Data Synchronization Across Heterogeneous Systems

DAS 839 NoSQL Systems

Submitted by:

Group 2:

Abhay Bhadouriya – MT2024003

Jainish Parmar - MT2024065

Jaimin Jadvani – MT2024064

Files Submitted:

- Comprehensive Report on Multi-Database Synchronization System
- student_course_grades.csv
- Code Files
- testcase_1.in to testcase_10.in

Comprehensive Report on Multi-Database Synchronization System

Introduction

Hey there! We're thrilled to share this updated report on our multi-database synchronization system, which now integrates Apache Hive, MongoDB, and MySQL with a fresh twist. This project has evolved a lot since our last submission, and we've poured our hearts into refining the codebase and adding a solid set of test cases. The system is all about keeping student grade data in sync across these databases, using a clever timestamp-based merging approach. We've enhanced the OpLogManager class to log operations more effectively, and the new test cases really help us validate everything. Below, you'll find the setup commands and an updated look at our project directory to give you a feel for how we've set things up.

- **Setup Commands:**
  - **MongoDB:**
    - sudo systemctl start mongod to kick off the MongoDB service.
    - mongosh to jump into the MongoDB shell.
    - show dbs to see the list of databases.
    - use grade_db to switch to our grade_db database.
    - show collections to check out the collections.
    - mongodb-compass for a handy GUI to visualize everything.
  - **MySQL:**
    - sudo systemctl start mysql to get the MySQL service running.
  - **Hive Server:**
    - start-yarn.sh to fire up YARN.
    - start-dfs.sh to get HDFS going.
    - hive --service metastore in the metastore directory to set up the metastore.
    - hive --service hiveserver2 --hiveconf hive.server2.thrift.port=10000 to launch HiveServer2 on port 10000.
  - **Python:**
    - source venv/bin/activate to activate our virtual environment.
    - python3 -m src.cleanup to clear out the databases and log files.
    - python3 -m src.main to run the main script and see it all in action.
- **Project Directory:**
  - logs/: Where we keep all our operation log files.
    - log_init: A starting point log or script, maybe a blank slate or template.
    - oplog_hive.log: Tracks all the Hive operation goodies like INSERTs and DELETEs.
    - oplog_mysql.log: Logs MySQL operations.
    - oplog_mongo.log: Holds MongoDB operation details.
  - src/: The heart of our code.
    - systems/: Where the database magic happens.
      - hive_system.py: Our HiveSystem class for Hive operations.

            - mongodb_system.py: MongoDBSystem for MongoDB interactions.
            - mysql_system.py: MySQLSystem to handle MySQL tasks.
        - cleanup.py: Takes care of cleaning up databases and logs.
        - config.py: Keeps all our configuration settings like credentials and paths.
        - main.py: Ties it all together with the merge sequence.
    - tests/: Home to our test cases.
        - testcase_1.in: The original test input file.
        - testcase_1.in to testcase_10.in: Our shiny new test case files.
    - venv/: Our Python virtual environment with all the dependencies.
    - student_course_grades.csv: An updated CSV with the latest student grade data.

1. Detailed Explanations of the Merge Logic

Overview of Merge Logic:

The merge logic is the backbone of our synchronization system—it's what keeps data flowing smoothly between Hive, MongoDB, and MySQL. We've designed it to pick the latest operation (based on timestamps) and push it from the source database to the target. This happens inside the merge method of each system class—HiveSystem, MongoDBSystem, and MySQLSystem—and follows a specific merge sequence we've outlined in the test*.in files:

- HIVE.MERGE(SQL)
- HIVE.MERGE(MONGO)
- SQL.MERGE(HIVE)
- SQL.MERGE(MONGO)
- MONGO.MERGE(HIVE)
- MONGO.MERGE(SQL)

The OpLogManager reads operations from the source log files (like oplog_hive.log), processes them, and applies changes to the target based on timestamp and grade checks. It's a pretty neat system once you see it in action!

