Triplex: A Distributed NoSQL Triple Store Prototype

Prachoday(IMT2021034)
Kushal Dasari(IMT2021035)
Raghunadh(IMT2021042)
Gowtham.k(IMT2021081)

I. Introduction

The project's focus is on creating a distributed NoSQL triple store prototype that leverages state-based objects, aiming to address the escalating demand for efficient management and querying of triple-formatted data, which comprises subject, predicate, and object elements. The endeavor holds significance in its potential to provide scalable and resilient solutions for handling diverse datasets within distributed environments, thus meeting the rising requirements for flexible data storage and retrieval mechanisms.

The main objectives of the project encompass the development of a prototype capable of executing fundamental operations such as querying, updating, and merging triples across multiple servers. Additionally, it seeks to establish synchronization between servers through a merge function, with each server employing distinct frameworks for data processing and storage. Furthermore, the project aims to construct a Triple datastore capable of storing subject, predicate, and object triplets across multiple server locations, ensuring the availability of the datastore even if one server goes offline. We created 3 servers and a client for doing this project.

II. System Architecture

Our system follows a client-server architecture style. We have the same data initially in all servers. We have three servers which are MongoDB, PostgreSQL, and HBase respectively. We have a client that can connect to any of the three servers. We utilized Flask to establish communication between the client and servers.

Each server runs on a specific port, and a client runs on a different port. Each server has a log file associated with it containing information related to updates done to that server. The client can communicate with the server of its choice and perform operations like querying, updating the server, or merging two servers. Additionally, the client can switch between servers even after connection.

The client can perform query and update operations from any server. It can also merge a server of its choice with the current server. Servers are responsible for handling query, update, or merge requests sent by the client. Each server in the system maintains a state-based object that represents the current state of triples stored in its local database.

We've employed state-based objects to store data on the servers. Each server holds data in the format of subject predicate object. The state-based object contains information about the triples, including subjects, predicates, and objects, allowing the server to track its local data. Each server maintains its individual state-based object, reflecting the current state of the triple store on that specific server. These state-based objects undergo updates whenever a merge operation is initiated, ensuring synchronization and consistency across all servers.

III. Methodology

Description of Methods for State-Based Objects:

1) **Query Method:**
   - The `query` method retrieves all triples associated with a given subject from the database.
   - It takes the subject as input and performs a database query to find all records where the subject matches the provided value.
   - The method then converts the query results into a list of dictionaries, each representing a triple, and returns this list.
2) **Update Method:**
   - The `update` method updates or inserts a new triple into the database.
   - It takes the subject, predicate, and object of the triple as input.
   - Using the appropriate database operation the method modifies the corresponding record in the database to set the object to the new value.
   - If no record matching the subject and predicate exists, the method creates a new record in the database with the provided values.
3) **Merge Method:**
   - The `merge` method merges data from another server into the current server's state.
   - It takes the IDs of the current server and the source server as input.
   - The method reads update log files from both servers to compare timestamps of updates for each triple.
   - For each triple in the source server's update log, the method checks if the update occurred after the last update in the current server's state.
   - If the update is more recent, it applies the update to the current server's state by calling the `update` method.

- This ensures that the current server's state is synchronized with the latest updates from the source server.

These methods provide a generic interface for managing the state of triple data in any database used as the backend for the state-based object. They facilitate querying, updating, and merging of triples across multiple servers in a distributed environment. **Programming Language and Frameworks:**

1) **Python as the Programming Language:**
   - Python is chosen due to its simplicity, readability, and extensive support for libraries and frameworks.
2) **Flask as the Web Framework:**
   - Flask is selected for building the RESTful API due to its lightweight nature and simplicity.
   - It provides essential tools for creating HTTP endpoints and handling requests and responses, making it suitable for developing the server-side component of the triple store.
3) **Database Libraries:**
   - Different databases require different libraries for interaction. Python libraries such as pymongo for MongoDB, psycopg2 for PostgreSQL,and HappyBase for Hbase.

