Transaction Management Technical Report

Clarissa Albarracin[1], Amor De Guzman[2], Reina Althea Garcia[3], Miko Santos[4]

[1]clarissa_albarracin@dlsu.edu.ph, [2]amor_deguzman@dlsu.edu.ph, [3]reina_garcia@dlsu.edu.ph, [4]miko_santos@dlsu.edu.ph

ABSTRACT

Having a distributed database system is essential for managing large-scale data efficiently and ensuring high availability and reliability. This paper explores the design and implementation of a distributed database system using a master-slave architecture to handle multiple nodes/servers together. The central master node is responsible for executing both read and write operations, while slave nodes manage read-only queries, with data replication ensuring consistency across the system. Key processes, including transaction logging, data replication, and node failure/data recovery mechanisms, are explained in detail. The web application interface, developed using Node.js, allows users to perform various operations such as creating, reading, updating, deleting, and generating reports on game data. The study provides valuable insights into the development and maintenance of effective distributed database systems.

KEYWORDS

distributed database, master-slave architecture, recovery system, data replication, data fragmentation

1. INTRODUCTION

A distributed database consists of several interconnected nodes or servers, each capable of functioning independently while also being able to replicate data with other nodes for recovery processes. This is one of the solutions for maintaining a scalable and flexible system whenever increasing data usage is expected. The problem that emerges from this topology is that data storage and retrieval at different nodes may raise issues since some topologies may let users perform write operations on a same data point simultaneously, resulting in data anomalies. Further, the development of global failure and recovery imposes challenges through requiring data redundancy and consistency across all servers.

The design of the distributed database consists of a three-node homogenous distributed database with a central node having all the data points from the steam_game dataset, whereas the other two nodes represent partial replicas, which contains games released before 2010 for the second node, and from and beyond 2010 for the third node. A web application was developed for users to interact with the distributed database. Also, the app supports common operation wherein users can add new games, search the game database, update existing game data points, delete existing game records, and view generated reports. Concurrency control is established through the implementation of write locks and global isolation levels. Further, logs are added in each node to maintain and facilitate recovery whenever a particular node crashes.

2. DISTRIBUTED DATABASE DESIGN

This section explains the setup of the distributed database system.

2.1 Database Schema

The dataset used in this project is the **Steam Games dataset**, which was initially provided in MCO1. During MCO1, the dataset required cleaning and fixing due to misalignment issues between the values and columns. We applied the same cleaning processes to this dataset for our current project to ensure consistency and correctness. The cleaned version has around 97,000 rows. Given the complexity of managing foreign keys in a normalized table, the group decided to use a single **denormalized table**. This approach simplifies the data structure by eliminating the need for complex joins and foreign key management, making the dataset easier to work with in the context of a distributed database. The group merged tables from the previous project, removing attributes that required foreign key and bridging tables, such as developers, and publishers. As a result, Figure 1 displays the schema of the final version of the table to be used in this project.

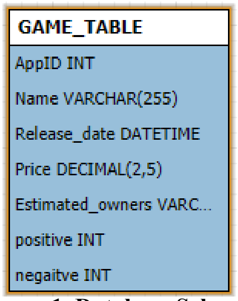


Figure 1. Database Schema

The `GAME_TABLE` is the main table to be used in this project, carefully selected to ensure relevance and efficiency in managing the data. It contains a combination of both quantitative and qualitative attributes that provide a comprehensive overview of each game, from basic identifiers to user feedback metrics. The primary contents of this table is shown in Table 1.

Table 1. Stream Games Dataset

Attributes	Description
`AppID`	A unique identifier assigned to each game.
`Name`	The title of the game.
`Release_date`	The official release date of the game.
`Price`	The cost of the game in the Steam store.
`Estimated_owners`	An estimated number of owners or purchasers of the game.
`positive`	The count of positive reviews received by the game.
`negative`	The count of negative reviews received by the game.

2.2 Distributed Database Topology

For the distributed database system, the table has been integrated into each node/server with partial replication implemented. It maintains consistency by containing the same tables and data across all nodes/servers. This setup is hosted on the DLSU CCS Cloud platform provided for the project. Although each node contains data from the same dataset, they differ in content: Node 1 contains all the Steam games, Node 2 holds games released before 2010, and Node 3 stores games released from 2010 onwards.

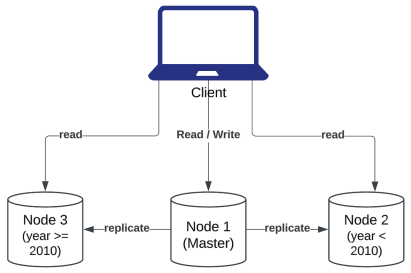


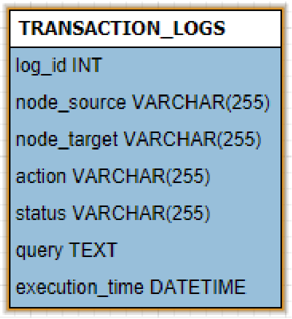
Figure 2. Master-Slave Implementation

To facilitate transactions between the user and the nodes/servers, the group developed a web application using Node.js. This web application allows users to perform a variety of operations, such as creating or inserting new games, reading game information through a search function, updating game information in the database using the AppID, removing existing game records, and viewing generated reports. The group decided to implement a **master-slave architecture** to ensure efficient and consistent data management. In this setup, Node 1, which serves as the central node, is designated as the master, while the other two nodes act as slaves. This architecture is designed to ensure that all updates are consistently propagated from the master node to the slave nodes. When all nodes are working, all transactions are initially processed by the central master node. The data is then replicated to the corresponding slave node based on the partition rule: games released before 2010 are stored in Node 2, and games released from 2010 onwards are stored in Node 3. This approach ensures that the data remains consistent and up-to-date across all nodes. In the event that Node 1, the central master node, becomes unavailable, the system is designed to redirect transactions to the appropriate slave node based on the same partitioning logic. This approach ensures that the system remains functional and that users can continue to perform their operations seamlessly, even if the central master node is down.

