DataSpan - A Distributed Triple Store Using State Based Objects

MUNAGALA KALYAN RAM - IMT2021023, IIIT-B, India
YUKTA ARAVIND RAJAPUR - IMT2021066, IIIT-B, India
BRIJ BIDHIN DESAI - IMT2021067, IIIT-B, India

ACM Reference Format:
Munagala Kalyan Ram - IMT2021023, Yukta Aravind Rajapur - IMT2021066, and Brij Bidhin Desai - IMT2021067. 2024. DataSpan - A Distributed Triple Store Using State Based Objects. *ACM Trans. Graph.* 37, 4, Article 111 (August 2024), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The project asks to design and implement a prototype distributed triple store by making use of state based objects. It requires implementing an application that connects **n** servers where each server is connected to a different database, i.e Postgres, MongoDB, Apache Hive, etc and each server contains the same copy of the **yago dataset**.

The client should be allowed to make updates and queries on the server of choice, independent of the other servers and on performing a merge operation, the servers should reach a common state.

This project's significance lies in coming up with an approach to ensure consistency across replicas where each replica is stored in a different database system.

2 SYSTEM ARCHITECTURE

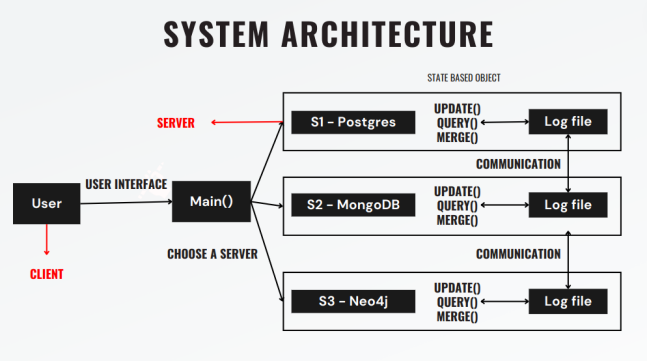

Fig. 1. System Architecture Design

Authors' Contact Information: Munagala Kalyan Ram - IMT2021023, kalyanram.munagala@iiitb.ac.in, IIIT-B, Electronic City, Bangalore, India; Yukta Aravind Rajapur - IMT2021066, yukta.rajapur@iiitb.ac.in, IIIT-B, Electronic City, Bangalore, India; Brij Bidhin Desai - IMT2021067, brijbidhin.desai@iiitb.ac.in, IIIT-B, Electronic City, Bangalore, India.

ACM 1557-7368/2024/8-ART111
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The diagram shown in [Fig 1] describes the system architecture we used for the project. This is a client server architecture where a server is a class that connects to a database.

The class here is the state based object which contains the **query, merge, update** methods which are implemented by utilizing the api's available to connect to the respective database through the application. Each database contains a table with the **subject, predicate, object** triple as its attributes.

All the updates done on a server are maintained in sharded log files which contain **sequence no., subject, predicate, object, timestamp**. The timestamp is important for maintaining consistency where we utilize the **last write wins** approach that takes the update with the highest timestamp value as the latest write done on a key (subject-predicate) and the value (object) present is updated across servers involved in the merge operation.

When the client enters the user interface (terminal-based), they have the option to choose the server after which the client can make use of the server's api's, i.e the methods offered by the state based object.

3 METHODOLOGY

3.1 Methods Implemented for State Based Object

(1) **Query (subject)** : Takes user input and creates a sql statement that is executed on the database and the output is displayed on the UI.
(2) **Update (subject,predicate,object)** : This takes the user input and checks if a given triple is already present, else it is updated and the update is added to the log file.
(3) **Merge (serverID)** : This takes the server to merge with as an input (remote server), then all the triples which are updated are iterated through and the latest update is taken based on the timestamp value.

3.2 Programming Languages and FrameWorks

(1) **Programming Language** : Java
(2) **Postgres** : Postgres [1] is installed in the system and kept running.
(3) **MongoDB** : MongoDB [2] is installed in the system and kept running.
(4) **Postgres Server** : Postgres JDBC and java.sql library used to implement the postgres server.
(5) **MongoDB Server** : MongoDB java drivers are used to connect to the mongodb database.
(6) **Neo4j** : Neo4j [7] is used as a graph database.
(7) **Neo4j Server** : Neo4j java driver is used to implement the Neo4j server.
(8) **JUnit** : JUnit [4] is a java library to write testing scripts.

