Distributed Electronic Health Record (EHR) System Using MongoDB and PostgreSQL

Srushti Pujari
Department of Computer Science
Arizona State University
Tempe, Arizona, USA
spujari6@asu.edu
1234081422

Sakshi Bajaj
Department of Computer Science
Arizona State University
Tempe, Arizona, USA
sbajaj21@asu.edu
1233409868

Bhavesh Haresh Khubnani
Department of Computer Science
Arizona State University
Tempe, Arizona, USA
bkhubnan@asu.edu
1229596630

Shrenik Podduturi
Department of Computer Science
Arizona State University
Tempe, Arizona, USA
spoddut3@asu.edu
1229935293

***Abstract*—The rapid growth of healthcare systems has highlighted the need for robust and scalable data management solutions. Traditional centralized databases often fail to address the challenges of latency, scalability, fault tolerance, and data consistency required for managing nationwide Electronic Health Records (EHR). This project introduces a distributed EHR system utilizing MongoDB for document-based data and PostgreSQL for relational transactional data. By implementing advanced techniques like sharding, partitioning, and replica sets, the system ensures low-latency access, fault tolerance, and data integrity while adhering to healthcare privacy regulations such as HIPAA. This paper presents the design, implementation, and evaluation of the system, showcasing its effectiveness in addressing modern healthcare data challenges.**

***Index Terms*—Distributed systems, MongoDB, PostgreSQL, Sharding, Partitioning, Fault tolerance, Healthcare data management**

I. Introduction

A. Background

The exponential growth of healthcare systems globally and the increasing demand for real-time access to patient records have necessitated the development of distributed solutions for Electronic Health Records (EHR). Centralized database architectures often struggle to meet the requirements of modern healthcare systems due to bottlenecks, single points of failure, and inefficiencies in scaling across regions. Distributed database systems offer a promising alternative by providing low-latency data access, fault tolerance, and the ability to handle massive datasets efficiently while maintaining data privacy and consistency.

Existing EHR solutions frequently rely on monolithic, region-specific databases, resulting in data silos that hinder seamless integration across institutions. This approach increases latency for patient data access and compromises the ability to provide timely healthcare services. Our project seeks to address these limitations by leveraging distributed databases, namely MongoDB and PostgreSQL, to create a scalable, efficient, and resilient healthcare data management system.

B. Problem Statement

Managing and accessing EHRs across a national network of hospitals, clinics, and healthcare providers presents significant challenges. Centralized systems often face bottlenecks and single points of failure, while poorly replicated systems result in high-latency data access and inconsistencies. These shortcomings can delay critical healthcare decisions and compromise data reliability and security.

This project aims to design and implement a distributed EHR system that:

- Provides low-latency access to patient records.
- Ensures robust data replication across regions for reliability and fault tolerance.
- Maintains data privacy and compliance with regulations such as HIPAA.
- Demonstrates resilience to failures, ensuring availability even during outages.

C. Objectives

- Design and implement a hybrid EHR system using MongoDB for unstructured NoSQL data and PostgreSQL for relational transactional data.
- Partition patient data geographically to optimize data access and reduce latency.
- Explore replication strategies to handle real-time updates and maintain fault tolerance across regions.

- Demonstrate consistency, availability, and scalability through rigorous testing and evaluation.
- Ensure the system complies with healthcare data standards like HIPAA to protect sensitive information.

D. Scope

The scope of this project encompasses the design, implementation, and evaluation of a distributed database system to manage healthcare data efficiently. The project focuses on:

- Developing a scalable system architecture using modern distributed database techniques.
- Implementing robust fault tolerance mechanisms, including replica management and failover handling.
- Providing healthcare professionals with a secure and user-friendly interface to manage patient records and operational data.
- Evaluating the system through comprehensive performance metrics, such as latency, throughput, and data consistency.

II. System Design

A. Overview

The system design leverages the strengths of both NoSQL and relational databases to handle the diverse nature of healthcare data. MongoDB is used to store unstructured and semi-structured data such as patient medical records, while PostgreSQL handles structured data like billing and admissions.

B. Architecture

The architecture integrates MongoDB and PostgreSQL in a hybrid distributed model, supported by middleware and user interfaces for seamless interaction. Figure 1 illustrates the architecture of the system.