To address node failures, the web application includes buttons for each node/server that can be toggled on and off, simulating the enabling and disabling of each node. These buttons, however, are hidden from the end-user and are only accessible for simulation and testing purposes. In addition, to ensure data consistency across all nodes/servers, write transactions—such as create, update, and delete operations—are processed in a systematic manner. The procedure begins by locking the relevant tables to prevent concurrent modifications. Next, the query is executed, and the transaction is logged. After committing the changes, the tables are unlocked, and the updates are propagated to the other nodes. This approach ensures that the data remains consistent and synchronized across the distributed database system. In cases where nodes are down, a robust recovery system has been implemented. For example, if a game released in 2011 is added to the central node (Node 1) and should be replicated to Node 3, but Node 3 is down, the replication will fail and the transaction will be logged as "PENDING." Once Node 3 is back online, a trigger function will check for any pending transactions and execute them to ensure that all nodes are up-to-date. This recovery approach ensures that the systems can handle failures and maintain data integrity, even when nodes are down.

2.3 Log Table

The `TRANSACTION_LOGS` table is used in storing transaction information, ensuring the system's transparency and reliability. It records key details such as the source node, the destination node where it should be replicated, the specific action that was performed (such as insert, update, or delete), the exact SQL query that was executed and the execution time of each transaction. This table is important for tracking the status of data across the distributed database system, particularly in situations where a node may be temporarily unavailable or disabled. In such cases, the `TRANSACTION_LOGS` table serves as a reference to determine which data needs to be replicated once the disabled node is restored. By storing detailed transaction records, the table ensures that the system can maintain data consistency across all nodes and servers.



TRANSACTION_LOGS
log_id INT
node_source VARCHAR(255)
node_target VARCHAR(255)
action VARCHAR(255)
status VARCHAR(255)
query TEXT
execution_time DATETIME

Figure 3. Transaction Schema

2.4 READ Transaction

For read transactions in a distributed database system that employs a master-slave architecture, users can search for the game they want using the AppID. Typically, all read queries are directed to the central master node, which is Node 1. This ensures that the most up-to-date and consistent data is retrieved efficiently. In the event that Node 1 becomes unavailable, the system automatically redirects read queries to Node 2 to maintain continuous access to the data. Should both Node 1 and Node 2 be down simultaneously, the system will then transfer read queries to Node 3. This approach ensures that read transactions can be handled smoothly, providing high availability and reliability for users when nodes fail.

2.5 WRITE Transactions

Other than the `READ` transactions implemented in the web application, users are also allowed to perform `WRITE` operations in the case of adding a new game record, editing, or deleting an existing record. Following the master-slave architecture, the master node handles all `WRITE` transactions with the remaining slave nodes handling `READ` operations. This allows for better performance in applications wherein operations are mostly `READ`-heavy.

Concurrent users introduce issues specifically related to conflicting data upon modifications. As a fix, concurrency control methods are applied with more detail on its implementation on the succeeding chapters. At this point, over reliance on Node 1 for all `WRITE` operations deems insufficient in case of its failure. As a failover, a slave node with the appropriate release year is promoted as master. Asynchronous replication of pending transactions is done in time Node 1 is back up to ensure global consistency across nodes. In addition to that, transaction recovery from issues such as replication failure to the node's unavailability is resolved by checking for pending transactions in the `transaction_log` table.

3. CONCURRENCY CONTROL AND CONSISTENCY

As multiple users perform read and write operations in the application simultaneously, the issue of data consistency across users becomes critical. To ensure integrity of data, concurrency control methods are implemented, such as transaction management, table locking, and usage of isolation levels (Repeatable Reads in this case). For each read/write operation, a transaction is first started and a table lock on the `GAME_TABLE` is used to avoid overlapping modifications. Once locked, the operation is executed to be followed by ending the transaction using the command `COMMIT` and unlocking the `GAME_TABLE`. For error handling during execution, a `ROLLBACK` command is enforced to maintain clean data by undoing all changes done by the current transaction (reverts back to the last `COMMIT`).

In the case of heavy load on the web application, relying on a single node can lead to inefficiency and slower operations, thereby hindering the application's performance. To ensure reliability during traffic surges, a master-slave architecture is adopted. In this distributed database system, three nodes are utilized: Node 1 acts as the master node, capable of both read and write operations on the database. Nodes 2 and 3 serve as slave nodes, handling only read operations to reduce complexity and improve data consistency across all nodes [4]. Although the centralization of control in a master-slave architecture is considered an advantage, the system's reliance on the master node can introduce bottlenecks, particularly in write-heavy applications, and makes it a single point of failure. This stands in stark contrast to the Peer-to-Peer architecture, where all units are treated equally, and the concept of a central node is absent [5].

In the master-slave setup, the master or central node has the capability to execute both read and write (create, read, update, and delete) queries. In contrast, the slave nodes can only handle read queries. During write transactions, data is first written to the master node, Node 1, and then replicated to the slave nodes, Node 2 and Node 3. For each write transaction, a log entry is created and stored in the `transaction_log` table, which is present in the databases of all nodes/servers. This log entry contains detailed information about the transaction, which is particularly useful for data recovery if a node is down and replication fails. When a down node/server is enabled using the toggle buttons, a function is triggered to check for any pending transactions in the `transaction_log` table of each database.

Table 2. Concurrency Test Result

Case	Transaction 1	Transaction II	P/F
1	READ X	READ X	P
2	UPDATE X	READ X	P
3	UPDATE X	UPDATE X	P

A test was considered correct if the two transactions would always end up with the same information after their respective operations. The results of these tests are shown in Table 2, where ACTION X signifies a certain transaction conducted on some data item in the database. All tests were run at least three times as seen on the Appendix. A common denominator here is that the successful completion of all test cases leads to the view and reasoning that the applied implementation in these two parts—global concurrency control and replication—works and behaves as expected logically. Crucial aspects that made it happen in critical implementation details lay with having Repeatable Read as isolation level and usage of explicit locks and writing the significant amount of logging and a reliable replication system for the control in place that will handle writes arriving on the other nodes.

4. GLOBAL FAILURE AND RECOVERY

There are many reasons why a system might fail, such as power outages, hardware failures, or data corruption, all of which can occur unexpectedly. In such situations, it is crucial to handle data recovery carefully to ensure data consistency across multiple users.

This project employs a master-slave architecture, with the master node being the primary point of failure. To mitigate this risk, the master node is transferred to another node if it fails, ensuring continued control over data. When executing write operations—such as creating, updating, or deleting game records—the system first checks if Node 1 is up and running. If Node 1 is operational, the operation proceeds as normal, with the modification reflected on Node 1 and replicated to either Node 2 or Node 3, depending on the release year of the game.

