A Technical Report on Transaction Management in a Distributed Database Management System

Sealtiel B. Dy
sealtiel_dy@dlsu.edu.ph
De La Salle University
Manila, Philippines

Kyle Carlo C. Lasala
kyle_lasala@dlsu.edu.ph
De La Salle University
Manila, Philippines

Maria Monica Manlises
maria_monica_manlises@dlsu.edu.ph
De La Salle University
Manila, Philippines

Camron Evan C. Ong
camron_evan_ong@dlsu.edu.ph
De La Salle University
Manila, Philippines

ABSTRACT

Distributed databases play a crucial role in enabling companies to efficiently and effectively provide data to their diverse user base. These require certain standards to ensure consistent data throughout the nodes of the database. This project attempts to recreate some of these standards on a smaller-scale distributed database of three nodes containing appointment data from SeriousMD. The database was subject to concurrent transactions from multiple users and crash simulation via a web application. The implementations to handle these cases were successful and effective, keeping the data in the database consistent throughout the tests.

KEYWORDS

distributed database, transactions, concurrency, data replication, data recovery

1 INTRODUCTION

With the ever-increasing data usage requirements of large companies, solutions that are more scalable and flexible are necessary. Distributed databases provide a range of benefits, making them a popular choice for enterprises seeking efficient data storage and management solutions. Through the implementation of this distributed system, the database is not constrained to a single computer or server. Instead, numerous devices, typically located in separate physical locations, store it, offering several enhancements. For example, the performance of transaction handling can be enhanced by optimizing the routing of traffic to reduce latency. This also enhances the levels of reliability and availability. While individual nodes may experience failures, the data remains accessible through other nodes throughout the network architecture. Furthermore, the process of expanding the database is simplified. Scaling up can be accomplished by straightforwardly incorporating additional nodes, which is a more efficient approach than vertically increasing the capacity of a single server. [4]

In this project, a distributed database with three nodes was constructed and developed, containing various appointment data from SeriousMD. Several strategies were implemented to simulate global concurrency, overall data consistency, and data recovery strategies for accurate data handling. Additionally, an accompanying web application, which represents a smaller version of a real-life tracker was also created for users to perform CRUD operations.

2 DISTRIBUTED DATABASE DESIGN

Data from the appointments section of SeriousMD's dataset served as the basis for the project. The cleaned version of around 300,000 data entries from the prior MCO was repurposed. Subsequently, the fields of the other tables were merged to incorporate additional pertinent data into the distributed database. One particular example is the clinics table's city and area fields, as these variables are essential for dividing the appointment data into smaller parts. It should be noted that this process would be a horizontal fragmentation, resulting in sub-tables or fragments that have all of the fields from the original table. Additionally, some fields, such as the `pxid` and `doctorid`, were eliminated as they were considered unnecessary. Essentially, in the original schema, these two were mainly utilized as foreign key references. However, as the new schema consists of a single denormalized table, the additional references were no longer necessary, and hence they were removed. The schema of the merged table is shown in Figure 1.

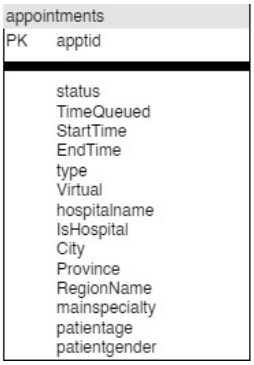

Figure 1: Database Model

This table has been integrated into a homogeneous distributed database featuring partial replication. The schema remains consistent across the three nodes/servers, all of which are hosted on virtual machines within the DLSU CCS Cloud infrastructure. Node 1, the central node, contains the entire table's data. Node 2 contains appointments whose region or location falls inside Luzon. Finally, Node 3 contains the appointments within Visayas and Mindanao.

A web application was created using Node.js to handle users' transactions to the distributed database. Through this, users can

perform CRUD operations, namely create, read, update, and delete appointment information as necessary. If all nodes are on, transactions are directed towards the central node. If the central node is disabled, transactions are redirected to the respective partitions. Node failure can also be simulated through the web app, as there are options to 'turn off' or disable nodes. It is important to note however that to preserve data transparency, this should be hidden to the user and is only available for simulation purposes. Additionally, a secondary web server was also created using Node.js to handle extra back-end operations, specifically concurrency replication and recovery procedures. The overall structure setup is shown in Figure 2. Note that each client can "connect" to each database in the distributed database, but this directing is handled by the web server itself.

