

Implementation of a Distributed Database System for the Internet Movie Database (IMDb)

Cua, Lander Peter E.¹, Gaba, Jacob Bryan B.², Gonzales, Mark Edward M.³, and Lee, Hylene Jules G.⁴

Department of Software Technology, College of Computer Studies, De La Salle University, Manila, Philippines

¹lander_peter_cua@dlsu.edu.ph, ²jacob_bryan_gaba@dlsu.edu.ph, ³mark_gonzales@dlsu.edu.ph, ⁴hylene_jules_lee@dlsu.edu.ph

ABSTRACT

Distributed databases are frequently used when processing huge volumes of data and allowing multiple users concurrent access to the database; however, despite these advantages it offers, this type of system comes with caveats, specifically with regard to maintaining data consistency. A three-node distributed database system for IMDb, consisting of a central node and two partial replicas (shards), was created as a proof of concept to address concurrency control issues, promote failure recovery, and support transaction management. The design employs locks implemented at the application layer and global isolation level for concurrency control and decentralized statement-based logging for replication and recovery. In evaluating this strategy, concurrent read and write transactions were performed at varying isolation levels to validate consistency; nodes were crashed while transactions were ongoing to test global failure recovery. Results show that this log-based approach allows for recovery and high availability, and combining exclusive locks with the READ COMMITTED isolation level enables concurrency while blocking transaction anomalies.

Keywords

Distributed Database, Concurrency Control, Failure Recovery, Transaction Management

1. Introduction

A distributed database is a set of loosely coupled sites, also referred to as *nodes*, that can function independently but are also able to access data stored in other nodes [37]. While this setup promotes scalability and availability, the storage of data and availability of data access at different nodes on a distributed database raises issues regarding concurrency control since some topologies permit users to update the same entry simultaneously, possibly leading to a lost update, or read an entry that is currently being updated, possibly resulting in read anomalies (e.g., dirty reads). Similarly, the implementation of recovery is complicated by the need to maintain data redundancy across all nodes [38]. These challenges are highlighted in the CAP theorem [41], which states that distributed systems can only implement two properties among consistency, availability, and partition tolerance.

The implemented distributed database system is a three-node homogeneous distributed database with a central node containing all the data and two nodes containing partial replicas (hereafter referred to as Node 2 and Node 3), which store data on movies released before 1980 and those released from 1980 onwards, respectively. Users interact with the distributed database system via a web application that supports fragmentation transparency. Moreover, they are able to perform CRUD operations: adding new entries, searching the database, updating existing entries, and

deleting existing entries. Concurrency control is implemented by enforcing write locks at the application layer and global isolation levels; statement-based logs are also maintained by each node to facilitate recovery in the event of crashes or disconnections.

The design of the distributed database system is intended to maximize the availability and distribute the load when accessing the database. For instance, the system can be used to manage movie catalogs fragmented according to access frequency, wherein Node 2 can be located in an area where users are more interested in older movies and Node 3 is likewise located in an area where users access newer movies more frequently. As the system supports a variety of write operations, it can be utilized by authorized users to modify the contents of the database; following the earlier example, these users could include the marketing staff involved in the creation of the catalog and sales managers who can delete entries that are no longer available for consumption.

2. Distributed Database

This section discusses the setup of the distributed database and the application use case. While the project uses a homogeneous setup as a proof of concept, the locking and logging mechanisms are implemented on the application layer, thus, in theory, allowing them to support a heterogeneous distributed database as well.

2.1 Database Schema

The distributed database system stores the Internet Movies Database by the Jožef Stefan Institute [39] (henceforth referred to as *IMDb IJS*). To avoid introducing complexities related to the management of foreign keys, IMDb IJS was denormalized into a single *movies* table, as seen in Table 1.

Table 1. Denormalized *movies* Table

Attribute	Data Type	Description
id	INT	Unique identifier (primary key)
title	VARCHAR	Title of the movie
year	INT	Year of release of the movie
genre	VARCHAR	Genre of the movie
rank	FLOAT	Rating of the movie
director	VARCHAR	Director of the movie
actor1	VARCHAR	First actor in the movie
actor2	VARCHAR	Second actor in the movie

In order to reduce the size of the table, rows were imploded based on multivalued attributes (namely *genre* and *actor*), and only one *genre* and two *actors* were kept. The complete code for the denormalization process is given in Appendix A.

2.2 Distributed Database Topology

Since the distributed database system needs to support read and write operations on all three nodes, it follows a master-master replication strategy. The flow of data is bidirectional, i.e., data are propagated to and from the nodes. In effect, the update-anywhere nature of this topology avoids a single point of contention albeit at the cost of complications with controlling concurrency and maintaining consistency [10].

Specifically, partial replication is in place, with the central node containing all the movies and the two other nodes storing only horizontal fragments. Horizontal fragmentation (sharding) helps distribute the load and allows the system to scale, thus benefiting applications that focus on transaction management.

Figure 1 illustrates the distributed database topology; the nodes are connected via the application layer. The logs are discussed in detail in Section 2.5 (Log Table).

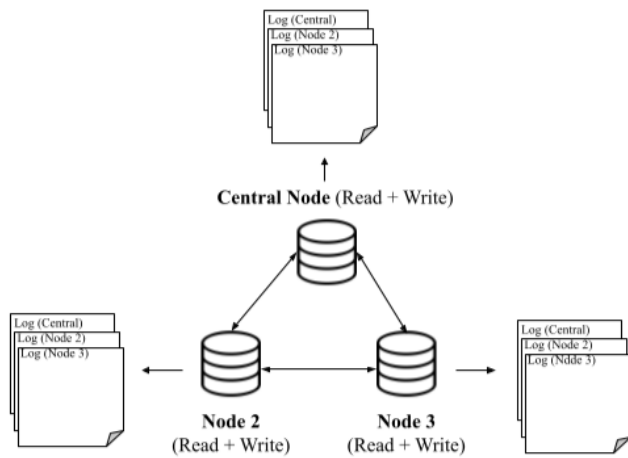


Figure 1. Distributed Database Topology

2.3 Remote Database Hosting

The distributed database is fully hosted on the cloud using the Amazon Relational Database Service (Amazon RDS), with each node corresponding to a database instance in the service and configured following the specifications in Appendix B. Note that a general-purpose solid-state drive was selected for the storage type to optimize support for the system’s intended usage: online transaction processing that expects only small I/O sizes [2].

The security group of the Amazon Elastic Compute Cloud was also modified to include an additional inbound rule (Appendix C) permitting applications outside the Amazon Virtual Private Cloud to access the remote database from any IP address (0.0.0.0/0) via the transmission control protocol (TCP). In the context of this project, these external applications include the user-facing web application and MySQL Workbench, with the latter extensively used during the initial import of the dataset into the three nodes, as well as during development and white-box testing.

2.4 Connecting the App to the Database

Written in JavaScript, the user-facing web application runs on Node.js, a cross-platform runtime environment built on top of the open-source V8 engine [31], and is deployed on Heroku, a cloud platform-as-a-service [35]. It is connected to the remote database

using the `mysql` driver (version 2.18.1) available through the Node Package Manager registry [30].

To manage the consumption of computational resources whenever a new connection has to be established, a connection pool was created for each node during application startup, with the host initialized to the Amazon RDS endpoint of the database and the port set to 3306, the default TCP port through which a MySQL client communicates with the MySQL server [28]. The connection limit was also specified to handle multiple connections for concurrent transactions, and the connection timeout was increased to 30 seconds to lengthen the window before aborting the attempt to complete the handshake (the 10-second default value proved insufficient to execute the experiments that involve global failure and recovery simulation). The code for instantiating a connection pool is given in Appendix D.

2.5 Log Table

The distributed database system employs a log-based approach to facilitate recovery in the event of node failures. Table 2 shows the information stored in a log. For illustrative purposes, Appendix E shows a sample log that records three transactions, all of which have been successfully committed in the central node.

Table 2. Information Stored in a Log

Attribute	Description
<code>node_id</code>	Numerical identifier of the node: 1 for the central node, 2 for Node 2, and 3 for Node 3
<code>lock_status</code>	Binary value indicating whether the node is locked: 0 for not locked and 1 for locked
<code>next_trans_record</code>	Identifier of the next transaction to be recorded in the log
<code>next_trans_commit</code>	Identifier of the next transaction to be committed
<code>id_new_entry</code>	ID (primary key) of the most recently inserted entry
<code>statements</code>	SQL queries executed (or to be executed) in the node, prefixed by a zero-based auto-incrementing identifier and delimited by three vertical pipes

This log-based approach is patterned after the recovery strategy that is utilized by most database management systems, including MySQL [24]. This system borrows the core ideas of MySQL’s global transaction identifiers and its dedicated system variables [25]. Each transaction is assigned a unique and global zero-based identifier. Every time a transaction is recorded, its identifier is assigned the value of `next_trans_record`, which is then incremented in preparation for the next transaction. In the end, the number of SQL queries in the `statements` attribute should always be equal to `next_trans_record`.