Core Principles:

- **Timestamp Precedence**: We always go with the operation that has the latest timestamp. If two timestamps are the same, we look at the grades to decide.
- **Operation Types**: We handle INSERT, UPDATE, READ, and DELETE. READs get logged but don't trigger merges, while the others drive the changes.
- **Idempotency**: Running the same merge again with the same log data won't mess things up unless there's a newer timestamp or a grade change.
- **HiveSystem Merge Logic**:
    - **Partition Management:** We check if partitions exist using SHOW PARTITIONS. For INSERT or UPDATE, we create a temp file with create_temp_data_file and load it with LOAD DATA LOCAL INPATH. For DELETE, we drop the partition with ALTER TABLE DROP.
    - **Timestamp and Grade Check**: A SELECT query grabs the timestamp from the target partition. If the source timestamp is newer or the grade's different, we update or insert. We use flags like should_insert (true if no partition exists) and should_update (true if the timestamp's older but grade mismatches or newer).
    - **Error Handling**: We wrap everything in try-except blocks. If something like a missing partition pops up, we log it and keep going—partial success is better than nothing!
- **MongoDBSystem Merge Logic**:
    - **Document Operations:** We use find_one to check documents by (student-ID, course-id). For INSERT/UPDATE, update_one with upsert=True sets the grade and timestamp. For DELETE, delete_one removes it.
    - **Decision Making**: We compare the document's timestamp with the source operation's. If the source is newer or grades differ, we update or insert.
    - **Indexing**: A unique index on (student-ID, course-id) keeps duplicates at bay and speeds things up.
- **MySQLSystem Merge Logic**:

  - **Relational Updates**: A SELECT fetches the timestamp and grade, and INSERT ... ON DUPLICATE KEY UPDATE handles INSERTs and UPDATEs. DELETE uses a standard query.
  - **Logic Flow**: We update if the source timestamp is newer or grades don't match.
  - **Transaction Control**: Each change gets its own commit to keep things intact.
- **Merge Sequence Execution**:
  - In main.py, we loop through system instances (like systems["HIVE"].merge("SQL")) to run the sequence.
  - This bidirectional sync ensures data moves everywhere, sorting out conflicts with timestamps.
- **Conflict Resolution Mechanism**:
  - **Timestamp-Based**: The latest timestamp always wins.
  - **Grade Mismatch**: If timestamps match but grades differ, we update to keep things accurate.
  - **Skip Logic**: Older timestamps get skipped unless grades differ.
- **Edge Cases**:
  - Hive missing partitions? We create them on the fly during inserts.
  - Duplicate keys in MongoDB/MySQL? Unique indexes save the day.
  - Null values? We ignore them for DELETE and READ.
- **Optimization Opportunities**:
  - Batch processing could cut down on query time.
  - Indexing timestamps might make comparisons zip along faster.
  - We're also thinking about parallelizing merges with some locking to avoid clashes.
- **Real-World Example**: Imagine a student's grade for CSE016 changes from A to B in Hive at 10:05 AM on April 18, 2025. When we run HIVE.MERGE(SQL), SQL picks up the B if it's the latest, and the log reflects the update.

2. Design of the Operation Log

Purpose and Role:

Our operation log, handled by oplog_manager.py, is like a diary for all CRUD operations. It's stored in files like oplog_hive.log, oplog_mysql.log, and oplog_mongo.log, and it's crucial for auditing and making the merge process work. It lets us track every change across the databases.

Structural Design:

- **Entry Format**: {op_id}, {operation} ({student-ID},{course-id},{grade},{timestamp})\n.
- **Components**:
    - op_id: A sequential number to keep things ordered.
    - operation: What happened—INSERT, READ, UPDATE, or DELETE.
    - student-ID, course-id: The key combo.
    - grade: Optional, left as None for READ or DELETE.
    - timestamp: An ISO-formatted time, defaulting to the current time if not set.

Implementation in oplog_manager.py:

- log_operation: Takes op_id, operation, key_tuple, grade=None, timestamp=None, unpacks the tuple, sets the timestamp, and appends to the log file. It tweaks the format for READ/DELETE vs. INSERT/UPDATE.
- read_log: Parses the log lines into a list of dictionaries for the merge to use.

Design Goals:

- **Persistence**: Append-only files mean we never lose a record.
- **Readability**: It's human-friendly, which is a lifesaver for debugging.
- **Flexibility**: Optional params make it adaptable to different operation types.

Storage and Access:

- Lives in logs/, with a file for each system.
- We use sequential read/write, which works great for single-threaded setups but could get tricky with multiple threads.

Security and Integrity:

- **Risks**: Without locking, multi-threaded access might corrupt the log.
- **Mitigation**: We're considering file locking or switching to a database-backed log down the line.

Scalability Considerations:

- It handles small logs like a champ, but big files might slow us down. We're eyeing indexed storage or pagination for the future. Plus, it's easy to add fields like user IDs if needed.