Ensuring data consistency and availability in a distributed database system with concurrent user access requires robust concurrency control and recovery mechanisms. The system for this project utilizes a master-slave architecture to coordinate updates from a central master node to replicas on various nodes, ensuring consistency maintenance. Write transactions are carefully and systematically controlled, with locks implemented, operations executed, and transactions logged before being asynchronously propagated to guarantee the integrity of the data. A recovery manager was created to supervise the recovery process by coordinating the replication of any missing transactions when a node is turned on after being turned off. This ensures that data consistency and system functionality are quickly restored in the event of failures or problems. By adopting a proactive approach, the amount of time that the system is not operational is reduced and the reliability of the system is improved.

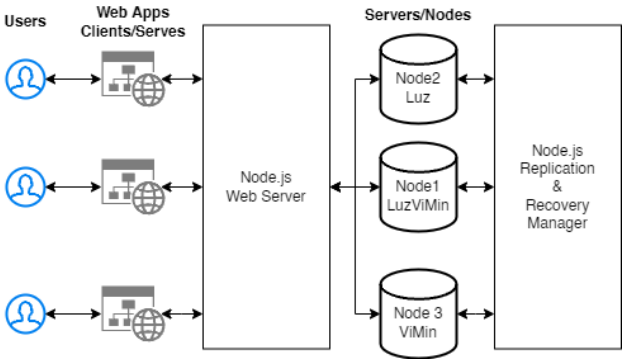

Figure 2: Client - Server Structure

3 CONCURRENCY CONTROL AND CONSISTENCY

Multiple users can access and query the web application at a time, hence, a robust implementation of concurrency control is required to maintain consistency in the distributed database.

A master-slave setup was created to satisfy the needs of the project. This is because the master-slave is generally simple to construct and implement, while also ensuring data consistency among the replicas in the various nodes. In this setup, the master can take both read (SELECT) and write (INSERT, UPDATE, DELETE) transactions while the slaves can only execute read transactions. As such, when a write transaction is to be executed, it is first done so on the master node before being replicated to the slave nodes. Being the fully replicated node, Node 1 is set up as the master node for this project. Note that a master-master setup would also work but is much more complicated to set up. For instance, conflict resolution between concurrent updates is required to prevent the nodes from reaching an inconsistent state. Inconsistency may also be reached if the databases lose contact with each other, thus proceeding to operate individually. [6]

When a write transaction (INSERT, UPDATE, DELETE) is sent to the master, the table will first be locked, then the operation will be performed, logged, and finally, the lock will be released. Should the operation fail, a ROLLBACK operation will be executed. This system allows the node to maintain data integrity within itself as users perform their various transactions. Note that this sort of implementation is akin to a Serializable isolation level. Additionally, the logging is implemented for the data replication itself, for checking that the replicated updates are performed correctly, and for use when failures in nodes occur. Transaction logs, generated alongside each transaction, document the modifications made to the database and its table rows. These logs store the updated data alongside additional pertinent details, including an ID counter and the node (aside from the master node) that will be affected.

The replication is done asynchronously, meaning the update should first be written to the master node before the update is propagated to the rest of the nodes. A separate web server, deemed as the recovery manager, checks the logs of each node. If the IDs of the latest log do not match, replication is then triggered using the same logs. The processes of replication and recovery will be further discussed in the next section.

In a sample scenario where a new appointment is inserted into the master/central node, the node is first locked while the INSERT query is being performed. When the query is complete, the table is unlocked. Using the generated logs, replication is then triggered asynchronously on the other nodes depending on the location of the appointment. Note that locking and unlocking will also be done on the node where replication will occur. For instance, if an appointment in Luzon is created, then Node 2, the node containing the Luzon fragment, will receive the same update to said appointment. On the other hand, if the appointment is in Visayas or Mindanao, then locking and replicating will be triggered on Node 3.

For this project, if the master node is down, since Node 2 and Node 3 work independently of one another, both of them are promoted to masters as write transactions can then be executed on either node before being propagated back to Node 1 once it is up again. For example, if a new appointment in Luzon is to be added to the database when Node 1 is down, the query will be sent to Node 2. Node 2 will first be locked as the INSERT query is being performed and will be unlocked again afterward. Logging will then be done and once Node 1 is available, replication from Node 2 to Node 1 will occur. Figure 3 shows the structure of the setup before asynchronous replication.

