Design and Implementation of a Distributed NoSQL Triple Store Prototype Using State-Based Objects

VIHAN VASHISHTH, International Institute of Information Technology, Bangalore, India
SHLOK AGRAWAL, International Institute of Information Technology, Bangalore, India
MADHAV SOOD, International Institute of Information Technology, Bangalore, India

This project focuses on the design and implementation of a distributed NoSQL triple store prototype utilizing state-based objects. The system architecture encompasses multiple servers, each employing distinct frameworks for data processing and storage. Through effective communication between backend systems and seamless translation of query languages, the prototype facilitates functionalities such as querying, updating, and merging of subject-predicate-object triples. The report details methodologies employed, implementation strategies, testing approaches, and evaluation results, showcasing the project's adherence to key course concepts and objectives. Overall, the prototype demonstrates scalability, robustness, and user-friendliness, serving as a comprehensive exploration of distributed NoSQL systems.

ACM Reference Format:
Vihan Vashishth, Shlok Agrawal, and Madhav Sood. 2024. Design and Implementation of a Distributed NoSQL Triple Store Prototype Using State-Based Objects. 1, 1 (April 2024), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the realm of database management systems (DBMS), selecting the right system is crucial for efficient data handling in software implementations. This paper presents an implementation-focused comparison of three popular DBMS solutions: PostgreSQL, MongoDB, and Neo4j. Each system offers unique features and capabilities tailored to different application requirements.

PostgreSQL, known for its adherence to SQL standards and transactional reliability [1], serves as a robust choice for relational data management. MongoDB, with its flexible document-oriented approach, caters to applications demanding scalability and schema flexibility [2]. Additionally, Neo4j's graph database architecture excels in managing interconnected data, making it suitable for applications emphasizing relationship mapping [4].

By providing an implementation of these systems in concurrency, this paper aims to guide developers and database administrators in selecting the most suitable DBMS for their projects. Through practical insights into data modeling, query execution, scalability, and ease of implementation, readers will gain valuable knowledge to inform their database system decisions. Whether managing structured data in PostgreSQL, handling flexible document structures in MongoDB, or navigating complex relationships in Neo4j, this paper offers practical guidance for implementing robust database solutions.

Authors' Contact Information: Vihan Vashishth, vihan.vashishth@iiitb.ac.in, International Institute of Information Technology, Bangalore, Bangalore, Karnataka, India; Shlok Agrawal, shlok.agrawal@iiitb.ac.in, International Institute of Information Technology, Bangalore, Bangalore, Karnataka, India; Madhav Sood, madhav.sood@iiitb.ac.in, International Institute of Information Technology, Bangalore, Bangalore, Karnataka, India.

ACM XXXX-XXXX/2024/4-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 SYSTEM ARCHITECTURE

Our project's system architecture seamlessly integrates PostgreSQL, MongoDB, and Neo4j to efficiently manage data in a distributed environment. We've designed the architecture with a focus on simplicity, scalability, and resilience, ensuring smooth operations across the three databases.

For PostgreSQL and MongoDB, we've implemented efficient data storage structures tailored to their respective strengths. In PostgreSQL, a single table named "triples" stores subject-predicate-object triples, while MongoDB utilizes a collection with the same name.

To synchronize state across distributed servers, we've adopted a log-based merging mechanism. Each server maintains a unique log file recording updates to its local state. During merging, servers exchange log files and apply updates based on sequence numbers, ensuring proper ordering and minimizing bandwidth usage. This approach offers scalability, resilience, and efficient bandwidth utilization, albeit with some storage overhead and complexity.

The addition of Neo4j introduces a graph-based data model, enhancing our system's querying capabilities. Nodes represent entities, labeled and indexed for efficient categorization and retrieval, while relationships between nodes represent predicates, further optimizing query execution speed. Log positions are stored as nodes labeled "LogPosition," with properties for server name and log position, indexed for quick retrieval. Updating log positions involves simply modifying node properties, ensuring consistency and efficiency.

