A Three-Node Implementation of a MySQL Distributed Database System

Andres Paolo Dalisay[1], Timothy Vincent Pinawin[2], Antoinne Bryce Salvador[3], Matthew Jericho Sy[4]
De La Salle University
[1]andres_dalisay@dlsu.edu.ph, [2]timothy_vincent_pinawin@dlsu.edu.ph, [3]antoinne_salvador@dlsu.edu.ph, [4]matthew_sy@dlsu.edu.ph,

ABSTRACT
In this paper, we aim to showcase the process of creating a web application that follows a three-node MYSQL distributed database system. We discuss how we extracted the raw data from the IMDB dataset and store it in their respective nodes. We also discuss how we implemented concurrency in our system. This enables concurrent access to the data even if we are accessing it from different nodes and web servers. Throughout the whole project, we also maintained consistency through our replication strategy. The use of log tables proved to be efficient in terms of managing the consistency of the whole distributed database system. Handling global failures was also managed through the use of log tables that were able to guide the system in syncing with other nodes. This project has demonstrated the basic knowledge needed in order to have a distributed database system that supports concurrency, consistency, and recovery.

Keywords
MySQL, Distributed Database, Concurrency, Consistency, Global Failure and Recovery.

1. Introduction
As time progresses, data is continuously created and manipulated. A single database would not be efficient enough to process multiple users and the problems that it may face such as concurrency and system failures. Hence, there is a need for a database that can handle this kind of situation. A distributed database with a concurrency and recovery system helps in mitigating these problems through replication and recovery strategies.

According to Öszu & Valduriez (1991) [2], a distributed database is a collection of multiple interrelated databases distributed over a network. Distributed databases are designed to store data in multiple locations and transactions are executed in parallel across the nodes.

In this project, we attempt to simulate a real-world scenario where a distributed database is used. Using the IMDB dataset, we implemented a distributed database system using three nodes. The first node is the central node which stores all of the available movies. The second node is used to store movies released before 1980. The last node stores the movies released after 1979.

Building distributed databases helps with issues of concurrency and failure and recovery. As evidenced by Berstein & Goodman (1981) [1], distributed databases can improve concurrency control by allowing multiple transactions to be executed simultaneously on different nodes, reducing the contention of resources.

2. Distributed Database Design
According to the project specifications, our distributed database required 3 nodes: Node 1 having all movie records regardless of year, Node 2 having all movie records with years before 1980, and Node 3 having all movie records with years on or after 1980. Other than the fragmented rows, the tables containing the movie records on all three nodes should be the same. This implementation is an example of a homogeneous database, which is a database system that has a uniform architecture and a consistent set of data and structures. In other words, all of the nodes in a distributed database are of the same type and use the same DBMS.

Because of the way the records are split between the nodes, the tables are an example of horizontal fragmentation. The year column determines what node shall the movie record be inserted into. Aside from being one of the requirements, horizontal fragmentation is beneficial in this situation as it allows parallel processing. This means that multiple queries can run on a certain node as efficiently as possible.

In order to prepare the sample data, we used Apache NiFi in order to perform ETL operations on the original imdb_ijs database. The database was condensed into a single table which contained the id, title, year, genre, director, and actor columns. Only movie records with non-null values were loaded into the new movies table. The new table and its values were then exported and set up as the movies table for the central node, or Node 1. Almost 173,000 records are in the Node 1 `movies` table.

Nodes 2 and 3 were to have all movies before 1980 and on or after 1980 respectively. In order to get the horizontally fragmented tables based on these year ranges, a simple SELECT query for both conditions were made, and the resulting tables were exported as .sql files. They were then imported into Nodes 2 and 3.

Figure 1 shows the nodes and their configuration.

