DAS 839: NoSQL Systems	April 29, 2025
EndTerm Project	
Instructor: Dr. Vinu E. Venugopal	

Data Integration and Synchronization Across Heterogeneous Systems

Hive, PostgreSQL, MongoDB

Introduction

In this project, we explore the concepts of data integration and synchronization across heterogeneous database systems. We work with three systems: Hive, PostgreSQL, and MongoDB. Each system stores the same dataset — a CSV file containing student course grades — in its native format. The schema includes five fields: `Student ID`, `Course ID`, `Student Name`, `Mail`, and `Grade`, with the composite key being the combination of `Student ID` and `Course ID`.

Our goal is to implement consistent CRUD functionality, focusing particularly on the `GET` (read) and `SET` (update) operations for the `Grade` field. Furthermore, we provide a mechanism to `merge` the state of one system with another using an operation log (oplog), thereby ensuring eventual consistency across all systems.

CRUD Operations and Supported Services

Hive

- **Create:** `CREATE TABLE`
- **Read:** `SELECT Grade FROM table WHERE student_ID='...' AND course_ID='...'`
- **Update:** `UPDATE` (with ACID support enabled)
- **Delete:** Limited support

PostgreSQL

- **Create:** `INSERT`
- **Read:** `SELECT Grade FROM table WHERE student_ID='...' AND course_ID='...'`
- **Update:** `UPDATE table SET Grade='...' WHERE student_ID='...' AND course_ID='...'`
- **Delete:** `DELETE`

MongoDB

- **Create:** `insertOne`, `insertMany`
- **Read:** `findOne({ student_ID, course_ID }, { Grade })`
- **Update:** `updateOne({ student_ID, course_ID }, { $set: { Grade } })`
- **Delete:** `deleteOne`

System Architecture

The architecture consists of four Python modules:

- `main.py`
- `hive_manager.py`
- `postgres_manager.py`
- `mongo_manager.py`

Each manager module implements three primary methods: `get()`, `set()`, and `merge()`.

If the (studentID, courseID) pair given from `set()`, `get()` operations from any system don't exist in the given dataset, a message will be displayed conveying the same.

Oplog Structure

Each system maintains an operation log (`oplog`) that records all read and write operations. The oplog is structured as follows:

```
new_database=# select * from oplogs
;
        timestamp        | operation | student-ID | course-id | new_grade
-------------------------+-----------+------------+-----------+-----------
 2025-04-30 06:43:08.498 | SET       | SID1033    | CSE016    | C
 2025-04-30 06:43:29.349 | SET       | SID1033    | CSE016    | B+
 2025-04-30 06:44:21.796 | GET       | SID1033    | CSE016    | X
(3 rows)
```

- **Timestamp**
- **Operation Type:** GET or SET
- **Student ID**
- **Course ID**
- **Grade:** Set value or `X` for GET

Only `SET` operations are considered during merge operations for synchronization. We also log the merge operations, but only if the operation requires a new value to overwrite an existing value in my Database. This is logged as a `SET` operation with the old timestamp from the merged database.

Merge Functionality

The `merge()` function in each system accepts another system as an argument. For example, `hive.merge(sql)` merges the state of the PostgreSQL database into Hive.

Implementation Details

- A key-value store is created with composite keys (`student_ID`, `course_ID`).
- The oplog of the calling system is traversed to populate the key-value store with the latest `SET` values based on timestamps.
- The structure of key is a tuple containing the studentID & courseID and value is a tuple containing timestamp, new grade & a flag which is equal to "local" to represent that the calling system has updated the log.

- The oplog of the other system is then processed, updating the key-value store with newer values. Here the flag is updated to "remote" to indicate that the other system has updated it.
- The final key-value pairs are then written into the calling system using `set()` operations, if they have the "remote" flag.