Moreover, every time a transaction is committed in a node, the value of `next_trans_commit` is updated to prevent queries from being re-executed or skipped and to permit auto-positioning even in the event of crashes. If all the transactions in the log have already been committed, the values of `next_trans_record` and `next_trans_commit` should be equal, as seen in Appendix E.

There are three major design decisions in which the system's log-based approach deviates from MySQL's implementation. The first difference is in relation to the logging format. Instead of MySQL's default row-based logging [26], this system follows statement-based logging. Although the former provides a higher degree of safety since logs contain events corresponding to the actual row changes, the statement-based format suffices, as the read and write transactions supported by this system do not include any of the nondeterministic stored programs and methods enumerated in the MySQL Reference Manual [26], precluding the possibility of triggering unsafe behaviors. Statement-based logging is also advantageous in terms of space efficiency since SQL statements occupy fewer bytes compared to changed rows.

Second, instead of recording in a binary file, a database table is instead used; therefore, each node has two tables: `movies` and `log`, with the statements stored inside a `LONGTEXT` field that supports 4 GB of characters. Aside from avoiding the overhead of storing and loading blob files, this also eliminates the need for workarounds to bypass the security restrictions of JavaScript's File System application programming interface (API), which, by default, blocks reading and writing onto a file unless otherwise permitted with direct consent from the end-user [8, 17].

Finally, each node maintains a copy of the logs of all the nodes in its `log` table. In this regard, the `log` table of each node has three rows, corresponding to the three nodes. Although propagating updates to the logs of all three nodes at the end of every transaction incurs additional space and time complexity, this approach prevents having single points of failure.

To justify this decision, consider the case when the central node fails and the user inserts a movie entry before 1980. It will be committed in Node 2, but the transaction will not execute in the central node during its down-time. While it is possible to indicate in Node 2's logs that the transaction has not yet been executed in the central node, the issue arises when the central node recovers at a time when Node 2 is down. The central node then cannot retrieve Node 2's logs to catch up on missed transactions.

This system's approach of having each node maintain a copy of the logs of all three nodes addresses this issue by providing the central node with the option to also consult the log pertinent to it that is stored in Node 3. This approach is partly motivated by the decentralized strategy of blockchain technology, where each node typically stores a complete copy of the entire blockchain [9].

Figure 1 provides a pictorial representation of this approach along with the topology. **For clarity, the log pertaining to node x that is stored in node y is hereafter referred to as (node x , node y) log.** Hence, the log pertaining to the central node that is stored in Node 2 is denoted by the ordered pair (central node, Node 2).

2.6 Read Transactions

The application supports two types of read (search) transactions: (i) retrieving all the movie entries and (ii) retrieving only a subset of the movie entries that match a specific search criterion.

2.6.1 Retrieving All Movie Entries

The application allows users to retrieve all the movie entries stored in IMDb IJS; the SQL code is given in Appendix F.

Since all the movie entries are retrieved, it is more efficient to fetch data directly from the central node rather than connecting to two separate nodes and joining the fragments, as handshaking and taking the union also incur network and performance overheads.

However, as shown in Figure 2, the unavailability of the central node forces the system to retrieve data from Nodes 2 and 3. This leads to three possible scenarios:

- If both partial replicas are available, the result set is complete since the results from both nodes can be merged.
- If exactly one of the partial replicas is down, the result set is incomplete since only the data from the available node can be retrieved.
- If both partial replicas are down, the result set is empty since all three nodes are unavailable.

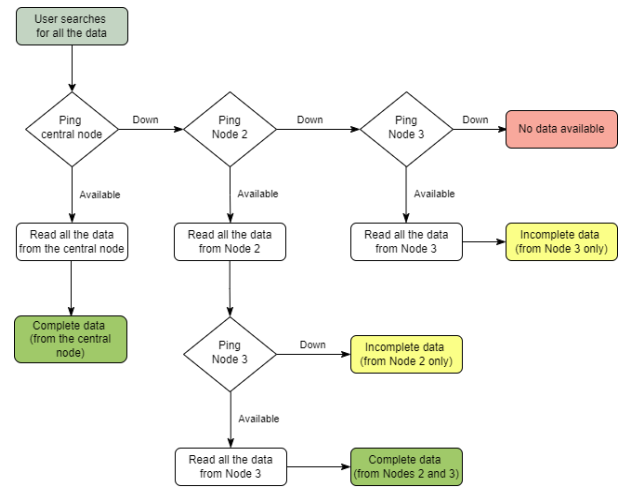


Figure 2. Retrieving All Movie Entries

2.6.2 Retrieving Based on a Search Criterion

The application allows users to retrieve a subset of the movie entries based on a specified search criterion. In particular, the user can search by movie ID, title, genre, director, or actor, and partial (substring) matches are returned; the SQL code is provided in Appendix G.

Since only a subset of the entries is needed, this read transaction prioritizes fetching from the partial replicas first in order to take advantage of the distributed database setup and avoid overloading the central node; it only connects to the central node if at least one of the replicas is down. Following the flowchart in Figure 3, the algorithm starts with checking the availability of a partial replica (arbitrarily selected as Node 2 in the implementation).

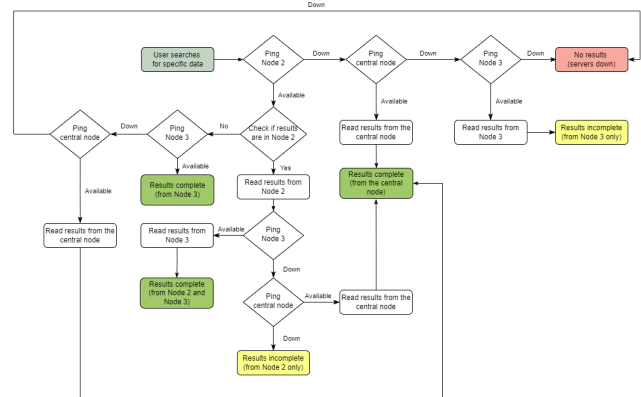


Figure 3. Retrieving Entries Based on Search Criterion

If Node 2 is available, it first checks whether this node contains relevant entries. If the answer is affirmative, it fetches them, then connects to the other partial replica (Node 3), leading to two possible scenarios:

- If Node 3 is available, it fetches the results from Node 3 and merges them with the results from Node 2, yielding a complete result set.
- If Node 3 is unavailable, it connects to the central node instead. If the central node is available, the results are fetched from it and returned (effectively discarding the partial result set from Node 2 to avoid duplicates). Otherwise, only the partial result set from Node 2, which may be incomplete, is returned.

If Node 2 does not contain relevant entries, it proceeds to connect to Node 3. Depending on this node's availability, there are two possible scenarios:

- If Node 3 is available, it fetches the results from this node, yielding a complete result set.
- If Node 3 is unavailable, it connects to the central node instead. If the central node is available, the results are fetched from it and returned, yielding a complete result set. Otherwise, the result set is empty, as none of the nodes containing relevant entries are available.

On the other hand, if Node 2 is unavailable, it prioritizes connecting to the central node in order to return a complete result set, provided that the central node is available. Inversely, the unavailability of the central node results in two possible scenarios depending on the availability of Node 3:

- If Node 3 is available, it fetches the results from this node. However, this result set may be incomplete.
- If Node 3 is unavailable, the result set is empty since all three nodes in the distributed database are down.

Read operations are not recorded in the logs since they do not mutate the database contents. In order to allow multiple users to retrieve information at the same time, they also do not request locks; however, since allowing a read while a write is happening opens the possibility of read anomalies, it is imperative to enforce proper isolation levels (discussed in Section 3). This is consistent with the implementations in database management systems like MySQL, where ordinary `SELECT` statements are nonlocking [22].

2.7 Write Transactions

The application supports three write transactions: (i) inserting a new movie entry, (ii) updating an existing entry, and (iii) deleting an entry. Note that, for write transactions, the happy path always starts with writing to the central node.

2.7.1 Inserting a Movie Entry

The application allows users to insert a movie entry given the title, year, genre, rank, director, and two actors. ID generation is handled by the application layer and automatically supplied in an auto-incrementing fashion. The SQL code is given in Appendix H.

As seen in Figure 4, if the central node is available, the system first checks if there is another (write) transaction already holding an exclusive lock on the `movies` table. To prevent issues related to locking, (e.g., deadlocks), a lock wait timeout is set to 100 seconds; if the lock is not granted within this 100-second window, the transaction is rejected, and the user is prompted to repeat it.

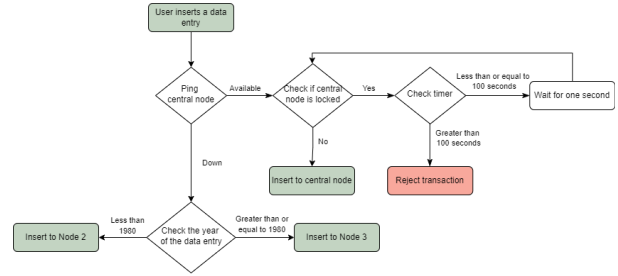


Figure 4. High-Level Algorithm for Inserting an Entry

Note that this lock is tracked using the `lock_status` attribute in the log table. For reference, the default lock wait timeout of InnoDB engine is 50 seconds [29]; setting the timeout to double this value was done to provide some leeway for the expected network overhead resulting from deployment to a cloud server.

