CS834: NoSQL Systems - Final Project

ANUJ ARORA, IMT2021050, IIITB
PANNAGA BHAT, IMT2021080, IIITB
NORI MEHER ASHISH, IMT2021085, IIITB
KADARU JASHWANTH REDDY, IMT2021095, IIITB

This is the final report for the end semester project by team 7.

Additional Key Words and Phrases: NoSQL, MongoDB, PostgreSQL, state based, merging

ACM Reference Format:
Anuj Arora, Pannaga Bhat, Nori Meher Ashish, and Kadaru Jashwanth Reddy. 2024. CS834: NoSQL Systems - Final Project. *ACM Trans. Graph.* 37, 4 (June 2024), 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The proliferation of data in various domains has necessitated the development of efficient data storage and retrieval systems. In this context, the problem statement entails the creation of a prototype for a distributed NoSQL triple store utilizing state-based objects. Triple stores are essential for managing semantic data, where information is structured into subject-predicate-object triples. The significance of this project lies in its potential to offer a scalable and fault-tolerant solution for storing and querying triples in a distributed environment.

The primary objective of the project is to design and implement a distributed NoSQL triple store capable of accommodating a variable number of servers (n). The scope encompasses the development of methods for querying, updating, and merging triples across multiple servers. Additionally, the project aims to incorporate advanced features such as sharding, replication, fault tolerance, dynamic scaling, backup and restore functionality, and cluster visualization.The primary objective of the project is to design and implement a distributed NoSQL triple store capable of accommodating a variable number of servers (4 in our case). The scope encompasses the development of methods for querying, updating, and merging triples across multiple servers. Additionally, the project aims to incorporate advanced features such as sharding, replication, fault tolerance, dynamic scaling, backup and restore functionality, and cluster visualization. The project aims to demonstrate proficiency in distributed NoSQL systems and showcase innovative solutions to address the challenges associated with managing semantic data in distributed environments. The code can be found at https://github.com/JashwanthKadaru/NoSQL-Final-Project.git

Authors' Contact Information: Anuj Arora, IMT2021050, IIITB, Anuj.Arora@iiitb.ac.in; Pannaga Bhat, IMT2021080, IIITB, Pannaga.Bhat@iiitb.ac.in; Nori Meher Ashish, IMT2021085, IIITB, meher.ashish@iiitb.ac.in; Kadaru Jashwanth Reddy, IMT2021095, IIITB, Jashwanth.Kadaru095@iiitb.ac.in.

ACM 1557-7368/2024/6-ART
https://doi.org/XXXXXXX.XXXXXXX

2 SYSTEM ARCHITECTURE

The architecture of the distributed NoSQL triple store prototype is designed to efficiently store and manage semantic data across multiple servers.The architecture encompasses several key components, including the client-server model, state-based object management, and the utilization of distinct frameworks for data processing and storage.

The architecture follows a distributed model, where multiple servers collaborate to store and process triples. At a high level, the system consists of a master server and multiple worker nodes, each responsible for specific tasks. The master server facilitates coordination and synchronization among the worker nodes, while the worker nodes handle data storage and processing operations.

2.0.1 ***Utilization of State Based Object.*** State-based objects play a crucial role in representing and managing triples within the system. Each triple is encapsulated as an object, where the subject, predicate, and object values are stored along with additional metadata, i.e timestamps. This approach enables efficient querying, updating, and merging of triples across distributed nodes while ensuring consistency and synchronization.

2.0.2 ***Client-Server Model and Services Offered.*** The system follows a client-server model, where clients interact with the master server to perform various operations such as querying for triples, updating triple values, and initiating merge operations between servers. Clients can access the system via terminal, providing seamless interaction with the distributed triple store.

2.1 Database Used

The servers are built with Python.
The project uses 3 different databases:

(1) MongoDB: NoSQL
(2) PostgreSQL: SQL
(3) SQLite3: SQL

Hive(NoSQL) has not been used due to issues faced in creating a table with transactions enabled.

