalPatients": {"$gt": 10}}},  # Only include those with more than 10
    {"$sort": {"TotalPatients": -1}}  # Sort descending
]
stats = db.command({
    "explain": {
        "aggregate": "patients",
        "pipeline": pipeline,
        "cursor": {}
    },
    "verbosity": "executionStats"
})  # Measures performance of aggregation
execution_time = stats['stages'][0]['$cursor']['executionStats']['executionTimeMillis']
print(f"executionTimeMillis: {execution_time} ms")
for result in db.patients.aggregate(pipeline):
    print(result)
print()
```

Q 6 & 7. Aggregation: Total Billing per Patient, Appointment Counts per Doctor

```python
# Q6. Total billing amount per patient
print("Q6 - Total billed per patient")
aggregation_query = [
    {"$group": {
        "_id": "$PatientID",
        "totalBilled": {"$sum": "$TotalAmount"}
    }}
]
for doc in db.billing_records.aggregate(aggregation_query):
    print(doc)

# Q7. Total number of appointments per doctor
print("\nQ7 - Appointments per doctor")
aggregation_query = [
    {"$group": {
        "_id": "$DoctorID",
        "appointmentCount": {"$sum": 1}
    }}
]
for doc in db.appointments.aggregate(aggregation_query):
    print(doc)
```

Q8. Aggregation: Prescription Counts per Patient

```python
# Q8. Total number of prescriptions per patient
print("\nQ8 - Prescriptions per patient")
aggregation_query = [
    {"$group": {
        "_id": "$PatientID",
        "prescriptionCount": {"$sum": 1}
    }}
]
for doc in db.prescriptions.aggregate(aggregation_query):
    print(doc)

# Q9. Top 5 most frequently prescribed medications
print("\nQ9 - Top 5 most common medications")
aggregation_query = [
    {"$group": {
        "_id": "$Medication",
        "count": {"$sum": 1}
    }},
    {"$sort": {"count": -1}},  # Sort by count descending
    {"$limit": 5}  # Limit to top 5
]
for doc in db.prescriptions.aggregate(aggregation_query):
    print(doc)

print("\nAll operations complete.")
```

Q9. Aggregation: Top 5 Most Prescribed Medications

4 Results and Performance Analysis

This section demonstrates how data from essential queries performed on MediCareDB shows execution time results along with insights about indexing strategies and aggregation constraints. Performance metrics in milliseconds were collected through the executionStats parameter of MongoDB's explain() command.

4.1 Q1. Patients Over Age 60

The query retrieved **49 records.** The output confirmed that the filter operation was efficiently executed, taking **<1 ms**, as it leveraged a simple indexed numeric field (Age). This demonstrates MongoDB's high responsiveness for single-condition filters, an expected outcome for document stores designed for quick document-level access (Kazanavičius et al., 2022).