Two clients, Client 1 and Client 2, were connected to the web application to test that the implemented methods worked properly during concurrent transactions. Client 1 first executes a transaction,

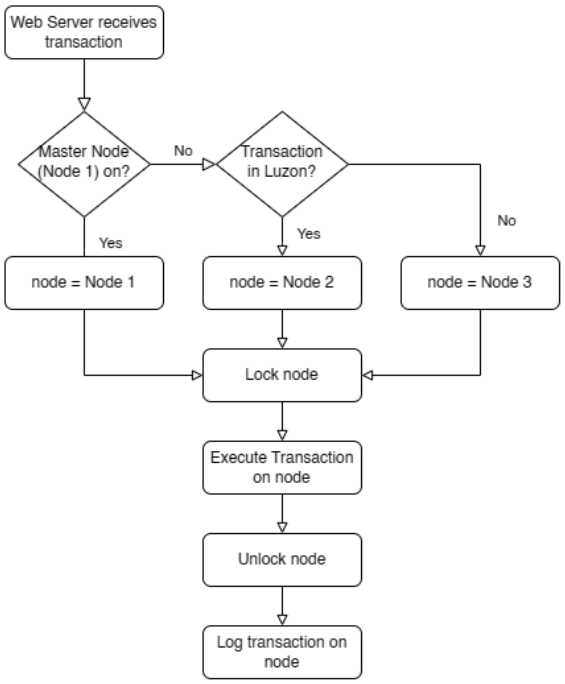

Figure 3: Master-Slave Structure

followed by client 2. A test was considered successful or correct if both clients received consistent information after their transactions. Table 1 shows a summary of the test results, with `action x` referring to a transaction on some data item in the database. Note that each test was conducted at least thrice.

Table 1: Concurrency Test Results

Case	Client 1	Client 2	Pass/Fail
1	Read X	Read X	Pass
2	Update X	Read X	Pass
	Delete X	Read X	Pass
3	Update X	Update X	Pass
	Delete X	Delete X	Pass

All test cases passed, implying that the implemented global concurrency control and replication are working as intended. This success is due to a mixture of details in implementation, such as simulating the Serializable isolation level using manual locks [2] and implementing a robust logging and replication system to handle write transactions on the other nodes.

4 GLOBAL FAILURE AND RECOVERY

Any node in a distributed database can fail or go offline without warning at any given point in time. This can result in data discrepancies between nodes, hence why data recovery is a crucial part of distributed databases. The DBMS needs a way to restore the consistency of the data in each node in case a failure, error, or crash occurs. Hence, recovery strategies were also constructed and implemented in this project.

A second, separate web server was created for handling recovery procedures and acts as the recovery manager. Each node has its own transaction log saved in the database which is mainly what the recovery manager checks to ensure consistency. The primary key (PK) of the `logs` table in each node is a tuple consisting of the `id` and `node`. The `node` refers to the fragmented node affected by the transaction and the `id` only increments based on the node involved. That is, when the master node is on, the first transaction involved with an appointment in Luzon will be saved in the master node's `logs` table with the following PK: `{id: 1, node: node_2}`. If another transaction with an appointment in Luzon is made after that, its `id` will then be 2. However, afterward, if a new transaction with an appointment in Visayas is executed, the `id` of that will still be 1 with a `node` of `node_3`. This is because the `id` of transactions involved in Luzon are independent of those in Visayas and Mindanao.

To ensure consistency, every 1250 milliseconds, the recovery function gets the ID of the latest log in each database. As seen in Figure 4, If both Node 1 and Node 2 are functioning properly, the ID of the latest transaction where `node = node_2` logged in Node 1 should be the same as the ID of the latest transaction in Node 2. If it is not the same, then replication will occur where the first missing transaction is copied to the node that is not up-to-date. This process is similar for the Visayas and Mindanao located appointments in Node 1 and Node 3. This allows the database to remain consistent even when nodes fail. Additionally, the system also covers transaction failure by employing a ROLLBACK operation whenever a query fails.

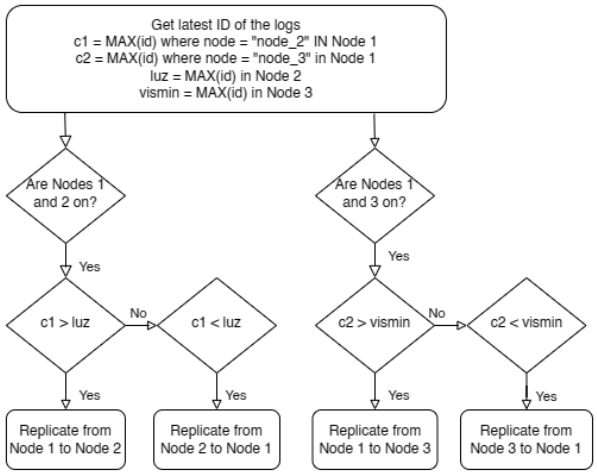

Figure 4: Recovery Algorithm

To illustrate once again, as seen in Figure 3 in scenarios where the user selects Node 1 (master node) but it is hidden or off, any queries will be redirected to the other nodes. This redirection will depend on the location of the appointment, again being either Luzon (Node 2) or Visayas and Mindanao (Node 3). All of the operations