Decision-making Process for Number of Servers (n) and Frameworks:

1) **Number of Servers (n):**
   - **Scalability and Fault Tolerance:** The decision on the number of servers (n) involves balancing scalability and fault tolerance. With **three servers**, the system can distribute the workload effectively and handle growing data volumes while maintaining operations even if one server fails.
   - **Redundancy and Availability:** Multiple servers increase redundancy, ensuring system availability in case of hardware failures or maintenance activities.
   - **Complexity and Resource Utilization:** While more servers could enhance fault tolerance, it also increases complexity and resource utilization. Three servers strike a practical balance, providing sufficient redundancy without overburdening resources.
2) **Frameworks for Data Processing and Storage:**
   - **PostgreSQL:** Chosen for its robustness, reliability, and support for ACID transactions. Its relational model suits structured data storage and offers features such as indexes and constraints for efficient querying.
   - **MongoDB:** Selected for its flexibility and scalability in handling semi-structured data like triples. Its document-oriented model aligns well with the triple data structure, enabling efficient storage and retrieval.
   - **HBase:** Chosen for its distributed storage capabilities and efficient querying of sparse data. It's suitable for storing large volumes of sparse data, such as triple data, and provides scalability and fault tolerance features.

IV. Implementation

1) **Data Structures used to store the triples:**
   - PostgreSQL server stores triples in table with 3 columns with each row assigned for each triple
   - MongoDB server stores data in collections. Collection holds the each triple as JSON structured documents.
   - In the HBase server, the triples are stored such that each row key corresponds to an index starting from 0 and incrementing sequentially. The table comprises a single column family named "a", which encompasses three columns: "subject", "predicate", and "object".
2) **Implementation process:**
   a) **Flask Application Setup:**
      - The Flask application is used as the front end to interact with the TripleStoreAPI for managing triple store operations.
      - Different routes are defined for handling queries, updates, merges, and other operations.
   b) **TripleStoreAPI Implementation:**
      i) **PostgreSQL Implementation:**
         - The PostgreSQL implementation uses a 'Config' class to handle database settings from a configuration file ('database.ini'). This class provides methods to load database parameters like host, port, database name, username, and password.
         - It connects to the database using the 'psycopg2' library and reads database settings from a configuration file.
         - The 'TripleStore' class manages triple store operations using a PostgreSQL database connection. It connects to the PostgreSQL database server and initializes a connection object.
         - The 'query' method retrieves all triples for a given subject from the PostgreSQL database using SQL queries.

```python
def query(self, subject):
    """Retrieve all triples for a given subject."""
    with self.conn.cursor() as cur:
        cur.execute('SELECT * FROM yago2 WHERE subject = %s', (subject,))
        results = cur.fetchall()
        columns = [desc[0] for desc in cur.description]  # Get column names
        formatted_results = []
        for row in results:
            formatted_results.append(dict(zip(columns, row)))
        return formatted_results
```

Fig. 1. query

- The 'update' method updates or inserts a new triple into the PostgreSQL database us-

ing SQL queries with UPSERT functionality, ensuring data integrity.

```
def update(self, subject, predicate, object):
    """ Update or insert a new triple into the database. """
    try:
        with self.conn.cursor() as cur:
            cur.execute('''
                WITH upsert AS (
                    UPDATE yago2
                    SET object = %s
                    WHERE subject = %s AND predicate = %s
                    RETURNING *
                )
                INSERT INTO yago2 (subject, predicate, object)
                SELECT %s, %s, %s
                WHERE NOT EXISTS (SELECT 1 FROM upsert);
            ''',
            (object, subject, predicate, subject, predicate, object))
            self.conn.commit()
            _write_to_log(subject,predicate,object,2)
    except (Exception, psycopg2.DatabaseError) as error:
        print(error)
```

