University of Glasgow | School of Computing Science

Towards transaction processing in big data systems

Kshitiz Bisht (2487508B)

April 14, 2023

ABSTRACT

Big data is generally a collection of structured, semi-structured, or non-structured data. These data are used in machine learning, business analysis, and security. Apache Hadoop is an open-source project used to analyze and store petabytes of data. Hadoop uses Hadoop distributed file system (HDFS) [23, 22] to store files in a Hadoop environment. Hadoop's advantages are high performance, cost-effectiveness, fault tolerance, high availability, and many more. These systems have lots of advantages, but one thing that separates them from traditional databases is transaction support. Transaction support is an essential aspect of data management in big data systems. The ability to provide transactions ensures that data consistency and reliability are maintained, even when dealing with large and complex datasets. Over the past couple of years, there has been a significant increase in the use of big data systems, such as Apache Hadoop, Apache Cassandra, and Apache HBase, which has led to a growing need for transactional support in these systems. This paper aims to propose an idea for transaction support in one of the famous big data systems, HBase. This paper highlights the importance of transactional support in big data systems and provides a comprehensive understanding of the challenges and opportunities in this area.

1. INTRODUCTION

Over the past couple of years, NoSQL[10, 24] databases have gained popularity. NoSQL databases were designed to overcome the limitations of traditional databases. One of the limitations was flexibility. NoSQL databases do not enforce a fixed schema. For example, adding a column in the traditional database would require changing the schema of the whole table, which could lead to increased memory usage as there would be a need to add an extra value in that column for other rows. But a NoSQL database could do this by adding a column. Another limitation is scalability. Traditional databases were single-node, which means that as data volume increased, databases required more powerful hardware to process. This led to an increase in expenses to maintain the database. Even the distributed databases did not scale well due to their consistency-first design. As NoSQL databases gained popularity, people realised there was a need for transaction support as transaction support is a critical part of the enterprise-level application. Transactions are sequences of operations executed together as a single unit, such that all the operations in a transaction are executed successfully or fail. To implement transactions in Big data systems, apache Hive [7] introduced transaction support. Hive supports transactions with the help of ORC files. Hive created deltas (small files) to represent a change. When many deltas are created, it does minor compaction to merge them into a single file. When these files are large enough, it performs major compaction to show changes.

Hive is a successful example of a big data system that can perform transactions which can be taken as a motivation to design transaction support for other databases. So we designed transaction support for one of the famous NoSQL databases, HBase. HBase was created based on Google's Big Table [8]. HBase is an open-source project. A distributed database handles large amounts of data. It is well-suited for use cases that require real-time access to large datasets, such as social media platforms, financial systems, and log data analysis. HBase provides several advantages over traditional databases, such as horizontal scalability, fault tolerance, and high availability. It can easily handle petabytes of data, distribute load across multiple nodes, and recover from node failures without losing data. Even with such powerful features, HBase provides limited support for transactions. Transactions in HBase provide row-level atomicity, which means that each transaction only operates on a single row of data and provides atomicity, consistency, and isolation guarantees for that row. HBase does not provide full ACID transactions like traditional relational databases[1].

Keeping this in mind, the project aims to ***develop a wrapper around the Hbase library to enable multi-row transactions in Hbase***. By exploring the aim, the paper provides several contributions

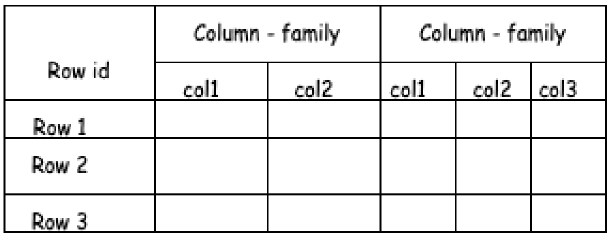
- Literature survey in the transactional database (Section 2). The concepts discussed in this section will discuss each part of the project, ideas related to it, and decisions taken from it.
- Design of library to implement transaction support (Section 3). The section will provide major design decisions taken in the project to implement the transaction support. It discusses the transaction monitor table to keep track of transactions, locks to lock the row in the table, deadlock detection, and staging writes.
- Algorithm for read and write operations (Section 3). The section will show the working of read and write operation that guarantees transaction support.
- detailed experiment (Section 4). The transaction support will be tested against YCSB. YCSB is an industry-standard used to benchmark the database. With the help of the YCSB, the database would be tested for latency, memory usage and batch operations. Apart from YCSB, the database will also be tested against data anomalies.

- Future work (section 5). Considering the limitations and assumptions taken during the project, the paper will explore future work that could help to introduce more efficient transaction support in big data systems.

2. BACKGROUND

2.1 HBase

HBase[6, 15, 25, 27] is an open-source NoSQL distributed database. HBase is modelled after Google's Big Table and runs on the Hadoop distributed file system (HDFS). HBase is a column-oriented database that stores data in a table. The tables are sorted by row. The row has a row key or row id that helps to identify the data in the table. The row can be described by several columns. Similar columns can be grouped to form a column family. E.g., in the table shown in fi1, the *col1* and *col2* are the column quantifier grouped together in a column called 'column-family'. The tables are divided into regions. A region is a contiguous range of table rows stored in a separate region server. Regions are dynamically split and merged to balance the load and ensure even data distribution across region servers. HBase uses HDFS for replication and fault tolerance. Each region is replicated across multiple data nodes in HDFS to ensure data availability in case of node failures. HBase also assigns each region to one or more region servers responsible for serving read and write requests for that region.



Row id	Column - family		Column - family		
	col1	col2	col1	col2	col3
Row 1					
Row 2					
Row 3					