We have 3 servers integrated to the application which are Postgres, MongoDB and Neo4j.

We have decided to use these databases in the servers as they have been covered in the course and it shows that our solution can accommodate different database architectures. We choose to use java as it provides driver implementations to connect to the databases described above and is easy to use.

4 IMPLEMENTATION

This section describes how the system architecture [Fig 1] is implemented using the frameworks and programming languages mentioned in [Section 3].

4.1 Database Setup

We first setup the mentioned databases in our systems and ensured that they are running.

We then run the postgres and mongodb scripts that stores the triples as follows :

(1) Postgres Server
- Table with fields (SUBJECT, PREDICATE, OBJECT)
- Table to store current sequence number with a single field (seq_no). Initiliaze to 1.

(2) Mongo Server
- JSON object collection with field (SUBJECT, PREDICATE, OBJECT)
- Single element collection 'current_seq' to store current sequence no with field 'seq_no'. Initiliaze to 1.

(3) Neo4j Server
- Relations where the predicate is the relation, and the nodes are subject and object.
- The relation points from subject node to object node.

The yago dataset (13 M) is loaded to the databases using the scripts present in the github repository.

Additionally, we have implemented indexing on each of the databases to improve efficiency of query and update. The script for indexing is present in the repository [9] .

For each of the databases, the indexing step mainly involves assigning the subject column as the index using the index function provided by the database.

```
Postgres server_postgres = new Postgres();
MongoDB server_mongo = new MongoDB();
//connect to postgres
Connection connection = server_postgres.connectToDatabase();

//connect to mongo
MongoCollection<Document> collection = server_mongo.getCollection();
```

Fig. 2. Connecting to different Servers

4.2 Main() Function

The main() function has the code to the user interface [Description in Section 6], initializes the servers [Fig 2] and maintain a unique id (Server ID), calls the necessary functions based on the user choices.

The main function also maintains a global nested map which maintains takes serverID as key and value as another map that maintains the last synced lines in the current server and the remote server.

4.3 Server Abstract Class

The abstract `Server` class offers a flexible foundation for distributed systems, especially in log file management and ensuring data consistency across servers. By abstracting functionalities, it enables easy extension and customization. The 'shardSize' variable sets the shard size (default: 100) for efficient log handling. With clear separation of concerns and well-defined interfaces, developers can swiftly integrate additional functionalities into diverse distributed systems.
Key functionalities:

- It contains all the paths to log files
- Efficient log file merging with support for sharding, allowing each log file to contain up to 100 lines and seamlessly transitioning to the next file when necessary.
- MergeLogs() is generalized for all servers so on extending this to a new server this is called which handles all internal merge logic.
- Helper function isNewerLine() that compares timestamps and returns a boolean value
- WriteLogs() function writes to appropriate log shard based on current sequence number.
- UpdateLastSynced() will update tha last sequence number synced till in the global hashmap after merge function is called.

4.4 How we Implemented Sharding

As mentioned above, the abstract class "Server" handles all the merging and writing logic with respect to log files. This is how it is done.

(1) Sharding
- Sharding is a technique used in distributed systems to horizontally partition data across multiple servers or storage devices.
- In the context of log files, sharding involves dividing the log entries into smaller, manageable chunks or shards, typically based on some criteria such as time intervals or numerical ranges.
- Each shard contains a subset of the total log entries, distributing the workload and improving system scalability and performance.

(2) Reading and Merging Logs
- When reading log files in a sharded environment, it's essential to track the position within each shard to resume reading from where the last synchronization occurred.
- The system maintains metadata about the last synchronized line for each shard, allowing it to resume reading from the correct position during subsequent operations.
- To optimize reading, the system calculates the shard and the starting position within that shard based on the last synchronized line.
- As log entries are processed, the system updates the metadata to reflect the latest synchronized position and implements the merge logic which is done using a hashmap.

(3) Writing Logs
- The `writeLog()` functionality is responsible for updating the log files with new entries after each update operation.
- It appends the update entry to the appropriate log file based on the shard size and the sequence number of the update.
- If a timestamp is not provided, it generates a timestamp using the current system time.
- The update entry is formatted with the sequence number, subject, predicate, object, and timestamp.
- Finally, the update entry is written to the corresponding log file using a `FileWriter`.