In the event that Node 1 fails, the master role is transferred to one of the remaining nodes: Node 2 is designated as the master if the game's release year is before 2010, while Node 3 takes over as master if the game was released in 2010 or later. Once Node 1 is back up, new data from Node 2 or Node 3 is replicated back to Node 1 to update its database.

Further facilitating the recovery process is the transaction logging which occurs whenever there is a write operation (e.g. INSERT, UPDATE and DELETE SQL commands) in a transaction. This allows previously down nodes to revisit the logs and identify changes not yet reflected in its own database. The status attribute from the TRANSACTION_LOGS table is utilized for this purpose and is logged as PENDING for every failed transaction due to node failure. Once replication of pending transactions is successful, its status in the TRANSACTION_LOGS is updated to COMPLETE.

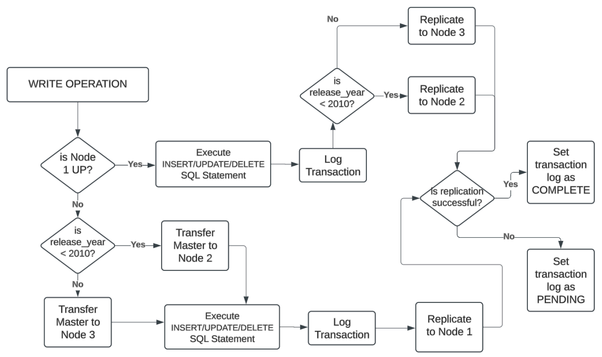
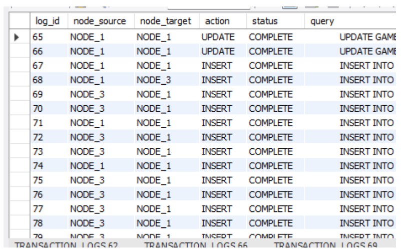

WRITE OPERATION
is Node 1 UP?
Yes
Execute INSERT/UPDATE/DELETE SQL Statement
Log Transaction
is release_year < 2010?
No
Replicate to Node 3
Yes
Replicate to Node 2
is replication successful?
Yes
Set transaction log as COMPLETE
No
Set transaction log as PENDING
No
is release_year < 2010?
Yes
Transfer Master to Node 2
No
Transfer Master to Node 3
Execute INSERT/UPDATE/DELETE SQL Statement
Log Transaction
Replicate to Node 1

Figure 4. Data Recovery Process

For example, when a new game is inserted into Node 1, the central/master node, the transaction is logged as COMPLETE once it successfully writes to the node (see Figure 5 below). This transaction is then replicated to the appropriate node based on the game's release year. If the game was released in 2011, it should be written to Node 3. If Node 3 is down at the time of replication, the transaction will fail, and its status will be logged as PENDING. When the disabled node (Node 3 in this case) is enabled again, the toggle function triggers a process that checks for any pending transactions. This process retrieves the pending transactions from the `transaction_log` table and completes the replication, ensuring data consistency across all nodes.



log_id	node_source	node_target	action	status	query
65	NODE_1	NODE_1	UPDATE	COMPLETE	UPDATE GAME
66	NODE_1	NODE_1	UPDATE	COMPLETE	UPDATE GAME
67	NODE_1	NODE_1	INSERT	COMPLETE	INSERT INTO
68	NODE_1	NODE_3	INSERT	COMPLETE	INSERT INTO
69	NODE_3	NODE_1	INSERT	COMPLETE	INSERT INTO
70	NODE_3	NODE_1	INSERT	COMPLETE	INSERT INTO
71	NODE_1	NODE_1	INSERT	COMPLETE	INSERT INTO
72	NODE_3	NODE_1	INSERT	COMPLETE	INSERT INTO
73	NODE_3	NODE_1	INSERT	COMPLETE	INSERT INTO
74	NODE_1	NODE_1	INSERT	COMPLETE	INSERT INTO
75	NODE_3	NODE_1	INSERT	COMPLETE	INSERT INTO
76	NODE_3	NODE_1	INSERT	COMPLETE	INSERT INTO
77	NODE_3	NODE_1	INSERT	COMPLETE	INSERT INTO
78	NODE_3	NODE_1	INSERT	COMPLETE	INSERT INTO
79	NODE_3	NODE_1	INSERT	COMPLETE	INSERT INTO

TRANSACTION_LOGS 62 TRANSACTION_LOGS 66 TRANSACTION_LOGS 69

Figure 5. Transaction Log Table Snippet

Testing the cases listed in the specifications involved disabling nodes. Cases 1 and 2 address basic transactions in scenarios where nodes are down while users continue sending queries then backs online down the line, central on the former and either Node 2 or 3 on the latter. In addition, Case 3 and 4 address global failure against recovery in scenarios where a node is down and has not been turned on, central node is down for the former and either Node 2 or 3 on the latter. The distributed database must perform a response to user requests, hence a test is considered a successful one if the transactions are processed correctly and the data used within the transaction remains consistent. As a sample test, case 1 involves disabling the central node and performing a READ transaction, which should be handled by one of the other nodes. The central node is then re-enabled and queried accordingly, ensuring the read results are identical. A summary of these test case results is provided in Table 3 and 4 where each condition was tested at least three times.

Table 3. Global Recovery for Disabled Node

Case	Action	Performed In	Disabled Node	P/F
1	READ X	Node 1	Node 0	P
2	READ X	Node 0	Node 1	P

Table 4. Failure to write for Disabled Node

Case	Action	Replicated from	Failure (to write) in	P/F
3	CREATE X	Node 1/2	Node 0	P
4	UPDATE X	Node 0	Node 1/2	P

5. DISCUSSION

The importance of using distributed databases in certain applications is derived from the size of the application's database and from the number of users simultaneously using the application. Distributed databases prevent inconvenience for said users by preventing errors such as unusability of the application when the central site holding all data is down, or slow transactions when many users are simultaneously using the application. The benefits of using a distributed database and the different concurrency techniques utilized for our project are further discussed by analyzing the group's concurrency and update strategy and the recovery strategy used for the Steam Games web application.

5.1 Concurrency and Update Strategy

By using appropriate isolation levels, transactions in a database can be executed concurrently since their usage allows for proper handling of data. The proper handling of data through usage of isolation levels ensures database consistency. Without using appropriate isolation levels, concurrent transactions will result in data inconsistency. The group chose to have an isolation level of Repeatable Read for all transactions, as an isolation level of Repeatable Read prevents phenomena such as dirty reads and non repeatable reads. As mentioned in Concurrency Control and Consistency, modifications to the data are also controlled using the commands `COMMIT` and `ROLLBACK`. The `ROLLBACK` command in particular allows for the database to return to "save points" by reverting back to the last `COMMIT` if an error occurs during a transaction.