Incorporating advanced features such as sharding log files and implementing a "recover()" method enhances fault tolerance and recovery capabilities. Sharding log files ensure efficient management and optimization of resources, while the "recover()" method enables swift restoration of the database from log files in case of failures.

Overall, our system architecture effectively leverages PostgreSQL, MongoDB, and Neo4j to provide a robust, scalable, and resilient solution for managing data in a distributed environment. With advanced features for fault tolerance and recovery, our architecture ensures smooth and uninterrupted operation, even in challenging conditions.

3 METHODOLOGY

The system implements a distributed state management solution for key-value pairs. Currently, PostgreSQL, MongoDB, and Neo4j are the database options. PostgreSQL offers a simple schema and

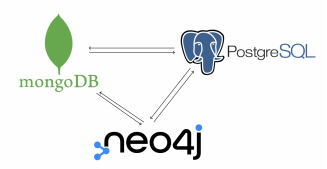

Fig. 1. Consistent Distributed System

efficient writes for large datasets, but requires additional logic for update conflict resolution. MongoDB provides flexibility for storing key-value pairs associated with subjects, simplifies querying, and scales well for smaller to moderate data volumes. However, it has limitations like document size limits and memory usage concerns for very large datasets. Neo4j introduces a graph-based approach to enhance query capabilities and data relationships.

For queries, the client specifies a subject, and the system retrieves the corresponding data (predicate, object, timestamp) from the designated server. Updates modify the value (object) associated with a subject-predicate pair. All three servers implement conflict resolution mechanisms to ensure the latest update prevails.

The key aspect of this system is maintaining consistency across servers. This is achieved through a log-based merging process. Each server maintains a log file that tracks updates (subject, predicate, object, timestamp). The merge operation allows servers to synchronize their state by processing the other server's update log. This ensures both servers eventually reflect the same state.

In the NoSQL triple store, timestamps serve as vital markers for conflict resolution during updates across distributed servers. When concurrent updates occur on different servers, timestamps provide a chronological order of events, enabling the system to discern the most recent modifications. By comparing timestamps associated with conflicting updates, our system ensures that the latest changes take precedence, preserving data consistency across distributed nodes. This timestamp-based conflict resolution mechanism guarantees that conflicting updates are resolved objectively and consistently, preventing data inconsistencies and safeguarding the integrity of the distributed database.

The codebase utilizes Python, due to its readability and extensive library ecosystem. The specific database servers (PostgreSQL, MongoDB, and Neo4j) offer different strengths and are well-suited for storing key-value pairs.

We are using three servers (n=3), they help us tackle issues like redundancy and improve fault tolerance. While additional servers could further enhance these aspects, it also increases complexity and resource requirements. PostgreSQL, MongoDB, and Neo4j were chosen for their unique strengths in handling distributed state management. PostgreSQL's simple schema and efficient writes complement MongoDB's flexibility and scalability. Neo4j's graph-based model enhances query capabilities, particularly for complex relationships. Together, they offer a versatile solution with robust conflict resolution mechanisms, ensuring data consistency across distributed nodes while accommodating diverse data structures and query requirements.

4 IMPLEMENTAION

This distributed state management system uses PostgreSQL, MongoDB, and Neo4j to store triples (subject, predicate, object) and implements log-based merging for consistency across servers. Each server maintains a log file with updates and sequence numbers. Merging involves comparing logs, applying updates with higher sequence numbers, and resolving conflicts (e.g., prioritizing latest timestamps). This approach offers efficiency, resilience, and scalability, but introduces storage overhead, complexity, and potential latency.

4.1 Data Structures

PostgreSQL employs a relational table named "triples" with the following columns:

(1) *subject* (text, not null, primary key part 1): Uniquely identifies the subject of the statement.
(2) *predicate* (text, not null, primary key part 2): Uniquely identifies the property or relation of the subject.
(3) *object* (text, not null): Represents the value associated with the subject-predicate pair.
(4) *timestamp* (BIGINT NOT NULL): Captures the timestamp of the update for conflict resolution.