3. Sample Logs and Test Cases

Sample Logs:

- **oplog_hive.log**:
    - 1, INSERT (SID1033,CSE016,A,2025-04-18T10:00:00)
    - 5, UPDATE (SID1033,CSE016,B,2025-04-18T10:05:00)
    - 17, DELETE (SID1033,CSE016,None,2025-04-18T10:10:00)
- **oplog_mysql.log**:
    - 1, INSERT (SID1310,CSE020,B,2025-04-18T10:01:00)
    - 3, UPDATE (SID1310,CSE020,C,2025-04-18T10:06:00)
    - 7, DELETE (SID1310,CSE020,None,2025-04-18T10:11:00)
- **oplog_mongo.log**:
    - 1, INSERT (SID1403,CSE006,A,2025-04-18T10:02:00)
    - 3, UPDATE (SID1403,CSE006,B,2025-04-18T10:07:00)

Test Cases:

- **Test Case 1: Initial Insert (Hive to SQL)**
    - **Setup**: Insert (SID1033, CSE016, A, 2025-04-18T10:00:00) in Hive.
    - **Action**: HIVE.MERGE(SQL)
    - **Expected**: SQL gets updated with A.
    - **Verification:** Check SQL with a query and peek at oplog_mysql.log.
- **Test Case 2: Update Conflict (Mongo to Hive)**
    - **Setup**: MongoDB has (SID1403, CSE006, A, 2025-04-18T10:02:00), Hive has B at 10:07:00.

  - **Action**: MONGO.MERGE(HIVE)
  - **Expected**: Hive keeps B since it's newer.
  - **Verification**: Query Hive and scan oplog_hive.log.
- **Test Case 3: Delete Propagation (SQL to Mongo)**
  - **Setup**: SQL has (SID1310, CSE020, B, 2025-04-18T10:01:00), MongoDB delete at 10:11:00.
  - **Action**: SQL.MERGE(MONGO)
  - **Expected**: MongoDB deletes if SQL's timestamp is newer.
  - **Verification**: Check MongoDB and oplog_mongo.log.
- **Test Case 4: Duplicate Timestamp**
  - **Setup**: Two INSERTs for (SID1033, CSE016, A, 2025-04-18T10:00:00) in Hive.
  - **Action**: HIVE.MERGE(SQL)
  - **Expected**: SQL gets just one insert.
  - **Verification**: Query SQL and look at oplog_mysql.log.
- **Test Case 5: Grade Mismatch**
  - **Setup**: MongoDB has (SID1403, CSE006, A, 2025-04-18T10:02:00), SQL has B at 10:02:00.
  - **Action**: MONGO.MERGE(SQL)
  - **Expected**: SQL switches to A due to the mismatch.
  - **Verification**: Query SQL and check oplog_mysql.log.
- **Test Cases 6-10: Diverse Scenarios**
  - **Setup**: These use new student-ID and course-id pairs from student_course_grades.csv, mixing INSERT, UPDATE, and DELETE across all databases.
  - **Action**: Run through the merge sequence with updated timestamps.
  - **Expected**: A consistent state across all databases.
  - **Verification**: Cross-check logs and run database queries.
- **Execution**: Just run python3 -m src.main, then dive into the logs and database states to see the results.

4. Behavior and Properties of Merge Operations

Behavior Overview:

The merges take operations from the source log and apply them to the target, guided by our timestamp and grade rules. We've added some debug prints to keep an eye on the process—it's like a live update!

Mathematical Properties of Merge Operations

Let's dive into the mathematical properties that shape our merge function—associativity, commutativity, and idempotency—and how they influence convergence, considering the operations (INSERT, UPDATE, READ, DELETE) each system supports.

- **Associativity**: This property means the order of grouping merges shouldn't matter—(A merge B) merge C should equal A merge (B merge C). For our system, this would hold if merging Hive to SQL then to MongoDB yields the same result as merging Hive to MongoDB then to SQL, assuming the latest timestamp always wins. In practice, our bidirectional sequence (e.g., HIVE.MERGE(SQL), SQL.MERGE(MONGO)) aims for this, but differences in system processing—Hive's partition handling, MongoDB's document model, and MySQL's relational structure—can complicate things. For instance, if a Hive partition creation fails during (Hive to SQL) merge, the next step (SQL to MongoDB) might miss updates, breaking associativity. Our try-except error handling helps, but we'd need retry mechanisms or a final consistency check to ensure this property holds across all scenarios.
- **Commutativity**: This asks if the order of merges matters—A merge B should equal B merge A. In our case, it's challenging due to the sequential log processing. Consider Hive with a DELETE for (SID1033, CSE016) at 10:10 AM and MongoDB with an INSERT at 10:11 AM. Merging Hive to MongoDB first deletes the data, while MongoDB to Hive first inserts it, leading to different outcomes. Our timestamp-based resolution (latest wins) mitigates this, but the order still impacts the result. Commutativity doesn't fully apply unless we enforce a global timestamp ordering or sequence numbers, which we haven't

implemented yet. This non-commutative behavior suggests we might need a centralized sync point in the future.