For the system's update strategy, operations which modify the database such as adding, editing, and deleting game records is primarily handled by the master node. These changes are then propagated across the appropriate slave node according to the games' release year to ensure all data modifications are reflected across all nodes. Each transaction of a WRITE nature is logged onto the TRANSACTION_LOGS to keep a record of transactions and their status for reference during node failures.

5.2 Recovery Strategy

Accounting for system errors includes having a solid recovery strategy employing techniques such as data fragmentation and replication, along with transaction logging. Our group's distributed database used a master-slave architecture design, wherein if the master node goes down, the responsibility of being the master node is passed to one of the former slaves, which is the best course of action in a central node crash.

Data fragmentation and replication are necessary as part of foolproofing against errors and providing convenience for accessing data. Data fragmentation is the splitting of data across multiple sites/nodes to allow for convenience of access and parallel processing. Users should not be able to tell the difference between using a database with fragmented or non fragmented data to ensure fragmentation transparency [2]. The distributed database our group designed fragments data horizontally as opposed to vertically, as the games are stored by release year. In the hypothetical situation that specific users of the web application are researching information on games released before 2010, the fragmented data allows for these users to access only the relevant tables with the needed data.

Data replication ensures that users have a smooth experience while using an application, as it allows for data to be available even when the central site of data goes down, as proven in demonstration Case 1 of Global Failure and Recovery, wherein the central node was simulated to have crashed. Despite the crash, users were still able to access the application as Nodes 1 and 2 together contain all data that was temporarily unavailable due to the crash. In the case of our application, if one of the slave nodes is down and a new game is added to the master node that is categorized, data replication will be accomplished once the concerned node is running again. The database keeps track of these missed replications by having a transaction log stored in the TRANSACTION_LOGS table. Inversely, the recorded transaction log will also ensure that any data that was created or updated during any moment of failure of the central node can be updated in the central node once it is running again.

Data transparency is supported by our data replication and update strategy since it allows for users to have a regular, smooth experience on our web application regardless of several kinds of simulated global crashes. Regardless of whether the central node crashes or one of the slave nodes crashes, the user is none the wiser since the application runs as usual.

By having Node 1 be the central node that contains all the data, Node 2 having replicate data on all games released before 2010, and Node 3 having replicate data on all games released on or after 2010, the distributed database combines data replication and fragmentation. The combination of data replication and fragmentation means that data replicated partially by partitioning data based on release year.

As previously mentioned, node recovery is supported by the facility of keeping transaction logs. A new transaction log is recorded for every transaction that uses a WRITE operation. Recorded in the transaction log are the following information: a unique log identifier for each log, the node wherein it was first recorded (node_source), the target node where data should be stored or replicated, the action done in the transaction, the status of the transaction, and the actual query itself. Since all details of any transaction that has a WRITE operation are recorded, even in the case that the master node is down, or one of the slave nodes is down, data replication will proceed normally once the affected nodes are up again. Hence, data will remain consistent throughout all nodes despite global failures, resulting in accurate, smooth transactions for end users at all times.

6. CONCLUSION

Distributed database systems are highly invaluable when it comes to handling large amounts of data that is accessed by multiple users simultaneously due to their global availability, fault tolerance, and scalability [1]. To create an effective system of this kind, multiple strategies for replicating, updating, and recovering data are essential to ensure data consistency across all locations or nodes. Focusing on one denormalized GAME_TABLE, this project successfully demonstrates these strategies centered on the system's master-slave architecture design.

The distributed database system employs a master-slave architecture with partial data replication and horizontal fragmentation across three nodes. Node 1 serves as the master node, storing all game data, while slave nodes 2 and 3 store games released before and 2010 onwards, respectively. Asynchronous replication allows for handling node failures by logging pending transactions and reflecting them once a node comes back online, ensuring data consistency without interrupting primary operations. Write operations are primarily managed by the master node, with modifications propagated to appropriate slave nodes and logged in the TRANSACTION_LOGS for tracking and recovery purposes.

Concurrency is managed through robust concurrency control methods, including table locking and a Repeatable Read isolation level to prevent overlapping modifications while maintaining data integrity. Transactions are carefully managed—started, locked, executed, committed, and unlocked—with a ROLLBACK command available for error handling. In the event of a master node failure, the system promotes a slave node based on game release year and recovers any pending transactions by checking the transaction log when the node returns online, thus maintaining continuous data availability and consistency across the distributed database system.

The development process of this distributed database system highlights the critical infrastructure behind modern, scalable applications. By implementing strategies like log-based recovery, transaction management, and a master-slave architecture, the project demonstrates how complex systems ensure data integrity and availability in scenarios with high user concurrency and potential system failures. This approach provides a practical blueprint for building resilient digital infrastructure that can dynamically adapt to network challenges, hardware failures, and unpredictable user loads – a crucial capability in our increasingly interconnected and data-driven world.

7. REFERENCES

[1] *Advantages of distributed databases for modern applications*. Macrometa. (n.d.). https://www.macrometa.com/articles/advantages-of-distributed-databases-for-modern-applications

[2] Anon. 2014. What is fragmentation? (March 2014). Retrieved December 3, 2024 from https://www.ibm.com/docs/vi/informix-servers/12.10?topic=SSGU8G_12.1.0%2Fcom.ibm.ddi.doc%2Fids_ddi_084.ht

[3] Haque, S. (2023, December 19). *What is a distributed database and when should you use one*. Fauna. https://fauna.com/blog/what-is-a-distributed-database-and-when-should-you-use-one

[4] Ogun, A. Y. (2022, May 1). *Master-slave database architecture in a Nutshell*. Medium. https://medium.com/@ayogun/master-slave-database-architecture-in-a-nutshell-e20a73e979d1

[5] *Master-Slave Architecture*. Dremio. (2024, July 23). https://www.dremio.com/wiki/master-slave-architecture/

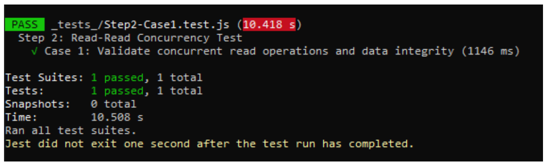
[6] *What is fragmentation?*. IBM. (2014, March). https://www.ibm.com/docs/vi/informix-servers/12.10?topic=SSGU8G_12.1.0%2Fcom.ibm.ddi.doc%2Fids_ddi_084.html