Our data uses a single table for simplicity and faster writes. The "subject" and "predicate" together act as a unique identifier, ensuring only one value exists for each combination. This fulfills your need for a single "object" per "subject-predicate" pairing. "Object" is a text field storing the value, and we use Unix Epoch Milliseconds for timestamps.

MongoDB follows a similar structure and the variable names used in PostgreSQL have the same meaning here:

(1) *Collection*: MongoDB utilizes collections to store related documents. In this case, the "triples" collection is employed to store all the triples in the database.
(2) *Document Structure*: Each document within the "triples" collection represents a single triple, comprising four main fields: *subject*, *predicate*, *object*, *timestamp*.
(3) *Indexes*: Indexes are created on the *subject*, *predicate*, and *object* fields to optimize query performance, enabling efficient retrieval of triples associated with specific subjects, predicates, or objects.
(4) *Log Positions*: Another collection, "log_positions," is initialized to track the log positions for MongoDB, PostgreSQL, and Neo4j servers, facilitating the merge operation between servers in a distributed environment.

With this structure, we can efficiently store and query triples based on subjects. Indexes are created on the subject, predicate, and object fields to optimize query performance for retrieving triples associated with specific subjects, predicates, or objects. Overall, this structure enables efficient storage and retrieval of triple data in MongoDB, facilitating fast querying and updating operations within the triple store.

Neo4j structures is designed to efficiently represent triple data:

(1) *Nodes*: Nodes represent entities such as subjects and objects in the data model. Each node can be labeled to categorize it, for example, "Subject" or "Object."
(2) *Relationships*: Relationships connect nodes and signify the predicates in the triples. They represent the connections between subjects and objects, labeled with the predicate names.
(3) *Indexes*: Separate indexes are created for subject nodes, object nodes, and relationships to optimize query performance. These indexes enable efficient lookup based on subject values, object values, and predicate names.
(4) *Timestamps*: Timestamps can be stored as properties on object nodes to track when the data was created or last updated. This enables versioning and auditing of the data.
(5) Nodes labeled *LogPosition* represent each log position. They contain properties: "server_name" distinguishes servers, "log_position" stores the actual position value.

4.2 Log-Based Merging

The process of merging data between MongoDB, PostgreSQL, and Neo4j involves a systematic exchange of information to ensure consistency and coherence across the two database systems. The MongoDB codebase utilizes a ‘merge‘ method to synchronize its data with the PostgreSQL database. Conversely, the PostgreSQL and Neo4j codebase employs a similar ‘merge‘ method for data synchronization. Despite the differences in underlying database architectures and query languages, all three systems adhere to a unified approach to data integration.

At the core of the log-based merging process is the concept of each server maintaining a unique log file that tracks updates made to its local state. These updates typically consist of operations such as insertions, deletions, or modifications of data. Each update recorded in the log file is associated with a sequence number, indicating the order in which the update was applied. This sequence number is crucial for ensuring that updates are merged in the correct order, thereby preserving the integrity of the data.

When two servers, let’s call them Server A and Server B, need to synchronize their states, they exchange their respective log files. The merging process begins with Server A requesting the log file from Server B. Server A then compares the timestamp of the updates in its own server with those in the log file from Server B. This comparison helps identify which updates from Server B need to be applied to Server A’s state.

During the merge, Server A applies subject, predicate pair updates from Server B’s log file that have a higher timestamp number than the pair in its own server, or if the pair is not present in its server at all. This ensures that only new updates are applied, preventing duplicate operations and maintaining the correct order of operations. Server B then follows with the same process.

After the merge is complete, both servers update their local log files to reflect the merged state. This includes appending any new updates received during the merge process. By updating their log files, servers ensure that their local states remain synchronized and up-to-date with the changes made on other servers.

