International Institute of Information Technology Bangalore

NoSql Systems

DAS 839

Project Report

Team Members:

Abhinav Kumar (IMT2022079)
Siddhesh Deshpande (IMT2022080)
Jinesh Pagaria (IMT2022044)
Krish Patel (IMT2022097)

May 13, 2025

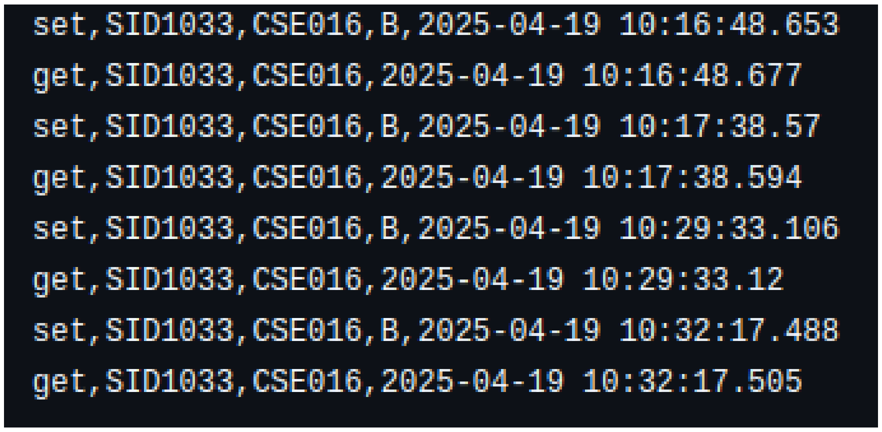
Problem Statement

In this project we have data redudantly loaded onto different systems and we peform a variety of operation like **SET**, **GET**, **MERGE** and we aim to Synchronize the data or the state of the systems involved. For this project the systems that we have chosen are **PostgreSql**, **Hive** and **MongoDB**.

Design of logs

The oplogs are stored in the following format :

- Get query - `get,<student-id>,<course-id>,<timestamp>`
- Set query - `set,<student-id>,<course-id>,<grade>,<timestamp>`



```
set,SID1033,CSE016,B,2025-04-19 10:16:48.653
get,SID1033,CSE016,2025-04-19 10:16:48.677
set,SID1033,CSE016,B,2025-04-19 10:17:38.57
get,SID1033,CSE016,2025-04-19 10:17:38.594
set,SID1033,CSE016,B,2025-04-19 10:29:33.106
get,SID1033,CSE016,2025-04-19 10:29:33.12
set,SID1033,CSE016,B,2025-04-19 10:32:17.488
get,SID1033,CSE016,2025-04-19 10:32:17.505
```

Figure 1: Sample oplogs

The oplogs don't have a grade field in get query as they don't change the data inside the datastore. The timestamp in the logs is used to compare with the timestamp inside datastore of the system which we want to merge with the system whose oplogs we are reading from. To determine if to update the grade or not during merge operation i.e. if the last_modified attribute in datastore is before the timestamp given in log file of other system's log then only we update the record value.

Design

We have added the 'last_modified' attribute to each tuple in datastore, which is set as the time of loading initially and then is changed when we run a set query or a merge query(only if the value was stale or lagged behind in time).In the oplogs we also add the timestamp for comparison during merge. We maintain an offset for other systems in order not to read the oplogs from beginning(just to avoid redundant checks not maintaining them is not going to affect the logic in anyway).

- For a **GET** operation we fetch the grade correspoding to the student-Id and course-Id and record it in the log with current timestamp.
- For a **SET** operation we modify the grade and 'last_modidfed' attribute to current time of operation and record it in the log with that timestamp.