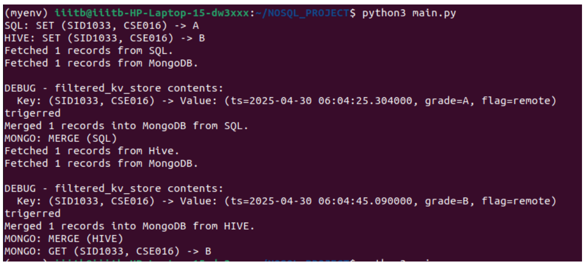
Merge Properties

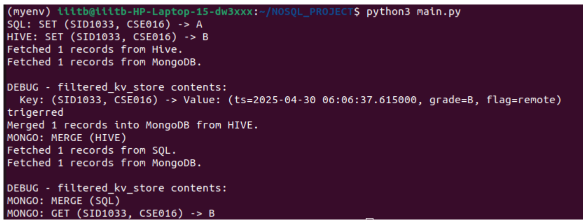
The merge operation defined for each system satisfies the following mathematical properties, which are critical to ensuring consistency and convergence in distributed systems.

Commutativity

Merging two operation logs in different orders results in the same final state:

$$A \oplus B = B \oplus A$$

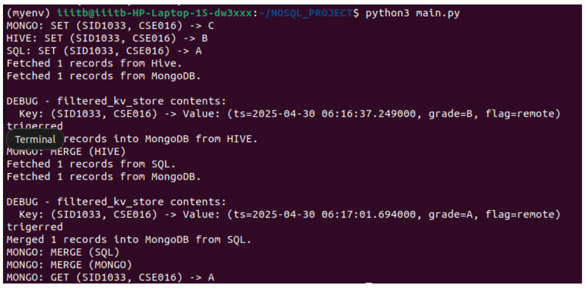


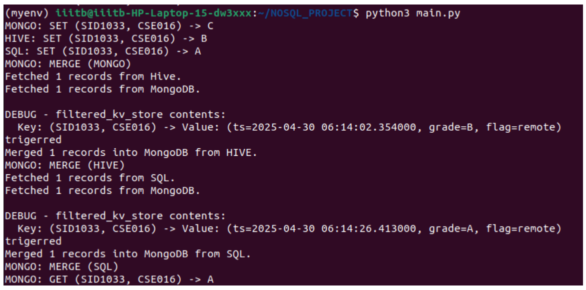


Associativity

The grouping of merges does not affect the final state:

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





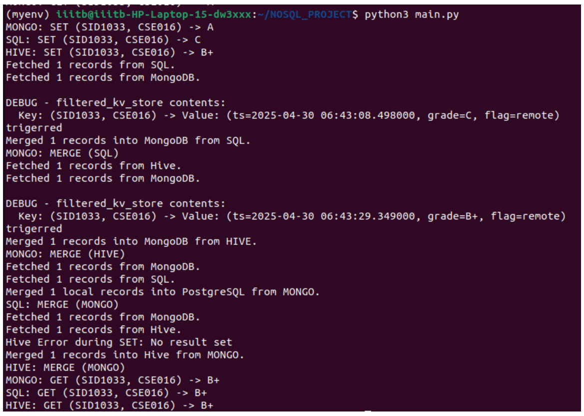
Idempotency

Merging a system with itself leaves it unchanged:

$$A \oplus A = A$$

Convergence

Once the views of all three databases became the same, they reached consistency.



Challenges Faced

- Hive connectivity issues due to port misconfiguration.
- Designing a merge logic that respects commutative properties; resolved via a deterministic key-value update strategy using oplogs.
- Initially we explored an alternative design, which achieved eventual consistency, but didn't achieve SEC. (Instead of updating merge as a set operation in the oplog with the old timestamp, you use a new timestamp).

Code Explanation

main.py

This script acts as the central driver for executing operations across the three database systems. It performs the following key functions:

- Imports the three manager classes: `HiveGradeManager`, `SQLGradeManager`, and `MongoDBGradeManager`.
- Initializes each manager instance using a common CSV file (`student_course_grades.csv`) and the appropriate database connection strings.
- Maps each system identifier (e.g., `HIVE`, `SQL`, `MONGO`) to its corresponding manager instance in a dictionary (`manager_map`).
- Reads instructions from an input file named `testcase_hive.in`, where each line specifies an operation in the format `SYSTEM.OPERATION`.
- Parses and delegates `SET`, `GET`, and `MERGE` operations to the appropriate manager. The `SET` command updates a student's grade for a given course, `GET` retrieves the current grade, and `MERGE` synchronizes the state of the current system with another using its oplog.
- Introduces a one-second delay between operations to ensure unique timestamps in the oplogs for consistent merging.