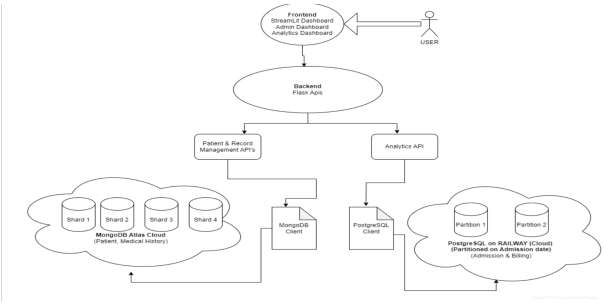

Fig. 1. System Architecture

- MongoDB: Deployed with four logical shards, each with replica sets (one primary and two secondaries). Sharding distributes data geographically to optimize query performance and balance workloads.
- PostgreSQL: Data is partitioned by admission date to optimize query performance for time-sensitive transactional data.
- Middleware: Flask provides API endpoints for data access and management, facilitating seamless interaction between databases and the frontend.
- Frontend: Streamlit delivers a responsive and user-friendly interface for healthcare providers to manage patient data and perform analytics.

C. Implementation Details

- Languages: Python for backend services, JavaScript for frontend interactivity.
- Databases: MongoDB for unstructured data, PostgreSQL for structured data.
- Tools and Frameworks: Flask for backend APIs, Streamlit for frontend UI, MongoDB Atlas for cloud-based NoSQL management, and Railway for PostgreSQL hosting.
- Libraries: pymongo for MongoDB operations, psycopg2 for PostgreSQL queries, and Pandas for data analysis and visualization.

D. MongoDB Sharding and Replication

To optimize performance and scalability, MongoDB employs sharding, where data is distributed across four logical shards based on the '$patient\_id$'. This ensures balanced data distribution and reduces query latency for large-scale datasets.

1) Shard Configuration:

- Shard Key: The '$patient\_id$' field is used as the shard key, ensuring that patient data is evenly distributed across the shards.
- Number of Shards: The system uses four shards ('$shard\_0$' to '$shard\_3$'), each responsible for a subset of the data based on the hash of the '$patient\_id$'.
- Replication: Each shard is replicated across three nodes to ensure fault tolerance. One node acts as the primary, while the other two are secondary replicas.

2) Replica Management and Node Failures: To handle node failures, the system employs a dynamic failover mechanism:

- Primary-Secondary Configuration: The primary node handles all write operations, while secondary nodes support read operations and act as backups.
- Simulated Node Failure: An API was implemented to simulate node failure. When a secondary node is selected to become the primary, the system updates the in-memory replica status and redirects subsequent queries to the new primary.
- In-Memory Status Tracking: Replica statuses ('$primary$, $secondary\_1$, $secondary\_2$') are maintained in memory, allowing the system to quickly adapt to changes in node availability.

3) Benefits of Sharding:

- Scalability: Data distribution across shards allows the system to handle large datasets efficiently.
- Reduced Latency: Queries are directed to specific shards, minimizing the volume of data scanned during operations.
- Fault Tolerance: Replication ensures data availability even during node failures, with seamless failover to secondary nodes.

E. PostgreSQL Partitioning

PostgreSQL utilizes partitioning to manage structured transactional data efficiently. The '$admissions$' table is partitioned by the '$admission\_date$' field, grouping records by admission year.

1) Partition Configuration:

- Partition Key: The '$admission\_date$' column is used as the partition key. Data is divided into partitions based on the year extracted from the '$admission\_date$'.
- Partition Tables: Each partition corresponds to a specific year (e.g., '$admissions\_2023$, $admissions\_2024$'), enabling efficient query execution for time-specific data.
- Default Partition: A default partition captures records that do not match existing partition criteria, ensuring no data is lost.

2) Benefits of Partitioning:

- Improved Query Performance: Queries targeting specific years only scan the relevant partition, reducing execution time significantly.
- Data Management: Partitioning simplifies archival and deletion of old data by allowing operations on individual partitions.
- Scalability: As new years are added, additional partitions can be created dynamically without impacting existing data.

F. Data Generation Process

- Synthetic Data Tools: We used the Synthia framework to generate structured patient demographics and admissions data, ensuring variation in age, gender, and geographic distribution.
- LLM Assistance: ChatGPT was utilized to generate realistic and diverse patient medical histories, including diagnoses, medications, and test results. This allowed for detailed and contextually accurate medical records.
- Manual Curation: The generated data was manually reviewed and refined to maintain quality and accuracy, ensuring that it adhered to clinical standards and was free of inconsistencies.