8. APPENDIX

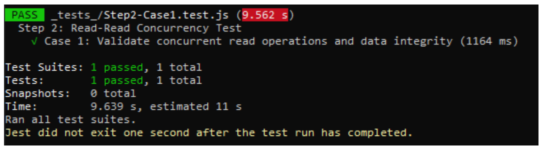
I. AUTOMATED TESTING

Step 2 : Concurrency Control and Consistency

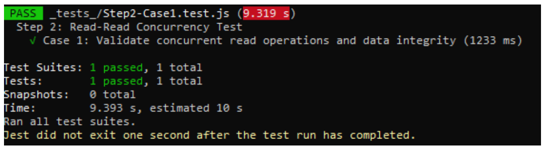
A. Concurrent transactions in two or more nodes are reading the same data item.



```
PASS  _tests_/Step2-Case1.test.js (10.418 s)
  Step 2: Read-Read Concurrency Test
    √ Case 1: Validate concurrent read operations and data integrity (1146 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        10.508 s
Ran all test suites.
Jest did not exit one second after the test run has completed.
```

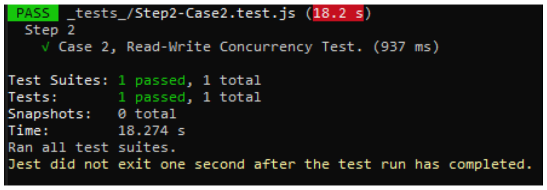


```
PASS  _tests_/Step2-Case1.test.js (9.562 s)
  Step 2: Read-Read Concurrency Test
    √ Case 1: Validate concurrent read operations and data integrity (1164 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        9.639 s, estimated 11 s
Ran all test suites.
Jest did not exit one second after the test run has completed.
```

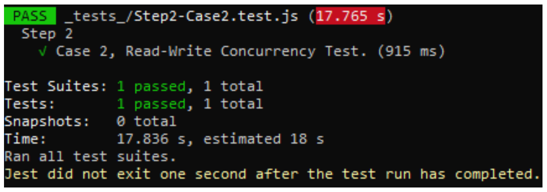


```
PASS  _tests_/Step2-Case1.test.js (9.319 s)
  Step 2: Read-Read Concurrency Test
    √ Case 1: Validate concurrent read operations and data integrity (1233 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        9.393 s, estimated 10 s
Ran all test suites.
Jest did not exit one second after the test run has completed.
```

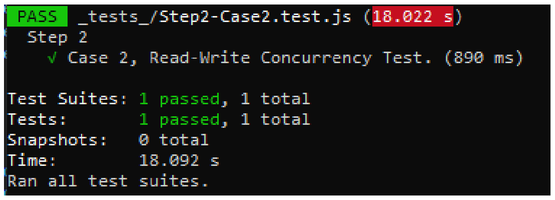
B. At least one transaction in the three nodes is writing (update) and the other concurrent transactions are reading the same data item.



```
PASS  _tests_/Step2-Case2.test.js (18.2 s)
  Step 2
    √ Case 2, Read-Write Concurrency Test. (937 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        18.274 s
Ran all test suites.
Jest did not exit one second after the test run has completed.
```

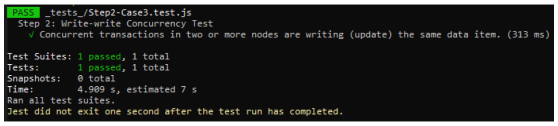


```
PASS  _tests_/Step2-Case2.test.js (17.765 s)
  Step 2
    √ Case 2, Read-Write Concurrency Test. (915 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        17.836 s, estimated 18 s
Ran all test suites.
Jest did not exit one second after the test run has completed.
```

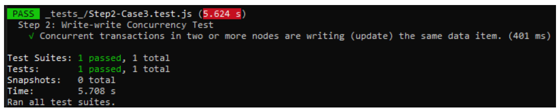


```
PASS  _tests_/Step2-Case2.test.js (18.022 s)
  Step 2
    √ Case 2, Read-Write Concurrency Test. (890 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        18.092 s
Ran all test suites.
```

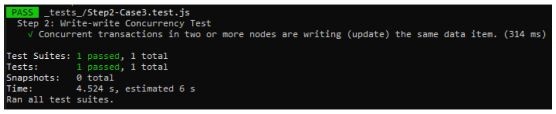
C. Concurrent transactions in two or more nodes are writing (update) the same data item.



```
PASS  _tests_/Step2-Case3.test.js
  Step 2: Write-write Concurrency Test
    √ Concurrent transactions in two or more nodes are writing (update) the same data item. (313 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        4.909 s, estimated 7 s
Ran all test suites.
Jest did not exit one second after the test run has completed.
```



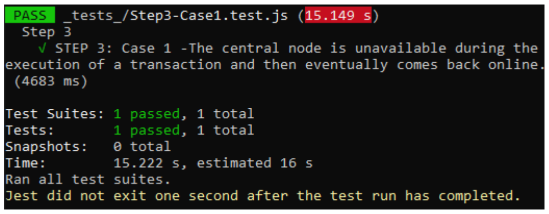
```
PASS  _tests_/Step2-Case3.test.js (5.624 s)
  Step 2: Write-write Concurrency Test
    √ Concurrent transactions in two or more nodes are writing (update) the same data item. (401 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        5.708 s
Ran all test suites.
```



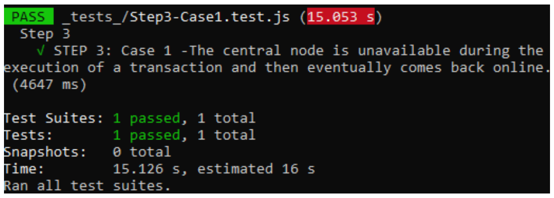
```
PASS  _tests_/Step2-Case3.test.js
  Step 2: Write-write Concurrency Test
    √ Concurrent transactions in two or more nodes are writing (update) the same data item. (314 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        4.524 s, estimated 6 s
Ran all test suites.
```