3 METHODOLOGY

The core functionality of the system revolves around state-based objects, which represent triples in the form of (subject, predicate,

object) tuples. Several methods have been implemented to facilitate the management of state-based objects:

- **Query**: The query method enables users to retrieve all triples associated with a given subject. This operation involves searching through the state-based objects stored across multiple servers and returning relevant triples to the client.
- **Update**: The update method allows users to modify the object value of a triple based on the provided subject-predicate combination. If the specified triple does not exist, a new entry is created. This operation ensures that the triple store remains up-to-date with the latest data changes.
- **Merge**: The merge method facilitates synchronization between servers by allowing users to initiate merge operations. This enables the exchange of state-based objects between servers, ensuring consistency and coherence across the distributed system.

Python was chosen as the primary programming language for the development of the prototype due to its simplicity, readability, and extensive ecosystem of libraries and frameworks. After careful analysis, we decided to configure the system with four servers, including one master server and three worker nodes. This setup strikes a balance between scalability and complexity, allowing us to demonstrate the functionality of a distributed NoSQL triple store while ensuring manageable system overhead.

4 HIVE INTEGRATION

While Hive effectively handles read-only queries for data retrieval, we encountered obstacles in facilitating read-write operations. Specifically, our attempts to create tables capable of accommodating write queries proved unsuccessful. Update and merge queries, which necessitate row modifications, were unable to execute within this framework. As a result, we faced limitations in leveraging Hive for tasks requiring data modification.
All necessary modifications to the `hive-site.xml` config file that enable transaction have benn done.

- Building a table in an external location meant that it could not accept the `'transactional'='true'` property.
- Building a table in the default warehouse throws an error because Hive was unable to access `user/hive/warehouse` directory in the Hadoop FS.

The `hadoop -chmod` command has no effect.
In summary, we are able to demo a 'query' operation with Hive, but not the 'merge' or 'update' operations. We have thus opted to use SQLite3 instead.

5 IMPLEMENTATION

5.1 Query

When the function `query(subject)` is invoked, the servers housing the shard potentially containing the specified subject are checked for its presence. These servers perform a `SELECT` type query and return all rows related to the subject.

```python
class SQLiteQueryConnector:
    def fetch_rows_related_to_subject(self, subject):
        """
        Function to fetch all rows related to a subject from the YAGO dataset.

        Args:
        - subject: The subject for which rows are to be fetched.

        Returns:
        - A list of rows related to the subject, length of the list, and success/failure status (bool).
        """
        try:
            query = 'SELECT * FROM YAGO WHERE subject = ?'
            self.cur.execute(query, (subject,))
            rows = self.cur.fetchall()
            return (rows, len(rows), True)
        except Exception as e:
            print(e)
            return ([], 0, False)
```

Fig. 1. Section of SQLite3 server performing a query for a subject value.

5.2 Update

Each server has a dictionary of updated rows in the format `(sibject, predicate):(object, timestamp)`. When `update(subject, predicate, new_object)` is called, the updating server first checks if the given subject-predicate pair is present. If it is, the row is updated with new object and the current time. Else, a new row is created with given object and current time. The dictionary is also updated to reflect this information.

```python
class MongoDBQueryConnector(DatabaseQueryConnector):
    def update_or_add_subject_predicate(self, subject, predicate, new_object, timestampArg=None):
        try:
            # Check if the subject and predicate already exist
            existing_row = self.collection.find_one({'subject': subject, 'predicate': predicate})

            if existing_row:
                # Update the existing row with the new object
                old_timestamp = existing_row['timestamp']
                old_object = existing_row['object']

                if timestampArg==None:
                    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S.%f")
                else:
                    timestamp = timestampArg

                self.collection.update_one({'_id': existing_row['_id']}, {'$set': {'object': new_object, 'timestamp':timestamp}})

                new_entry = {"timestamp": timestamp, "subject": subject, "predicate": predicate, "object": new_object}
                self.updateModifications[(subject, predicate)] = {"new_object": new_object, "timestamp": timestamp}
                self.isModified = True
                return ({"new_row": {"subject":subject, "predicate": predicate, "object": new_object, "timestamp":timestamp},
                         "old_row": {"subject":subject, "predicate": predicate, "object": old_object, "timestamp":old_timestamp},
                         "status": True})
            else:
                # Add a new row with the subject, predicate, and object
                if timestampArg==None:
                    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S.%f")
                else:
                    timestamp = timestampArg

                new_row = {'subject': subject, 'predicate': predicate, 'object': new_object, 'timestamp': timestamp}
                self.collection.insert_one(new_row)

                new_entry = {"timestamp": timestamp, "subject": subject, "predicate": predicate, "object": new_object}
                self.updateModifications[(subject, predicate)] = {"new_object": new_object, "timestamp": timestamp}
                self.isModified = True
                return ({"new_row": {"subject":subject, "predicate": predicate, "object": new_object, "timestamp":timestamp},
                         "old_row": {},
                         "status": True})
        except Exception as e:
            print(e)
            return ({"new_row": {},
                     "old_row": {},
                     "status": False,
                     "error": e})
```