Fig. 2. update

- The 'writetolog' method writes update information to a log file (update_log.txt) for auditing and tracking purposes. It records the timestamp, subject, predicate, and object of each update operation.
- The List method updates the dictionary (l) with the latest update timestamps and clears the temporary dictionary (l1).
- The merge method reads the log file of the other server that we are merging with and compares the timestamps of the (subject,predicate) pairs and merge the latest updates into the current server.

```
def merge(self,id, source_id):
    """Merge data from another server."""
    file_a = "update_log.txt" + str(id)
    file_b = "update_log.txt" + str(source_id)

    # Open both files in read mode
    with open(file_a, "r") as file_a, open(file_b, "r") as file_b:
        lines_a = file_a.readlines()
        for line_a in lines_a:
            a=line_a.split(":")
            print(a)
            a_timestamp=a[0]+":"+a[1]+":"+a[2]
            parts = line_a.split(",")
            a_subject = parts[0].split(":")[-1].strip()
            a_predicate = parts[1].split(":")[-1].strip()
            a_object = parts[2].split(":")[-1].strip()
            found = False
            key = (a_subject,a_predicate)
            if key in l:
                tb = l[key]
                a_time = datetime.strptime(a_timestamp, "%Y-%m-%d %H:%M:%S")
                if tb:
                    b_time = datetime.strptime(tb, "%Y-%m-%d %H:%M:%S")
                    if b_time > a_time:
                        found = True  # Ignore update

            if not found:
                store.update(a_subject,a_predicate,a_object)

    store.List()
```

Fig. 3. merge

- To make the comparision simpler we used dictionary datastructure to store the (subject,predicate),timestamp pairs. These pairs are stored from reading the log file of the current server. It also reduces complexity of merge operation significantly, as we are not involved in file reading.
- Please ensure that we restart the server if we do any updates to the log file to eliminate any inconsistency with dictionary in the code.

Implementation of MongoDB using PyMongo and HBase using HappyBase are quite similar to PostgreSQL server implementation. Only changes are the query languages to query and update, and syntax,libraries to connect to the respective servers.

ii) **Usage of log files:**

- Each backend system (PostgreSQL, MongoDB, HBase) maintains its separate log file (update_log.txt) to track update operations.
- The log files record the timestamp, subject, predicate, and object of each update or merge operation for auditing and synchronization purposes.
- During merge operations, the log files are used to compare timestamps and ensure that updates are applied in the correct order based on timestamp comparisons.
- If a log file limit is reached (e.g., 200 entries), the system returns a message indicating that manual flushing of log files are required by first synchronizing the servers, ensuring data consistency, and preventing data overloading, loss.

iii) **Mapping Query Languages:**

- Mapping query languages between different backend systems involves translating queries from one system's syntax to another system's syntax.
- For example, SQL queries in PostgreSQL are translated to MongoDB query syntax when performing similar operations in MongoDB (e.g., querying triples, updating triples).
- HBase operations use HBase-specific APIs and methods for querying and updating data, which may differ significantly from SQL or MongoDB syntax.
- The TripleStore class abstracts these differences by providing consistent methods (query, update, merge) that internally handle the translation or execution of queries based on the backend system in use.
- Mapping query languages ensures that the same logical operations (e.g., querying a triple, updating a triple) can be performed

seamlessly across different backend systems without requiring the user to manage system-specific syntax or APIs directly.

iv) **Flask Routes and API Endpoints:**
- Flask routes are defined for handling different operations such as querying, updating, and merging triples.
- API endpoints are designed to receive JSON data for updating and merging triples.
- Error handling is implemented to manage cases like log file limits and database connection errors.

v) **Integration with Front End:**
- The Flask application integrates these backend functionalities with a user-friendly front end.
- HTML templates are used for rendering forms and displaying results.

V. Testing

We have the YAGO dataset, which contains many triples of subject, predicate, and object. We utilized it for testing our system. To ensure the functionality of our application, we employed a systematic approach. Initially, we performed query operations across all three servers, cross-verifying to ensure consistency in the results obtained.

Subsequently, to assess the update operation, we executed updates on a single server, specifying subject, predicate, and object values. After confirming that the request was successful, we queried the same subject and checked whether the database was updated.

For merging, we ensured that data with the latest timestamp resides in the database of the server to which the client is connected. We used query operations to ensure that the merge was successful. We queried in our cuurent server so that we get the correct result. Regarding challenges faced during testing, HBase is taking significant time to perform update and merge operations. As the YAGO dataset contains a lot of data, loading data has taken more time in all databases.

VI. User Interface

The user interface developed for the distributed NoSQL triple store prototype provides users with an intuitive platform to interact with the system's functionalities. The entry point has server selection, and the client has to select which server he wants to connect to.

Subsequently, the user is presented with options to query, update, or merge data. If the user chooses to query, they are directed to a page where they input the subject they wish to query. Upon submitting the subject, the server retrieves and presents the triples associated with the subject.