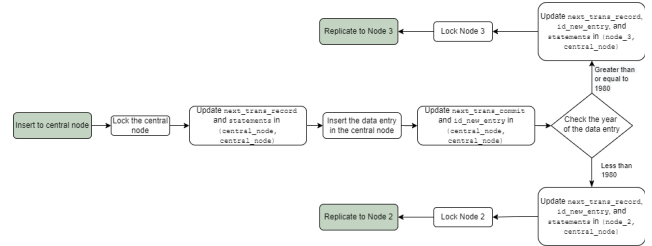


Figure 5. Inserting to the Central Node

If there is no exclusive lock on the central node (or once the lock is released within the 100-second window), the transaction places an exclusive lock on the node's `movies` table, and insertion to the central node commences following the flowchart in Figure 5; as introduced in Section 2.5, the notation (x, y) refers to the log pertaining to node x that is stored in node y . Essentially, the system records the transaction in the log pertaining to the central node, executes and commits the insertion, and records that it has been committed. Before proceeding to replication, it first updates the log pertaining to the relevant partial replica.

For comparison, MySQL follows an execute-log-commit pipeline where the relay of the log file to the replica happens after logging, albeit in a separate thread [27]. To simplify the implementation in this system, the stages are done sequentially, and the execute and commit steps are bundled together. This simplification makes it imperative for logging to take precedence to ensure that a record of the transaction persists even in the event of failures.

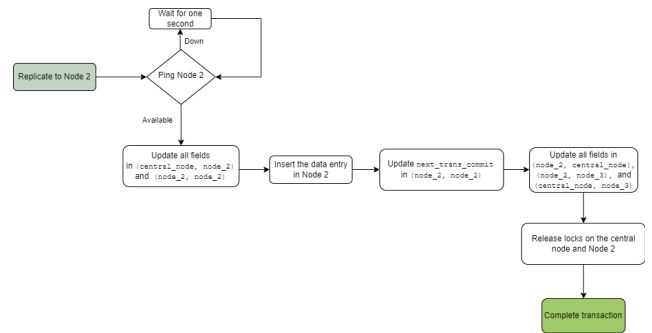


Figure 6. Replicating to Node 2 (From the Central Node)

Assuming that the relevant partial replica is Node 2, there are two possible scenarios based on its availability. If it is available, it

follows the replication process in Figure 6, which is analogous to inserting to the central node but with the addition of two steps at the end: (i) propagating the updated logs to all three nodes and (ii) releasing the exclusive locks on the central node and Node 2, signaling that the system is ready for another write transaction.

On the other hand, if Node 2 is unavailable, a background process is launched to monitor its status every second. Once the node has recovered, it will catch up on the missed transactions by referring to the pertinent logs stored in the two other nodes (Figure 7). In case of conflicts between the logs, the longer one is considered canonical. This choice is justified by the fact that logs record the accumulation of transactions; a similar idea is employed in the longest chain rule used in blockchain technology [7].



Figure 7. Catching Up After Node 2's Recovery

Returning to the start of the flowchart in Figure 4, if the central node is down at the beginning of the transaction, the insertion is routed to either Node 2 or 3 depending on the year. Assuming that it is routed to Node 2, the transaction is aborted if the said node is down or stays locked beyond the 100-second window. Otherwise, the insertion scheme follows an analogous flow to the one shown in Figure 5 albeit deferring the replication until the central node recovers. Upon its recovery, it will use the logs to catch up on the missed insertion, following an analogous logic as in Figure 7. The complete flowcharts are provided in Appendices I and J.

2.7.2 Updating and Deleting a Movie Entry

The application allows users to update and delete existing entries. In particular, the title, genre, rank, director, and two actors can be updated, with the ID serving as the basis for selecting the entry. Likewise, the entry for deletion is specified by supplying its ID. The SQL queries are given in Appendices K and L.

The algorithms for both updating and deletion follow the same core logic as that for insertion. However, there are some minor differences; for instance, these transactions also have to issue read operations to retrieve the year and identify in which partial replica the entry is stored. Furthermore, the log attribute `id_new_entry` remains unchanged since no new movie entry is added. The flowcharts for the update strategy are given in Appendices M to R while those for the delete strategy are in Appendices S to X.

2.8 Issues and Considerations

Aside from the design decisions mentioned in the preceding sections, writing the web application in JavaScript and interfacing it with MySQL via the `mysql` driver also entailed further issues and considerations. First, although security is an ancillary issue in this project, it may be important to note that a distributed database system can be vulnerable to SQL injection attacks [34]. Following the recommended approach in the documentation of `mysql` [11], this system's application layer uses parameterized SQL queries (with `?` as stand-ins for user-supplied values, as seen in Appendices G, H, K, and L) whenever possible to block this exploit.

Second, while most database management systems follow a multithreaded approach, JavaScript only has a single thread that is based on the event loop [18]. To reconcile these, the application's

source code extensively employs the language's asynchronous programming features that are also supported in the Node.js runtime environment on which the system runs. In this regard, the base code for a database insertion that incorporates the discussed concurrency control and recovery strategies features deeply nested callback methods, as shown in Listing 1.

Listing 1. Callback Base Code for Database Insertion

```
pool.getConnection(func(err, c) {
  c.ping(func(err) {
    c.beginTransaction(func(err) {
      log(func(query, param, result) {
        c.query(func(query, param, result) {
          conn.commit(func(err) {
            log(func(query, param, result) {
              // ...
            });
          });
        });
      });
    });
  });
});
```

In the actual implementation, Listing 1 was augmented to handle errors returned by the callbacks. Whenever possible, the transaction is rolled back; otherwise, the error is simply thrown. As can be surmised, this resulted in a markedly deep nesting of callback functions; colloquially referred to as “callback hell,” this is a widely acknowledged problem in JavaScript and in distributed systems [12], resolved via `async` and `await` functions [1].

In spite of this aforementioned issue related to asynchronous programming, it proved advantageous in creating the background process for monitoring the status of a crashed node (Listing 2). Combined with `setTimeout` and recursion, the method can, thus, be repeatedly invoked without blocking the other functions.

Listing 2. Background Monitoring of Crashed Node

```
function monitor() {
  if (c != undefined)
    c.ping(func(err) {
      t = setTimeout(monitor, 1000);
      if (!err) clearTimeout(t); // connected
    });
  else
    setTimeout(monitor, 1000);
}
```

Finally, the use of parameterized queries to prevent SQL injection (instead of the simpler concatenation of user-supplied values) also introduced an additional layer of complexity to parsing the statements recorded in the logs, as the user-supplied values (such as title and genre) have to be first extracted from the stored query; to this end, utility methods that invoke string slicing were written.

2.9 Application Use Case

The implementation of the application as a distributed database allows for the optimization of database reads and increases its availability in the event of node connectivity issues and crashes. The application is particularly useful in cases where one system holds a centralized copy of the data while other systems assist in distributing the load by storing partially replicated fragments.

To illustrate, suppose a movie rental company maintains a catalog of its available movies, as well as additional information in which customers are often interested, such as the genre, release year, director, and actors. The company has two main branches, one of which is frequented by customers who prefer renting older movies while the other is frequented by customers who mostly rent newer movies. To exploit these trends, the company maintains two partial replicas of the central node; the first replica

contains data on movies released before 1980 and is managed by the first branch, and the second contains data on movies released during or after 1980 and is managed by the second branch.

When customers search for information related to a movie in one of the branches, the application first examines the contents of the node managed by the local branch before accessing either the central node or the node managed by the other branch, depending on their availability (i.e., if the information they are looking for is not available in the local node). Maintaining a smaller amount of data in the branch nodes optimizes the search processes, as the application would only need to scan a smaller set of entries. Moreover, if the central node happens to crash or be rendered inaccessible, customers can still access the catalog using the data from the local node, and vice versa.

Meanwhile, when the company staff decides to add new movies, update details, or delete movies from the catalog, the application first updates the data in the central node before replicating the same operation on either of the local nodes depending on the release year of the movie entry. Similar to the searching function, maintaining replicas maximizes the system’s availability even in the event that a node becomes inaccessible.

For example, suppose the central node crashes while an update is being performed, prompting the system to redirect the transaction to the relevant local node (instead of rejecting it altogether). Since the application’s search operation is also routed to this local node, a customer querying the catalog can see the update reflected. To guarantee eventual consistency, this change is propagated to the central node when it comes back online. Similarly, if either of the local nodes crashes, the replication of the update to the relevant local node can be deferred without compromising availability.