Figure 1: Table-based format of HBase

Different components of Hbase are **HMaster**, which is the main server in Hbase architecture. It is responsible for assigning regions to the different region servers. It also monitors the servers' workloads and can reassign table regions to achieve load balancing. HMaster also monitors the health of all the other region servers; if a region server fails, it reassigns the regions to other region servers. It can also change the schema and other metadata operations requested by the client, handling create or delete operations and monitoring the health of the region server. **Region servers** hold the table's regions. It handles basic operations such as create, read, update or delete (CRUD). It has several components, such as a write-ahead log(WAL), Memstore, block cache, and HFile. WAL contains details about the database changes that have not been committed yet. WAL is done for fault tolerance Memstore is a write cache that holds the data before it is flushed to disk. Block cache is a read cache that holds the recently read data, and HFile is the file that stores the data after it is flushed. **Zookeeper** is an open-source project responsible for handling communication within Hbase. It keeps track of all region servers and HMaster. It is also responsible for keeping track of failed server or network partitions. Zookeeper is the first point of contact whenever a client makes a read-and-write request, as it redirects the client to the server. Whenever a read request is made to Hbase, it is sent to the zookeeper first, as it is the first point of contact in this architecture. The zookeeper then gives the location of the META table to the client. The META table keeps a list of all the regions. This table has a binary tree-like structure and stores the information in key-value pairs where the key is the region start key, the region id and the value is the region server. The client then gets to the META table, where it gets the location of the region server holding that data. The client then goes to the region server and checks the block cache first, as it may contain the data from the client first. If it finds the data server will return the data. If it does not find the data, then it checks the Memstore. If the data is present in Memstore, it will return it. If not then the last option is to search the HFile. If the data is actually available in the system, it will be available in the HFile. After the search, the data is written in the block cache and returned to the client.
When a written request is made, the request is first written in WAL. Once the entry has been made in WAL, the data is written in Memstore. Memstore acts as a buffer before the data is flushed into the disk. Once the memstore reaches its threshold, the data is flushed into HFile.

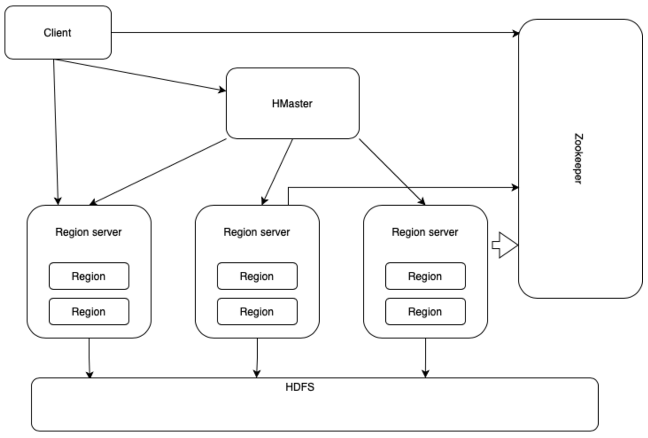


Figure 2: HBase architecture

2.2 ACID Transaction

ACID[18, 31, 29] transactions are properties that make a transaction reliable. It makes sure that the database always remains in a consistent state. ACID stands for four properties: **Atomicity**, meaning that all transaction operations (read, write, update and delete) are treated as a single unit. Either the entire statement or nothing will execute, preventing data loss. **Consistency**, it means that transactions made in the table are done in a defined way which could not lead to inconsistency in future. **Isolation** ensures that the transactions don't interfere with each other. Each transaction executes in a way that it is the only transaction. **Durability** means that the transaction history will be maintained even in case of system failure.

2.3 Concurrency control mechanism

The concurrency control[18] mechanism ensures that the database offers transactional properties. It regulates the database to ensure that transactions do not interfere with each other and accesses the data in a way that maintains the consistent state of the database. Some of the common concurrency control mechanisms are as follows

2.3.1 2 Phase locking

It is a widely used concurrency control mechanism in database systems to ensure data consistency and reliability. In 2PL[30, 21, 11], locks restrict access to shared resources and prevent conflicting updates that could lead to data inconsistencies. In the first phase, known as the "growing phase," the transaction requests a lock on a resource before accessing it. Once the lock is acquired, the transaction can perform its operations. During this phase, locks can be acquired but not released until the transaction reaches the second phase. In the second phase, known as the "shrinking phase," the transaction releases the locks it acquired in the first phase, allowing other transactions to access the resource. Once all locks are released, the transaction is said to have been committed, and its updates become permanent.

2.3.2 Optimistic concurrency control

It is another mechanism used in database management systems to control access to shared resources. In OCC[16, 30], transactions do not acquire locks before accessing shared resources. Instead, transactions read the needed data and store the values they will update. Before committing, a transaction verifies that other transactions have not updated the values it reads during its execution. If no conflicts are detected, the transaction commits, and its updates become permanent. However, if a conflict is detected, the transaction rolls back and restarts with the updated data. OCC relies on the assumption that conflicts are rare.

2.3.3 Timestamp ordering

This is yet another mechanism where each transaction is assigned an increasing timestamp[20, 17, 30]. The timestamp could be a system clock or a counter. Transactions read and write without acquiring locks. Instead, every object is tagged with the timestamp of the last transaction that attempts to write or read it. The read-and-write operation occurs as follows; T/O Read - if the write timestamp of an object is greater than the transaction's timestamp, then the transaction aborts to prevent the transaction from accessing the objects in the future. Else, the transaction reads the object and updates the timestamp of the object. T/O Write - If the transaction's timestamp is smaller than the write or read timestamp of the transaction, then the transaction aborts and restarts. Else, the transaction is allowed to write the object and update the timestamp of the object.

2.3.4 Discussion

All the concurrency control mechanisms have some disadvantages associated with them. The timestamp ordering has a problem with rollback; if one of the transactions has to roll back, other transactions related to that transaction that has a later timestamp have to be rolled back, which can affect the performance of the database. With two-phase locking, as we will deal with a lock, we have to make a detection/prevention algorithm with the help of the dependency graph. Optimistic concurrency control, on the other hand, does not need locks, but this technique does not perform well under high contention. For this project, we decided to go with two-phase locking.

2.4 Version storage

DBMS maintains multiple versions of a single object. The DBMS's version storage scheme specifies how the system stores these versions and what information each version contains.DBMS uses the tuples' pointer field to create a version chain per logical tuple, allowing it to find a version visible to a particular transaction at a time. Some approaches include **append-only storage**[30, 19]. In this storage, The DBMS first obtains an empty slot from the table for the new tuple version to update an existing one. The current edition's content is then copied to the new version. The changes are then applied to the tuple in the newly assigned version slot. The new version can be stored in two different ways in this storage. If the new version is stored at the end of the chain, it is called oldest-to-newest. Otherwise, if the new version is stored at the end, it is called newest-to-oldest. **Time travel storage** [30, 19] is similar to the append-only scheme but stores the older data version in a separate table. On every update, the DBMS copies the master version to a new slot in the time-travel table. It then modifies the master version stored in the main table and updates pointers. Last, **delta storage** [30, 19], on every update, copies only the values modified to the delta storage and overwrites the master version. Transactions can recreate the old version by applying the delta in reverse order.

