Data Synchronization Across Heterogeneous Systems

Mohan Raj Vandana
IMT2022123

Harshavardhan Tangudu
IMT2022104

Saketh Sivananda Manchiraju
IMT2022085

I. Introduction

This project implements data synchronization mechanisms across multiple heterogeneous database systems — specifically MongoDB, PostgreSQL, and Apache Hive. We focus on maintaining data consistency across these systems while supporting independent read and update operations.The significance of this project lies in handling datasets in distributed environments, catering to the increasing demand for flexible data storage and retrieval mechanisms.

The project's primary objectives include developing a prototype capable of performing essential operations such as querying, updating, and merging data across multiple servers,and enabling synchronization between servers via a merge function, each employing distinct frameworks for data processing and storage.

The core functionality revolves around:

- Independent `GET` and `SET` operations for each database system
- An operation logging (oplog) mechanism to track changes
- A `MERGE` functionality that synchronizes data states between systems based on operation logs
- Support for composite primary keys (`student_id`, `courseid`) for uniquely identifying records

The implementation follows object-oriented programming principles in Python to create modular and extensible code that abstracts the heterogeneous nature of the underlying systems while providing a unified interface for data operations.

II. System Architecture

We have designed a distributed NoSQL system consisting of three servers, each containing identical data. Each server maintains a database with a table to store the actual data and we maintained an oplog file (operation log) on each server. This oplog records all update operations that happened in its database and plays a crucial role in the merge and synchronization processes across servers.

Client can communicate with the server of it's choice and perform operations like querying,updating the server,merging two servers.

Server offers services like querying rows having particular (`student_id`, `course_id`), updating object value of rows having a given (`student_id`, `course_id`), and merging the current server with other server, i.e., synchronizing two servers.

Fig. 1. Architecture

The architecture follows a service-oriented approach where each database system exposes standard `GET` and `SET` operations, while maintaining operation logs that can be used for state synchronization.

III. Methodology

A. *Input File:* `testcase.txt`

The input to the system is a file named `testcase.txt`, which consists of a sequence of operations. Each line corresponds to a database operation and has the format:

```
<counter>, <DB_NAME>.<OPERATION((<STUDENT_ID>,
<COURSE_CODE>), <GRADE>)
<DB_NAME>.MERGE(<OTHER_DB>)
```

Here, `<counter>` serves as a logical timestamp (or operation counter). The supported operations are:

- `SET((<student_id>, <course_id>), <grade>)`: Stores the given grade for a student-course pair.
- `GET((<student_id>, <course_id>))`: Retrieves the grade associated with the student-course pair.

- `MERGE(<other_db>)`: Synchronizes data between two databases based on timestamp ordering.

An example snippet from `testcase.txt`:

```
1, HIVE.SET((SID1033, CSE016), D)
1, MONGO.SET((SID1310, CSE020), C)
MONGO.MERGE(PSQL)
```

B. Main Driver Execution

The main driver code is responsible for reading the operations sequentially from `testcase.txt` and invoking the corresponding method on the appropriate database server. Initialization of the database handlers is done then, each line from the input file is parsed, and the corresponding `SET`, `GET`, or `MERGE` function is called on the target server instance.

C. Merge Functionality

The `MERGE` operation is implemented to synchronize the data between two databases: a *source* and a *target*. The goal is to ensure consistency, using the **latest-write-wins** strategy based on operation timestamps (counters).

IV. Implementation Details

A. Data Model:

The data structure is based on the `student_course_grades.csv` file with the following schema:

- **student_id** (string): Part of composite primary key
- **course_id** (string): Part of composite primary key
- **roll_no** (string): Student's roll number
- **email_id** (string): Student's email address
- **grade** (string): Student's grade in the course

B. Database Connectors

We implemented separate connector classes for each DB `HiveTripleStore`, `MongoTripleStore`, and `PostgresTripleStore` to interact with Hive, MongoDB, and PostgreSQL respectively. Each class encapsulates connection logic and provides a consistent services for database operations like `GET`, `SET`, and `MERGE`.

Hive Connector: Initialized using host and port, the Hive connector sets up a Beeline/JDBC connection. The database (`grade`) is selected via `connect_to_db()`.

MongoDB Connector: Connects to a local MongoDB instance using host and port. The target database is selected similarly.

PostgreSQL Connector: Uses host, port, user credentials to establish a connection. The specified database is accessed with `connect_to_db()`.