This file abstracts the orchestration logic from the underlying database details, focusing purely on parsing and dispatching user-defined operations.

hive_manager.py

This file defines the `HiveGradeManager` class, which manages student grades in an Apache Hive database. Its main responsibilities include table initialization, data access (get/set), operation logging, and merging updates from external systems (SQL, MongoDB). The key functionalities are:

- **Initialization:** Connects to Hive and creates two tables in the `new_database` schema—`grades` for storing student grades and `oplogs` for recording operations.
- **execute():** A generic helper method that runs SQL queries on Hive and returns results for `SELECT` queries.
- **get():** Retrieves a student's grade for a specific course and logs the operation in `oplogs`.
- **set():** Updates a student's grade in the `grades` table using `INSERT OVERWRITE` with conditional logic, and logs the operation.
- **log2():** Similar to `set()`, but designed to apply externally sourced changes during a merge, using a provided timestamp for proper ordering.
- **merge():** Imports the most recent `SET` operations from another system's oplog (either SQL or MongoDB), resolves conflicts based on timestamps, and applies newer external changes to the local Hive instance.
- **Operation Logging:** All `GET` and `SET` operations are recorded in the `oplogs` table with a millisecond-precision timestamp, enabling eventual consistency and cross-system merges.

Overall, this class ensures that Hive can act both as a primary data store and a replica target, maintaining consistency with other systems through a log-based synchronization strategy.

postgres_manager.py

This Python script defines a class `SQLGradeManager` that manages student grades using a PostgreSQL database, and supports data synchronization from Hive and MongoDB sources. It performs the following key functions:

- `__init__`: Initializes the database engine, metadata, and sets up tables using `initialize_tables()`.
- `initialize_tables`: Defines and creates two tables:
  - `grades` – holds student grades with a composite primary key (`student-ID`, `course-id`).
  - `oplogs` – logs operations (`GET`, `SET`) with timestamps.

  The method also loads initial data from a CSV file into the `grades` table.
- `get(student_id, course_id)`: Retrieves a student's grade for a specific course and logs the read operation.
- `set(student_id, course_id, new_grade)`: Updates the grade for a given student-course combination and logs the write operation.
- `merge(source_system)`: Synchronizes updates from an external system (`Hive` or `MongoDB`) into the local SQL database. It performs conflict resolution by keeping the most recent grade (based on timestamps) and logs each applied remote change.
- `_log_operation` and `_log_operation2`: Internal methods to insert records into the `oplogs` table, either with the current or provided timestamp.

This design ensures consistency and auditability of grade updates across multiple systems, while resolving conflicts by preferring the most recent data.

mongo_manager.py

The `mongo_manager.py` script defines the `MongoDBGradeManager` class, which interfaces with a MongoDB database to manage student course grades. It performs the following key functions:

- **Initialization:** Connects to MongoDB, drops and recreates the `grades` and `oplogs` collections, sets up a compound index on `student-ID` and `course-id`, and loads initial data from a CSV file.
- **Grade Retrieval (`get`):** Fetches the grade for a specific student-course pair and logs the access in the `oplogs` collection.
- **Grade Update (`set` and `log2`):** Updates an existing grade and logs the update operation. The `log2` method is a variant used during merge operations with an externally provided timestamp.
- **Logging (`_log_operation`):** Inserts a timestamped log entry into `oplogs` for each GET or SET operation.
- **Merge Functionality:** The `merge` method integrates grade updates from external sources (PostgreSQL or Hive) into MongoDB. It collects and compares the latest timestamped SET operations and applies only newer remote updates to MongoDB while logging them using `log2`.

This class ensures synchronization across databases and maintains a detailed log of all interactions.

Conclusion

This project demonstrates a robust approach to synchronizing data across heterogeneous systems using abstracted operations and operation logs. Through consistent and commutative merges, we can ensure eventual consistency and correctness across all platforms involved.

Team Members

- **IMT2022034** - Tarun Kondapalli
- **IMT2022065** - Amruth Gadepalli
- **IMT2022100** - Tahir Khadarabad