In the update page, the user is prompted to input the subject, predicate, and object values. After providing these values, the client sends them to the server, and the updates are executed accordingly. If the update is successful, a confirmation message is displayed. Otherwise, an appropriate failure message is provided.

On the merge page, the client is required to input the server ID with which they want to merge. After submitting the server ID, the request is sent to the server, and the merge operation is executed successfully.

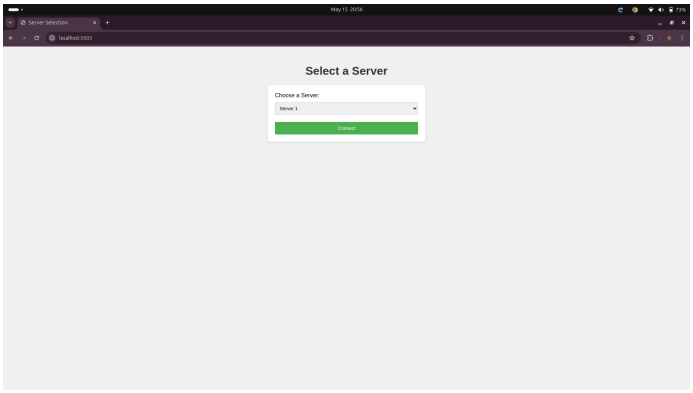
This is the workflow through which the user interacts with the system, making use of the various functionalities provided by the user interface.

VII. Advanced Features

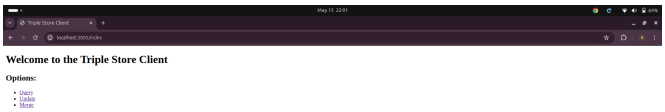
Flask serves as a versatile framework for client-server communication in web applications, offering a straightforward approach to defining routes, handling requests, and generating responses. Through its routing mechanism, Flask allows developers to map URL paths to specific functions, enabling precise control over how clients interact with the server. By defining view functions for each route, Flask facilitates request processing, data manipulation, and response generation, ensuring seamless communication between clients and the server. With built-in support for various response formats, including JSON, HTML, and plain text, Flask enables developers to tailor responses to the specific needs of clients. So, we chose flask for server-client communication.

VIII. Evaluation & Results

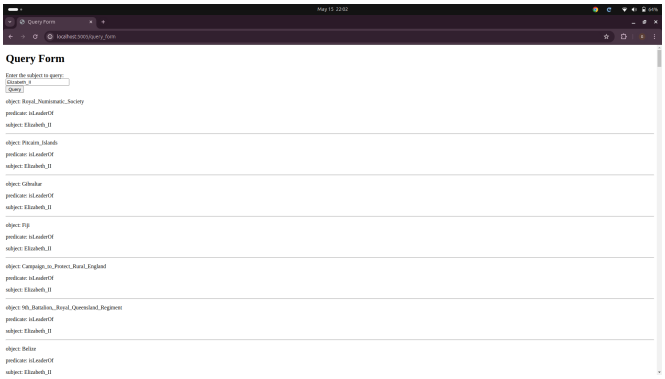
We will have a dropdown of what server to select and client can select the server which he wants.



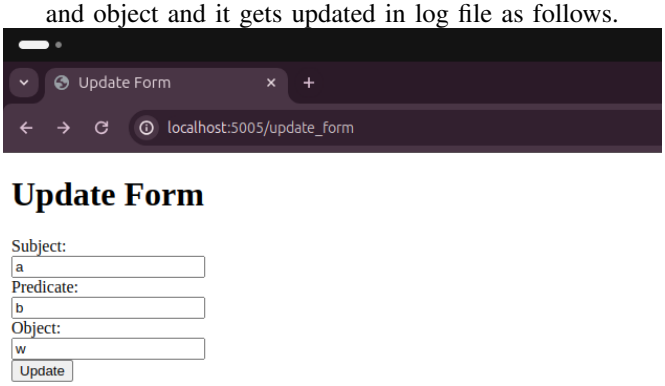
After connecting to the required, we will get options of querying, updating or merging.