Step 3: Global Failure And Recovery

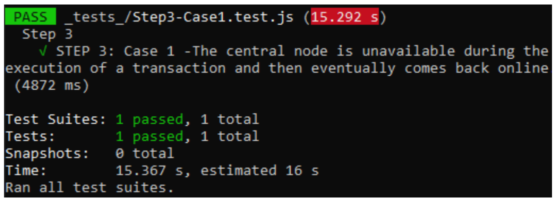
A. The central node is unavailable during the execution of a transaction and then eventually comes back online



```
PASS  _tests_/Step3-Case1.test.js (15.149 s)
  Step 3
    √ STEP 3: Case 1 -The central node is unavailable during the
execution of a transaction and then eventually comes back online.
 (4683 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        15.222 s, estimated 16 s
Ran all test suites.
Jest did not exit one second after the test run has completed.
```



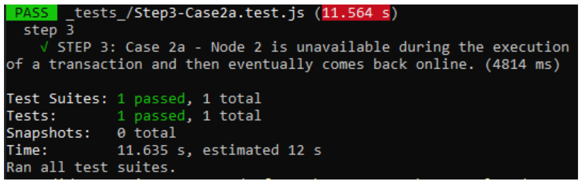
```
PASS  _tests_/Step3-Case1.test.js (15.053 s)
  Step 3
    √ STEP 3: Case 1 -The central node is unavailable during the
execution of a transaction and then eventually comes back online.
 (4647 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        15.126 s, estimated 16 s
Ran all test suites.
```



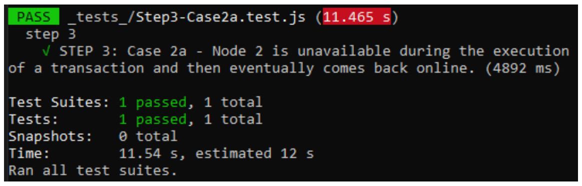
```
PASS  _tests_/Step3-Case1.test.js (15.292 s)
  Step 3
    √ STEP 3: Case 1 -The central node is unavailable during the
execution of a transaction and then eventually comes back online
 (4872 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        15.367 s, estimated 16 s
Ran all test suites.
```

B. <partial node> is unavailable during the execution of a transaction and then eventually comes back online

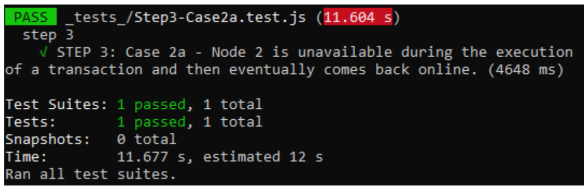
a. Node 2



```
PASS  _tests_/Step3-Case2a.test.js (11.564 s)
  step 3
    √ STEP 3: Case 2a - Node 2 is unavailable during the execution
of a transaction and then eventually comes back online. (4814 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        11.635 s, estimated 12 s
Ran all test suites.
```

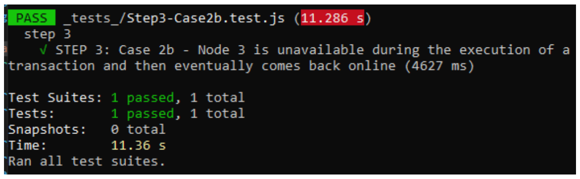


```
PASS  _tests_/Step3-Case2a.test.js (11.465 s)
  step 3
    √ STEP 3: Case 2a - Node 2 is unavailable during the execution
of a transaction and then eventually comes back online. (4892 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        11.54 s, estimated 12 s
Ran all test suites.
```

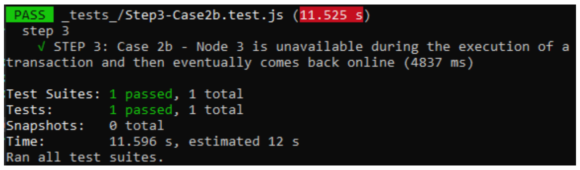


```
PASS  _tests_/Step3-Case2a.test.js (11.604 s)
  step 3
    √ STEP 3: Case 2a - Node 2 is unavailable during the execution
of a transaction and then eventually comes back online. (4648 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        11.677 s, estimated 12 s
Ran all test suites.
```

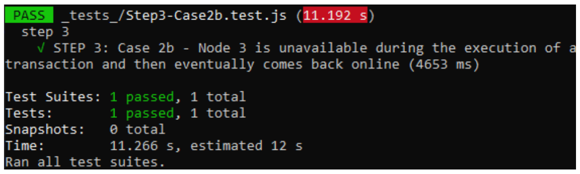
b. Node 3



```
PASS  _tests_/Step3-Case2b.test.js (11.286 s)
  step 3
    √ STEP 3: Case 2b - Node 3 is unavailable during the execution of a
transaction and then eventually comes back online (4627 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        11.36 s
Ran all test suites.
```

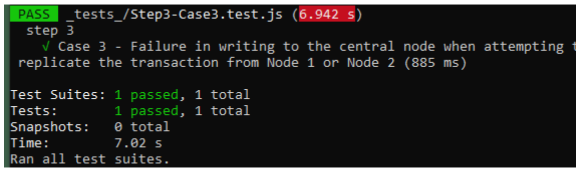


```
PASS  _tests_/Step3-Case2b.test.js (11.525 s)
  step 3
    √ STEP 3: Case 2b - Node 3 is unavailable during the execution of a
transaction and then eventually comes back online (4837 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        11.596 s, estimated 12 s
Ran all test suites.
```

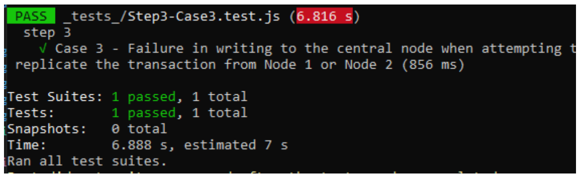


```
PASS  _tests_/Step3-Case2b.test.js (11.192 s)
  step 3
    √ STEP 3: Case 2b - Node 3 is unavailable during the execution of a
transaction and then eventually comes back online (4653 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        11.266 s, estimated 12 s
Ran all test suites.
```

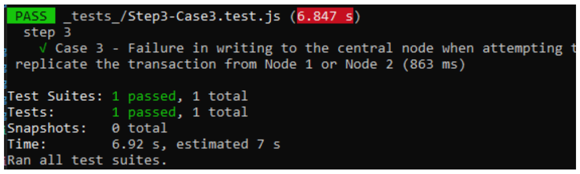
C. Failure in writing to the central node when attempting to replicate the transaction from Node 1 or Node 2