3. Concurrency Control and Consistency

This section discusses the experiments conducted in relation to concurrency control and consistency. As explained in Section 2.6 and 2.7, the distributed database system employs a lock-based mechanism, where read transactions are non-locking but write transactions hold and request exclusive locks. However, since reading is permissible while a write transaction is ongoing, it is imperative to determine and enforce the proper isolation level.

3.1 Experiment and Simulation Setup

An experiment was conducted for the purpose of determining the proper isolation level. Note that changing the transaction level globally requires an administrator privilege; ergo, it cannot be switched from only the application layer. To this end, a parameter group was created in the Amazon RDS interface [13], with the autocommit parameter set to 0 and transaction_isolation parameter set to the isolation level being tested (Figure 8).

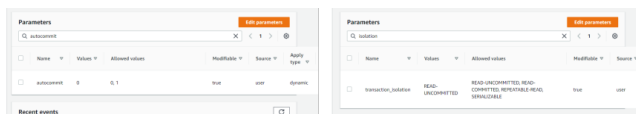


Figure 8. Amazon RDS Parameter Group

Disabling the autocommit parameter is necessary to prevent SQL statements from forming single transactions on their own and to allow them to be included as part of multi-statement transactions. To evaluate the concurrency control and consistency, the deployed web application was opened on three different machines situated in separate geographical locations. Three cases were simulated, with each experiment conducted thrice:

First, all transactions are reading. The three end-users searched for the entry with ID 174202 (arbitrarily chosen) at the same time. Since this read operation is lightweight and executes in less than a second under normal internet connectivity conditions, a sleep statement was inserted to trigger an artificial delay. The resulting interleaving of the transactions is shown in Table 3.

Table 3. Interleaving (All Transactions are Reading)

User 1	User 2	User 3
START TRANSACTION; SELECT * FROM movies WHERE id = 174202;		
DO SLEEP(20); COMMIT;	START TRANSACTION; SELECT * FROM movies WHERE id = 174202; DO SLEEP(20); COMMIT;	START TRANSACTION; SELECT * FROM movies WHERE id = 174202; DO SLEEP(20); COMMIT;

Second, all transactions are writing. The three end-users updated the movie entry with ID 174202 at the same time, albeit supplying different values for its title (as underscored in Table 4). Similar to the previous case, a sleep statement was inserted to trigger an artificial delay. In order to test the lock wait timeout mechanism, two subcases were considered:

- The exclusive locks are released within the lock wait window. This was simulated by setting the number of seconds the thread is put to sleep to a value that is less than 100 seconds (set to 10 in the experiments).
- The exclusive locks are not released within the lock wait window. This was simulated by setting the number of seconds the thread is put to sleep to a value that is above 100 seconds (set to 200 in the experiments).

Table 4 shows the resulting interleaving of the transactions. The parameter of DO SLEEP() depends on which subcase is tested; for brevity, placeholders are used for the user-supplied values.

Table 4. Interleaving (All Transactions are Writing)

User 1	User 2	User 3
START TRANSACTION; UPDATE movies <u>SET title = aa,</u> genre = yy, ... WHERE id = 174202;		
DO SLEEP(xx); COMMIT;	START TRANSACTION; UPDATE movies <u>SET title = bb,</u> genre = yy, ... WHERE id = 174202; DO SLEEP(xx); COMMIT;	START TRANSACTION; UPDATE movies <u>SET title = cc,</u> genre = yy, ... WHERE id = 174202; DO SLEEP(xx); COMMIT;

Lastly, one transaction is writing and two other transactions are reading. An end-user updated the entry with ID 174202 while two other end-users read the same entry at the same time. Similar to the other cases, a sleep statement was inserted to trigger an artificial delay. Specifically, the experiments sought to simulate a dirty read; thus, the read transactions took place between the execution and commit steps of the write transaction [37]. The interleaving of the transactions is provided in Table 5; for brevity, placeholders are used for user-supplied values.

Table 5. Interleaving (Transactions are Reading and Writing)

User 1	User 2	User 3
START TRANSACTION; UPDATE movies SET title = aa, genre = yy, ... WHERE id = 174202;		
DO SLEEP(20);	START TRANSACTION; SELECT * FROM movies WHERE id = 174202; COMMIT;	START TRANSACTION; SELECT * FROM movies WHERE id = 174202; COMMIT;
COMMIT;		

Since the application layer of the distributed database system bundles the search functionality inside a transaction, i.e., the SELECT queries in Appendices F and G are always enclosed inside the beginTransaction() callback, it is guaranteed that there is only one read operation inside a transaction. This design decision precludes the possibility of having non-repeatable and phantom reads since both these anomalies require multiple read statements within the same transaction [4]. Hence, the only read anomaly simulated in the experiments was a dirty read.

3.2 Results and Validation

For validation, the actual results of each set of transactions were recorded and compared against the expected results for each of the four InnoDB isolation levels: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. The result sets were checked through both the deployed web application and MySQL Workbench. The complete test scripts are provided in a separate document accompanying this paper.

3.2.1 All Transactions are Reading

For all four isolation levels, the experiments showed that the system permits concurrent read transactions. Moreover, there were no inconsistencies in the result set, i.e., all three end-users were able to retrieve the information relating to the movie entry with ID 174202. This is aligned with the expected results since the SELECT statements are nonlocking and do not mutate the database contents; technically, the isolation level is immaterial.

3.2.2 All Transactions are Writing

For all four isolation levels, the experiments showed the same results, as summarized in Table 6. Essentially, the lock-based mechanism implemented at the application layer resulted in the queuing and sequential execution of the write transactions.

When the exclusive locks were released within the 100-second lock wait timeout window, the write transactions still pushed through, following the order in which they were received by the application. However, when the locks were not released within the said window, only the first transaction (i.e., the one holding the exclusive lock) was committed; the other two transactions (i.e., those requesting the lock) were rejected, and the users were prompted to try again at another time since the servers are busy.

Table 6. Results (All Transactions are Writing)

Case	Behavior
Exclusive locks are released within the lock wait timeout window.	The transactions issued by Users 1, 2, and 3 were executed in this order. <i>The title of the movie entry was changed to cc.</i>
Exclusive locks are not released within the lock wait timeout window.	The transaction issued by User 1 executed. The transactions issued by Users 2 and 3 were rejected. <i>The title of the movie entry was changed to aa.</i>

This is in consonance with the expected results following the flow of the algorithm described in Section 2.7; since the execution of write transactions is handled via locking, the isolation level is technically immaterial. The results also showed that the design decision and implementation of the lock wait timeout mechanism prevented lock contention issues. For comparison, the same approach is followed by major database management systems, including MySQL [20], PostgreSQL [33], and IBM Db2 [14].

3.2.3 One Transaction is Writing and Two Other Transactions are Reading

Unlike in the previous cases, the results are dependent on the set isolation level, as presented in Table 7.

Table 7. Results (One Transaction is Writing and Two Other Transactions are Reading)

Isolation Level	Behavior
READ UNCOMMITTED	<i>Dirty read:</i> The search transaction returns the updated values before the update transaction is committed.
READ COMMITTED	The search transaction returns the original value (i.e., the value prior to the execution of the update statement).
REPEATABLE READ	
SERIALIZABLE	If autocommit is disabled, a lock contention issue occurs. Otherwise, the search transaction returns the updated values <i>after</i> the update transaction is committed.

These results are consonant with the expected results: dirty read is blocked at all isolation levels except at the lowest tier. Under the hood, the READ COMMITTED isolation level takes advantage of read views, which are snapshots used by InnoDB to save read-only copies of the data; a new read view is generated after a consistent read operation [20]. REPEATABLE READ also utilizes snapshots but refers only to the first snapshot generated by the first consistent read while SERIALIZABLE forces the transactions to emulate a serial execution.

Note that, in performing the experiments for the `SERIALIZABLE` isolation level, disabling `autocommit` resulted in a lock contention issue. MySQL's reference manual [20] explains that, at this isolation level, turning `autocommit` off converts ordinary `SELECT` statements to locking and blocking reads, thus conflicting with the exclusive lock held by the update transaction [23].

Since only dirty reads have to be blocked via the isolation level and the other read anomalies are blocked by the design of the application layer, **it suffices to set the isolation level of this distributed database system to `READ COMMITTED`**. For reference, `READ COMMITTED` is also the default isolation level of some major database management systems, including Microsoft SQL Server [16] and PostgreSQL [32].

4. Global Failure Recovery

This section discusses the experiments conducted in relation to global failure recovery. As explained in Sections 2.5 to 2.7, the distributed database system employs a statement-based logging approach for the replication and recovery of write transactions. Furthermore, in the event of the unavailability of the central node, the application layer is programmed to redirect the request to a partial replica; ergo, a transaction is expected to be rejected only if both the central node and the relevant partial replica are down.

4.1 Experiment and Simulation Setup

The foremost challenge in setting up the experiments to validate the system's global failure recovery strategy is simulating a crash. Since the database is deployed on a cloud service, the machine's operating system cannot directly interact with a database instance or issue an explicit crash directive [36]. To this end, Sharif [36] suggests some workarounds, such as overflowing the buffer and forcing a reboot by flushing the table with a read lock.