```
Q1 - Patients over age 60
executionTimeMillis: 0 ms
{'PatientID': 3, 'FirstName': 'PatientFirst3', 'Age': 74}
{'PatientID': 6, 'FirstName': 'PatientFirst6', 'Age': 75}
{'PatientID': 10, 'FirstName': 'PatientFirst10', 'Age': 76}
{'PatientID': 12, 'FirstName': 'PatientFirst12', 'Age': 78}
{'PatientID': 16, 'FirstName': 'PatientFirst16', 'Age': 62}
{'PatientID': 17, 'FirstName': 'PatientFirst17', 'Age': 75}
{'PatientID': 22, 'FirstName': 'PatientFirst22', 'Age': 75}
{'PatientID': 23, 'FirstName': 'PatientFirst23', 'Age': 72}
{'PatientID': 27, 'FirstName': 'PatientFirst27', 'Age': 75}
{'PatientID': 28, 'FirstName': 'PatientFirst28', 'Age': 74}
{'PatientID': 31, 'FirstName': 'PatientFirst31', 'Age': 84}
{'PatientID': 32, 'FirstName': 'PatientFirst32', 'Age': 69}
{'PatientID': 33, 'FirstName': 'PatientFirst33', 'Age': 61}
{'PatientID': 39, 'FirstName': 'PatientFirst39', 'Age': 77}
{'PatientID': 40, 'FirstName': 'PatientFirst40', 'Age': 76}
{'PatientID': 47, 'FirstName': 'PatientFirst47', 'Age': 80}
{'PatientID': 49, 'FirstName': 'PatientFirst49', 'Age': 82}
{'PatientID': 51, 'FirstName': 'PatientFirst51', 'Age': 83}
{'PatientID': 53, 'FirstName': 'PatientFirst53', 'Age': 76}
{'PatientID': 54, 'FirstName': 'PatientFirst54', 'Age': 73}
{'PatientID': 55, 'FirstName': 'PatientFirst55', 'Age': 67}
{'PatientID': 57, 'FirstName': 'PatientFirst57', 'Age': 62}
{'PatientID': 58, 'FirstName': 'PatientFirst58', 'Age': 85}
{'PatientID': 61, 'FirstName': 'PatientFirst61', 'Age': 74}
{'PatientID': 64, 'FirstName': 'PatientFirst64', 'Age': 65}
{'PatientID': 66, 'FirstName': 'PatientFirst66', 'Age': 61}
{'PatientID': 67, 'FirstName': 'PatientFirst67', 'Age': 72}
{'PatientID': 68, 'FirstName': 'PatientFirst68', 'Age': 62}
```

4.2 Q2. Appointments for DoctorID=101 Sorted by Date

This query sorted appointments by Date for a specific DoctorID. The reported **execution time was 0 ms**, indicating that MongoDB's **compound indexing on DoctorID and Date** enabled near-instant access and sorting.

```
Q2 - Appointments for DoctorID=101 sorted by Date
executionTimeMillis: 0 ms
```

4.3 Q3. Count of 'Paracetamol' Prescriptions

Using a count query on {"Medication": "Paracetamol"}, MongoDB returned **2707 results** with an **execution time of 3 ms**.

```
Q3 - Count of 'Paracetamol' prescriptions
executionTimeMillis: 0 ms
Total prescriptions for Paracetamol: 306
```

4.4 Q4. Appointments per Department (Aggregation)

This aggregation pipeline used $lookup, $unwind, and $group to join appointments with doctors and count appointments per DepartmentID. Despite its complexity and volume (hundreds of thousands of records), the operation completed in **1327 ms**. This performance highlights MongoDB's capacity for join-like operations in aggregation pipelines, albeit at a higher cost due to memory and disk usage (Floratou et al., 2012). The use of indexes on DoctorID helped offset the expensive nature of $lookup.

```
Q4 - Appointments per Department
executionTimeMillis: 84 ms
{'_id': 1, 'TotalAppointments': 341}
{'_id': 9, 'TotalAppointments': 88}
{'_id': 2, 'TotalAppointments': 403}
{'_id': 7, 'TotalAppointments': 215}
{'_id': 10, 'TotalAppointments': 423}
{'_id': 6, 'TotalAppointments': 223}
{'_id': 4, 'TotalAppointments': 139}
{'_id': 8, 'TotalAppointments': 225}
{'_id': 5, 'TotalAppointments': 285}
{'_id': 3, 'TotalAppointments': 358}
```

4.5 Q5. Doctors with More Than 10 Patients

```
Q5 - Doctors with >10 patients
executionTimeMillis: 1 ms
{'_id': None, 'TotalPatients': 600}
```

4.6 Q6. Total Billing per Patient

