Project report

> Team Members and their contributions
>
> - Subham Agarwala - IMT2022110
> - Divyam Sareen - IMT2022010
> - Owais - IMT2022102

Problem Statement

In distributed environments, the same dataset may be stored and updated independently across multiple heterogeneous systems. This can lead to inconsistencies in data values when systems do not share updates directly.

The goal of this project is to address the problem of data synchronization across three different systems — MongoDB, Hive, and PostgreSQL — that each maintain a redundant copy of a student course grades dataset. Each system supports `GET` and `SET` operations for accessing and modifying the grade corresponding to a `(student_id, course_id)` pair.

Due to independent updates, the systems may diverge. To resolve this, a `MERGE(source_system)` function is implemented, which uses operation logs instead of direct access to data. These logs record the sequence of `GET` and `SET` operations performed in each system.

The objective is to ensure that the systems can be brought back into a consistent state by applying these operation logs in a defined order, without violating causality or overwriting newer updates.

Summary of What We Have Done

This project implements a synchronization system between three heterogeneous data storage platforms: **MongoDB**, **Hive**, and **PostgreSQL**. Each system maintains a redundant copy of the same dataset — *student_course_grades*, uniquely identified by a `(student_id, course_id)` pair.

The main features of our implementation include:

- **SET and GET Operations**: Each system supports `SET(table name, column name, student id, course id, new_value)` that takes to update a grade, and `GET(table name, column name, student id, course id)` to retrieve it. These operations are invoked through a centralized controller, that is the main file, and are also recorded in that system's **operation log** (oplog) along with the merge operations.
- **Operation Logs**: Each system maintains its own oplog, which captures all executed `SET`, `GET` and `MERGE` commands in timestamp order. These logs are used later during synchronization.
- **MERGE Operations**: A `MERGE` command is used to sync one system with another. During a merge, the controller reads the source system's oplog and `SET` operation to each field, on the basis of the last update made to that field. This is done with the help of the timestamps maintained in the oplog. The implementation of this feature is used by using a map to track each change made while

parsing the oplog, and for each key in the map, value is updated sequentially on the basis of timestamp, and then only the final values of the map are written down to the database system.

- **Testcase Execution**: A test case file (`testcase.in`) defines the sequence of `SET`, `GET`, and `MERGE` operations across systems. The centralized driver parses and executes each command in order, ensuring controlled testing of all functionalities.
- **No Direct Data Access Between Systems**: All synchronization is based solely on oplogs. One system cannot directly query another's data.

This approach simulates an environment where systems update independently and are later synchronized through operation replay. The solution demonstrates how consistency can be achieved using logs, without requiring shared state or real-time coordination between systems.

Components of Code:

1. DatabaseDAOInterface

This interface serves as the contract for interacting with different databases in the system, abstracting the data access layer for MongoDB, PostgreSQL, and Hive.

It defines the methods required to perform CRUD operations on data for synchronization, such as `set()` and `get()`.

- **Operations**:
  - `set()`: Updates the value for a given `(tablename, columnname, student_id, course_id)` tuple.
  - `get()`: Retrieves the grade for a given `(tablename, columnname, student_id, course_id)` tuple.

```
public interface DatabaseDaoInterface {
    // Method to set grade for a given student-course pair
    void set(String tableName, String columnName, String studentId, String courseId, String grade);

    // Method to get grade for a given student-course pair
    String get(String tableName, String columnName, String studentId, String courseId);
}
```

2. Main

The `main()` method orchestrates the execution of the entire data synchronization process by parsing the input commands and invoking corresponding operations like `SET`, `GET`, and `MERGE`.

It allows for using `testcase.in` file to run the testcases.

The main method reads a sequence of commands (SET, GET, MERGE) from an input file and calls the appropriate database operations on the respective systems (MongoDB, Hive, PostgreSQL).

- **Flow**:
  1. Initialize systems (MongoDB, Hive, PostgreSQL).

2. Load the data into the databases, if required, by asking prompts from the user.
3. Parse the input test case.
4. Execute the sequence of operations defined in the test case (SET, GET, MERGE).

3. Oplog.java

The `Oplog` class is used to track the sequence of operations (SET/GET) performed on a database system, storing each operation with a timestamp.

The fields stored are:

String tableName,
String studentID,
String courseID,
String column,
String newValue,
ZonedDateTime timestamp,
String operationType

Tracks operations to facilitate merging and synchronization across systems.

- **Operations**: Each operation is logged with a timestamp and the details of the operation (SET/GET and its arguments).

4. CsvToSystemImporter

This class imports the data from the CSV file into the systems (MongoDB, Hive, PostgreSQL), mapping the data into the respective tables.

Reads the student course grades data from a CSV file and populates the corresponding tables in each system.

- **Operations**:
  - Read the CSV.

- Insert data into MongoDB, Hive, and PostgreSQL.

5. SystemDao

The `SystemDao` classes provide the necessary functionality to interact with a specific database system (e.g., MongoDB, Hive, PostgreSQL), offering the `SET` , `GET` and `MERGE` operations. This class implements the `DatabaseDaoInterface`.

Encapsulates database-specific logic for performing `SET`, `GET` and `MERGE` operations.

- **Operations**: This class would implement database interactions for each system.
  - For MongoDB, it interacts with MongoDB's Java driver.
  - For Hive, it uses Hive's JDBC interface. Our Hive setup also uses Postgres as a metastore.
  - For PostgreSQL, it uses PostgreSQL's JDBC driver.

  We have implemented the functions that DatabaseDAOInterface wants us to implement.