For the purposes of this project, failure is induced via the `mysql` API method `destroy()`; this triggers an immediate termination of the underlying socket regardless of the existence or the status of queued processes or events. This also ensures that the terminated connection will not accept any incoming callback [11], which also matches the expected behavior when a node crashes unexpectedly or becomes unavailable. The `destroy()` method is then injected into the code depending on the failure scenario.

For the read transactions, the gist of the failure scenarios is having a node (or a set of nodes) unavailable during the transaction. To expand the breadth of the test coverage, all seven (7) possible combinations of unavailable nodes were considered: the three cases where only one node is unavailable, the three cases where two nodes are unavailable, and the single case where all three nodes are unavailable. To simulate these, a `destroy()` method is invoked at the start of the transaction.

For the write transactions, six (6) cases are considered. Recovery occurs by establishing a new connection that is taken from the connection pool via the API method `createConnection()` and directing it towards the crashed node. In the experiments, three down-time windows were simulated: recovery was scheduled five (5), sixty (60), and three hundred (300) seconds after the crash. For each case, the experiment was repeated three times.

- **Case 1: Failure to write to the central node when attempting to replicate the transaction from Node 2 or Node 3.** Reiterating the discussion in Sections 2.5 to 2.7, the application's default behavior is to write first on

the central node before replicating to the relevant partial replica; consequently, replicating from a partial replica will be triggered only if the central node is down.

Therefore, `destroy()` was invoked before the start of the transaction to deliberately crash the central node and invoked anew when replication to the central node commences (i.e., at the starting juncture of Appendix J).

- **Case 2a: Failure to write to Node 2 when attempting to replicate the transaction from the central node.** The `destroy()` method was called after the transaction has been committed to the central node but before it could be replicated to the partial replica (i.e., at the starting juncture of Figure 6). The test input was a movie released prior to 1980.
- **Case 2b: Failure to write to Node 3 when attempting to replicate the transaction from the central node.** This is identical to the previous case, but the test input was a movie released from 1980 onwards.
- **Case 3: The central node is unavailable during the transaction, then eventually comes back online.** The `destroy()` method was invoked before the start of the transaction to deliberately crash the central node.
- **Case 4a: Node 2 is unavailable during the transaction, then eventually comes back online.** The `destroy()` method was invoked before the start of the transaction to deliberately crash Node 2, and the test input was a movie released prior to 1980.
- **Case 4b: Node 3 is unavailable during the transaction, then eventually comes back online.** This is identical to the previous case, but the node crashed was Node 3 and the test input was a movie released from 1980 onwards.

These experiments were conducted for the three types of write transactions supported by the system (update, insert, and delete).

4.2 Results and Validation

The general approach to validating the recovery strategy consists of (i) a white-box examination of the Heroku logs (to monitor the processes being executed at the back-end) and the contents of the `log` table (accessed via MySQL Workbench) and (ii) a black-box test of the attributes related to the expected output or the contents of the database at each node. The black-box results were checked from both the deployed web application and MySQL Workbench.

The expected behavior pertinent to the former is based on the algorithms in Section 2 (and summarized in Table 10) while the expected results pertinent to the latter are presented in the succeeding subsections. The complete test suites and results are compiled in a separate document accompanying this paper.

4.2.1 Read Transactions

Since read transactions do not change the `log` table of any of the nodes, three attributes of multiple-entry result sets were instead checked to validate their correctness, the latter two of which are motivated by boundary value analysis:

- Cardinality of the result set and attributes of the results
- Correctness of the attributes of the entry with the lowest movie ID
- Correctness of the attributes of the entry with the highest movie ID

If the read transaction returns only a single entry (e.g., searching by ID), the correctness was ascertained by comparing the attributes of the displayed entry vis-à-vis the entered search criterion.

To illustrate, when a test search was performed using *Action* as the search criterion, the application returned 11574 entries for the case where all three nodes are available; Table 8 presents the cardinalities of the results sets depending on the availability of the nodes, and Appendix Y shows the attributes of the entries with the lowest and highest IDs. These matched the expected results based on the algorithm described in Section 2.6.

Table 8. Results After Searching for *Action*

Unavailable Node	Cardinality	Description
Central Node	11574	Full result set
Node 2	11574	Full result set
Node 3	11574	Full result set
Central Node & Node 2	7900	Partial result set
Central Node & Node 3	3674	Partial result set
Nodes 2 & 3	11574	Full result set
All three nodes	0	Empty result set

The third column of Table 8 provides a general description of the results, evincing that the system is able to return the maximal cardinality of the result set that is possible based on the available nodes: out of the seven cases simulated, four (4) yield a full result set, two (2) yield a partial set, and one (1) yields an empty set.

4.2.2 Write Transactions

For the black-box testing, the correctness of insert transactions was evaluated based on three attributes:

- Correctness of the attributes of the entry added to the database vis-à-vis the user input
- Cardinality of the *movies* table in the nodes (the total number of entries in Nodes 2 and 3 should be equal to the number of entries in the central node)
- Mutual exclusivity of the data in Nodes 2 and 3 (since they are horizontal fragments)

To illustrate, when a test insert of a movie released in 1970 was performed, the cardinalities of the three nodes changed depending on the failure scenario simulated, as seen in Table 9. For a basis of comparison, the central node, Node 2, and Node 3 had 173394, 92003, and 81391 entries, respectively, prior to this insertion.

Similarly, update transactions were evaluated based on the correctness of the new information stored in the database and its existence in the proper nodes; entries with year before 1980 should only be in the central node and Node 2 while those from 1980 onwards should only be in the central node and Node 3. Lastly, delete transactions were evaluated based on the absence of the entry from the database; it should no longer be retrievable via the web application and should not be stored in any of the nodes.

Table 9. Result Set Cardinality Based on Failure Scenario

Case	Central Node	Node 2	Node 3
1	173394 (while crashed)	92004	81391
2a	173395	92003 (while crashed)	81391
3	173395	92004	81391
4a	173395	92004	81391

The actual results of the experiments matched the expected results. Finally, in relation to the white-box examination of the Heroku logs and the *log* table, Table 10 summarizes the expected states of the nodes and logs based on the system's recovery strategy, which is orthogonal across the three write transactions (insert, update, and delete). For simplicity, Table 10 assumes Node 2 as the relevant partial replica. For all cases, the final check upon recovery was verifying the exact consistency of the *log* table and the logical consistency of the *movies* table across all the nodes.

Table 10. Expected States of the Nodes and Logs

Case	Expected States of the Nodes and Logs
1	<p>The transaction is directed to Node 2, where it is executed and committed, and its logs are updated.</p> <p>Hence, even though the central node fails during replication, a record of the transaction still persists in the (central node, Node 2) log, which is the log pertaining to the central node and is stored in Node 2 (cf. Appendices I, Q, and W).</p> <p>It can be verified that the central node is down by checking that the <i>mysql</i> API method <i>ping</i> throws an error. While it is down, <i>next_trans_commit</i> is behind <i>next_trans_record</i> in the (central node, Node 2) log (a copy of which is found in Appendix Z). Upon recovery, it can then use this log and the said markers to catch up (cf. Appendices J, R, and X).</p>
2a 2b	<p>The transaction is directed to the central node, where it is executed and committed, and its logs are updated.</p> <p>Hence, even though Node 2 fails during replication, a record of the transaction still persists in the (Node 2, central node) log (cf. Figure 5, Appendices N and T).</p> <p>While Node 2 is down, <i>next_trans_commit</i> is behind <i>next_trans_record</i> in the (Node 2, central node) log. Upon recovery, it can then use this log and the said markers to catch up and reach eventual consistency of database contents and logs (cf. Figure 6, Appendices O and U).</p>
3	The explanation for Case 1 applies as well.
4a 4b	The explanation for Cases 2a and 2b applies as well.

Handling Case 3 also justifies the inclusion of *id_new_entry* in the log as a computationally inexpensive technique for the partial replicas to generate a unique ID even without access to the central node. By extension, this also supports the design decision of transferring the responsibility of autoincrementing the *id* attribute from the database server to the application layer.

5. Discussion

The results of the experiments showed that the log-based approach allowed for system recovery and high availability while the combination of exclusive locks with the `READ COMMITTED` isolation level enabled concurrency while blocking transaction anomalies. However, apart from the design and implementation of the application layer, the platform on which the database is hosted also proved to be an important consideration in the setup of the simulations. For example, setting global isolation levels required administrator privilege; to work around this restriction, the created instances were first included in a parameter group, which was modified to configure the isolation level [5].

Another related attribute salient to the conduct of the simulations was the `autocommit` parameter. In MySQL, it is enabled by default [19], allowing for basic concurrency and recovery control abstracted to users of this database management system; however, using the system with `autocommit` enabled preemptively blocked dirty reads, as transactions could not be interleaved in such a way that a read would be executed before a currently executing write on the same entry is committed. Thus, it had to be toggled for the concurrency control experiments to be performed. After all, the goal of the application’s design is to serve as a proof of concept that, in theory, can also support heterogeneous setups.