2.5 Compare and swap

Compare-and-swap is a synchronisation technique used to implement atomic operations on shared memory locations. This technique is often used in multi-threading processes to update the value without interfering with other processes. To compare and swap, the method reads the current value in shared memory, Compares it with the expected value. If the comparison succeeds, change the value; otherwise, the operation fails.

This whole operation is atomic, meaning the operation succeeds or fails. Compare-and-swap is needed to ensure that only one transaction can acquire the lock on a row at a time. The problem with this technique is that the compare-and-swap takes place in hardware, but in our project, it will be done over the network as we have to acquire the lock on that table. To overcome this issue, HBase provides a similar API known as CheckAndMutate[4]. This API allows atomic read-write operations on a specific row. The API works by verifying the current value based on the expected criterion provided by the client, and if the criterion succeeds, the method can perform operations such as inserting, updating and deleting a row while guaranteeing atomicity.

2.6 Isolation Level

Isolation[13, 28] is one of the core properties of the ACID transaction. Isolation level refers to the degree to which transactions are isolated from each other in a database. It determines the consistency and accuracy of data when multiple transactions are running concurrently. Different isolation levels provide different guarantees on the consistency and accuracy of data in a database. Different isolation levels are as follows.

1. Read Uncommitted - This isolation level allows a dirty read, which means that the uncommitted changes mad-

e by the transaction can be observed by other transactions. It is the lowest isolation level.

2. Read Committed - This isolation level does not allow dirty read means that the transaction does not allow read or write to other transactions until it commits.

3. Repeatable Read - This isolation level guarantees that the transaction always sees the same data snapshot. This level holds the read lock on all rows it references the data from and writes lock on the rows the transaction inserts, updates or deletes. It is the most restrictive isolation level.

4. Serializable - In this isolation level, transactions are executed as if they occurred in some order, even if multiple transactions occur simultaneously in the database.

5. Snapshot Isolation - This isolation provides a consistent view of the database at a specific point in time. All the transactions in this isolation level sees a snapshot of the database as it existed at the beginning of the transaction, regardless of any changes made by other transactions that are not yet committed. This isolation level allows multiple transactions to read the same data simultaneously without blocking each other.

This project will aim to achieve snapshot or serializable isolation.

2.7 Transaction support in Big data systems

Transaction Support in Big data Systems is currently under research, as combining the performance of Hadoop with the reliability of transactions can make a robust database. One of the main challenges with NoSQL databases like HBase is the need for built-in support for transactions. As a result, NoSQL databases typically sacrifice transaction support to achieve greater scalability and performance. Researchers have explored different approaches to enable transaction support to big data systems to address this issue. One common approach is to use a two-phase commit protocol[5]. This protocol follows the master-slave protocol, where the master (or coordinator) is usually the transaction's origin, and the rest are slaves (or participants). As the name suggests, this protocol has 2 phases, **voting phase** and **decision phase**. During the voting phase, the coordinator sends a prepare-to-commit message to all the participants. The participants receive the messages, validate the details of the transactions and send the reply to the coordinator. The coordinator receives a 'yes' or 'no' reply from the participant. This is where the decision phase start. The transaction will only commit if the coordinator receives 'yes' from all the participants. Otherwise, the transaction aborts if a participant replies with 'no'. The problem with the approach is that if the coordinator fails or crashes for some reason, the participants would have to block the resources until they receive the message from the coordinator. Zookeeper atomic broadcast (ZAB)[14] was designed to overcome this issue. It uses leader based approach to decide whether to commit or not. Whenever a leader receives a request from the client, the leader generates a sequence number and leader's epoch (represents when the leader started to lead.) for that transaction and sends it to all the followers. When the followers receive the request, they add the transaction to the list and send an acknowledgement message to the leader. When

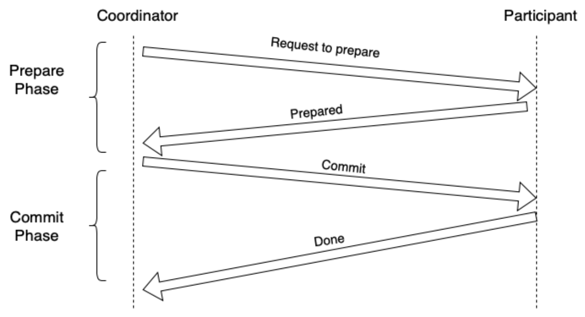


Figure 3: 2 phase commit protocol.

the leader has acknowledged the message from followers, it sends a committed message. The follower will only commit the transaction unless that transaction's sequence number is greater than any sequence number in the list. One of the main advantages of ZAB over 2PC is that the leader always has a consistent view of the system. The leader would know if a new node is added or removed from the system, unlike 2PC, where there are a fixed number of participants. Another advantage is that the new leader is elected through the election process if the leader fails.

Some developments for transaction support in big data systems include the Optimized row column (ORC)[12, 2]. It is column-oriented storage designed for Hadoop workloads. ORC supports ACID transactions in Hive. Apache Hive is a data warehouse that provides an SQL-like interface to the users to analyze petabytes of data. In Hive, a table to support transactions should have an ORC format, and transaction property should be true. The table is automatically partitioned into different buckets. The system adds deltas to the bucket whenever a change is made in the table. When enough deltas are in the bucket, the system automatically performs minor compaction to merge all the deltas into a single file. Once that file is large enough, the system performs major compaction to re-write the base to incorporate the deltas. Apache Phoenix[3] is another example of a big data system that supports transactions. It is an open-source, relational database engine used for OLTP. It works as an SQL driver to provide transactional guarantees.

3. IMPLEMENTATION

In this paper, we introduce a novel approach to introduce transaction support in one of the famous big data systems, HBase. Transactions in HBase currently provide row-level atomicity, meaning each transaction only operates on a single row of data and provides atomicity, consistency, and isolation guarantees for that row. The aim of this paper is to enable cross-row atomicity. This paper designs multi-row transaction support for HBase by implementing **locks** and **transaction monitor table**. The lock is not the traditional row lock offered by HBase[26]. Instead, this lock is created dynamically in the row as a new column to lock it during operations. During the transaction, a transaction may die in between, which can lead to the lock not being released by the transaction. The transaction monitor table can mitigate this problem by keeping track of all the live transactions. When a transaction starts, it is registered with the transaction monitor system along with a unique transaction ID. It sends a heartbeat (update its timestamp) after every 5 seconds to this table which helps monitor the transaction's status.