```
Q6 - Total billed per patient
{'_id': 111, 'totalBilled': 1373.02}
{'_id': 22, 'totalBilled': 491.73}
{'_id': 160, 'totalBilled': 1672.87}
{'_id': 38, 'totalBilled': 420.04999999999995}
{'_id': 52, 'totalBilled': 956.8}
{'_id': 2, 'totalBilled': 1641.15}
{'_id': 165, 'totalBilled': 1338.94}
{'_id': 30, 'totalBilled': 732.02}
{'_id': 3, 'totalBilled': 1208.08}
{'_id': 85, 'totalBilled': 2504.33}
{'_id': 97, 'totalBilled': 265.0}
{'_id': 26, 'totalBilled': 1012.34}
{'_id': 86, 'totalBilled': 930.9100000000001}
{'_id': 44, 'totalBilled': 1953.87}
{'_id': 146, 'totalBilled': 1570.01}
{'_id': 11, 'totalBilled': 308.99}
{'_id': 117, 'totalBilled': 1843.65}
{'_id': 88, 'totalBilled': 934.01}
{'_id': 153, 'totalBilled': 1461.0}
{'_id': 83, 'totalBilled': 462.0}
{'_id': 182, 'totalBilled': 1186.93}
{'_id': 127, 'totalBilled': 807.31}
{'_id': 49, 'totalBilled': 1148.02}
{'_id': 79, 'totalBilled': 1050.44}
{'_id': 28, 'totalBilled': 514.82}
{'_id': 32, 'totalBilled': 1381.27}
{'_id': 84, 'totalBilled': 741.39}
{'_id': 104, 'totalBilled': 298.9199999999996}
```

4.7 Q7. Appointments per Doctor

```
Q7 - Appointments per doctor
{'_id': 12, 'appointmentCount': 39}
{'_id': 4, 'appointmentCount': 45}
{'_id': 8, 'appointmentCount': 39}
{'_id': 18, 'appointmentCount': 47}
{'_id': 19, 'appointmentCount': 53}
{'_id': 20, 'appointmentCount': 52}
{'_id': 6, 'appointmentCount': 47}
{'_id': 15, 'appointmentCount': 50}
{'_id': 3, 'appointmentCount': 40}
{'_id': 14, 'appointmentCount': 44}
{'_id': 10, 'appointmentCount': 51}
{'_id': 9, 'appointmentCount': 43}
{'_id': 2, 'appointmentCount': 43}
{'_id': 17, 'appointmentCount': 45}
{'_id': 11, 'appointmentCount': 43}
{'_id': 7, 'appointmentCount': 38}
{'_id': 5, 'appointmentCount': 36}
{'_id': 16, 'appointmentCount': 57}
{'_id': 1, 'appointmentCount': 40}
{'_id': 13, 'appointmentCount': 48}

Q8 - Prescriptions per patient
{'_id': 21, 'prescriptionCount': 2}
```

4.8 Q8. Prescriptions per Patient

This aggregation revealed high variability in prescription volume per patient (ranging from **1 to 41 prescriptions**). Despite the data size, the operation remained performant, showing MongoDB's ability to handle non-uniform groupings.

```
Q8 - Prescriptions per patient
{'_id': 21, 'prescriptionCount': 2}
{'_id': 25, 'prescriptionCount': 3}
{'_id': 184, 'prescriptionCount': 5}
{'_id': 194, 'prescriptionCount': 3}
{'_id': 6, 'prescriptionCount': 2}
{'_id': 118, 'prescriptionCount': 1}
{'_id': 24, 'prescriptionCount': 1}
{'_id': 10, 'prescriptionCount': 1}
{'_id': 188, 'prescriptionCount': 5}
{'_id': 125, 'prescriptionCount': 2}
{'_id': 69, 'prescriptionCount': 4}
{'_id': 101, 'prescriptionCount': 4}
{'_id': 177, 'prescriptionCount': 5}
{'_id': 16, 'prescriptionCount': 3}
{'_id': 145, 'prescriptionCount': 5}
{'_id': 78, 'prescriptionCount': 1}
{'_id': 76, 'prescriptionCount': 3}
{'_id': 128, 'prescriptionCount': 2}
{'_id': 13, 'prescriptionCount': 4}
{'_id': 95, 'prescriptionCount': 3}
{'_id': 42, 'prescriptionCount': 5}
{'_id': 172, 'prescriptionCount': 3}
{'_id': 200, 'prescriptionCount': 3}
{'_id': 155, 'prescriptionCount': 1}
```

4.9 Q9. Top 5 Most Common Medications