C. Database Libraries

We used `pymongo`, `psycopg2`, and `pyhive` for MongoDB, PostgreSQL, and Hive, respectively. These libraries were selected for their reliability, compatibility, and ease of integration.

All connectors follow a unified initialization pattern: create an instance, connect to the server, and select the working database. This modular design supports scalability and extension to additional backends.

D. Core Functions:

1) GET Operation

The GET operation retrieves the grade for a specific student-course combination:

- Input: student_id, course_id
- Output: grade (or None if not found)
- effect: Records the operation in the system's oplog

2) SET Operation

The SET operation updates or inserts a grade for a specific student-course combination:

- Input: student_id, course_id, grade
- Output: Boolean indicating success
- effect: Records the operation in the system's oplog

3) MERGE Operation

The MERGE operation synchronizes the state between two systems, This function is in the main driver :

- Input: Source system oplog and Target system oplog
- Output: None
- effect: Applies relevant SET operations from the source system's oplog to the target system

1) Retrieve the operation logs (oplogs) from both the source and target databases.
2) Maintain a dictionary to track the latest update for each `(student_id, course_id)` key based on the counter value.
3) For each entry in the source's oplog:
   - If the key is not present in the target's oplog, apply the `SET` operation in the target.
   - If the key is present in both oplogs, compare the counters:
     - If the source has a higher (more recent) counter, apply the update to the target.
     - Otherwise, ignore the source update, as the target already has the latest value.

This ensures that each key in the target database reflects the most recent write across both systems after the merge completes.

E. Operation Log (Oplog) Architecture:

Each database system maintains its own operation log with the following structure:

- **Timestamp/counter**: counter /time stamp of the operation
- **Operation type**: `GET` or `SET`
- **Parameters**: For `GET` operations - (student_id, course_id), for `SET` operations - ((student_id, course_id), grade)

F. Merge Functionality:

The merge operation is designed to synchronize the state of one system with another based on operation logs. The implementation considers:

- **Idempotency**: An operation is said to be *idempotent* if applying it multiple times has the same effect as applying it once. In the context of our system, we only perform `SET` operations during synchronization or merge. These operations are inherently idempotent because setting a value repeatedly with the same data and timestamp results in no further change after the first application.
- **Commutativity**: Commutativity implies that the order in which we merge operation logs does not affect the final result, though it may affect which system ends up holding the merged state.

  Consider two systems, A and B, each maintaining an operation log. When we perform:
  - `A.merge(B)`: System A incorporates all of B's operations that are newer than its own.
  - `B.merge(A)`: System B incorporates all of A's operations that are newer than its own.

  Although A and B each retain different copies of the final state, the *content* (i.e., the union of all relevant operations based on timestamp comparison) is the same in both. Thus, merging is commutative in terms of the resulting logical state, though not in terms of which physical system contains it.
- **Associativity**: Associativity means the grouping of merge operations does not affect the final outcome. That is,

  `MERGE(A, MERGE(B,C))= MERGE(MERGE(A,B),C)`

  When synchronizing multiple databases or systems, we can combine any two systems first and then merge with a third without changing the result.
  - *Example*: Suppose A, B, and C are systems with disjoint operation logs. Whether we compute `MERGE(A, MERGE(B, C))` or `MERGE(MERGE(A, B), C)`, the final content (union of all three logs, considering timestamp-based conflict resolution) remains the same.

G. Main Execution Logic:

The `main()` function, which orchestrates the reading and processing of database operations from a file named `test_cases.txt`. The goal is to parse, interpret, and execute a series of `GET`, `SET`, and `MERGE` operations across three different database systems: PostgreSQL, MongoDB, and Hive.

Reasons for Choosing Databases:

Hive:

- Scalability: Hive is built on top of Hadoop, providing scalability by distributing data across multiple nodes in a cluster.
- Integration with Hadoop Ecosystem: Hive seamlessly integrates with Hadoop components such as HDFS and YARN, offering a cohesive big data processing environment.

MongoDB:

- Flexible Schema Design: MongoDB's document-oriented structure allows for flexible schema design, suitable for storing triples with varying structures.
- High Performance: MongoDB is optimized for high performance with features like indexing and sharding, enhancing query performance and scalability.
- Horizontal Scalability: MongoDB supports horizontal scalability through sharding, facilitating the distribution of data across multiple nodes.