- For the **MERGE** operation for the system we read the log of other system from the offset and if a set operation is encountered we get the 'last_modified' attribute of that row and if the value of timestamp in the log is greater than the last_modified the then we set the the grade for the according the log and also modify the last_modified attribute with the timestamp on the log. We also update our own log with a set operation with the timestamp that we just read from the other log's set operation.This step is required to ensure that no matter in what order systems do a merge they all should have the changes reflected in them. We also increase the offset of the system whose logs we are reading as we traverse so that we now from where to start reading the next time we do a merge with this system from the current system.

Our Program can run in 2 modes :

- Interpreter mode : Give operation to be perfomed in a specific format one by one and wait for that operation to be executed and to exit just type exit.
- Testcase mode : reads from testcase file.

Merge Logic

We store the 'last_modified' attribute for each tuples/records in the datastores denoting the last time the data was modified. We change this whenever a set query is executed. Initially the time of loading data is set. It is used to compare the timestamps in logs for merge.
We also maintain offsets for other systems inorder to not not read the full logs every time merge query is executed.
for the merge logic we do the following :

1. read the log file of the system to merge from,starting from the offset and consider only the set queries as get queries don't change the data.
2. For every set query get the 'last_modified' attribute from the datastore if the data is stale i.e. the last_modified attribute is earlier than the timestamp in log file then do the following :
3. set the grade and last_modified attributes values to to grade and timestamp in log file.
4. update the log file of the this current system which has called its own merge function with a set query because we have made a update to our data store that needs to be mentioned in the log so that when some other system tries to merge with this system then the merge happens correctly.
5. update the offset of the other system internally after reading each line of the log .

ACI Properties

Let the merge function be denoted by $\cup$. So $A \cup B$ mean State of B is merged in A.

Associativity

$$A \cup (B \cup C) = (A \cup B) \cup C$$

We achieve Associativity property by merging based on later timestamp and writing it log when we change the record with same timestamp this allows us to merge three system in any precidence and get the same end resut in the final system merge.

Commutativity

$$A \cup B = B \cup A$$

The System is not commutative in nature because when we call A.MERGE(B) then state of B is merged in A and hence the state of A and B are not same because the way A.merge(B) is defined doesnt say that A and B have same state . Hence the system is not commutative.