Fig. 2. Section of MongoDB server performing an update. Presence of existing row is checked first.

5.3 Merge

When `merge` is called between 2 servers, they exchange their dictionaries containing the update information. Each server will then iterate through the dictionary. If the timestamp in the dictionary is greater than the timestamp corresponding to the same subject-predicate pair in the database, the object and timestamp are overwritten with the new information.

```python
class PostgreSQLQueryConnector(DatabaseQueryConnector):
    def mergeSelf(self, modifications_dic):

        Args:
        - modifications_dic: A Dictionary containing modifications with timestamps.

        Returns:
        - True, otherwise raises exception.
        """
        try:
            for (subject, predicate), modification in modifications_dic.items():
                # Get the most recent modification for each subject-predicate pair
                new_object = modification['object']
                new_timestamp = modification['timestamp']

                # Check if the subject and predicate already exist
                query = 'SELECT * FROM public."Yago 13" WHERE subject = %s AND predicate = %s'
                self.cur.execute(query, (subject, predicate))
                existing_row = self.cur.fetchone()

                if existing_row:
                    # Update the existing row with the new object
                    if datetime.strptime(new_timestamp, "%Y-%m-%d %H:%M:%S.%f") > existing_row[3]:
                        update_query = 'UPDATE public."Yago 13" SET object = %s, timestamp = %s WHERE subject = %s AND predicate = %s'
                        self.cur.execute(update_query, (new_object, new_timestamp, subject, predicate))
                else:
                    # Add a new row with the subject, predicate, and object
                    insert_query = 'INSERT INTO public."Yago 13" (subject, predicate, object, timestamp) VALUES (%s, %s, %s, %s)'
                    self.cur.execute(insert_query, (subject, predicate, new_object, new_timestamp))

            self.mergedAt = datetime.now()
            print("Merge operation completed.")
            self.conn.commit()
            return True
        except Exception as e:
            self.conn.rollback()
            raise e
```

Fig. 3. Section of PostgreSQL server that performs merge. Note the dictionary input and comparison of timestamps

6 ADVANCED FEATURES

In addition to the core functionalities of the distributed NoSQL triple store, several advanced features have been incorporated into the prototype to enhance its capabilities and address specific requirements of distributed systems.

- **Sharding & Replication**: Sharding enables the distribution of data across multiple servers, improving scalability and performance. In our implementation, sharding is complemented by replication, where multiple copies of data are maintained across different nodes to ensure fault tolerance and high availability. This combination of sharding and replication enhances the resilience of the system against node failures and provides seamless access to data even in the presence of faults.
- **Load Balancing**: It involves distributing incoming client requests across multiple servers in a balanced manner to optimize resource utilization, maximize throughput, and minimize response times. Master Server assigns node to the client.
- **Fault Tolerance Mechanisms**: To ensure uninterrupted service in the event of node failures, fault tolerance mechanisms have been implemented. These mechanisms include health checks and automatic failover, which detect node failures and redistribute shards to healthy nodes dynamically. By automatically adapting to changes in the cluster topology, the system maintains data consistency and availability, minimizing downtime and ensuring reliability.
- **Shard Management**: Shard Management allows cluster to reassign shards dynamically between nodes, allowing flexibility. Shard Management allows the cluster to reassign shards dynamically between nodes, providing flexibility for administrators. This enables them to optimize resource utilization by balancing shard distribution across the cluster based on factors like node capacity and incoming data load.
- **Cluster Visualization**: A dashboard or visualization tool has been developed to provide insights into the current state of the cluster. This visualization tool offers administrators a comprehensive view of node status, shard distribution, and performance metrics, allowing them to monitor and manage the cluster effectively.