Implementing concurrency control and recovery strategies, as opposed to using mechanisms built into existing database management systems, highlighted the complexities of distributed database system configuration. One such complexity is promoting fragmentation transparency; this was accomplished by providing access to the web application through a single URL and allowing users to perform read and write operations using a single set of input fields as shown in Figure 9. Abstracted to the end-users, determining which nodes to access is handled by the application layer, with the algorithm designed in light of the availability of the nodes and the comparative computational requirements of performing the operation.

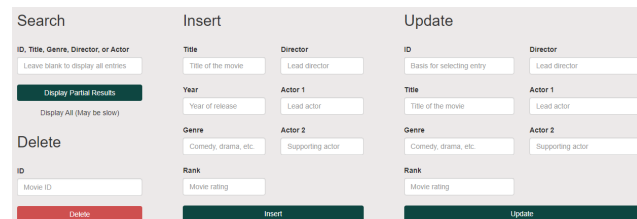


Figure 9. Fragmentation Transparency (Application Interface)

The use of the master-master replication strategy for the system topology also introduced some challenges. As highlighted by Elmasri and Navathe [10], algorithms maintaining concurrency across all three nodes—both for the actual values of the databases and those of the logs stored in each node—can quickly become complex; nevertheless, the management of redundant copies of the logs across all three nodes was still preferred to maximize persistence in the event of node failure. The logs were maintained akin to a decentralized system, similar to the distributed ledger technology recently popularized by blockchain [15].

While the system’s failure recovery strategy was implemented independent of server-specific solutions, concurrency control was implemented through the use of InnoDB isolation levels. Its success in this system, as evinced in the results of the experiments conducted, may serve as a proof of concept both for the validity

or utility of the distributed database management strategies in this application and for the versatility of implementing hybrid database management protocols compared to the high cost incurred by implementing all protocols independently and the restrictions caused by relying solely on existing systems [40].

6. Conclusion

The implementation of a distributed database system for a denormalized version of the IMDb served a twofold purpose: from a practical perspective, the resulting web application may serve as an eventually consistent and failure-resistant point of access to the IMDb dataset, which can be useful for film producers, vendors, and consumers. From a technical perspective, the implementation of various distributed database management concepts, as well as techniques associated with other data items (e.g., distributed ledgers) in the development of the web application evinces the utility of existing database management techniques and the potential benefit of integrating these with protocols and standards used for managing other types of data. Moreover, the assessment of their intended functionality vis-à-vis different concurrency and failure management issues served as an evaluation of the strengths and weaknesses of distributed database systems, as well as an exploration of important considerations involved in implementing concurrency and failure control that balances trade-offs between availability and consistency.

The evaluation of the distributed database system was done in two phases: first, three sets of transactions, comprising combinations of reading and writing transactions, were performed on the system at various transaction isolation levels to determine the possibility of read anomalies in concurrent operations. Second, different sequences of node crashes and recoveries were orchestrated at selected junctures of the replication to ascertain the effectiveness of the failure control mechanisms implemented in the system. The performance of the system was evaluated by comparing the actual results of these experiments to expected results prepared as part of the test scripts for the application.

As a result of the system design—which ensures that a transaction can only include a single `SELECT` query—the non-repeatable read and phantom read anomalies are excluded from the database transaction types that can be executed by end-users. Thus, the most appropriate isolation level for the system in terms of concurrency control was `READ COMMITTED`, which prevents dirty read anomalies while minimizing the locks set on the database, thus maximizing concurrency and availability [21].

Meanwhile, the system proved resilient against a variety of failure scenarios, including crashes that occurred while accessing the central node and the partially replicated nodes, as well as those that occurred while replicating transaction results to the central node or the partial replicas. In particular, the use of decentralized statement-based logs was proven to be an effective failure management strategy, as the statements can be extracted from the logs and executed once the crashed node becomes available. Additionally, the use of locks only for write operations in conjunction with the `READ COMMITTED` isolation level maximizes the concurrency of the system while preventing dirty reads. However, while the application was able to handle the simulated concurrency and failure control issues, its implementation can be improved by using modern JavaScript’s `async` and `await` functions to improve the maintainability of the source code and by exploring threading to augment the speed of the transaction execution pipeline.

7. References

- [1] Anderson, K. 2016. *Asynchronous programming in Javascript*. CSCI 5828: Foundations of Software Engineering. Retrieved from <https://books-library.net/files/download-pdf-ebooks.org-1506362349Sf7I0.pdf>
- [2] Amazon. n.d. *Amazon EBS volume types*. AWS. Retrieved from <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html>
- [3] Amazon. n.d. *Amazon virtual private cloud (Amazon VPC)*. AWS. Retrieved from <https://aws.amazon.com/vpc/>
- [4] Amazon. n.d. *Aurora MySQL isolation levels*. AWS. Retrieved from https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/AuroraMySQL.Reference.html?fbclid=IwAR1wZAFU_AuxLOFiAOjCOMCCpBDj7Dom8UJj2tSCbUUEzxNMxz_2EiTylk#AuroraMySQL.Reference.IsolationLevels
- [5] Amazon. 2020. *How do I enable functions, procedures, and triggers for my Amazon RDS MySQL DB instance?*. AWS. Retrieved from <https://aws.amazon.com/premiumsupport/knowledge-center/rds-mysql-functions/>
- [6] Bernstein, P.A. and Goodman, N. 1981. *Concurrency control in distributed database systems*. Computing Surveys, 13(2), 185-221. Retrieved from <https://people.eecs.berkeley.edu/~brewer/cs262/concurrency-distributed-databases.pdf>
- [7] Blum, E., Kiayias, A., Moore, C., Quader, S., and Russell, A. 2019. *The combinatorics of the longest-chain rule: Linear consistency for proof-of-stake blockchains*. In Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms. Retrieved from <https://epubs.siam.org/doi/10.1137/1.9781611975994.69>
- [8] Chromium. 2019. *Controlling access to powerful web platform features*. Retrieved from <https://chromium.googlesource.com/chromium/src/+/lkgr/docs/security/permissions-for-powerful-web-platform-features.md>
- [9] Crosby, M., Nachiappan, Pattanayak, P., Verma, S., and Kalyanaraman, V. 2015. *BlockChain technology*. Sutardja Center for Entrepreneurship & Technology Technical Report. Retrieved from <https://scet.berkeley.edu/wp-content/uploads/BlockchainPaper.pdf>
- [10] Elmasri, R., & Navathe, S.B. 2016. *Fundamentals of database systems (7th ed.)*. Pearson.
- [11] Github. *MysqLjs/mysql*. Retrieved from <https://github.com/mysqLjs/mysql>
- [12] Gómez, E.Z., López, P.G., and Mondéjar, R. 2015. *Continuation complexity: A callback hell for distributed systems*. In: Hunold S. et al. (eds) Euro-Par 2015: Parallel Processing Workshops. Euro-Par 2015. Lecture Notes in Computer Science, vol 9523. Springer, Cham. https://doi.org/10.1007/978-3-319-27308-2_24
- [13] Huawei. 2021. *Applying a parameter template*. Retrieved from https://support.huaweicloud.com/intl/en-us/usermanual-rds/rds_05_0018.html
- [14] IBM. 2014. *Lock waits and timeouts*. Retrieved from <https://www.ibm.com/docs/en/db2/10.5?topic=management-lock-waits-timeouts>
- [15] Marco Polo Network. 2018. January 30. *Difference Blockchain and DLT*. Retrieved from <https://marcopolonetwork.com/articles/distributed-ledger-technology/>
- [16] Microsoft. 2021. *Set transaction isolation level (transact-SQL)*. SQL Docs. Retrieved from <https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-ver15>
- [17] Mozilla. 2022. *File system access API*. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/File_System_Access_API
- [18] Mozilla. 2021. *Javascript*. Retrieved from <https://developer.mozilla.org/en-US/docs/Web/JavaScript>
- [19] MySQL. n.d. *13.3.1 Start transaction, commit, and rollback statements*. MySQL 8.0 Reference Manual. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/commit.html>
- [20] MySQL. n.d. *15.21.5 InnoDB Error Handling*. MySQL 8.0 Reference Manual. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/innodb-error-handling.html>
- [21] MySQL. n.d. *15.7.2.1 Transaction Isolation Levels*. MySQL 8.0 Reference Manual. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html>
- [22] MySQL. n.d. *15.7.2.3 Consistent Nonlocking Reads*. MySQL 8.0 Reference Manual. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/innodb-consistent-read.html>
- [23] MySQL. n.d. *16.7.2.4 Locking Reads*. MySQL 9.0 Reference Manual. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/innodb-locking-reads.html>
- [24] MySQL. n.d. *17.1.1 Binary log file position based replication configuration overview*. MySQL 8.0 Reference Manual. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/binlog-replication-configuration-overview.html>
- [25] MySQL. n.d. *17.1.3.1 GTID format and storage*. MySQL 8.0 Reference Manual. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/replication-gtids-concepts.html>
- [26] MySQL. n.d. *17.2.1.1 Advantages and disadvantages of statement-based and row-based replication*. MySQL 8.0 Reference Manual. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/replication-sbr-rbr.html>
- [27] MySQL. n.d. *18.1.1.1 Source to replica replication*. MySQL 8.0 Reference Manual. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/group-replication-primary-secondary-replication.html>
- [28] MySQL. n.d. *MySQL port reference tables*. MySQL 8.0 Reference Manual. Retrieved from