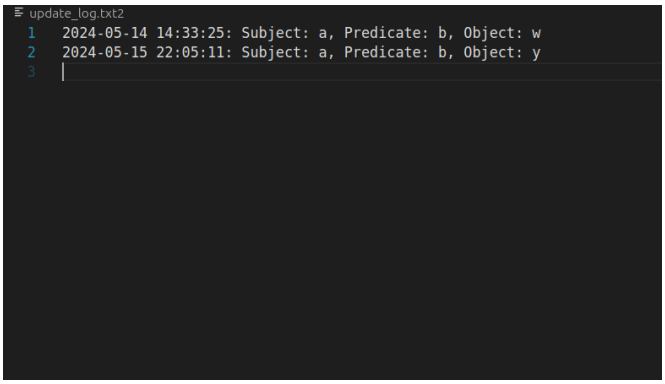


We will query by entering the subject and obtain the result.

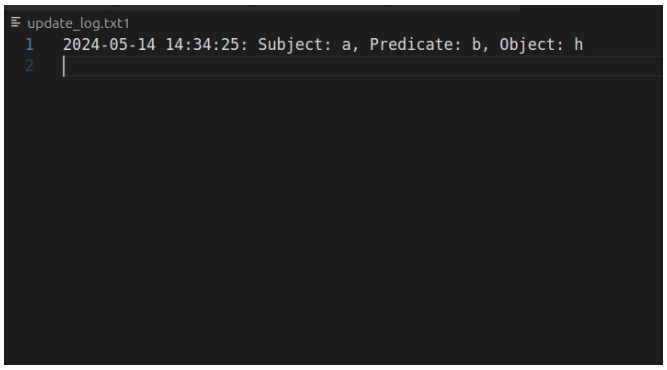


Now if user wants any triple, he will give subject, predicate and object and it gets updated in log file as follows.

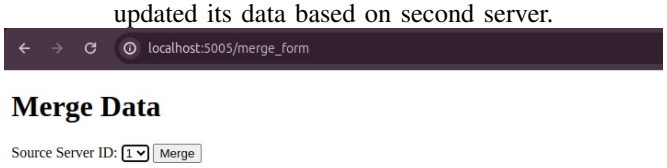


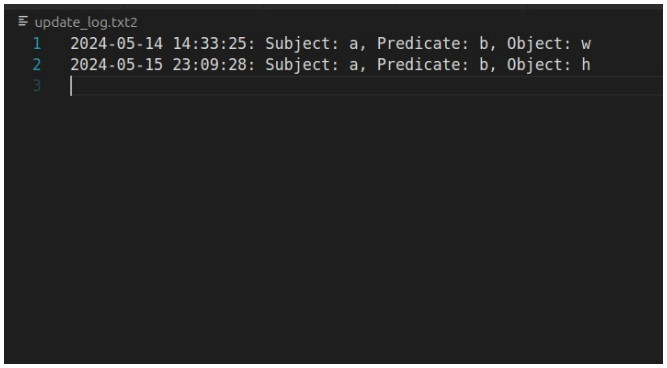


This is the log file of first server.



We then got a merge request in second server to merge with first server. As we know that latest data is in second server, we have to update first server. We can clearly see the updated log file of first server showing that first server has updated its data based on second server.





IX. Conclusion

In conclusion, we have successfully designed and implemented a prototype of a distributed NoSQL triple store utilizing state-based objects. The system follows a client-server architecture, with three servers (MongoDB, PostgreSQL, and HBase) and a client interface developed using Flask.

The system architecture ensures data consistency and integrity by maintaining state-based objects on each server, representing the current state of the triple store. These state-based objects are updated during merge operations to synchronize data across servers.

The implementation includes methods for querying, updating, and merging triples, along with appropriate data

structures and libraries for interacting with different backend systems. Testing was performed using the YAGO dataset, verifying the functionality of the application through systematic testing procedures.

The user interface provides an intuitive platform for users to interact with the system, offering options for querying, updating, and merging data. Through server selection and simple form inputs, users can perform various operations seamlessly.

Overall, the prototype demonstrates the feasibility of implementing a distributed NoSQL triple store using state-based objects, providing users with efficient data management and manipulation capabilities across multiple servers.

X. References

Please find the source code here,
https://github.com/kushaldasari/nosql.