done at this time will be logged. When Node 1 is online again, the second web server will begin replicating all the relevant logs that need to be replicated, specifically those involving write operations. In scenarios where either Node 2 or Node 3 is offline, then the transaction will simply be executed in Node 1 as it is the master node. Similarly, once the offline node comes online again, then the relevant write transactions will be replicated onto it.

To test the cases listed in the specifications, nodes had to be disabled. Cases 1 and 2 deal with basic transactions in scenarios where nodes are down but users are still sending queries. A distributed database should be able to respond to users' requests, hence, a test is considered passed if A) the transaction is handled successfully, and B) the data is consistent. A sample test is simulated by first disabling Node 1. A READ transaction is sent by a client, which should be responded to by one of the other nodes. Node 1 is then re-enabled and queried the same way, and the results from the reads should be the same. A summary of the results of these test cases is shown in Table 2. Note again that each test was conducted at least thrice.

Table 2: Recovery from Disabled Node

Case	Action	Performed In	Disabled	Pass/Fail
1	Read X	Node 2	Node 1	Pass
	Read X	Node 3	Node 1	Pass
2	Read X	Node 1	Node 2	Pass
	Read X	Node 1	Node 3	Pass

The next set of cases deals with failed replications. These can occur when a node is down, but the master sends a replication, or when an executing query is interrupted, possibly due to the server crashing. This was simulated by sending an UPDATE transaction while the specified nodes were unavailable. The node will then be enabled, to which replication should be triggered. Given this, a test is passed if the `Fail` node can replicate the transaction successfully after it is available again. A summary of the tests and results is shown in Table 3. Each test case was ran at least thrice.

Table 3: Recovery from Failure

Case	Action	Replicated From	Fail	Pass/Fail
3	Update X	Node 2	Node 1	Pass
	Update X	Node 3	Node 1	Pass
4	Update X	Node 1	Node 2	Pass
	Update X	Node 1	Node 3	Pass

All set-ups and tests for global failure and recovery passed and are working as intended. Both READ queries in cases 1 and 2 returned the same corresponding appointment data, and the database was successful in recovering and replicating the re-enabled nodes in cases 3 and 4.

5 DISCUSSION

Creating and setting up a fully functioning distributed database requires a lot more time and effort in comparison to a single system database. This project deploys three such nodes on virtual machines via the DLSU CCS Cloud. On one hand, MySQL has built-in functionalities for automated replication via the NDB cluster [2], but applying that solution abstracts the process, defeating the purpose of the project. Additionally, those built-in functionalities cannot simulate partial replication on three nodes. Hence, facilities to handle concurrency, replication, and recovery procedures on each node were created from scratch to fit the requirements of the specifications and the distributed database.

Triggers are a type of procedure called by an event happening, such as a transaction being accomplished, and generating logs as a result. These logs were important in the replication and recovery strategies that were implemented in this distributed database project. A row-based logging approach was used, storing the new data, an ID, the execution time, statement or action, and finally the node involved. All of this stored data made replicating and recovery much simpler because the log itself already included all the necessary information. [2]

Of note, a properly distributed database should also hide details such as how the data is fragmented, and the number of nodes, and overall should provide a seamless feel to the users [1]. In applying this to the distributed database, the primary web server directs all clients to the central/master node if said node is available or online. In cases where the central node is down, transactions are instead directed to the two slave nodes depending on the location of the query. That said, for simulation purposes, anyone accessing the recovery web server can also disable or turn off a node. In a real-life scenario or deployment, perhaps only database administrators who are testing the database would see these options. On the other hand, the normal user would only be able to see a base web application delivering the data, as if it were stored in a single database.

The distributed database system was tested on the cases listed in the specifications. It performed outstandingly and passed all cases successfully. For concurrency control, the system provided consistent information as users were accessing and updating the data concurrently. It was also able to respond to users' usage and queries, recovering and keeping data consistent even through the simulated node outages and crashes. Given this, the system is successful within the bounds of the specifications of the project. The combination of locking, unlocking, and logging all contributed to the capabilities of the distributed database.