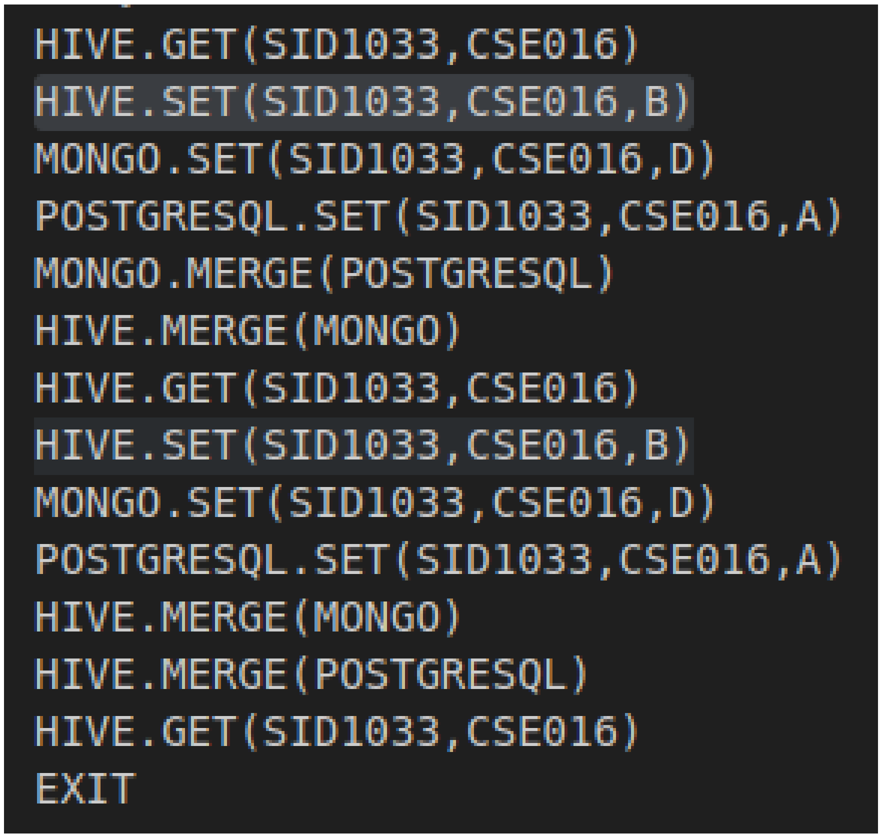
Idempotency

$$A \cup A = A$$

We achieve idempotency property by simply returning when same system is called for merge i.e. when $A.MERGE(A)$ is called we simply return from function without doing anything.