When the transaction makes a 'write' request, the transaction's ID is inserted in the Monitor table. The transaction also starts a background process, sending heartbeats to the global table to show it is alive. After registering the transaction, The transaction stores the changes locally until the commit. When the transaction decides to commit, it acquires the lock on the row and writes the new data.

Since the writes are locally stored, the transaction first checks the local data for' read' requests. If the data is present locally, it returns that data. Otherwise, the transaction first acquires the lock on the row in the table and returns the value.

In both cases, if the transaction fails to acquire the lock for the first time, the transaction waits for the locks to be released. Then, it waits a specific time and tries to acquire the lock again. The delay for the retry is increased by 1 second each time. If the transaction fails to acquire the lock after six retries, it checks the monitor table for the status of the other transaction that holds the lock. If the timestamp of the old transaction hasn't been updated for more than 5 seconds, the new transaction assumes the other transaction to be dead and forcefully acquires the lock. However, if the old transaction is alive, the new transaction aborts itself.

3.1 Transaction monitor system

In a database, it is necessary to keep the status of live transactions in the database to ensure consistency and avoid conflicts. We approach this problem by implementing a transaction monitor table. The project created a table called "Monitor", it is necessary to remember that no other table in the database can take this name. The role of the transaction monitor system is to keep track of all the transactions occurring in the database. This table can be queried by a transaction to check the status of other live transactions. A transaction monitor table is created with the name 'Monitor' if the table is not present in HBase. The monitor system is a table that has two columns. The first column is used to store the transaction ID, and the second column is used to store the transaction's timestamp. The second column is updated with the heartbeat. When the monitor table is created inside the HBase, the transaction will regularly start the thread process to update its timestamp. These updates occur after every 5 seconds. These updates are necessary because if the difference between the last update of the transaction and the current timestamp is more than 5 seconds, it means that the transaction died. When the transaction is completed, it stops sending heartbeats to monitor. Lastly, Using a transaction monitor table increases system overhead because each transaction must perform its real operations and send heartbeats monitor table. However, it's an essential part of our database due to the advantages of guaranteeing consistency and preventing conflicts.

3.2 Lock

To implement this approach, the main problem was to decide how to lock row in Hbase. Different solutions were discussed to tackle this issue. Zookeeper's distributed lock was taken into consideration. The idea was to use Zookeeper's lock as a service and lock the objects to use. But, with this approach, the load on zookeepers will increase, leading it to become the centre spot of Hbase, which was undesirable. Another solution was to implement our server for the lock, but that would have taken much more time. Ultimately, the solution was to create a locked column through custom API whenever a transaction wanted to operate on a row. First, check whether the lock column exists on that row to perform an operation on a row. If it does not have the lock column, then the transaction creates a lock column on that object and returns the object. But if it has a lock column, it will fail to acquire the lock.

A transaction can only operate on the row once it has acquired the lock on that row. Therefore, we used the CheckAndMutate API provided by HBase to obtain the lock on the row. The CheckAndMutate checks whether the row we are trying to acquire lock already has a lock column. If the row does not have a lock column, it mutates the row to have a lock column and returns true. Otherwise, if the row already has a lock column, it returns false. The lock on the row holds the transaction ID that acquires it. The transaction checks whether the table exists to acquire the lock on the row. If the table does not exist, then the method throws an exception. Otherwise, it gets the table. After getting the table, there can be three cases-

1. **Case 1: Row already exists and transaction already has lock** - After getting the table, the method checks whether the row exists. If the row exists and the transaction has already acquired the lock on the row from before then, the method returns true.

2. **Case 2: Row already exists, but the transaction does not have lock** - If the row exists and the transaction has no lock. The method performs CheckAndMutate on the row. If the CheckAndMutate succeeds, the transaction acquires the lock on the row. However, if it fails, it means that another transaction is holding the lock. In that case, we do exponential backoff. In this process, we continuously try to acquire the lock on the row, but after each try, we increase the delay by times 2. Finally, if one of the requests succeeds in exponential backoff, we acquire the lock on the row.If we fail to acquire the lock even after exponential backoff, we will check the transaction status, as it might be possible that the transaction dies without releasing the lock. In that case, we get the transaction id holding that lock and verify its status from the transaction monitor. If the last update of the timestamp was more than 5 seconds, the transaction is dead. When the transaction is considered dead, we forcefully acquire the lock on the row. Otherwise, if the transaction is alive, we abort the current transaction trying to acquire the lock.

3. **Case 3: Row does not exist in the table** - In this case, if the row does not exist. Then the transaction makes a new row on that table and performs CheckAndMutate to acquire the lock on the new row.

3.3 Deadlock

In terms of database, a deadlock is a situation where two transactions wait for each other to release the lock. As a result, none of the transactions can proceed. Databases often have multiple algorithms to detect and resolve the deadlock. Some of the different approaches to dealing with deadlock are