PostgreSQL:

- ACID Compliance: PostgreSQL ensures data consistency, integrity, and durability, essential for a triple store.
- Rich Feature Set: PostgreSQL offers advanced features such as complex queries, indexing, and support for advanced data types.
- Mature Ecosystem: PostgreSQL has a mature ecosystem with extensive community support, documentation, and third-party tools.

Summary:
Overall, the main logic provides a deterministic, timestamp-based synchronization and logging mechanism across heterogeneous database systems. It supports fine-grained operational control and facilitates eventual consistency by leveraging operation logs for replay and conflict resolution.

V. Testing : Consistency Verification via Merge

To validate the correctness of our `MERGE` operation, we designed a controlled test scenario. We used records for five students from `grades.csv` file . We then independently modified these records in each system with different grades and timestamps to simulate divergence.

The student-course pairs and their latest grade updates (based on logical timestamps) are listed below:

- (`SID1033`, `CSE016`) → Grade `C` at timestamp 10
- (`SID1310`, `CSE020`) → Grade `A` at timestamp 12
- (`SID1132`, `CSE004`) → Grade `B` at timestamp 14
- (`SID1378`, `CSE018`) → Grade `A` at timestamp 9
- (`SID1428`, `CSE014`) → Grade `B` at timestamp 11

After invoking the `MERGE` operation across all database systems, each should converge to a consistent state reflecting

the most recent updates. The expected final grades after synchronization are:

```
C, A, B, A, B
```

This test confirms that our merge logic correctly resolves data conflicts based on timestamps and ensures eventual consistency across heterogeneous backends.

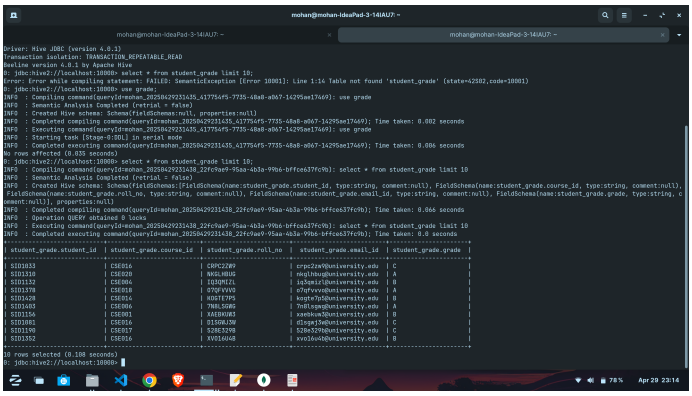

Fig. 2. Hive

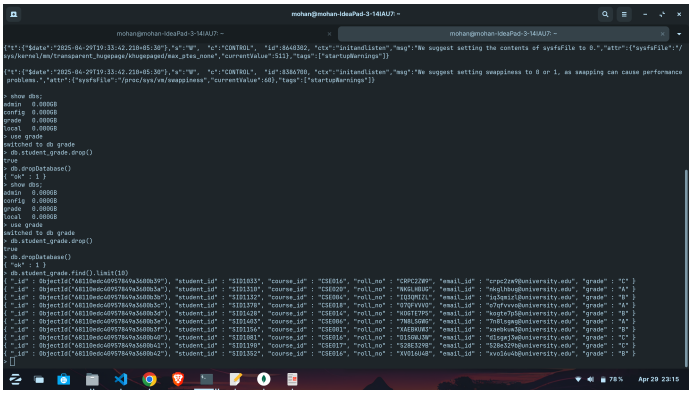

Fig. 3. Mongo

VI. Conclusion

This project successfully implements data integration and synchronization mechanisms across MongoDB, PostgreSQL, and Hive. The implementation provides:

- Independent `GET` and `SET` operations for each system
- Merge functionality that ensures eventual consistency

The system's design ensures that data remains consistent across all databases despite their heterogeneous nature, and the merge operation's mathematical properties guarantee convergence to a consistent state regardless of operation order.

VII. Individual Contributions

- **Mohan Raj (IMT2022123)**: 34% – Implemented Hive connector, and the merge execution logic in the main driver.
- **Saketh (IMT2022085)**: 33% – Implemented mongo connector, designed operation log structure, and merge functionality.
- **Harshavardhan (IMT2022104)**: 33% – Implemented PostgreSQL connector, handled test cases file in Main driver(regex expressions) and created test cases.