<https://dev.mysql.com/doc/mysql-port-reference/en/mysql-p-orts-reference-tables.html>

- [29] Nichter, D. 2013. *MySQL 5.5 lock wait timeout: patience is a virtue, and a locked server*. Percona. Retrieved from https://www.percona.com/blog/2013/02/28/mysql-5-5-lock-wait_timeout-patience-is-a-virtue-and-a-locked-server/
- [30] NPMJS. 2020. *MySQL*. Retrieved from <https://www.npmjs.com/package/mysql>
- [31] OpenJS Foundation. 2022. *Node.js is a JavaScript runtime built on Chrome's v8 JavaScript engine*. Retrieved from <https://nodejs.org/en/>
- [32] PostgreSQL. n.d. 9.3. *Read committed isolation level*. Retrieved from <https://www.postgresql.org/docs/7.2/xact-read-committed.html>
- [33] PostgreSQLCo. n.d. *Lock_timeout*. Retrieved from https://postgresqlco.nf/doc/en/param/lock_timeout/
- [34] Devi, R., Venkatesan, R., and Koteeswaran, R. 2016. *Detection and preclusion of SQL injection in a distributed environment using contemporary approach*. International Journal of Pharmacy and Technology 8(4). 23171-23180.
- [35] Salesforce. 2022. *Heroku*. Retrieved from <https://www.heroku.com/>
- [36] Sharif, A. 2017. November 17. *How to fail or crash your MySQL instances for testing*. Severalnines. Retrieved from <https://severalnines.com/database-blog/how-fail-or-crash-your-mysql-instances-testing>
- [37] Silberschatz, A., Korth, H.F. and Sudarshan, S. 2020. *Database System Concepts (7th ed.)*. McGraw-Hill Education, New York.
- [38] Son, S.H. and Choe, K.M. 1988. Techniques for database recovery in distributed environments. *Information and Software Technology*, 30(5), 285-294. DOI: 10.1016/0950-5849(88)90021-3
- [39] Vavpetič, A., Perovšek, M., Kranjc, J., and Lavrač, N. 2015. ILP Datasets. (April 2015). Retrieved from http://kt.ijs.si/janez_kranjc/ilp_datasets/
- [40] Vincent, S. 2018. *Database management and design*. Retrieved from https://stevevincent.info/CIS2210_2018_12.htm
- [41] Yue, P. and Tan, Z. 2018. 1.06 - GIS databases and NoSQL databases. *Comprehensive Geographic Information Systems*, 50-79. <https://doi.org/10.1016/B978-0-12-409548-9.09596-8>

8. Appendix

Appendix A. Denormalization of the movies Table

```
SET session group_concat_max_len=1500000;

-- Denormalize into a single table
CREATE TABLE exploded_movies
SELECT movies.id, title, year, genre, `rank`,
CONCAT(directors.first_name, CONCAT(" ",
directors.last_name)) AS director,
CONCAT(actors.first_name, CONCAT(" ",
actors.last_name)) AS actor
FROM movies
```

```
JOIN movies_directors
ON movies.id = movies_directors.movie_id
JOIN directors
ON directors.id =
movies_directors.director_id
JOIN roles
ON roles.movie_id = movies.id
JOIN actors
ON actors.id = roles.actor_id

JOIN movies_genres
ON movies.id = movies_genres.movie_id
ORDER BY id;

-- Implode based on multivalued attributes
CREATE TABLE movies
SELECT id, title, year, genre, `rank`,
director,
SUBSTRING_INDEX(GROUP_CONCAT(DISTINCT
actor), ',', 1) as actor1,
SUBSTRING_INDEX(GROUP_CONCAT(actor), ',',
-1) as actor2
FROM exploded_movies
GROUP BY id;

-- Retain only one genre and two actors
UPDATE movies
SET actor2 = CASE
WHEN actor1 = actor2 THEN actor2 = NULL
ELSE actor2
END
```

Appendix B. Amazon RDS Configuration

Attribute	Value
Database Engine	InnoDB – MySQL 8.0.27
Instance Class	db.t2.micro
Virtual Central Processing Unit	1 core
Random Access Memory	1 GB
Storage Type	General-purpose solid-state drive (gp2)
Storage Capacity	20 GiB
Maximum Storage Threshold	1000 GiB
Database Port	3306

Appendix C. Inbound Rule for Allowing External Application Access to the Remote Database

Attribute	Value
Type	MySQL/Aurora
Protocol	TCP
Port Range	3306
Source (CIDR Block)	0.0.0.0/0

Appendix D. Instantiation of Connection Pool

```
const centralPool = MySQL.createPool({
  connectionLimit: 20,
  host: process.env.CENTRAL_URL,
  port: process.env.DB_PORT,
  user: process.env.CENTRAL_USERNAME,
  password: process.env.CENTRAL_PASSWORD,
  database: DATABASE,
  connectTimeout: 30000
});
```

Appendix E. Sample Log

Attribute	Value
node_id	1
lock_status	0
next_trans_record	3
next_trans_commit	3
id_new_entry	174204
statements	0 INSERT INTO movies (id, title, year, genre, `rank`, director, actor1, actor2) VALUES (174201,'testEntry', 1909, 'Comedy', 8, 'Lander', 'Hylene', 'Mark') 1 INSERT INTO movies (id, title, year, genre, `rank`, director, actor1, actor2) VALUES (174202,'testEntry1', 1976, 'Comedy', 8, 'Lander', 'Hylene', 'Mark') 2 INSERT INTO movies (id, title, year, genre, `rank`, director, actor1, actor2) VALUES (174203,'testEntry3', 2013, 'Comedy', 8, 'Lander', 'Hylene', 'Mark')

Appendix F. SQL Code for Retrieving All Movie Entries

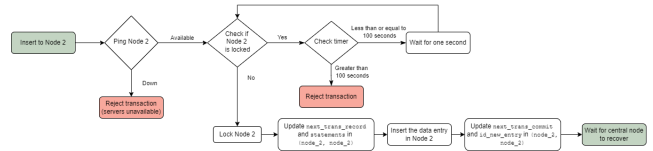
```
SELECT * FROM movies
```

Appendix G. SQL Code for Retrieving Entries Based on Search Criterion

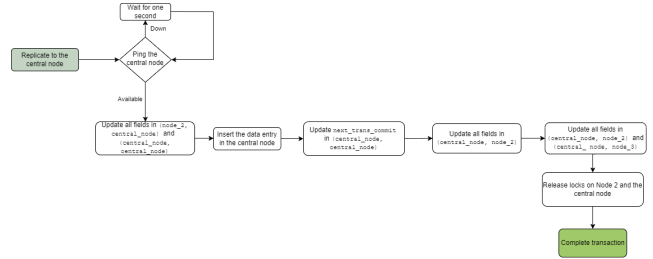
```
SELECT *
FROM movies
WHERE id = ? OR title LIKE ? OR genre LIKE ?
      OR director LIKE ? OR actor1 LIKE ?
      OR actor2 LIKE ?
```

Appendix H. SQL Code for Inserting Entries

```
INSERT INTO movies (title, year, genre, `rank`,
  director, actor1, actor2)
VALUES (?, ?, ?, ?, ?, ?, ?)
```



Appendix I. Inserting to Node 2 (Central Node is Down)



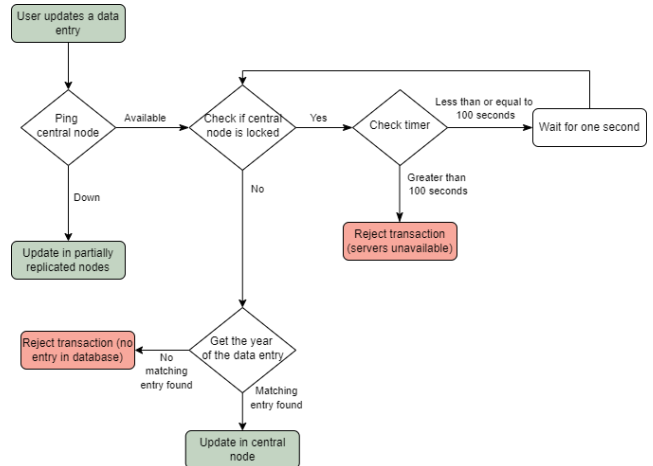
Appendix J. Replicating the Insert to the Central Node (from Node 2)