7 TESTING & RESULTS

7.1 Time Complexity

- **Query**: Instant i.e O(1)
- **Update**: Instant i.e O(1)
- **Merge**: Linear in number of unique updates i.e O(n)

7.2 Sample run

7.2.1 Query. A query for subject <Jaroslav_Volek> was successfully performed in 0.9002 seconds.

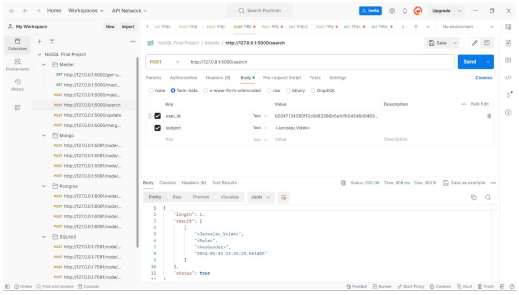

Fig. 4. POST request returns the row in the 'result' value

Fig. 5. Console output for query

7.2.2 Update. A single update was performed successfully in 2.4244 seconds.

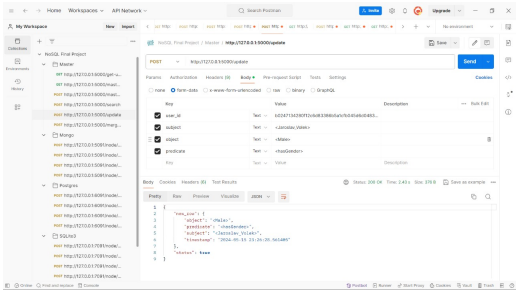

Fig. 6. POST request returns a success message

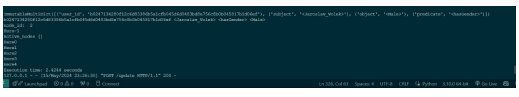

Fig. 7. Console output for update

7.2.3 Merge. The merge operation was performed after one update between servers 2 and 3.

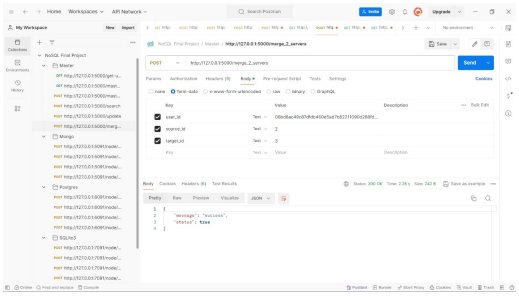

Fig. 8. The POST request sent via the Postman extension returned a success

The merge operation took 2.2729 seconds to complete.

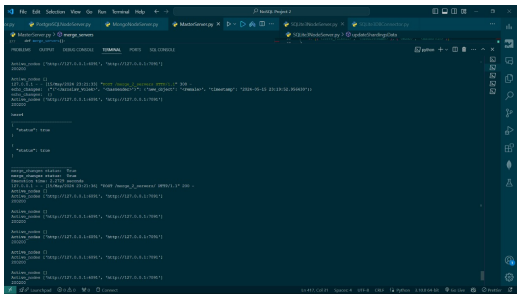

Fig. 9. Console output for the merge operation

8 CONCLUSION

In this project, we successfully established a distributed database across three servers, each employing different database types. Initially aiming to implement Hive, challenges within the Hadoop file system prompted us to pivot towards an alternative SQL-based database system in SQLite3. The project helped us understand sharding and replication to build a resilient program. Through navigating these challenges, we gained a deeper understanding of distributed database management.