4.5 Database Classes

Postgres

```
public static void queryTriple(Connection connection, String
    ↪  subject) throws SQLException {
    String sql = "SELECT␣*␣FROM␣sample_yago␣WHERE␣subject␣=␣?
        ↪ ";

    PreparedStatement statement = connection.prepareStatement
        ↪ (sql);
    statement.setString(1, subject);

    ResultSet resultSet = statement.executeQuery();
    while (resultSet.next()) {
        String retrievedSubject = resultSet.getString("
            ↪ subject");
        String retrievedPredicate = resultSet.getString("
            ↪ predicate");
        String retrievedObject = resultSet.getString("object"
            ↪ );
        System.out.println("Subject:␣" + retrievedSubject + ",
            ↪ ␣Predicate:␣" + retrievedPredicate + ",␣
            ↪ Object:␣" + retrievedObject);
    }
}
```

Listing 1. Query Triple Stored in Postgres with JDBC

MongoDB

```
public static Map<String,String[]> queryTriple(
    ↪ MongoCollection<Document> collection, String subject
    ↪ ) {

    Map<String, String[]> results= new HashMap<>();
    Bson filter = subject.isEmpty() ? null : Filters.eq("
        ↪ subject", subject);
    for (Document document : collection.find(filter)) {
        String retrievedSubject = document.getString("subject
            ↪ ");
        String retrievedPredicate = document.getString("
            ↪ predicate");
        String retrievedObject = document.getString("object")
            ↪ ;
        String [] tmp= {retrievedPredicate,retrievedObject};
        results.put(retrievedSubject, tmp);

        // System.out.println("Subject: " + retrievedSubject
            ↪ + ", Predicate: " + retrievedPredicate + ",
            ↪ Object: " + retrievedObject);
    }
    return results;
}
```

Listing 2. Query Triple Stored in MongoDB Java Drivers

Neo4j

```
public static void queryTriple(Driver driver, Scanner
    ↪ scanner){

    System.out.print("Enter␣subject␣to␣query:␣");
    scanner.nextLine();
    String subjectValue = scanner.nextLine(); // Consume
        ↪ extra newline

    try (Session session = driver.session()) {

        // Run Cypher query
        String query = "MATCH␣(s:Entity␣{name:␣
            ↪ $subjectValue})-[p]->(o)␣RETURN␣s,␣p,␣o";

        Result result = session.run(query,parameters("
            ↪ subjectValue",subjectValue));

        while(result.hasNext()){
            Record record = result.next();
            Node subject = record.get("s").asNode();
            Relationship predicate = record.get("p").
                ↪ asRelationship();
            Node object = record.get("o").asNode();

            System.out.println("Subject:␣" + subject.get("
                ↪ name") + ",␣Predicate:␣" + predicate.
                ↪ get("predicate") + ",␣Object:␣" +
                ↪ object.get("name"));
        }
    }
    catch(Exception ex){
        ex.printStackTrace();
    }
```

Listing 3. Query Triple Stored in Neo4j with Java Drivers

Update() and Query() :

- Queries are constructed based on the user input.
- The Driver corresponding to the server translates the query into the database's internal format and communicates with the server.
- Only the update queries are written to the log file. The log file has the following format : **Sequence No, subject, predicate, object, timestamp**.

Merge() :

- It uses the global nested map defined in the main function to locate the last synced updated in the current and remote server's log files.
- Following which the remaining updates are maintained in a hashmap which will be used to implement the last-write-wins approach.
- The hashmap is maintained each triplet as HASHMAP (KEY = (SUB,PRED), VALUE = (OBJECT, TIMESTAMP)).
- - The map is updated for a key only if the timestamp is newer during the merge operation. This is where the last-write-wins concept comes into play. The merge operation is done both ways to ensure that the servers are in sync.

5 TESTING

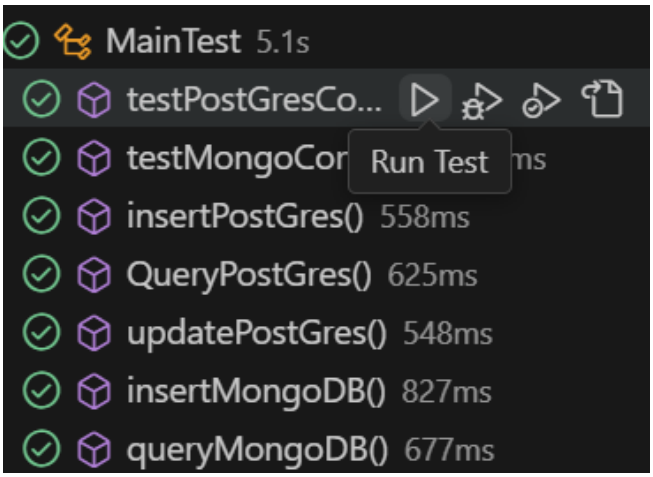

Fig. 3. JUnit Testing on update, query, connection

5.1 Testing Approach

The testing of the application is done by using JUnit and manual testing.