Appendix K. SQL Code for Updating Entries

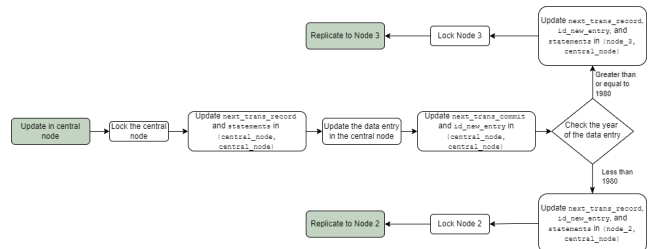
```
UPDATE movies
SET title= ?, genre = ?, `rank` = ?,
    director = ?, actor1 = ?, actor2 = ?
WHERE id = ?
```

Appendix L. SQL Code for Deleting Entries

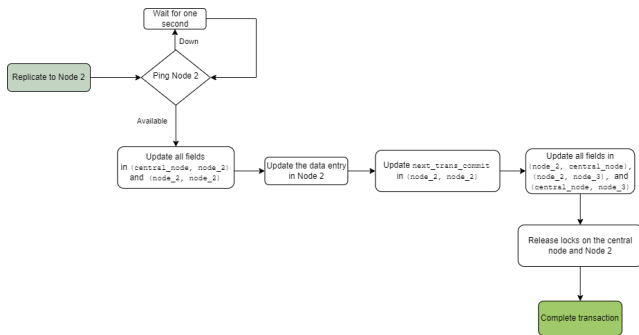
```
DELETE FROM movies WHERE id = ?
```



Appendix M. High-Level Algorithm for Updating an Entry



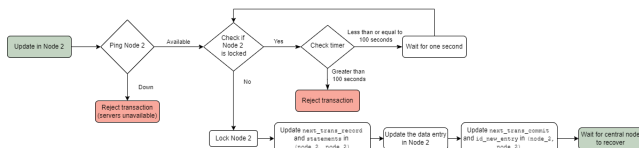
Appendix N. Updating in the Central Node



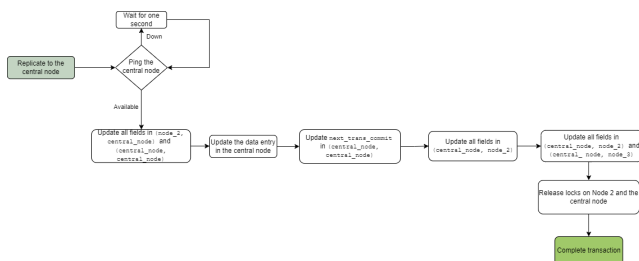
Appendix O. Replicating the Update to Node 2 (from the Central Node)



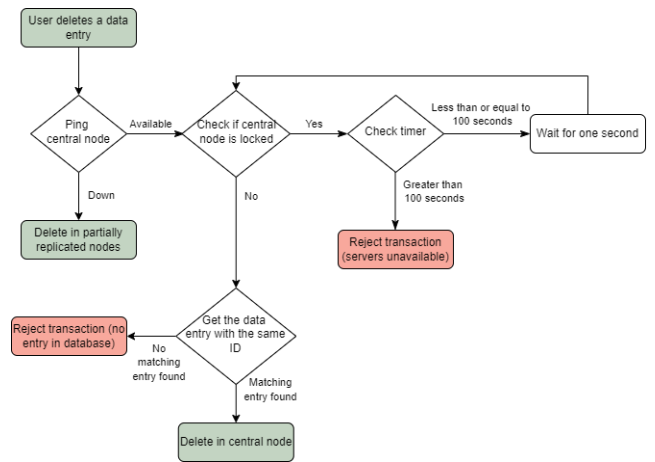
Appendix P. Determining the Relevant Partial Replica



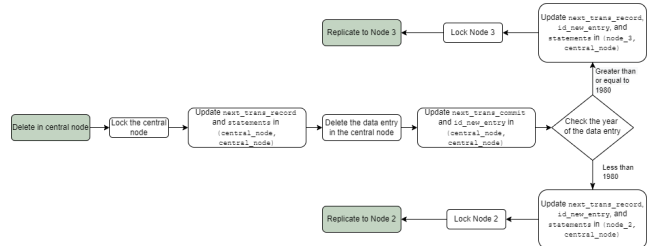
Appendix Q. Updating in Node 2 (Central Node is Down)



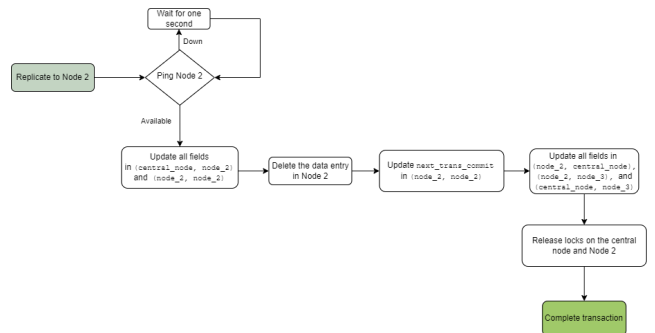
Appendix R. Replicating the Update to the Central Node (from Node 2)



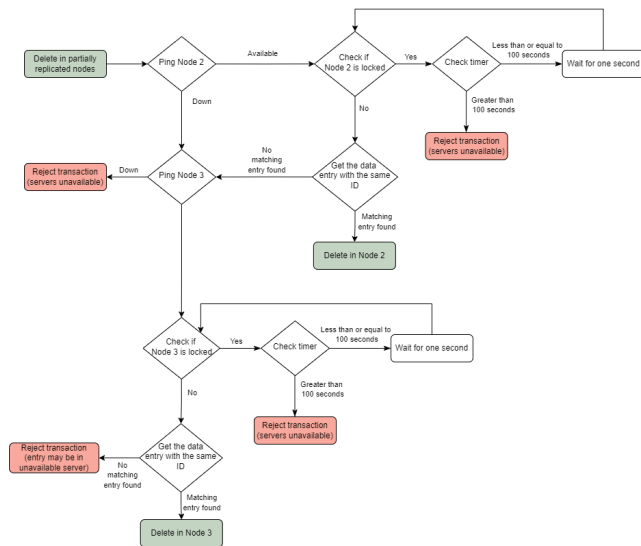
Appendix S. High-Level Algorithm for Deleting an Entry



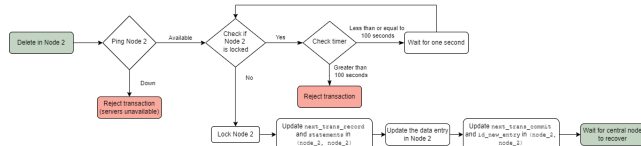
Appendix T. Deleting from the Central Node



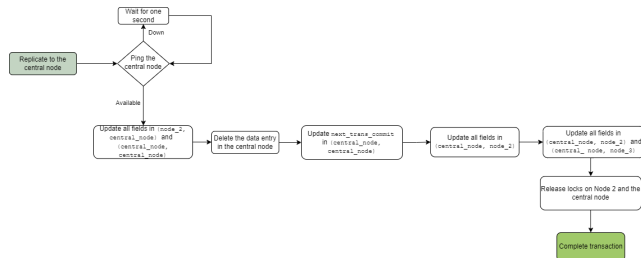
Appendix U. Replicating the Delete to Node 2 (from the Central Node)



Appendix V. Determining the Relevant Partial Replica



Appendix W. Deleting from Node 2 (Central Node is Down)



Appendix X. Replicating the Delete to the Central Node (from Node 2)

Appendix Y. Attributes of the Entries with the Lowest and Highest IDs After Searching for Action

Attribute	Entry with Lowest ID	Entry with Highest ID
id	18	173317
title	\$windle	ltimo narco, El
year	2002	1992
genre	Action	Action
rank	5.4	NULL
director	K.C. Bascombe	Víctor Herrera Zenil
actor1	Alain Goulem	Edgardo (I) Gazcón
actor2	Jack Daniel Wells	Patricia Rivera

Appendix Z. (central node, Node 2) Log After Inserting an Entry (Global Failure and Recovery Case 1)

Attribute	Value
node_id	1
lock_status	0
next_trans_record	3
next_trans_commit	2
id_new_entry	174204
statements	<pre> 0 INSERT INTO movies (id, title, year, genre, `rank`, director, actor1, actor2) VALUES (174201,'testEntry', 1909, 'Comedy', 8, 'Lander', 'Hylene', 'Mark') 1 INSERT INTO movies (id, title, year, genre, `rank`, director, actor1, actor2) VALUES (174202,'testEntry1', 1976, 'Comedy', 8, 'Lander', 'Hylene', 'Mark') 2 INSERT INTO movies (id, title, year, genre, `rank`, director, actor1, actor2) VALUES (174203,'testEntry3', 1970, 'Comedy', 8, 'Lander', 'Hylene', 'Mark') </pre>