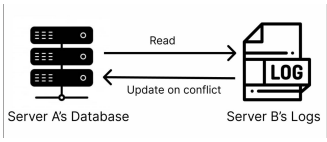

Fig. 2. Merging

The log-based merging process offers several advantages for distributed systems. It is efficient because only updates need to be exchanged between servers, reducing bandwidth usage. It is resilient because log files serve as a reliable record of changes, enabling servers to recover from failures or inconsistencies by replaying updates from the log. Additionally, the process is scalable as each server maintains its own log file, allowing for parallel merging and avoiding centralized bottlenecks.

4.3 Querying

These query methods serve to retrieve data from different database systems: MongoDB, Neo4j, and PostgreSQL. In MongoDB, the method searches for documents in the triples collection matching a specific subject, returning tuples containing subject, predicate, object, and timestamp. In Neo4j, it utilizes a Cypher query to find nodes labeled as "Subject" with the specified value, traversing relationships to retrieve associated predicate, object, and timestamp values. Meanwhile, in PostgreSQL, the method executes an SQL query to select records from the triples table where the subject matches the input parameter, returning tuples with subject, predicate, object, and timestamp fields.

4.4 Updating

The update methods are designed to modify existing data entries in their respective database systems: PostgreSQL, Neo4j, and MongoDB. In PostgreSQL, the method constructs and executes an SQL query to insert or update a record in the triples table, ensuring that the object value and timestamp are updated. Similarly, in Neo4j, it employs a Cypher query to find and update existing nodes representing the subject and predicate, adjusting the object value and timestamp accordingly. In MongoDB, the method utilizes an update operation to modify documents in the triples collection, setting the new object value and timestamp. In all cases, the _write_to_ log method is invoked to log the update operation for tracking and synchronization purposes.

5 TESTING

The prototype utilizes a script named yago_test.py to simulate data updates and validate the consistency of information across the three database servers, PostgreSQL, Neo4j, and MongoDB. This script leverages the YAGO dataset, a massive collection of facts stored in a tab separated values format. The YAGO dataset, with approximately 13 million rows, serves as a realistic data source to simulate real-world usage. The script iterates through the YAGO entries, alternately inserting data (subject, predicate, object) into PostgreSQL and MongoDB.

5.1 Test Execution and Challenges

While the entire 13 million rows wouldn't be feasible for testing due to time constraints, the script was successfully run for the first 10,000 entries. During this process, tests were conducted every 1,000 insertions.

These tests involved:

(1) Data Insertion Verification: Ensuring the data inserted into both PostgreSQL and MongoDB matched the original data from the YAGO file.
(2) Merge Validation: Simulating a merge between the two databases and verifying the merged data's consistency. This involved checking if the merge logic correctly resolved any conflicts based on timestamps (latest timestamp wins).

6 ADVANCED FEATURES

The system incorporates advanced features to enhance log file management and database recovery. Firstly, log files are sharded, restricting each to a maximum of 100 entries. This approach ensures efficient handling of log data while preventing files from becoming overly large. Additionally, the "recover()" method is introduced to facilitate database restoration from log files. This method meticulously iterates through the log files, processing each entry to update the database accordingly. By leveraging these advanced functionalities, the system optimizes log file organization, promotes efficient database recovery processes, and contributes to overall system robustness and reliability.

(1) **Benefits:** The system's enhanced manageability is achieved through the sharding of log files, streamlining file management processes and mitigating the risk of file corruption. By dividing log files into smaller, more manageable segments, the system facilitates easier storage and retrieval of log data. Additionally, fault tolerance is bolstered with the integration of the "Recover()" method, which swiftly restores the database from log files in the event of a failure. This feature significantly reduces downtime by efficiently recovering database operations, ensuring system continuity and reliability even in the face of unexpected disruptions.
(2) **Limitations:** The assumed indestructibility of log files poses a significant risk to data integrity, as the loss or corruption of these files could lead to irreversible data loss. Additionally, inconsistencies in timestamps may arise, highlighting the need for more precise time-tracking mechanisms to ensure data accuracy. Moreover, scalability concerns emerge with the merge method, as its efficiency may diminish when handling a large number of servers. Addressing these challenges requires implementing more advanced scaling solutions capable of efficiently managing and merging data across multiple servers, ensuring optimal performance and reliability as the system scales.