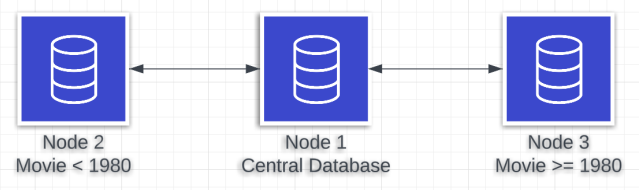

Figure 1. Database Relationship Diagram

Table 1. Example Data in Node 2

year	id	title	genre	director	actor
1979	1152	Jaws	Horror	Steve Spielberg	Roy Schieder
1972	1812	The Godfather	Drama	Francis Ford Coppola	Marlon Brando
1968	1566	2001: A Space Odyssey	Sci-Fi	Stanley Kubrik	Keir Dullea

$$movie_1 = \sigma_{year < 1980}$$

Table 2. Example Data in Node 3

year	id	title	genre	director	actor
1993	12314	Jurassic Park	Action	Steve Spielberg	Sam Niell
2012	12442	Avengers	Action	Joss Whedon	Chris Evans
2022	13244	The Batman	Action	Matt Reeves	Robert Pattinson

$$movie_2 = \sigma_{year >= 1980}$$

In terms of replication, the setup that we built is a partial replication ~~setu~~p. Node 2 and Node 3 only contain subsets of the data in Node 1. The main replication system, together with the update strategy, is further explained in the Concurrency Control and Consistency section.

As mentioned by Silberschatz et al. (2019) [4], data replication is important as it exhibits availability, parallelism, and reduced data transfer. Availability comes in when a failure of a node occurs. The whole system will not go down just because of a single node failure. Parallelism is when queries can be simultaneously processed by our database servers. Lastly, reduced data transfer occurs since a site has a partial local copy of the database.

With all the configurations and coding that we've done, this implementation can be considered a Master-Master or Multimaster setup. The updates happen on nodes that the web server is currently running on. The responsibility for synchronization falls on the replication system.

3. Concurrency Control and Consistency

A Global Concurrency Control is a process that checks multiple transactions if it can access and modify data concurrently without violating consistency and integrity in a distributed database system. As discussed by Öszu & Valduriez (2020) [3] in their book, concurrency control mechanisms are used to ensure that multiple transactions executing concurrently in a distributed database do not interfere with one another and that the database remains in a constant state. However, global concurrency control verifies that the transactions executed at different nodes of the distributed database execute in a mutually consistent manner.

To maintain consistency between all three nodes, each node's databases were set up with log tables. Nodes 2 and 3 have one log table each, while Node 1 has two, corresponding to log records done on movie records before 1980 and on or after 1980. Each log table is connected to MySQL triggers which store records into them based on certain actions done to their respective `movies` tables. For example, if an INSERT query is executed, the `insert_movie` trigger is executed, which then inserts a record into the log table containing the action (in this case an INSERT), the time the action was done, and the values of the new record such as the name and year of the movie. Triggers were made for INSERT, UPDATE, and DELETE queries. SELECT queries were excluded since they have no bearing on changes in the `movies` table.

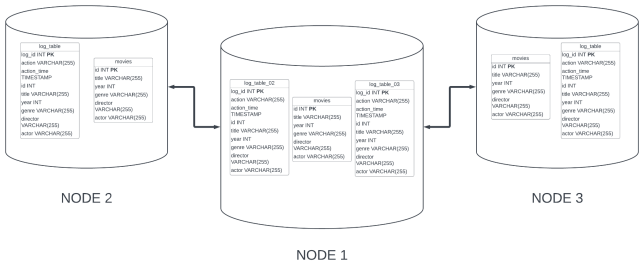

Figure 2. Node database setup with respective log tables.

Each log has a log ID which is used in order to compare the states of each database. For example, Node 1's log table for records before 1980 (`log_table_02`) is synced to Node 2's log table, which should have consistent log records based on the MAX log id. If `log_table_02` has one less ID than Node 2's log table, then that means Node 1 has to be synchronized with Node 2. This synchronization is done with our app-level synchronization functions.