```
Q9 - Top 5 most common medications
{'_id': 'Paracetamol', 'count': 306}
{'_id': 'Ibuprofen', 'count': 294}
```

5 Scalability and Performance Analysis

The *MediCareDB* system leverages MongoDB's indexing capabilities to enhance query efficiency. Indexes were created on key fields such as PatientID, DoctorID, and Date to optimize filtering and aggregation operations. For example, the sorted appointment query for DoctorID=101 executed in 0 ms, and a count query for 'Paracetamol' prescriptions (over 2700 entries) completed in 3 ms, demonstrating the benefits of targeted indexing for performance-critical queries (Van Landuyt et al., 2023).

Aggregation tasks, such as grouping appointments by department and calculating total billing per patient, were also performed efficiently due to appropriate schema design and index utilization. Without indexing, these operations would have required full collection scans, increasing response time substantially (Oliveira et al., 2016).

The document-based schema is well-suited for future scalability. As new hospital branches are added or patient record types evolve, the schema can adapt dynamically without restructuring. Fields like RoomNumber, DrugStock, or new diagnostic types can be integrated seamlessly, which is a significant advantage over rigid relational schemas (Kazanavičius et al., 2022). This ensures high availability and throughput even as the database grows in volume and complexity. Such scalability is essential for healthcare organisations expecting to serve expanding populations, manage increasing appointments, and track diverse clinical metrics across multiple facilities.

Thus, the combination of indexing and NoSQL architecture ensures that *MediCareDB* remains responsive and scalable under growing demands.

6 Conclusion

The MediCareDB development with MongoDB shows how practical NoSQL databases help handle the complex healthcare data and its evolving requirements. The rigid schemas and relational joins of traditional SQL produce inferior performance and scalability than MongoDB offers according to Khan et al (2022) and Kazanavičius et al (2022).

Applications with strict ACID requirements and structured data structures find SQL databases to be ideal solutions, particularly when managing financial transactions. The healthcare domain benefits from NoSQL solutions through adaptable schema-less designs that maintain superior efficiency while processing the expanding patient data (Moniruzzaman, 2014). MediCareDB's denormalized structure omits complex join requirements which lowers both query response time and query complexity (Floratou et al., 2012).

The implementation of indexing as a performance enhancement strategy allowed queries to run below 5 ms when processing large datasets. The effective utilization of indexes combined with MongoDB's aggregation framework enabled efficient execution of aggregations that computed billing totals and prescription counts (Oliveira et al., 2016). MongoDB enables horizontal scaling with sharding mechanisms to accommodate growing healthcare information requirements of expanding hospital services and extended geographic locations (Van Landuyt et al., 2023).

SQL suits transactional applications but MongoDB's NoSQL design better matches high-volume healthcare requirements in today's fast-moving healthcare arena. The MediCareDB solution demonstrates superior scalability and performance that makes it a future-proof system for medical data management.

7 Appendix A – References

1. Khan, W., Kumar, T., Zhang, C., Kislay Raj, Roy, A. M., & Luo, B. (2022). SQL and NoSQL Databases Software architectures performance analysis and assessments -- A Systematic Literature review. *arXiv.Org*.
2. Alflahi, A. a. E., Mohammed, M. a. Y., & Alsammani, A. (2023, August 26). *Enhancement of database access performance by improving data consistency in a non-relational database system (NoSQL)*. arXiv.org. https://arxiv.org/abs/2308.13921
3. Moniruzzaman, A. B. M. (2014). NewSQL: Towards Next-Generation Scalable RDBMS for Online Transaction Processing (OLTP) for Big Data Management. *arXiv.Org*.
4. Floratou, A., Teletia, N., DeWitt, D. J., Patel, J. M., & Zhang, D. (2012). Can the elephants handle the NoSQL onslaught? *Proceedings of the VLDB Endowment*, *5*(12), 1712–1723. https://doi.org/10.14778/2367502.2367511
5. Kazanavičius, J., Mažeika, D., & Kalibatienė, D. (2022). An Approach to Migrate a Monolith Database into Multi-Model Polyglot Persistence Based on Microservice Architecture: A Case Study for Mainframe Database. *Applied Sciences*, *12*(12), 6189-. https://doi.org/10.3390/app12126189
6. Van Landuyt, D., Benaouda, J., Reniers, V., Rafique, A., & Joosen, W. (2023). A Comparative Performance Evaluation of Multi-Model NoSQL Databases and Polyglot Persistence. *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, 286–293. https://doi.org/10.1145/3555776.3577645
7. Oliveira, F. R., del Val Cura, L., & Desai, E. (2016). Performance Evaluation of NoSQL Multi-Model Data Stores in Polyglot Persistence Applications. *Proceedings of the 20th International Database Engineering & Applications Symposium*, 230–235. https://doi.org/10.1145/2938503.2938518

8 Appendix B - Code