A realization that came from this project is that data replication is necessary because having a fragment or small copy of the database makes CRUD operations much easier. If there was no fully replicated node, then the master-slave structure would not work as well. Creating reports for the whole database would also involve additional operations that merge information from fragmented nodes. However, pure data replication can also be costly as all nodes will have to be updated every time a new transaction is executed. As such, data fragmentation is also important and it can also reduce the total storage needed for the database. [7] This project shows that for a database with three nodes, having 1 fully replicated node and 2 partially replicated nodes is enough for all operations to continue even when 1 node is shut off.

Other than that, data replication itself also benefits users because it keeps the data of each fragment consistent with each other [5]. This means that in cases where a node goes offline, another node can provide the data. In cases where the node fails to replicate a

transaction, this failure can be noted and reran. All of this works to keep data in the distributed database to be updated and accurate, thus allowing users to still receive data even in cases where nodes fail. Given this, a robust replication system, involving logs and even checkpoints is necessary to support proper node recovery. Along with these, a recovery manager is also needed to roll back failed replications for atomicity or redo the transaction's updates to maintain durability. In larger-scale projects, a backup mechanism can also be useful in cases wherein data discrepancy is too large. The entire database can be rolled back to this backup which would contain a snapshot of a recent consistent state.

6 CONCLUSION

A small-scale distributed database with three data nodes was developed to bring appointment information to its relevant users. A central node served as the master wherein all transactions would be directed to it, while two other nodes containing partitions of the data acted as backups. Locks were used to implement global concurrency and mechanisms for replication and recovery were also designed using logs. All of these were tested using automated testing simulating multiple users accessing the database and node crashing.

Through testing, the developed mechanisms were effective in handling concurrency and recovery within the scope of the specifications. Locks successfully kept concurrency as they simulated higher isolation levels, namely Serializable. Node disabling allowed testing for usability in cases of node unavailability and crashing, and both the logging and the recovery manager were successful in keeping the data consistent via the developed replication system.

That said, perhaps in a larger-scale project, these implementations may fall short in certain aspects. This project was, admittedly, very small scale and helped to provide foundational concepts in distributed database development and management, but there are still multitudes of other topics that were out of scope. Some examples include deadlocks, to which there are multiple types of [3], full outages, and even poor performance due to long query times. Hence, further improvements can be made by doing more research and exploring other techniques, such as synchronous replication, master-master implementations, and full replication depending on the intended scale and required operational time of the database.

REFERENCES

[1] Devi Indriani Ario Bijak Prabawa and Farah Kalsum. 2021. *Transparencies in DBMS.* Binus University. https://sis.binus.ac.id/2021/03/10/transparencies-in-dbms/

[2] David Axmark and Michael Widenius. 2021. *MySQL 8.0 reference manual.* MySQL. http://dev.mysql.com/doc/refman/8.0/en/index.html

[3] Geeks for Geeks. 2021. *Distributed System – Types of Distributed Deadlock.* Geeks for Geeks. https://www.geeksforgeeks.org/distributed-system-types-of-distributed-deadlock/

[4] GeeksforGeeks. 2023. *Advantages of Distributed database.* Retrieved April 10, 2024 from https://www.geeksforgeeks.org/advantages-of-distributed-database

[5] IBM. n.d.. *What is data replication?* IBM. https://www.ibm.com/topics/data-replication

[6] Ben Meehan. 2024. *Master-slave Master-master replication in Databases.* Retrieved April 10, 2024 from https://iq.opengenus.org/master-slave-and-master-master/

[7] Dhiraj Rane and M. P. Dhore. 2016. Overview of Data Replication Strategies in Various Mobile Environment. *IOSR Journal of Computer Engineering* (2016). https://doi.org/10.1109/ICCIKE47802.2019.9004233

7 APPENDIX

7.1 Appendix A: Sample Logs

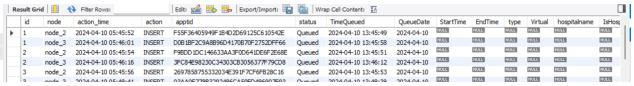

Figure 5: Sample Logs

8 DECLARATIONS

8.1 Declaration of Generative AI in Scientific Writing

During the preparation of this work, the author(s) used ChatGPT and Quillbot in order to write drafts of non-documentation sections of the paper. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication.

8.2 Record of Contribution

All authors contributed to the creation of the technical report. Contributions to the different processes involved in this study are as follows:

- Sealtiel B. Dy - System Development, Test Cases
- Kyle Carlo C. Lasala - System Development, Test Cases
- Maria Monica Manlises - System Development
- Camron Evan C. Ong - System Development