We have two synchronizer functions: `sync_central` and `sync_fragment`. The latter syncs a given fragment node, either Node 2 or Node 3, with Node 1. This is done by querying the maximum log_id values in the given node's log table and the respective Node 1 log table, and checking if there is a difference between them. If it turns out that Node 1 has a higher maximum log_id, then the function will take all the log records from the last log_id on the fragment log table until the latest. Each log record is then parsed into appropriate queries for the fragment node to execute, making its `movies` table consistent with Node 1.

On the other hand, `sync_central` has a similar process but instead checks both Node 2 and Node 3's log tables for new changes and collates them into a list of queries for Node 1 to execute. This list is sorted by each log record's timestamp in order to do each table update chronologically.

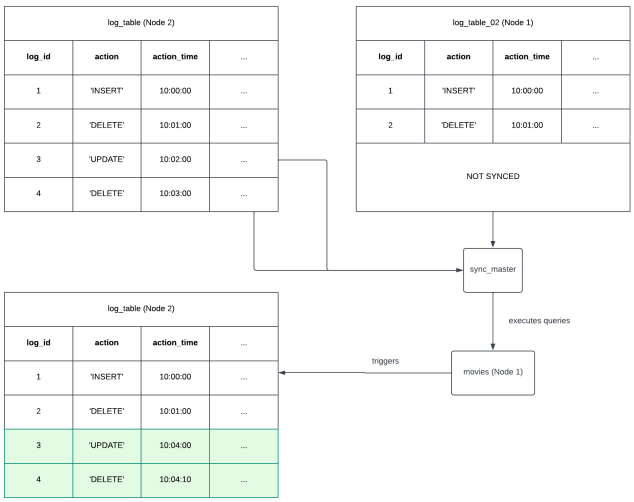

Figure 3. Process flow for sync_master function's Node 2 to Node 1 sync.

~~For testing, we followed the test cases given in the project details~~. We simulated reading a data item by refreshing the web application. Refreshing the web application is considered as a read since we are always displaying the last 200 entries in our home page, sorted by ID in descending order. Editing and deleting ~~a data item is done by clicking on the edit button beside each item, wherein th~~e user will be redirected to a page where they can edit the title, year, genre, director, and actor or delete the item from the database itself.

In order to facilitate our testing, we added a 5 second sleep delay in our ~~code for~~ transactions in order to simulate the transaction not being committed yet. Another transaction is run when the delay is active in order to simulate concurrency.

For case #1, "concurrent transactions in two or more nodes are reading the same data item." We simulated this by having Node 2 and Node 3 ~~clicking the edit button~~ of the same data item. Both users in Node 2 and Node 3 should see the same and consistent version of the data item.

For case #2, "at least one transaction in the three nodes is writing (update / delete) and the other concurrent transactions are reading the same data item.". We simulated this by editing the director, title, and actor field of a data item in Node 1. The other nodes will refresh the page when Node 1 is executing the transaction. Users in Node 2 and Node 3 should not see the edited fields when the transaction in Node 1 is not yet finished.

For case #3, "concurrent transactions in two or more nodes are writing (update / delete) the same data item.". We simulated this by using the same steps in Case #1, however, both users in Node 2 and Node 3 will edit the same data field. The data field that was experimented on was the director field and both users will submit the edited data item at the same time. We expected that it will only commit the user that was last in committing based on the time stamp.

After conducting the experiments using the test cases, we found that the distributed database system we implemented was able to handle concurrency and maintain data consistency.

For Case #1, both Node 2 and Node 3 were able to read the same data item consistently, showing that the system can handle concurrent reads.

For Case #2, Node 2 and Node 3 were not able to see the changes made by Node 1 until the transaction was finished, which shows that the system can handle concurrent reads and writes while maintaining data consistency. However, when the Isolation level is set to READ UNCOMMITTED, both Node 2 and Node 3 see the changes made by Node 1 immediately even if the transaction was not finished. This is due to how the Isolation level at READ UNCOMMITTED allows transactions to read uncommitted data modifications made by other transactions. In other words, there is no locking or blocking of the data item being edited in Node 1.