- **JUnit** : This library is currently used to test the update, query and connection operations of the Postgres and MongoDb servers as seen in [Fig 3].
- **Manual Testing** : The merge operation is tested by manually making updates to the each server through the UI and the log files are checked after the merge operation to check if the last-write-wins approach is implemented correctly.

5.2 Dataset Description

The YAGO dataset is a comprehensive knowledge base that organizes human knowledge into structured data. It includes information about entities and their relationships sourced from sources like Wikipedia and WordNet. The data is structured as triples of subject-predicate-object.

A sampled version of the yago dataset was used to perform the testing by updating the object value of various keys (subject-predicate) and performing merge. We have now tested the application on the entire dataset (13 M) and have not faced any issues.

5.3 Issues

We had faced issues with testing the merge operation and the log file operations (sharding, updating, etc) using JUnit.

Hence, we have performed manual testing for merge function on test cases that cover a wider range of user interactions.

6 ADVANCED FEATURES

We have included the following features outside the specified objectives in the project guidelines

(1) Sharding Log Files

Benefits:

- Scalability: Sharding horizontally partitions log data, allowing for distributed storage and processing across multiple servers. This enables seamless scaling as the system grows, accommodating increased data volume without sacrificing performance.
- Improved Performance: By distributing workload across shards, the system can parallelize log processing tasks, leading to faster response times and reduced latency for data retrieval and analysis.

Limitations:

- Complex Shard Management: Managing a large number of shards introduces complexity in terms of shard creation, distribution, and rebalancing. This overhead requires careful planning and coordination to maintain optimal shard performance.
- Synchronization Overhead: Maintaining data consistency and synchronization across shards can be challenging, particularly in distributed environments with high concurrency. Implementing effective synchronization mechanisms is crucial to prevent data inconsistencies and ensure data integrity.

(2) Indexing in Databases

Benefits:

- Improved Query Performance: Indexes allow to quickly locate specific documents or records based on the subject field. By indexing commonly queried fields, databases can efficiently retrieve data, leading to faster query performance.

Limitations :

- Increased Storage Overhead: Indexes require additional storage space to store metadata and index structures. This overhead can become significant, especially for large datasets with numerous indexed fields. As a result, indexing can increase the overall storage requirements of the database.

7 USER INTERFACE

We have implemented a terminal based user interface [Fig 4] . The user is provided with options to choose a server of his choice.

After picking the server, the user can perform the following operations independent of the other servers **update, query, merge, exit**. On exit, the user can once again choose the server to execute further operations.

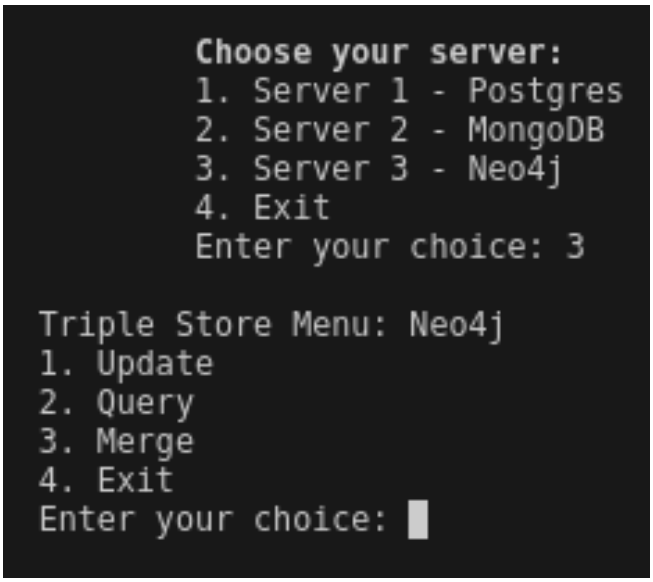

Fig. 4. Terminal Based User Interface

8 EVALUATION AND RESULTS

Introducing sharding brings down the time complexity taken for merge to directly proportional to the number of unseen updates. This is in the user's hands as per the design of the project but if done regularly then it will be more efficient as we don't have to read lot of logs, rather only a constant number(≈ shardSize) from a single file.