7 USER INTERFACE

Our project features a user-friendly interface designed for effortless interaction with our PostgreSQL, Neo4j, and MongoDB databases. This interface operates through a simple command-line format, allowing users to input commands and parameters easily. Users can perform various tasks such as querying specific data, updating records, and merging data between servers by typing commands like "query," "update," or "merge" followed by the server name and necessary details.

The interface ensures input accuracy by validating commands and providing helpful prompts in case of errors or insufficient information. It initializes connections to both databases at the beginning of the session, enabling users to start working right away. Upon completion of tasks, it gracefully closes these connections, maintaining system integrity.

This intuitive interface simplifies database management, making complex operations accessible to users without extensive technical knowledge. It functions as a virtual assistant, streamlining tasks and handling technical complexities behind the scenes. Whether retrieving information, making edits, or integrating data, users can rely on the interface for seamless execution, enhancing productivity and user experience.

8 EVALUATION AND RESULTS

In evaluating our prototype's functionality, we rigorously tested it using two Python scripts: basic.py and Yago_test.py. The former focused on fundamental operations, such as querying and updating data in PostgreSQL, Neo4j, and MongoDB servers, ensuring the core functionalities functioned as intended. Meanwhile, Yago_test.py, leveraging a dataset of approximately 13 million rows, assessed the system's performance under significant data volumes. While testing limitations arose due to the dataset's size, tests were conducted on the initial 10,000 rows to evaluate performance and validate operations.

Despite these constraints, the prototype demonstrated successful data insertion, updating, and merging operations, affirming its functionality and correctness across varying dataset sizes. In comparison with our project objectives, the prototype closely aligns with our goals of enabling efficient interaction with PostgreSQL and MongoDB databases. Future enhancements, including implementing a database recovery mechanism, aim to address scalability challenges and further enhance system resilience and performance.

The initial Yago test run with 10,000 entries took approximately 2 minutes and 40 seconds. This highlights the potential time challenges associated with testing using the entire 13 million-row dataset

9 CONCLUSION

As we conclude this project, We're pleased to reflect on the journey we've taken to develop a robust and efficient database system. Our collaborative efforts have led us to carefully consider the strengths of PostgreSQL, MongoDB, and Neo4j, allowing us to leverage their unique features for effective data management. By embracing a pragmatic approach, we've implemented a streamlined single-table structure in PostgreSQL and a flexible document-oriented model in MongoDB, alongside Neo4j's intuitive graph database capabilities.

Throughout our development process, we've introduced practical enhancements such as log file sharding and the "recover()" method. These features enhance fault tolerance and simplify system management, reflecting our commitment to practical solutions.

Yet, we recognize the challenges ahead, including ensuring log file durability and addressing scalability concerns.

Looking forward, our focus remains on continuous improvement and refinement. By addressing these challenges thoughtfully, we aim to enhance the reliability and scalability of our system, ensuring it meets the evolving needs of our users.

10 REFERENCES

[1] *PostgreSQL, a powerful open source object-relational database system with over 35 years of active development that has earned it a strong reputation for reliability, feature robustness, and performance https://www.postgresql.org/.*

[2] Y. Gu, S. Shen, J. Wang and J. -U. Kim, "Application of NoSQL database MongoDB," *2015 IEEE International Conference on Consumer Electronics Taiwan*, Taipei, Taiwan, 2015, pp. 158-159, doi: 10.1109/ICCE-TW.2015.7216831.

[3] López, Félix and Cruz, Eulogio. (2015). Literature review about Neo4j graph database as a feasible alternative for replacing RDBMS. Industrial Data. 18. 135. 10.15381/idata.v18i2.12106.