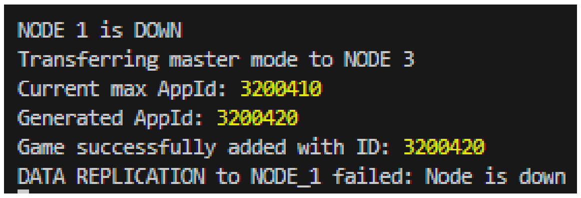
```
PASS  _tests_/Step3-Case3.test.js (6.942 s)
  step 3
    √ Case 3 - Failure in writing to the central node when attempting t
  replicate the transaction from Node 1 or Node 2 (885 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        7.02 s
Ran all test suites.
```



```
PASS  _tests_/Step3-Case3.test.js (6.816 s)
  step 3
    √ Case 3 - Failure in writing to the central node when attempting t
  replicate the transaction from Node 1 or Node 2 (856 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        6.888 s, estimated 7 s
Ran all test suites.
```



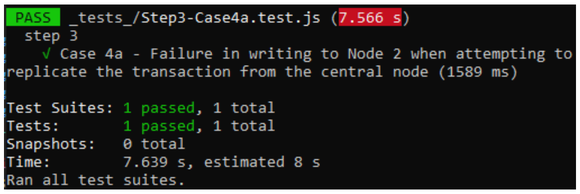
```
PASS  _tests_/Step3-Case3.test.js (6.847 s)
  step 3
    √ Case 3 - Failure in writing to the central node when attempting t
  replicate the transaction from Node 1 or Node 2 (863 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        6.92 s, estimated 7 s
Ran all test suites.
```



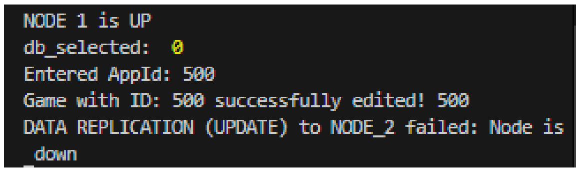
```
NODE 1 is DOWN
Transferring master mode to NODE 3
Current max AppId: 3200410
Generated AppId: 3200420
Game successfully added with ID: 3200420
DATA REPLICATION to NODE_1 failed: Node is down
```

D. Failure in writing to <partial node> when attempting to replicate the transaction from the central node

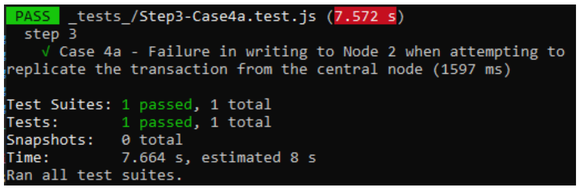
a. Node 2



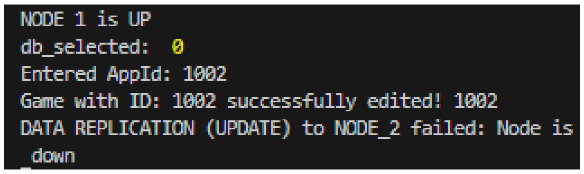
```
PASS  _tests_/Step3-Case4a.test.js (7.566 s)
  step 3
    √ Case 4a - Failure in writing to Node 2 when attempting to
replicate the transaction from the central node (1589 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        7.639 s, estimated 8 s
Ran all test suites.
```



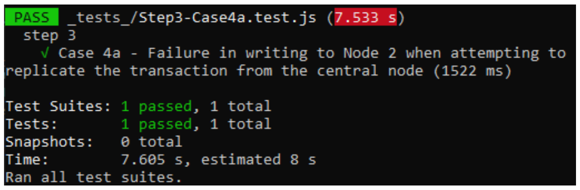
```
NODE 1 is UP
db_selected:  0
Entered AppId: 500
Game with ID: 500 successfully edited! 500
DATA REPLICATION (UPDATE) to NODE_2 failed: Node is
 down
```



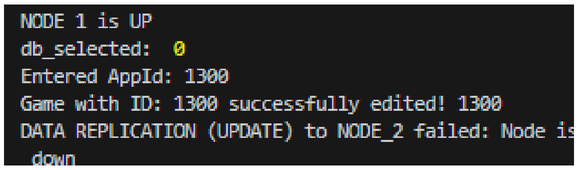
```
PASS  _tests_/Step3-Case4a.test.js (7.572 s)
  step 3
    √ Case 4a - Failure in writing to Node 2 when attempting to
replicate the transaction from the central node (1597 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        7.664 s, estimated 8 s
Ran all test suites.
```



```
NODE 1 is UP
db_selected:  0
Entered AppId: 1002
Game with ID: 1002 successfully edited! 1002
DATA REPLICATION (UPDATE) to NODE_2 failed: Node is
 down
```

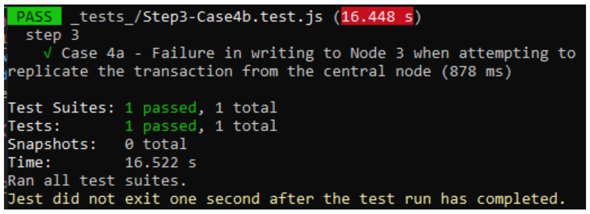


```
PASS  _tests_/Step3-Case4a.test.js (7.533 s)
  step 3
    √ Case 4a - Failure in writing to Node 2 when attempting to
replicate the transaction from the central node (1522 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        7.605 s, estimated 8 s
Ran all test suites.
```

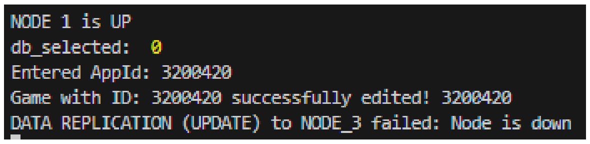


```
NODE 1 is UP
db_selected:  0
Entered AppId: 1300
Game with ID: 1300 successfully edited! 1300
DATA REPLICATION (UPDATE) to NODE_2 failed: Node is
 down
```