For Case #3, the system was able to handle concurrent writes to the same value in a record by committing changes one at a time. However, we also tried to modifying the same record but for different columns. In reality, both modifications should be able to push through with the update. However, our code implementation makes it so that all the columns are set again when an UPDATE query is sent. It does not detect whether a change was only made to certain columns. E.g., if only the title was edited in a transaction, the app will still take the old values of the other columns and use it in the query:

```
UPDATE FROM movies (title, year, …) VALUES
(data.title, data.year, …) WHERE id
=data.id;
```

This makes it so that if another concurrent update transaction were to run that only edits the artist field, it would still run the same query which still has the unchanged title. As a result, the first transaction would not necessarily take effect.

In order for the data to be consistent, we also take into consideration when a user wanted to change the year property of a movie record. There is a possibility that there will be a transfer of records from one node to another. An example of this is shown below. In the movie Jaws, the original year is 1975 and this record is stored in Node 1 and in Node 2.

Table 3. Example of Editing Year Property

year	id	title	genre	director	actor
1979	115	Jaws	Horror	Steve Spielberg	Roy Schieder

Table 4. After Editing Year Property

year	id	title	genre	director	actor
1989	115	Jaws	Horror	Steve Spielberg	Roy Schieder

When someone changes the year into 1980 and above, our system can handle the change in nodes. To be specific, the process for this is as follows:

First, we run a delete on the node that we are currently accessing. Then, we insert this deleted record into the correct node. It will then be replicated either to Node 1 or to Node 3, depending on the node that we are accessing. Even if we are in Node 1, we still need to perform the delete and insert operations as we need to use the different log tables for the mentioned operations.

4. Global Failure and Recovery

In a distributed database system, recovery plays an important role in maintaining the consistency of the whole system. It involves restoring a system to a consistent state after a failure, such as a server or node being shut off. Without recovery strategies, data may become inconsistent between databases and may lead to missing records and inaccurate values.

Our recovery strategy involves the use of the log tables and app-level synchronizer functions.

The synchronizer functions are called after every transaction to ensure that appropriate changes are replicated onto the involved nodes. In addition, the web app also calls these synchronizer functions every 1 second in order to keep the databases consistent even when no transactions are happening. It also serves as an automatic recovery system for downed nodes as once a node recovers, it immediately tries to sync its data with other nodes.

In order to test our recovery system, we followed the given test cases specified by the project details. We simulated node failures by stopping both the web server and MySQL database on the "failed" nodes.

For case #1, "the central node is unavailable during the execution of a transaction and then eventually comes back online," we turned off Node 1 and added, edited, and deleted a movie record on Node 2 in 3 separate iterations. These transactions should push through to Node 2's database. After Node 1 recovers, the transactions made on Node 2 should be replicated on Node 1.

For case #2, "Node 2 or Node 3 is unavailable during the execution of a transaction and then eventually comes back online," the same process as case #1 was done, except Node 2 was turned off and transactions were done to Node 1. Node 2 should have the new transactions made on Node 1 after it recovers.

For cases #3 and #4, "failure in writing to the central node when attempting to replicate the transaction from Node 1 or Node 2," and "failure in writing to Node 2 or Node 3 when attempting to replicate the transaction from the central node," similar steps were taken where the designated "failed" node is turned off and transactions are done on the other nodes. However, these cases look at what the "online" nodes should be doing when another node is down. In our case, the online nodes should repeatedly try to sync to the downed node until it recovers.