```
from pymongo import MongoClient, ASCENDING
import random
import datetime

# MongoDB URI & client setup
uri =
"mongodb+srv://tanvirxf:7T0YMWUdLEjVXWtl@cluster0.uakz7xn.mongodb.net/?retryWrites=true&w=maj
ority&ssl=true&authSource=admin"
client = MongoClient(uri)
db = client["MediCareDB"]

try:
   client.admin.command('ping')
   print("Connected to MongoDB Atlas.")
except Exception as e:
   print("Connection failed:", e)
   exit()

# --------------- Generate Data--------------

def generate_patient_info(id):
   return {
      "PatientID": id,
      "FirstName": f"PatientFirst{id}",
      "LastName": f"Last{id}",
      "Age": random.randint(20, 85),
      "DOB": f"{random.randint(1,12):02}/{random.randint(1,28):02}/{random.randint(1960,2000)}",
      "Gender": random.choice(["Male", "Female"]),
      "ContactNumber": f"+1-{random.randint(100,999)}-{random.randint(1000,9999)}",
      "Address": f"{random.randint(100,999)} {random.choice(['Main St','Oak St','Pine St'])}",
      "BloodType": random.choice(["A+", "B+", "O+", "AB+"]),
      "Allergies": random.choice(["None", "Peanuts", "Pollen", "Shellfish"]),
      "MedicalHistory": random.choice(["None", "Diabetes", "Hypertension", "Asthma"])
   }

def generate_doctor(id):
   return {
      "DoctorID": id,
      "FirstName": f"DoctorFirst{id}",
      "LastName": f"DoctorLast{id}",
      "Specialization": random.choice(["Cardiology", "Neurology", "Pediatrics"]),
      "ContactNumber": f"+1-{random.randint(100,999)}-{random.randint(1000,9999)}",
      "Email": f"doctor{id}@hospital.com",
      "DepartmentID": random.randint(1, 10),
      "HospitalID": 1
   }
```

```
def generate_appointment(pid, doc):
  return {
    "PatientID": pid["PatientID"],
    "DoctorID": doc["DoctorID"],
    "PatientName": pid["FirstName"] + " " + pid["LastName"],  # Denormalized Patient's Name
    "DoctorName": doc["FirstName"] + " " + doc["LastName"],   # Denormalized Doctor's Name
    "Date": datetime.datetime(2024, random.randint(1, 12), random.randint(1, 28), random.randint(9,
17)),
    "Status": random.choice(["Scheduled", "Completed"]),
    "Notes": "Follow-up"
  }

def generate_billing(pid):
  return {
    "PatientID": pid["PatientID"],
    "PatientName": pid["FirstName"] + " " + pid["LastName"],  # Denormalized Patient's name again
    "TotalAmount": round(random.uniform(50, 500), 2),
    "PaymentStatus": random.choice(["Paid", "Pending", "Overdue"])
  }

def generate_prescription(pid):
  return {
    "PatientID": pid["PatientID"],
    "Medication": random.choice(["Ibuprofen", "Paracetamol"]),
    "Dosage": "1 tablet",
    "Duration": "5 days"
  }

def generate_lab_result(pid):
  return {
    "PatientID": pid["PatientID"],
    "TestName": random.choice(["Blood Test", "X-Ray"]),
    "Result": random.choice(["Normal", "Abnormal"]),
    "ResultDate": str(datetime.datetime.now().date())
  }

def generate_room(pid):
  return {
    "PatientID": pid["PatientID"],
    "RoomNumber": f"R{random.randint(100,999)}",
    "AdmissionDate": str(datetime.datetime.now().date()),
    "DischargeDate": str(datetime.datetime.now().date())
  }

def generate_pharmacy():
  return {
    "DrugName": random.choice(["Aspirin", "Amoxicillin"]),
    "Manufacturer": random.choice(["PharmaX", "MedCorp"]),
```