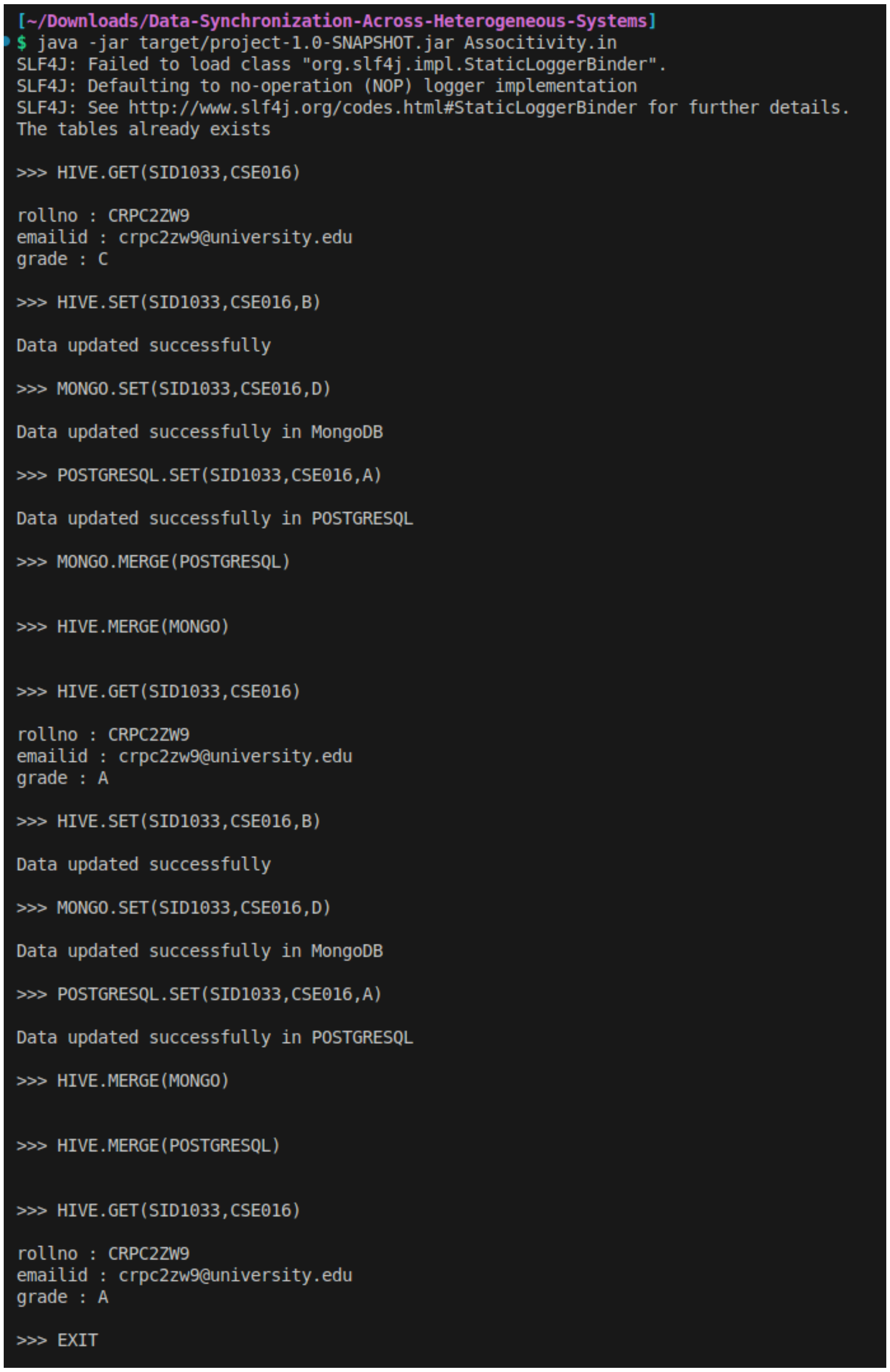
Testcase Execution

Associativity



```
HIVE.GET(SID1033,CSE016)
HIVE.SET(SID1033,CSE016,B)
MONGO.SET(SID1033,CSE016,D)
POSTGRESQL.SET(SID1033,CSE016,A)
MONGO.MERGE(POSTGRESQL)
HIVE.MERGE(MONGO)
HIVE.GET(SID1033,CSE016)
HIVE.SET(SID1033,CSE016,B)
MONGO.SET(SID1033,CSE016,D)
POSTGRESQL.SET(SID1033,CSE016,A)
HIVE.MERGE(MONGO)
HIVE.MERGE(POSTGRESQL)
HIVE.GET(SID1033,CSE016)
EXIT
```

Figure 2: Associativity testcase

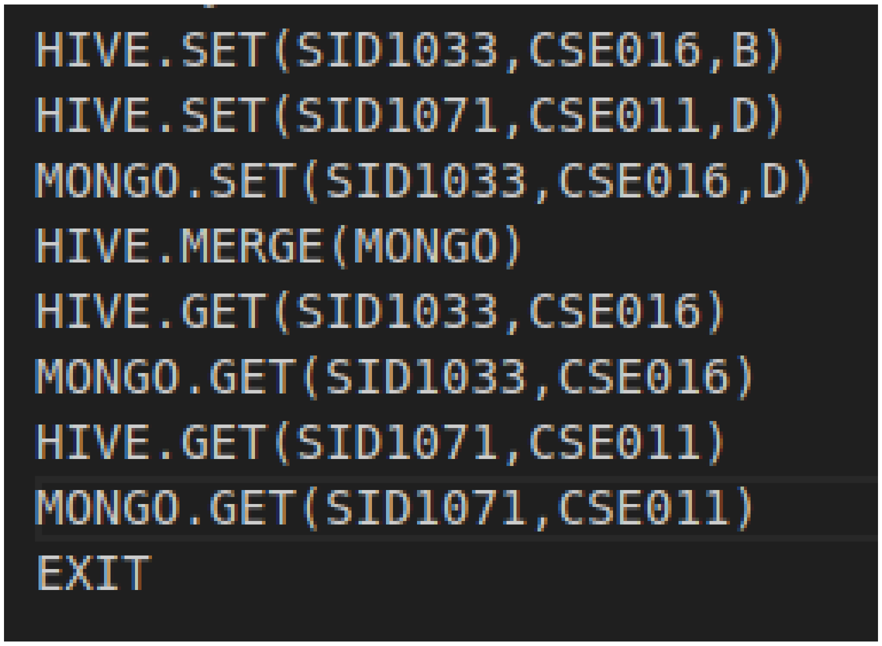


```
[~/Downloads/Data-Synchronization-Across-Heterogeneous-Systems]
$ java -jar target/project-1.0-SNAPSHOT.jar Associtivity.in
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
The tables already exists

>>> HIVE.GET(SID1033,CSE016)

rollno : CRPC2ZW9
emailid : crpc2zw9@university.edu
grade : C

>>> HIVE.SET(SID1033,CSE016,B)

Data updated successfully

>>> MONGO.SET(SID1033,CSE016,D)

Data updated successfully in MongoDB

>>> POSTGRESQL.SET(SID1033,CSE016,A)

Data updated successfully in POSTGRESQL

>>> MONGO.MERGE(POSTGRESQL)


>>> HIVE.MERGE(MONGO)


>>> HIVE.GET(SID1033,CSE016)

rollno : CRPC2ZW9
emailid : crpc2zw9@university.edu
grade : A

>>> HIVE.SET(SID1033,CSE016,B)

Data updated successfully

>>> MONGO.SET(SID1033,CSE016,D)

Data updated successfully in MongoDB

>>> POSTGRESQL.SET(SID1033,CSE016,A)

Data updated successfully in POSTGRESQL

>>> HIVE.MERGE(MONGO)


>>> HIVE.MERGE(POSTGRESQL)


>>> HIVE.GET(SID1033,CSE016)

rollno : CRPC2ZW9
emailid : crpc2zw9@university.edu
grade : A

>>> EXIT
```