Through these test cases, we found that our system was resilient and kept records consistent despite node failures. Because of our log table-based replication system, any updates done to the tables were taken into account and were replicated by other nodes after recovery. Cases #1 and #2 all resulted in the downed nodes being consistent with the other nodes after its recovery. Cases #3 and #4 showed that each node in our system continually tries to establish a synchronization connection between all nodes, downed or not, which makes post-recovery replication updates practically seamless. The full test script and results are included as a supplement to this report.

5. Discussion

Throughout the duration of this project, we were able to simulate a real-world application of a distributed database system. We successfully explored and implemented the concepts of consistency, concurrency, and failure and recovery.

We saw how important it is to maintain a consistent distributed database across all nodes. Without consistency, the system becomes unreliable and will lead to different problems. One example of how a database can become inconsistent is when a node failure occurs and there is no recovery strategy in place to update any changes made on other nodes. This will result in the nodes having inconsistent copies of the data.

In testing concurrency and transactions in our distributed database, we realized that the CRUD functions of our web-app only use one query each, which may have affected the results of our tests. In some cases, multiple queries in a single transaction are needed in order to demonstrate certain concurrency problems. This is why most of our test cases more or less pass.

The concept of failure and recovery highlighted the importance of having multiple nodes and its replication system. Our replication system relies on our log tables per node, with the exception of Node 1, which has two log tables that are for supporting consistency with Node 2 and Node 3. As mentioned in the previous chapter, we did not encounter any issues when a node goes down. Any updates made to the system will be replicated once the node goes back up. With our recovery system, we realized how crucial it is for big companies to guarantee that their system would not crash. They must have a robust recovery and replication system.

In connection to this, another realization is distributed database systems have a greater advantage in terms of security. If a centralized database is compromised, data integrity is not anymore secured. In the case of a distributed database, an attack on one node would not be that impactful to the system. Database administrators may intentionally turn off this node and everything will seem normal.

We also realized how distributed database systems can handle scalability well, specifically, horizontal scaling. According to AWS (n.d.) [5], horizontal scaling is done by adding more computers to the system. In our case, it is when we add more nodes to our system. This will benefit the performance of our system by equally distributing the load of each node.

6. Conclusion

For this project, we built a distributed database system for a condensed imdb_ijs dataset. This was made to simulate concurrency control and recovery strategies that are commonly implemented in real-world databases and applications.

We implemented concurrency control through the use of transactions and isolation levels. Through the given test cases, we were able to observe that our system seemed to run concurrent transactions in a serialized way, regardless of isolation level. This behavior may be due to the fact that most of our transactions only execute one query at a time.

To maintain consistency as well as implement a recovery strategy, we implemented a log table-based replication system for our database and web-app. This ensures that data are correctly replicated into their corresponding nodes based on their year values. The log table was also used as a way to record any new changes that may need to be done in order to keep the nodes consistent. Automatic synchronization was also implemented for the purpose of automatic recovery after node failures.

Testing our system for global failures and recovery, we found that it sufficiently handled downed nodes and was able to update recovered nodes when records were inserted or updated in other nodes.

Overall, this project was able to give us a more hands-on approach to learning about distributed database systems and the mechanisms that keep them from being inconsistent and inaccurate.

7. References

[1] Bernstein, P. A., & Goodman, N. A. (1981). Concurrency Control in Distributed Database Systems. ACM Computing Surveys, 13(2), 185–221. https://doi.org/10.1145/356842.356846

[2] Özsu, M. T., & Valduriez, P. (1991). Distributed database systems: where are we now? IEEE Computer, 24(8), 68–78. https://doi.org/10.1109/2.84879

[3] Özsu, M. T., & Valduriez, P. (2019). Principles of Distributed Database Systems. Springer Nature.

[4] Korth, H. F., Silberschatz, A., & Sudarshan, S. (2019). Database System Concepts. McGraw-Hill Education.

[5] Horizontal scaling. (n.d.). AWS Well-Architected Framework. Retrieved April 17, 2023, from https://wa.aws.amazon.com/wat.concept.horizontal-scaling.en.html