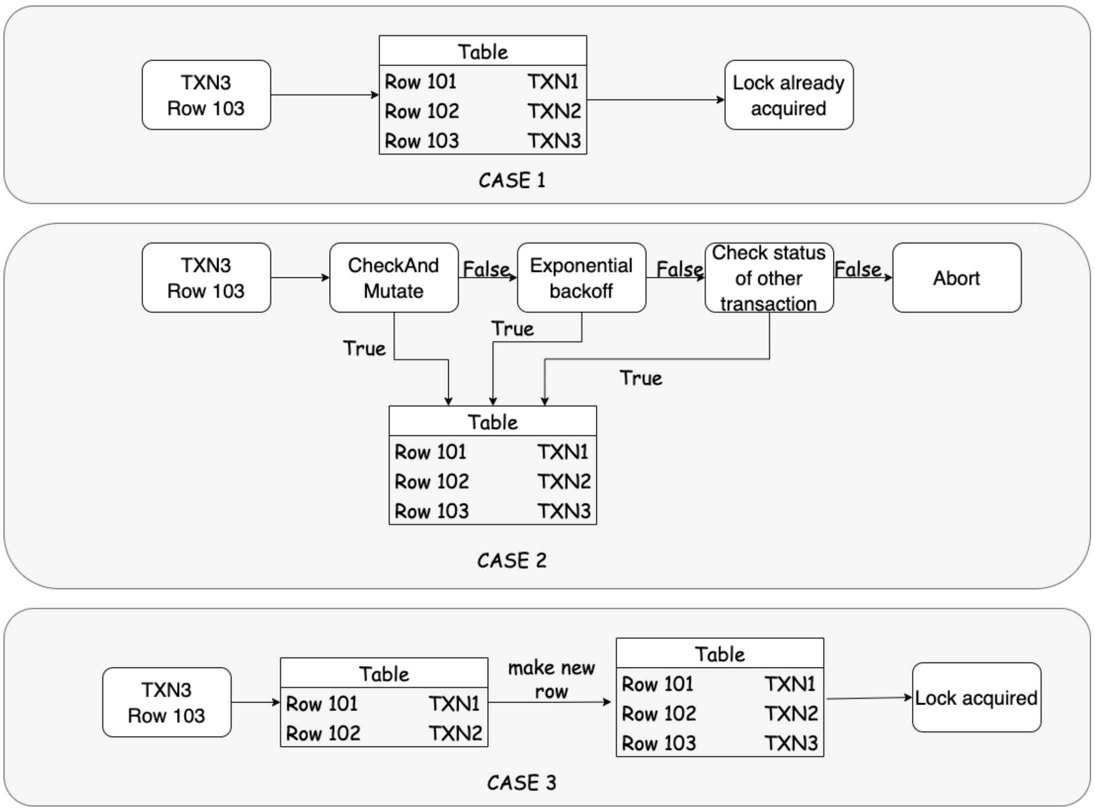


Figure 4: case(i) transaction 'txn3' already has lock on row 103, case(ii) transaction 'txn3' acquires lock on row 103, case(iii) transaction 'txn3' makes a new row 103 with lock

3.3.1 Deadlock detection

In this method, the database has an algorithm that detects the deadlock. The algorithm uses a dependency graph whenever a deadlock occurs. A dependency graph is a directed graph. In this graph, all the transactions are represented as nodes, and an edge drawn from one node to another represent that transaction is waiting to lock held by another transaction. Whenever a cycle is detected in the graph, like in fig 5, the algorithm resolves it by either aborting one of the transactions or by both.

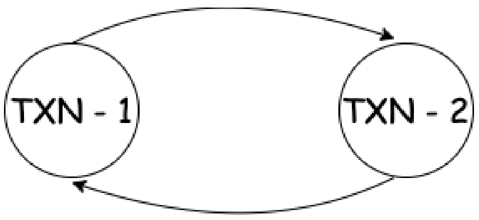


Figure 5: Dependency graph showing transactions 'txn1' and 'txn2' are waiting for each other to release the locks

3.3.2 Preventing deadlock

Some databases solve the deadlock problem by avoiding situations that can lead to deadlock. Some of the schemes are **wait-die**; in this method, if two transactions want to acquire a lock on the same row, then the databases check the transactions' timestamp. After checking the timestamp, the database lets the transaction with the latest timestamp complete and the transaction with the oldest timestamp has to wait for the row to be released. Another method is called **wound wait**, which lets the transaction with the oldest timestamp complete its process in the same scenario, and the transaction with the latest timestamp must wait. This project used the preventing deadlock technique to resolve the deadlock, but we used our technique to acquire locks in a manner that prevents deadlock. We will sort all the rows and acquire the lock in that order to implement this. E.g., if transaction A tries to acquire the lock on rows 3,2,1, transaction B starts to acquire the lock on rows 2 and 3. Transaction A and B will acquire the lock in order 1,2,3 and 2,3, respectively. If transaction A acquires the lock on row 2 first, then transaction B would have to wait before acquiring the lock on row 3.

3.4 Read and write operation

3.4.1 Write

The write operation can be explained with the help of fig6. First, the transaction registers itself with the transaction monitor table and sends heartbeats. Then, all the writes in the transactions are staged in the HashMap. Staging the writes is necessary as it can help to improve the latency because there is a possibility that data required by the client is already present in the cache. Another reason to stage the writes is to provide a more accessible mechanism to abort the

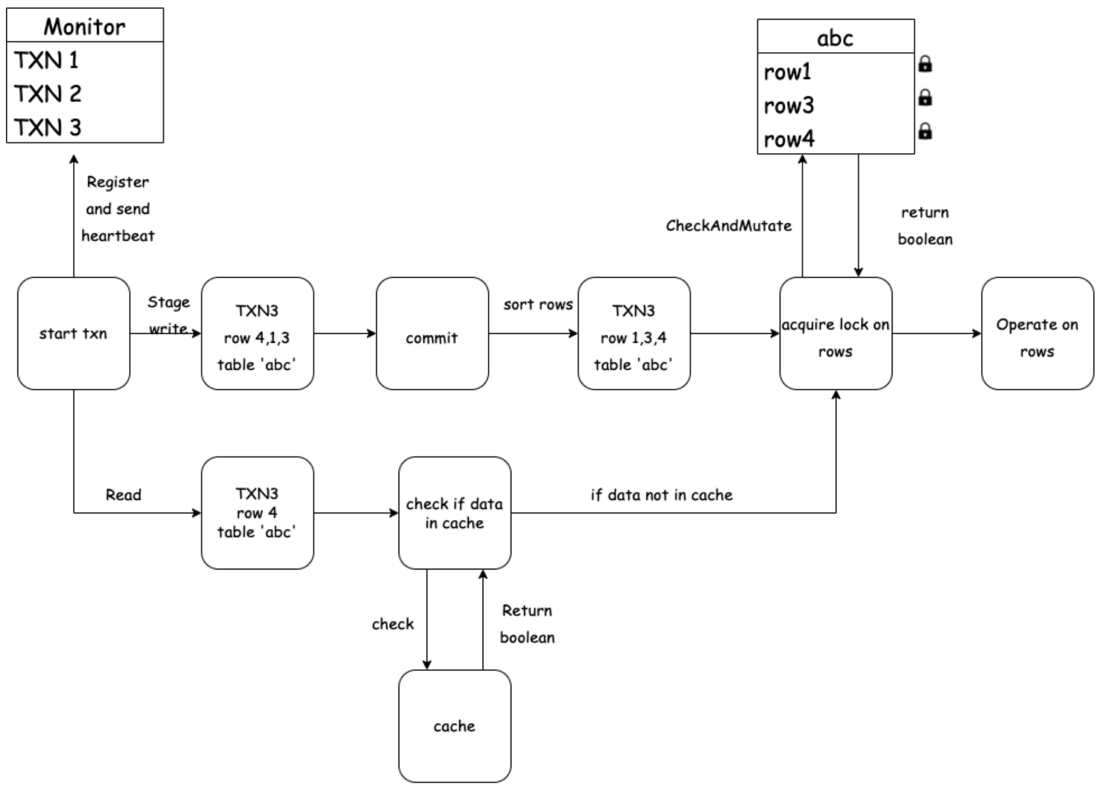


Figure 6: Flowchart of read and write operation

transaction. E.g., if, during a process, the client decides to abort the transaction, there is no need to undo the changes in the table as it has not been committed. The HashMap stores the table name, row, column and data. When all the writes are done, and the client commits the transaction. All the rows inside the HashMap are sorted; this can be seen in the fig6 when the transaction 'txn3' tries to operate on rows, the system sorts the rows from 4,1,3 to 1,3,4. When all the rows are sorted, the transaction starts the growing phase, acquiring the lock on the rows. When all the locks are acquired, the transaction operates on the rows and returns true.