Figure 3: output of Associativity testcase

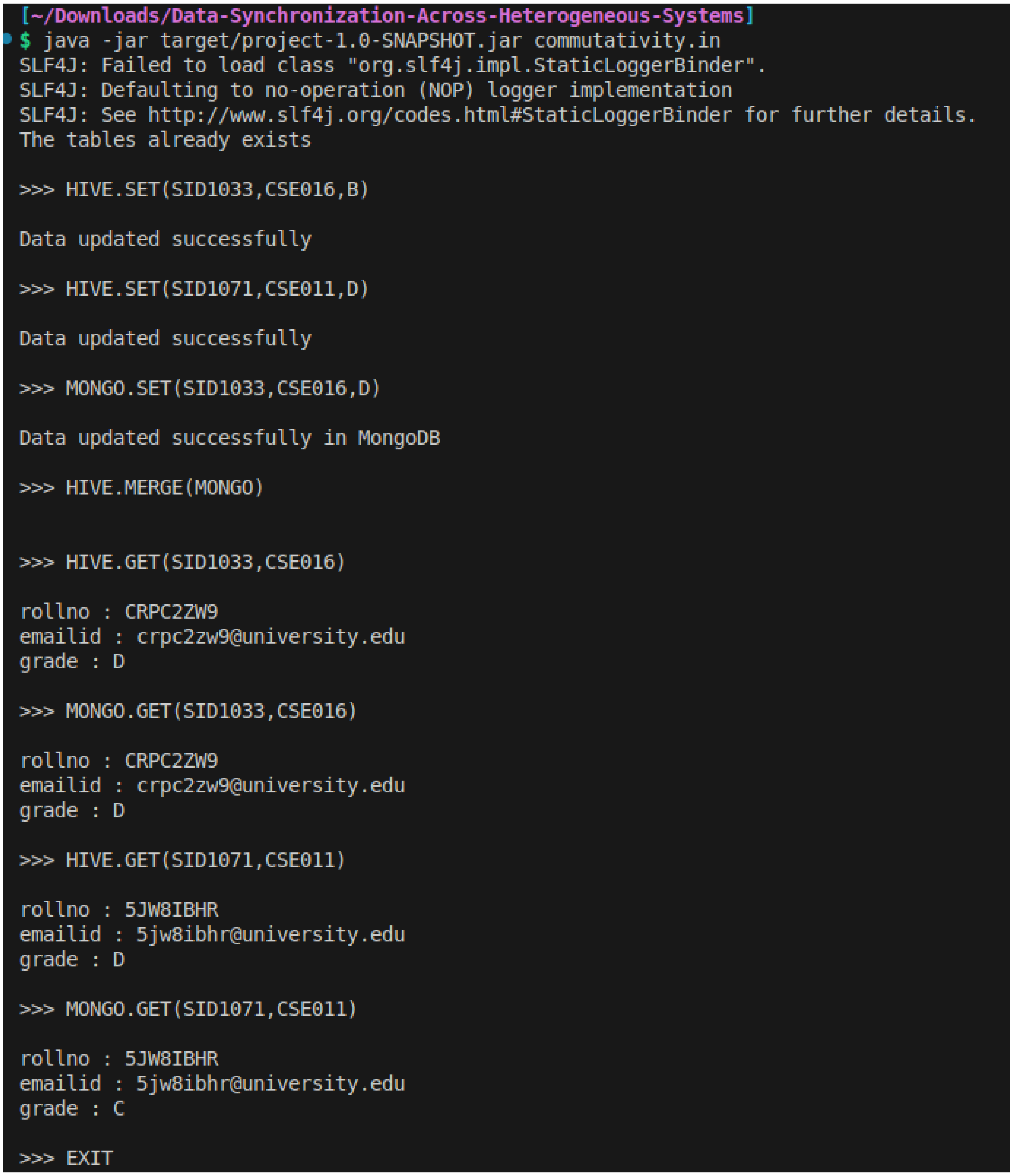
The above testcase demonstrate the Associativity property of our system as we can see that `HIVE.GET(SID1033,CSE016)` returns the same value even after doing the merge in 2 ways.
$A \cup (B \cup C) = (A \cup B) \cup C$ here A is Hive, B is Mongo and C is PostgreSql.