The time taken to execute queries on the complete yago dataset is greatly reduced by using indexing.

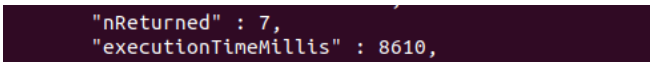

Fig. 5. Time taken to query without indexing in MongoDB

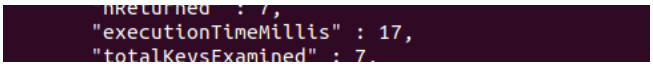

Fig. 6. Time taken after indexing in MongoDB

From [5] and [6] we can see that the query time reduced significantly from 8610 ms to 17 ms after indexing in the case of a query in mongodb. Similarly, in [7] and [8] the query time after indexing reduced from 6581 ms to 468 ms in Neo4j. Finally in postgres [9] and [10], we can see the query time reduce from 8984 ms to 0.68 ms after indexing

9 CONCLUSION

Overall we have implemented 3 servers, with fully persistent and working functionalities using Java. We have achieved to closely simulate a triple store which can be moderately scaled as sharding is done and indexing allows for optimized queries. Some challenges we faced was maintaining and concurrently working on a big codebase as a team.

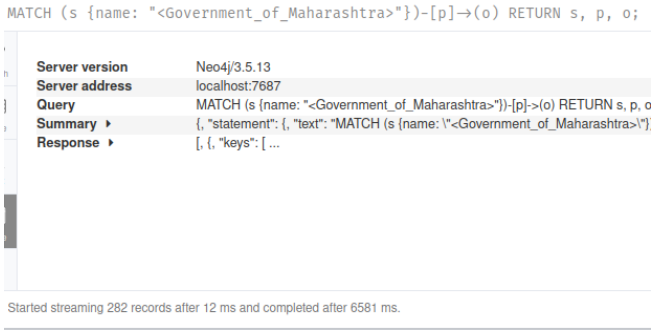

Fig. 7. Time taken without indexing in Neo4J

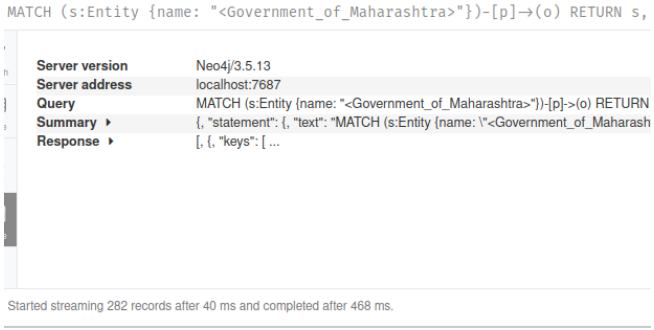

Fig. 8. Time taken after indexing in Neo4J

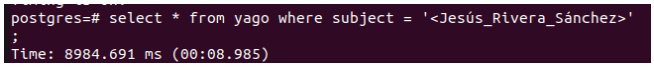

Fig. 9. Time taken after without indexing in Postgres

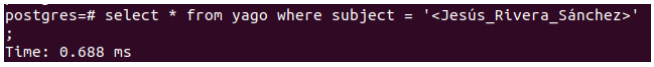

Fig. 10. Time taken after indexing in Postgres

REFERENCES

[1] Postgres Installation: https://www.digitalocean.com/community/tutorials/how-to-install-and-use-postgresql-on-ubuntu-20-04
[2] MongoDB Installation: https://www.digitalocean.com/community/tutorials/how-to-install-mongodb-on-ubuntu-18-04-source
[3] Java Maven Repository : https://mvnrepository.com/
[4] JUnit - Java Library For Testing : https://www.javatpoint.com/junit-tutorial
[5] Yago Dataset: https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/downloads/
[6] Hadoop Installation: https://medium.com/@vikassharma555/hadoop-installation-on-windows-wsl-2-on-ubuntu-20-04-lts-single-node-d604729ea0ca
[7] Neo4j Download : https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-neo4j-on-ubuntu-20-04
[8] Working with MongoDB in Java: https://www.baeldung.com/java-mongodb
[9] Github Repo : https://github.com/yuktaX/Distributed_NoSQL_TripleStore