3.4.2 Read

The read operation can be understood with the help of Fig6. First, the transaction registers itself with the transaction monitor table and sends heartbeats. To perform the read, we first check the HashMap. If the data required by the client is already present in the HashMap, we return that data. Otherwise, we acquire the lock on the row and then return that data.

4. EVALUATION

4.1 YCSB

To benchmark the performance of the database, we will use the Yahoo cloud serving benchmark (YCSB)[9]. It is an open-source framework for evaluating the performance of various cloud-based data serving systems. YCSB provides a set of workloads that simulate real-world applications and data access patterns. These workloads can be customized to match the specific needs of a particular application or use case. The YCSB dataset could be downloaded from the link https://github.com/brianfrankcooper/YCSB. After downloading the dataset, it could be built by following the instructions on the GitHub page of YCSB. After installing YCSB, the YCSB API can be extended to create a custom binding for the project. To do this, we created a new Java class that implements the DB interface provided by YCSB. This interface contains methods that need to implement to perform various database operations such as insert, read, update, and delete. The JAR file of the interface along with the project is used for testing. In the following subsections, we conducted different experiments such as memory usage; to compare the memory utilized by standard HBase and transaction-supported HBase (HBase-T), latency; to compare the average time taken by standard HBase and transaction-supported HBase to complete a set of operations and throughput; to compare the number of operations performed per seconds by standard HBase and transaction supported. All the experiments were performed on a MacBook Pro with a 2 GHz Quad-Core Intel Core i5 processor and 16 GB 3733 MHz memory. No other programs were running during the experiment to get maximum performance.

4.1.1 Unlocking vs lazy unlocking

This experiment compares two strategies proposed by the project, **Unlocking** and **Lazy Unlocking**. In the unlocking strategy, the transaction releases the lock after committing, but in lazy-unlocking, the transaction does not release the lock. In lazy-unlocking, the transaction will die while holding the locks, and if another transaction needs to acquire the lock on that row, it would have to acquire the lock from the dead transaction forcefully. Lazy-unlocking could be preferred when a new transaction always operates on a new row and does not need a row acquired by another transaction. The possible advantage of this strategy should be decreased latency as the transaction is not releasing the lock. To verify the assumption, the results of this experiment can be seen in 7, where it shows the difference between the memory used by standard and transaction-supported. HBase. The results

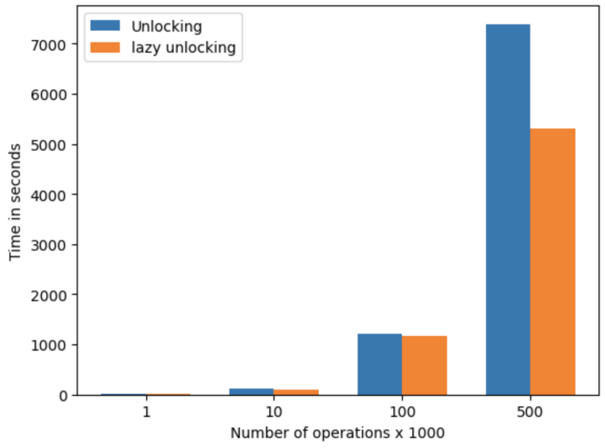


Figure 7: comparison between unlocking and lazy unlocking

show that lazy-unlocking takes less average time to complete a set of operations. This could be because the transaction does not have an extra operation of deleting the lock column family. This could be clearly seen in the experiment in 500000 operations, with a difference of almost 2000sec in the average time taken. But, the Lazy-unlocking strategy would surely be useless when the transactions work on the same set of rows. For further experiments, we used the lazy-unlocking strategy to maximize the performance of the HBase-T.

4.1.2 *Latency*

In this experiment, we compare the latency or the average time taken by standard HBase and HBase-T to perform some operations. To perform this experiment, we scale the number of operations as 1000, 10000, 100000, and 500000. For each number of operations, the experiment was performed five times, e.g. For 1000 operations; the experiment was conducted five times for both standard and transaction support. We expect an increase in latency. The results of this experiment can be seen in 8, where it shows the latency between different standards and HBase-T, respectively. These results have two main observations

1. **Increased latency** - As the number of operations increases, the time taken to complete the transaction increases.

2. **Latency difference** - There is a significant difference in latency between the standard and HBase-T. E.g., in 500000 operations, there is a difference of 10^2 factor between them. We think it was because of the lock-acquiring process, as the transaction has to connect with the table twice, first to acquire the lock and second to operate on the row, which could result in increased latency.