Commutativity



```
HIVE.SET(SID1033,CSE016,B)
HIVE.SET(SID1071,CSE011,D)
MONGO.SET(SID1033,CSE016,D)
HIVE.MERGE(MONGO)
HIVE.GET(SID1033,CSE016)
MONGO.GET(SID1033,CSE016)
HIVE.GET(SID1071,CSE011)
MONGO.GET(SID1071,CSE011)
EXIT
```

Figure 4: Commutativity testcase



```
[~/Downloads/Data-Synchronization-Across-Heterogeneous-Systems]
$ java -jar target/project-1.0-SNAPSHOT.jar commutativity.in
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
The tables already exists

>>> HIVE.SET(SID1033,CSE016,B)

Data updated successfully

>>> HIVE.SET(SID1071,CSE011,D)

Data updated successfully

>>> MONGO.SET(SID1033,CSE016,D)

Data updated successfully in MongoDB

>>> HIVE.MERGE(MONGO)


>>> HIVE.GET(SID1033,CSE016)

rollno : CRPC2ZW9
emailid : crpc2zw9@university.edu
grade : D

>>> MONGO.GET(SID1033,CSE016)

rollno : CRPC2ZW9
emailid : crpc2zw9@university.edu
grade : D

>>> HIVE.GET(SID1071,CSE011)

rollno : 5JW8IBHR
emailid : 5jw8ibhr@university.edu
grade : D

>>> MONGO.GET(SID1071,CSE011)

rollno : 5JW8IBHR
emailid : 5jw8ibhr@university.edu
grade : C

>>> EXIT
```

Figure 5: output of Commutativity testcase

The above testcase demonstrate the Non-Commutativity property of our system as we can see that `HIVE.GET(SID1071,CSE011)` and `MONGO.GET(SID1071,CSE011)` don't return the same value

Steps to run

1. download the project folder on the system.
2. open the terminal in the root of the project and run `mvn clean install` to build the project
3. for running the jar file please run `java -jar target/project-1.0-SNAPSHOT.jar` for interpreter mode and `java -jar target/project-1.0-SNAPSHOT.jar <testcase file>` for testcase mode

Note - We assume the data is redundantly loaded in all the datastores and all of them are running on their default ports.

Contributions

1. Siddhesh Deshpande(IMT2022080) - Implemented the merge, get, set for Hive.
2. Abhinav Kumar(IMT2022079) - Implemented the merge, get, set for PostgreSQL.
3. Krish Patel(IMT2022097) - Implemented the merge, get, set for MongoDB.
4. Jinesh Pagaria(IMT2022044) - Implemented logger.java file which has functions such as read and write that will read or write to the log files.

All have had equal 25% each contribution in the project.