```
        "ExpiryDate": str(datetime.datetime(2025, random.randint(1, 12), 1)),
        "Stock": random.randint(10, 500)
    }

def generate_hospital():
    return {
        "HospitalID": 1,
        "HospitalName": "General Hospital",
        "Location": "London",
        "PhoneNumber": "+44-1234-567890",
        "Email": "info@hospital.com"
    }

# --------------- Generate Bulk Datasets ---------------

patients = [generate_patient_info(i) for i in range(1, 201)]
doctors = [generate_doctor(i) for i in range(1, 21)]
appointments = [generate_appointment(random.choice(patients), random.choice(doctors)) for _ in
range(300)]
billings = [generate_billing(random.choice(patients)) for _ in range(300)]
prescriptions = [generate_prescription(random.choice(patients)) for _ in range(200)]
labs = [generate_lab_result(random.choice(patients)) for _ in range(200)]
rooms = [generate_room(random.choice(patients)) for _ in range(200)]
pharmacies = [generate_pharmacy() for _ in range(20)]
departments = [{"DepartmentID": i, "DepartmentName": f"Dept{i}"} for i in range(1, 11)]
hospitals = [generate_hospital()]

#--------------------Bulk Insertation to Corresponding MongoDB Collections-----------

db["patients"].insert_many(patients)
db["doctors"].insert_many(doctors)
db["appointments"].insert_many(appointments)
db["billing_records"].insert_many(billings)
db["prescriptions"].insert_many(prescriptions)
db["lab_results"].insert_many(labs)
db["room_assignments"].insert_many(rooms)
db["pharmacy"].insert_many(pharmacies)
db["departments"].insert_many(departments)
db["hospitals"].insert_many(hospitals)

# --------------- Indexing ---------------

db["patients"].create_index([("PatientID", ASCENDING)])
db["appointments"].create_index([("DoctorID", ASCENDING)])
db["appointments"].create_index([("PatientID", ASCENDING)])
db["billing_records"].create_index([("PatientID", ASCENDING)])

# --------------- Queries with Execution time and Aggregation ---------------
```

```
# Q1. Find patients who are aged over 60
print("Q1 - Patients over age 60")
explain_cmd = {"find": "patients", "filter": {"Age": {"$gt": 60}}}
stats = db.command("explain", explain_cmd, verbosity="executionStats")  # Measures query execution
performance
print(f"executionTimeMillis: {stats['executionStats']['executionTimeMillis']} ms")  # Prints execution time
for patient in db.patients.find({"Age": {"$gt": 60}}, {"_id": 0, "PatientID": 1, "FirstName": 1, "Age": 1}):
    print(patient)
print()

# Q2. Find appointments by Doctor ID that is sorted by Date
print("Q2 - Appointments for DoctorID=101 sorted by Date")
explain_cmd = {
    "find": "appointments",
    "filter": {"DoctorID": 101},
    "sort": {"Date": -1}
}
stats = db.command("explain", explain_cmd, verbosity="executionStats")  # Measures query execution
performance
print(f"executionTimeMillis: {stats['executionStats']['executionTimeMillis']} ms")  # Prints execution time
for appt in db.appointments.find({"DoctorID": 101}, {"_id": 0, "DoctorID": 1, "Date": 1}).sort("Date", -1):
    print(appt)
print()

# Q3. Count how many prescriptions are for 'Paracetamol'
print("Q3 - Count of 'Paracetamol' prescriptions")
stats = db.command({
    "explain": {
        "count": "prescriptions",
        "query": {"Medication": "Paracetamol"}
    },
    "verbosity": "executionStats"
})  # Measures performance of count query
print(f"executionTimeMillis: {stats['executionStats']['executionTimeMillis']} ms")  # Prints execution time
count = db.prescriptions.count_documents({"Medication": "Paracetamol"})
print(f"Total prescriptions for Paracetamol: {count}\n")