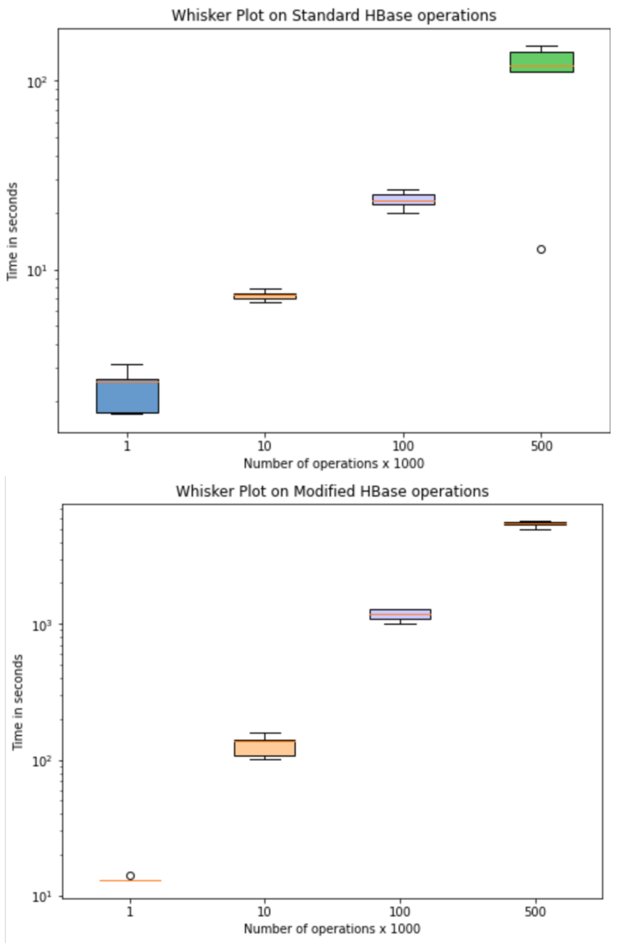
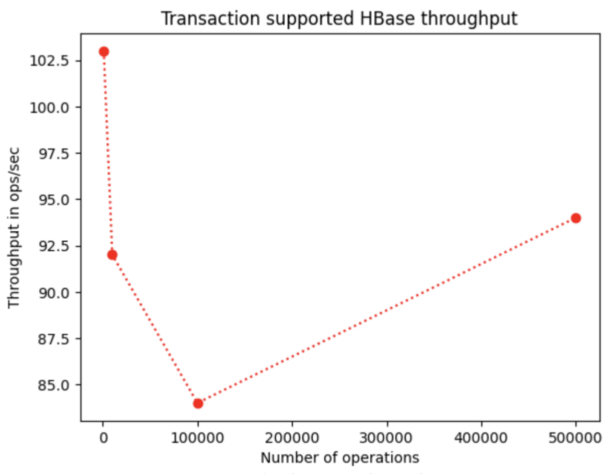


Figure 8: Latency difference between HBase and HBase-T

4.1.3 *Throughput*

In this experiment, we compare the throughput of standard HBase and HBase-T to perform some operations.
To perform this experiment, we scale the number of operations from 1000, to 10000, 100000, and 500000. For each number of operations, the experiment was performed five times, e.g. For 1000 operations; the experiment was conducted five times for both standard and transaction support. We expect an increase in latency. The results of this experiment can be seen in 9 and 9, where it shows the difference in throughput between standard HBase and HBase-T. In this experiment, we observed that the throughput is directly proportional to the number of operations in the case of standard HBase. But in transaction-supported HBase, the throughput decreased and then increased. This could be due to the implementation of 2 phase locking used in the project. The decrease in throughput could be because the concurrency control mechanism is trying to adapt and optimize transaction execution, and as the number of operations increased, the concurrency control mechanism started to stabilise.



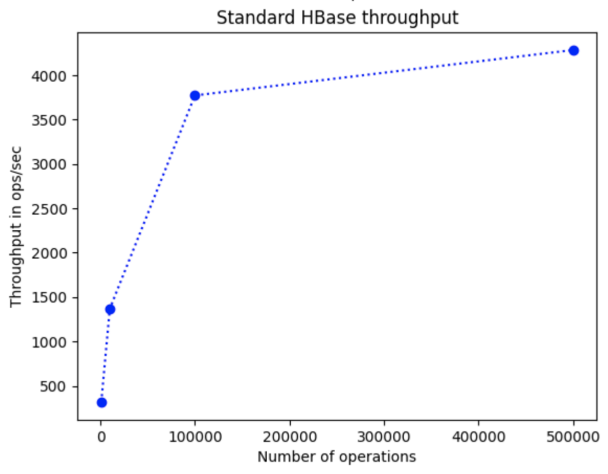


Figure 9: Throughput difference between HBase and HBase-T

4.1.4 Memory usage

This experiment compares the memory used by standard HBase and HBase-T after performing all the operations on the database. To perform this experiment, we scaled the operations as 1000, 10000, 100000, and 500000. At each scaling factor, we get the memory utilised by the database. The memory utilised can be obtained by accessing the server on http://localhost:16010/. We expect an increase in memory usage. The results of this experiment can be seen in 10, where it shows the difference between the memory used by standard and HBase-T
These results have two main observations

1. **Linear relationship** - A linear relationship exists between the number of operations and memory used.

2. **Memory difference** - We were expecting a memory difference due to the extra column created for lock, but the result shows that there is no significant difference in a small number of operations, and as the number of operations increases, we can observe an increase in memory usage. This increase in memory could be because of the lock column created by the transaction to acquire the lock. The memory utilized in the lock column may be so small that when the number of operations is significantly higher, only there can be a noticeable difference. In the future, we can also perform an experiment to analyse the memory the lock utilises in a single row.

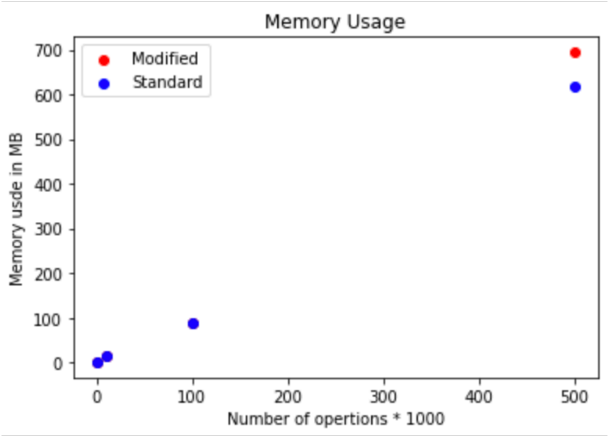


Figure 10: Memory utilisation between HBase and HBase-T

4.2 Evaluating concurrency control anomalies

Concurrency control anomalies are unexpected anomalies that may occur when two or more transactions are trying to modify the same data. To test our database against data anomalies, we will design a few small-scale experiments to replicate the anomalies.

1. **Lost Update** - Occurs when two transactions try to update an object simultaneously, and one transaction overwrites the changes of the other. For example, transaction X reads a row. Transaction Y reads the same row, adds 20, and rewrites it. Transaction X subtracts 50 from the row that it previously read and rewrites the row. Transaction Y's update is lost. To test this behaviour, we will start two transactions concurrently with the help of threads. Transaction A will try to modify row1 and row2, and transaction B will try to modify row2 and row3. We ran this experiment multiple times and noticed that only one transaction could change row2 at a time. The other transaction had to wait for the previous one to commit. Only after committing the other transaction could modify row2. This experiment proves that the database does not allow the lost update.