How We are connecting to each database

MongoDB

- **Driver**: The official MongoDB Java driver (`com.mongodb.client.MongoClient`) is used to connect to MongoDB.
- **Connection**: A `MongoClient` is created using `MongoClients.create(uri)` where the `uri` specifies the MongoDB server connection string (e.g., `mongodb://localhost:27017`).
- **Data Import**: After establishing the connection, data is imported by converting CSV rows into BSON documents and inserting them into a MongoDB collection using methods like `insertOne()` or `insertMany()`.

Hive

- **Driver**: The Hive JDBC driver (`org.apache.hive.jdbc.HiveDriver`) is used for connecting to Hive.
- **Connection**: A connection to Hive is established through Spring framework's `jdbcTemplate` . We make a BasicDataSource with the following parameters - Driver, URL, User, Password. This DataSource is used to initialize a new object of the jdbcTemplate. We are accessing the hiveserver through port 10000.
- **Data Import**: Data is loaded into Hive tables using `LOAD DATA LOCAL INPATH` commands once the connection is established.

PostgreSQL

- **Driver**: The PostgreSQL JDBC driver (`org.postgresql.Driver`) is used to establish the connection.
- **Connection**: A connection to PostgreSQL is created using `DriverManager.getConnection(url, user, password)` where `url` is the PostgreSQL connection string (e.g., `jdbc:postgresql://localhost:5432/postgres_db`).

- **Data Import**: After the connection, data is typically inserted into PostgreSQL tables using `PreparedStatement` objects to execute batch inserts for efficiency.

Merging Logic

- **Identify the Source and Target Systems**:
  The first step in the merge process is identifying the systems involved in the synchronization. The source is typically the external system (e.g., MongoDB, Hive), and the target is the system where the merged data is eventually stored (e.g., PostgreSQL).
- **Load Operation Logs (Oplogs)**:
  Operation logs (or oplogs) are files or records that track changes (like updates, inserts, or deletes) in the source system. They contain information like:
  - The record being changed (identified by a unique key).
  - The field being modified.
  - The new value of the modified field.
  - A timestamp indicating when the change was made.

  The merge logic loads these oplogs from both the source and the target systems. The logs help in comparing changes made in both systems and determining what needs to be synchronized.
- **Compare Changes from Both Systems**:
  Once the oplogs from both systems are loaded, the merge logic compares the entries to determine which records need to be updated. This comparison is typically done based on:
  - **Timestamps**: If the external system's oplog contains a more recent timestamp than the target system's oplog, it indicates that the external system's data is more up-to-date.
- **Update Target System with Latest Data**:
  After determining which records are newer or missing in the target system, the merge logic proceeds to update the target system. This typically involves:
  - **Updating existing records**: If a record exists in the target system but needs an update, the logic replaces the old values with the new ones from the source system.

  The update is performed based on the latest information from the source system, ensuring that the target system stays synchronized with the external source.
- **Log and Track Updates**:
  After each update operation, the logic logs the changes made to the target system. This log typically includes information such as:
  - The unique identifier of the record.
  - The fields that were updated.
  - The new values that were applied.
  - A timestamp to track when the update was made.

Testing the Code

To validate the correctness of the implementation, a structured testing approach was followed using a test input file named `testcase.in`. This file contains a chronological sequence of operations including `SET`, `GET`, and `MERGE` commands across the three systems: MongoDB, Hive, and PostgreSQL.

```
HIVE.GET(student_course_grades, grade, SID1033, CSE016)
HIVE.SET(student_course_grades, grade, SID1033, CSE016, B)
HIVE.GET(student_course_grades, grade, SID1033, CSE016)

HIVE.GET(student_course_grades, grade, SID1310, CSE020)
HIVE.SET(student_course_grades, grade, SID1310, CSE020, C)
HIVE.GET(student_course_grades, grade, SID1310, CSE020)
MONGO.GET(student_course_grades, grade, SID1310, CSE020)
MONGO.SET(student_course_grades, grade, SID1310, CSE020, F)
MONGO.GET(student_course_grades, grade, SID1310, CSE020)
HIVE.MERGE(MONGO)
HIVE.GET(student_course_grades, grade, SID1310, CSE020)
MONGO.GET(student_course_grades, grade, SID1310, CSE020)
```

Each command in the file follows the format:

- `SYSTEM.OPERATION(arguments)`
- Example: `HIVE.GET(student_course_grades, grade, SID1033, CSE016)`, `HIVE.MERGE(SQL)`, `MONGO.GET(student_course_grades, grade, SID1310, CSE020)`

The testing process involves the following steps:

1. **Initialization**: All three systems are initialized with the same csv. Their operation logs (`hive_oplog.csv`, `mongo_oplog.csv`, `sql_oplog.csv`) are initially empty, and track every change made to the initial table.
2. **Execution of Commands**: The main controller reads the test case file line by line and executes each command in the given order. For every `SET` and `GET` operation, an entry is recorded in the respective system's operation log. `MERGE` operations are executed based on log replay from the source system to the target system.
3. **Conflict Handling**: Test cases also include scenarios where systems have conflicting updates. These help verify that the implementation respects the ordering of operations and avoids overwriting newer updates during a merge. This is solved by the timestamps stored in the oplog.
4. Test cases include:
    - `GET` operations without prior `SET`.
    - Chained merges such as `MONGO.MERGE(HIVE)` followed by `SQL.MERGE(MONGO)`.
    - Multiple `SET` operations to the same entry from different systems to check for conflicting writes.

Individual Contibutions

Subham Agarwala: I implemented the portion with MongoDB, I have done one third of the part, i.e. 34%

Divyam Sareen: I implemented the portion with Hive. I have done one third of the part as well, i.e. 34%

Mohammad Owais: I have implemented the part with Postgres. I have also done one third of the work. 34%.