G. Data Structure

The data is represented as JSON documents for MongoDB, capturing unstructured medical records, while PostgreSQL stores structured transactional data like admissions and billing. Below is an example of a MongoDB document:


```
{
    "patient_id": "a4683dca",
    "name": "Sheena Freeman",
    "age": 60,
    "gender": "Male",
    "contact_details": {
        "phone": "(883)598-6531",
        "email": "abigailbridges@santiago-cabrera.
            com",
        "address": "003 Young Pine\nRicardoport, HI
            16725"
    },
    "medical_records": {
        "medical_history": [
            {
                "visit_id": "V000",
                "visit_date": "2022-01-09",
                "reason": "Follow-up",
                "diagnosis": "Fracture",
                "observations": "Swelling and pain
                    around injury, apply cold packs
                    and painkillers.",
                "medications": [
                    {
                        "name": "Paracetamol",
                        "dose": "356mg",
                        "frequency": "Twice daily",
                        "duration": "13 days"
                    }
                ],
                "tests": [
                    {
                        "test_name": "Blood Test",
                        "results": {
                            "value": 126
                        },
                        "recommendations": "Increase
                             fiber intake to manage
                            blood sugar and
                            cholesterol levels."
                    }
                ]
            }
        ]
    }
}
```


H. Data Distribution

- MongoDB: Stores unstructured data, such as patient demographics, contact details, and detailed medical records, including medical history, diagnoses, medications, and test results.
- PostgreSQL: Manages structured data related to patient admissions and billing, including admission dates, discharge summaries, billing amounts, and payment statuses.

I. Data Strategy

The data strategy ensures scalability, privacy, and compliance:

- Synthetic Data Generation: Generated synthetic patient data to replicate real-world scenarios while protecting privacy.
- Data Distribution:
  - Patient demographics, medical histories, and prescriptions stored in MongoDB.
  - Admissions, billing, and operational data stored in PostgreSQL.
- Compliance: Ensured all data management processes adhered to HIPAA standards for data privacy and security.

III. Methodology and Techniques

This chapter highlights the methodologies and distributed database concepts employed:

- Sharding: Implemented in MongoDB to distribute data geographically and improve performance for read and write operations.

- Partitioning: Used in PostgreSQL to manage time-sensitive data efficiently, reducing query execution times.
- Replica Management: MongoDB replicas ensured high availability and reliability, with automatic failover to secondary nodes during failures.
- Consistency Checks: Verified data synchronization across primary and secondary replicas to maintain system integrity.

IV. Evaluation

The system was evaluated against the following metrics:

- Latency: MongoDB queries showed a 20% improvement in response times after sharding. PostgreSQL queries on partitioned tables executed 30% faster than on non-partitioned tables.
- Fault Tolerance: The system handled replica failures gracefully, redirecting queries to available nodes with minimal downtime.
- Scalability: The hybrid architecture supported increased data volumes without compromising performance.
- Data Insights: Streamlit dashboards provided actionable insights into shard performance, query latency, and replica health.

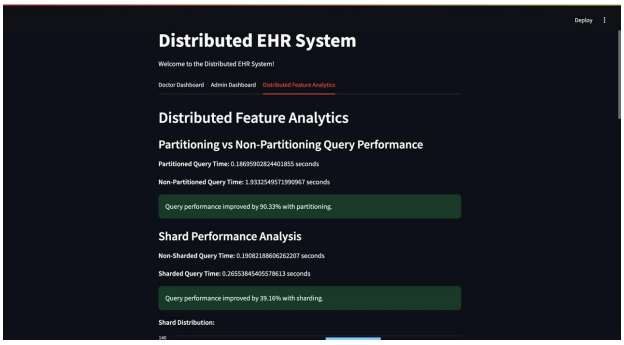

Fig. 2. Analytics

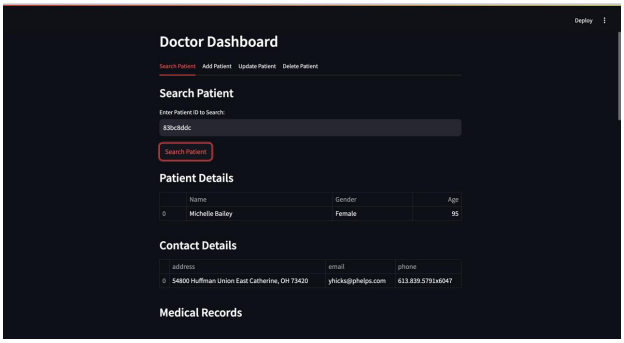

Fig. 3. Doctor Dashboard

V. Conclusion and Future Work

The Distributed EHR System successfully integrates MongoDB and PostgreSQL to address the challenges of healthcare data management. By leveraging distributed database concepts, the system achieves scalability, low-latency access, and fault tolerance while ensuring compliance with healthcare standards.

Fig. 4. Shard Performance

Future work will focus on:

- Integrating machine learning algorithms for predictive analytics in healthcare.
- Enhancing real-time capabilities for emergency response scenarios.
- Expanding the system to include advanced data visualization and reporting features.

Acknowledgments

We thank the faculty at Arizona State University for their guidance and support throughout this project.

References

[1] MongoDB Documentation. [Online]. Available: https://www.mongodb.com/docs

[2] PostgreSQL Documentation. [Online]. Available: https://www.postgresql.org/docs