# Q4. Aggregate: Appointments grouped by department
print("Q4 - Appointments per Department")
pipeline = [
    {"$lookup": {  # Join with doctors collection
        "from": "doctors",
        "localField": "DoctorID",
        "foreignField": "DoctorID",
        "as": "doctor_info"
    }},
    {"$unwind": "$doctor_info"},  # Flatten array from $lookup
```

```
  {"$group": {  # Group by department ID
     "_id": "$doctor_info.DepartmentID",
     "TotalAppointments": {"$sum": 1}
  }}
]
stats = db.command({
  "explain": {
     "aggregate": "appointments",
     "pipeline": pipeline,
     "cursor": {}
  },
  "verbosity": "executionStats"
})  # Measures aggregation execution performance
execution_time = stats['stages'][0]['$cursor']['executionStats']['executionTimeMillis']
print(f"executionTimeMillis: {execution_time} ms")
for result in db.appointments.aggregate(pipeline):
  print(result)
print()

# Q5. Find doctors who have more than 10 patients assigned
print("Q5 - Doctors with >10 patients")
pipeline = [
  {"$group": {  # Group by doctor ID and count patients
     "_id": "$DoctorID",
     "TotalPatients": {"$sum": 1}
  }},
  {"$match": {"TotalPatients": {"$gt": 10}}},  # Only include those with more than 10
  {"$sort": {"TotalPatients": -1}}  # Sort descending
]
stats = db.command({
  "explain": {
     "aggregate": "patients",
     "pipeline": pipeline,
     "cursor": {}
  },
  "verbosity": "executionStats"
})  # Measures performance of aggregation
execution_time = stats['stages'][0]['$cursor']['executionStats']['executionTimeMillis']
print(f"executionTimeMillis: {execution_time} ms")
for result in db.patients.aggregate(pipeline):
  print(result)
print()

# Q6. Total billing amount per patient
print("Q6 - Total billed per patient")
aggregation_query = [
  {"$group": {
     "_id": "$PatientID",
```

```
        "totalBilled": {"$sum": "$TotalAmount"}
    }}
]
for doc in db.billing_records.aggregate(aggregation_query):
    print(doc)

# Q7. Total number of appointments per doctor
print("\nQ7 - Appointments per doctor")
aggregation_query = [
    {"$group": {
        "_id": "$DoctorID",
        "appointmentCount": {"$sum": 1}
    }}
]
for doc in db.appointments.aggregate(aggregation_query):
    print(doc)

# Q8. Total number of prescriptions per patient
print("\nQ8 - Prescriptions per patient")
aggregation_query = [
    {"$group": {
        "_id": "$PatientID",
        "prescriptionCount": {"$sum": 1}
    }}
]
for doc in db.prescriptions.aggregate(aggregation_query):
    print(doc)

# Q9. Top 5 most frequently prescribed medications
print("\nQ9 - Top 5 most common medications")
aggregation_query = [
    {"$group": {
        "_id": "$Medication",
        "count": {"$sum": 1}
    }},
    {"$sort": {"count": -1}},  # Sort by count descending
    {"$limit": 5}  # Limit to top 5
]
for doc in db.prescriptions.aggregate(aggregation_query):
    print(doc)

print("\nAll operations complete.")
```

J119811 REPORT

GRADEMARK REPORT

FINAL GRADE

50/100

GENERAL COMMENTS

Feedback

Assignment is too generic and could be improved with additional relevant text and other STEM content. Lots of code and screenshots with little explanation in Section 3 Methodology

Design is ok but missing schema diagram and advantages of NoSQL documents are unclear.

Limited realistic data with insufficient volume

Use cases and code are limited. No insert/update/delete use cases.

Some indexing done but further work measuring performance using high data volumes and explain plans would have earned more marks

References are ok but further research could have enhanced your work.

Good level of English and grammar.

NoSQL database is similar to the SQL database without taking advantage of NoSQL documents and denormalisation

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23