- **Idempotency**: This is a strong point for us! It means repeating the same merge with the same input doesn't change the result after the first run. In Test Case 4, merging Hive to SQL with two identical INSERTs for (SID1033, CSE016, A, 10:00 AM) results in a single update, proving idempotency. Our logic skips older timestamps unless grades differ, and Hive's partition checks, MongoDB's unique indexes, and MySQL's ON DUPLICATE KEY UPDATE support this. It's a stability anchor, allowing us to rerun merges safely, especially after errors.

Reflecting on Convergence

Convergence is about all systems reaching the same data state over time, and these properties play a big role given our operation types.

- **How Convergence Happens**: Our bidirectional merge cycle propagates the latest operations. If Hive inserts (SID1403, CSE006, A) at 10:02 AM and MongoDB updates it to B at 10:07 AM, running the full sequence aligns all databases on B. Test Case 3 shows a DELETE from SQL syncing to MongoDB if the timestamp is newer. Convergence depends on completing the cycle, and our fault tolerance (logging errors and continuing) supports this, though delays (e.g., Hive's HDFS latency) can slow it down. A reconciliation step could catch any missed updates.
- **Impact of Operation Types**:
    - **INSERT**: Seeds new data, driving initial convergence. Duplicates are handled by indexes, ensuring smooth spread.
    - **UPDATE**: Refines data, overriding older values with newer grades, pushing systems toward alignment.
    - **DELETE**: Removes data, but conflicts arise. If SQL deletes at 10:11 AM and MongoDB inserts at 10:12 AM, the delete only applies if its timestamp wins, affecting convergence timing.
    - **READ**: Logged for auditing but doesn't merge, so it supports verification rather than convergence.
- **Property Influence on Convergence**:

- **Associativity**: If associative, grouped merges (e.g., (Hive to SQL) to MongoDB) would converge the same way. Partial failures suggest we need retries to maintain this.
- **Commutativity**: Non-commutativity means merge order impacts the path to convergence. A global ordering could help, but it's complex.
- **Idempotency**: This boosts convergence by allowing repeated merges to stabilize data, especially after errors, as seen in Test Case 5's grade mismatch resolution.

Performance:

It depends on log size and query speed—Hive's HDFS ops can be the slowest. Sequential execution keeps races at bay, but we're exploring ways to speed it up.

Fault Tolerance:

Our try-except blocks catch errors like missing partitions, logging them and moving on. It's resilient!

Scalability:

It scales with log size, and indexing helps a lot. MySQL's keys are a big win here.

Limitations:

No cross-database transactions mean partial updates are a risk. Identical timestamps lean on grade checks, which can be shaky.

Future Adjustments:

- Adding sequence numbers could break timestamp ties.
- Batch merges might boost performance.
- Parallel processing with locking could speed convergence, though it needs careful design.

Observed Behavior:

Timestamps are ticking up nicely (e.g., 10:00:00 to 10:10:00 on April 18, 2025), and skipped operations show our selectivity is working.

5. Lessons Learned and Team Reflections

Working on this project has been a rollercoaster! We've learned so much about handling different database systems and syncing them in real-time. The timestamp logic was a brain teaser at first, but seeing it work across Hive, MongoDB, and MySQL felt like a victory. Jaimin nailed the Hive partition stuff, Abhay's MongoDB indexing saved us from duplicates, and Jainish's MySQL tweaks kept everything tight. The new test cases were a game-changer—writing testcase1.in to testcase9.in forced us to think through every edge case. Next time, we'd start with more tests upfront to catch issues early. We're proud of this, but there's always room to grow—maybe adding a UI or more error alerts in the future!

6. Conclusion

This updated system is a solid step forward in multi-database synchronization. With enhanced merge logic, a robust log design, comprehensive test cases, and a deeper understanding of mathematical properties like associativity, commutativity, and idempotency, we've built something reliable and scalable. The addition of student_course_grades.csv and the new test files has made our validation process rock-solid. We're excited to see where this can go—perhaps integrating more databases, optimizing for larger datasets, or testing convergence with mock failures.

Thanks for checking out our work!