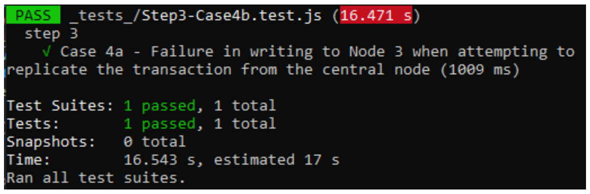
b. Node 3



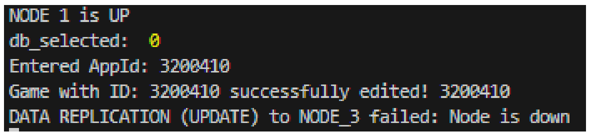
```
PASS  _tests_/Step3-Case4b.test.js (16.448 s)
  step 3
    √ Case 4a - Failure in writing to Node 3 when attempting to
replicate the transaction from the central node (878 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        16.522 s
Ran all test suites.
Jest did not exit one second after the test run has completed.
```



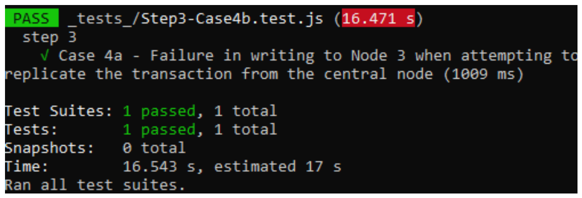
```
NODE 1 is UP
db_selected:  0
Entered AppId: 3200420
Game with ID: 3200420 successfully edited! 3200420
DATA REPLICATION (UPDATE) to NODE_3 failed: Node is down
```



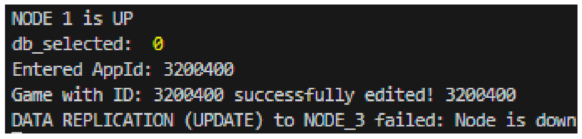
```
PASS  _tests_/Step3-Case4b.test.js (16.471 s)
  step 3
    √ Case 4a - Failure in writing to Node 3 when attempting to
replicate the transaction from the central node (1009 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        16.543 s, estimated 17 s
Ran all test suites.
```



```
NODE 1 is UP
db_selected:  0
Entered AppId: 3200410
Game with ID: 3200410 successfully edited! 3200410
DATA REPLICATION (UPDATE) to NODE_3 failed: Node is down
```



```
PASS  _tests_/Step3-Case4b.test.js (16.471 s)
  step 3
    √ Case 4a - Failure in writing to Node 3 when attempting to
replicate the transaction from the central node (1009 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        16.543 s, estimated 17 s
Ran all test suites.
```



```
NODE 1 is UP
db_selected:  0
Entered AppId: 3200400
Game with ID: 3200400 successfully edited! 3200400
DATA REPLICATION (UPDATE) to NODE_3 failed: Node is down
```

II. TEST SCRIPTS

Step 2 : Concurrency Control and Consistency

*/*Every test should begin with a fresh node index initialization*/*

A. Concurrent transactions in two nodes are reading the same data item.

db_selected1	db_selected2	ids	titles	titles (Expected)	titles (Actual)	P/F
0	1	10	Counter-Strike	Counter-Strike	Counter-Strike	**P**
		30	Day of Defeat	Day of Defeat	Day of Defeat	**P**
		40	Deathmatch Classic	Deathmatch Classic	Deathmatch Classic	**P**

B. At least one transaction in the three nodes is writing (update) and the other concurrent transactions are reading the same data item

/ Other game attributes needed for the update remain to be constant on the script*/*

db_selected1	db_selected2	ids	Game Name (pre-test)	testName1 (Expected)	testName 1 (Actual)	P/F
0	1	['130', '730']	X*	testNameB1	testNameB1	**P**
			X*	testNameB2	testNameB2	**P**
			X*	testNameB3	testNameB3	**P**

X : current *Name* on the database respective of the selected AppID *ids* and node used (*db_selected1* or *db_selected_2*)

C. Concurrent transactions in two or more nodes are writing (update) the same data item.

/ New Testname1 and id every test run*/*

db_selected1	db_selected2	id	Game Name (pre-test)	testName1 (Expected)	testName1 (Actual)	P/F
0	1	3200090	X*	testNameC6	testNameC6	**P**
		3200410	X*	testNameC7	testNameC7	**P**
		3200380	X*	testNameC8	testNameC8	**P**

X : current *Name* on the database respective of the selected AppID *ids* and node used (*db_selected1* or *db_selected_2*)

Step 3: Global Failure And Recovery

/ Every test should begin with a fresh node index initialization*/*

A. The central node is unavailable during the execution of a transaction and then eventually comes back online

ids	Disabled	Action	Performed In	Pass/Fail
50	Node 0	Read	Node 1	**Pass**
630	Node 0	Read	Node 2	**Pass**

B. <partial node> is unavailable during the execution of a transaction and then eventually comes back online

a. Node 2

ids	Disabled	Action	Performed In	Pass/Fail
100	Node 1	Read	Node 0	**Pass**

b. Node 3

ids	Disabled	Action	Performed In	Pass/Fail
630	Node 2	Read	Node 0	**Pass**
3200420	Node 2	Read	Node 0	**Pass**

C. Failure in writing to the central node when attempting to replicate the transaction from Node 1 or Node 2

ids	Disabled	Action	Performed ?	Pass/Fail
-	Node 0	Create	X	**Pass**

/ All game attributes needed for the create remain to be constant on the script*/*

D. Failure in writing to <partial node> when attempting to replicate the transaction from the central node

a. Node 2

ids	Disabled	Action	Performed ?	Pass/Fail
1300	Node 1	Update	X	**Pass**
20200	Node 1	Update	X	**Pass**

/ Other game attributes needed for the update remain to be constant on the script*/*

b. Node 3

ids	Disabled	Action	Performed ?	Pass/Fail
3200400	Node 2	Update	X	**Pass**
630	Node 2	Update	X	**Pass**

/ Other game attributes needed for the update remain to be constant on the script*/*

III. Web Application Link

https://stadvdb-mco2-stream-games.onrender.com

IV. GitHub Link

https://github.com/Mikosantos/mco2

9. Declarations

9.1 Declaration of Generative AI in Scientific Writing.

During the preparation of this work the author(s) used ChatGPT in order to refine explanations, and assist with drafting content for this report. 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.

9.2 Record of Contribution.

P1: Clarissa Albarracin **P2:** Amor De Guzman
P3: Reina Althea Garcia **P4:** Miko Santos

P1	Web Application (Frontend and backend; Database Connections, Create, Update), Technical Report
P2	Technical Report
P3	Web Application (Backend; Search, Delete, Data Recovery, Data Replication), Technical Report
P4	Web Application (Frontend and Backend; Generate Report), Technical Report, Test Cases