2. **Dirty reads** - occurs when one transaction modifies a row, and a second transaction reads the row before the first is committed. If the first transaction terminates and is rolled back, the second transaction has read data that does not exist. Given the nature of our project, it should be impossible for the transaction to read uncommitted data, as the transaction can only read the data after acquiring a lock on a row, and a lock can only be acquired if a row is not used by any transaction or the transaction just released the lock. But to still verify this behaviour, we conducted an experiment. In this experiment, transaction A will modify the data in row1. After modifying row1, the transaction will wait with the sleep method provided by Java, but transaction A will continue to send heartbeats to

the transaction monitor. During sleep, transaction B will try to read row1. We ran this experiment multiple times and noticed that transaction B was not able to able to acquire the lock as transaction A still had the lock on row1 resulting in aborting transaction B. This experiment proves that the database does not allow dirty reads.

3. **Non-repeatable reads** - occurs when a transaction reads a row twice, each time getting a different result. This can occur if a transaction reads a row, another transaction updates it, and then the first transaction rereads it and gets a different value the second time. To verify this, we conducted the same experiment as dirty reads. Non-repeatable is not possible with this database, as the transaction will not be able to modify the row if another transaction has already acquired the lock on that row.

5. DISCUSSION

To summarize this paper, we proposed a solution to implement transaction support in HBase. HBase currently provides limited transaction support by guaranteeing atomicity in single-row operations. But, our design would enable HBase to provide atomicity across rows. Furthermore, we provided this support by implementing a transaction monitor system that a transaction could query to verify the status of other transactions in the database. All the writes are staged in a HashMap. Only when the client starts to commit the transaction would first sort all the rows and then acquire the lock in that order. The lock is created dynamically by adding a new lock column in the row. After all the locks are obtained, the transaction would start operating on the rows. If a transaction wants to read a row, it would first check if the data is already present in the cache, and if the data is not available in the cache, it would then provide the data from the table.
To test the efficiency of our database, we performed multiple experiments with the help of YCSB. To perform the experiments, we scaled the number of operations from 1000, to 10000, 100000, and 500000. We checked the difference in memory used by standard HBase and our HBase-T. We observed the difference in memory utilized when the number of operations is near 500000. The also performed an experiment to analyse latency, and our solution took relatively longer than the standard HBase because the project was complex. We also conducted a small-scale experiment where we analyzed our project against data anomalies. Another experiment was to compare the throughput between standard HBase and HBase-T. We observed that standard HBase throughput increases as the number of operations increases, but the HBase-T throughput decreases to 100000 operations, then increases. We also compared two unlocking strategies, unlocking and lazy-unlocking, for releasing the locks. After performing the YCSB test, we also performed small-scale experiments to evaluate concurrency control anomalies where we observed that the project did not allow lost updates or dirty and non-repeatable reads.

5.1 Limitations and Future work

In this section, we will discuss the limitations of the project along with the strategies to overcome the limitations.

1. **Synchronised clock** - The project currently assumes that the clock of all the clients is synchronised, meaning that at one point in time, no two clients will have the same timestamp for a transaction. This ensures that even if two clients start transactions from different time zones and are trying to operate on the same row. They should not consider each other dead, as the difference in time would be too much. To resolve this issue, the project could implement a vector clock. A vector clock is an algorithm that provides a consistent view of all processes occurring in the distributed system. The algorithm marks every process in the system with a vector that has a length of size N, where N is the number of processes in the system. At the start, all the clocks are set to zero, and whenever an event occurs in the database, the vector clock is incremented by 1. This vector clock could be used to assign timestamps to transactions in the database or update timestamps in the monitor table.

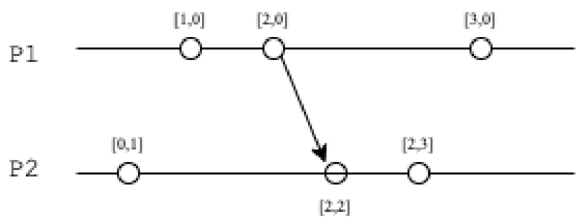


Figure 11: Working of vector clock

2. **Recovery Protocol** - The project currently has no measure if a transaction fails between committing. If a transaction fails between writing on a row, there is no measure to undo the changes. To improve it in future, the project could design a protocol that implements a write-ahead log (WAL). The WAL is one of the common techniques used in the database to ensure recoverability. The log could contain details such as transaction Id, row Id, old and new data. These log files contain details of all the changes in the database and could be stored on a server or locally, depending on the requirements. In case of system failure, the database could rerun the WAL files to undo/redo the changes, thus maintaining the consistent state of the database. The system could delete old files to ensure there are not too many log files.
Another recovery measure could be to use a commit timestamp. These timestamps could be used to recover from system failure and prevent data loss. The way it can be used is that in case of system failure, if the commit timestamp of the transaction is before the system crash, the database would know that these transactions are committed and are safe. If the commit timestamp of the transaction is after the system crash, the database would have to undo/redo the changes done by the transaction.

3. **Testing transaction support** - The project uses YCSB to benchmark the database, but it does not properly

check the transaction part of the database. All the operations done in the experiment are part of one transaction. In future, we could do batch operations where after every 100 operations, we start a new transaction. But, to check the efficiency of the transactional part of the database, we would have to create conflicts where two transactions would try to acquire the lock on the same row. This strategy has progressed as the developers work towards a new dataset called YCSB+T.

4. **Alternate way to stage** In this version, All the versions in the database are stored according to timestamp, meaning that the version with the greatest timestamp will be the most recent. The transaction starts by acquiring the lock on the rows. It first checks if the lock column exists or not. If it does exist, it means that some other transaction is already using it, and it aborts or restarts. Otherwise, it makes a locked column and locks the object. The checking and locking of objects occur with the help of *CheckAndMutate*. After locking the object, the transaction stores its timestamp locally and performs operations on it. After modifications, it checks if the object's timestamp is still the same when it's time to commit. If the timestamp is different, the transaction aborts because some other transaction has modified it. If the version number remains the same, the transaction will put the modified object with a new timestamp and set the most significant bit to 1. This bit change will ensure that the object's recent version appears first in the list. After the transaction is committed, all the locks acquired by the transactions are released.

Acknowledgments. I would like to thank my supervisor Dr Nikos Ntarmos for guiding and helping me to overcome the challenges I faced throughout the project.

6. REFERENCES

[1] Apache. Apache hbase acid. `https://hbase.apache.org/acid-semantics.html`.

[2] Apache. Apache orc. `https://orc.apache.org/docs/`.

[3] Apache. Apache phoenix. `https://phoenix.apache.org/`.

[4] Apache HBase. CheckAndMutate. `https://hbase.apache.org/devapidocs/org/apache/hadoop/hbase/client/CheckAndMutate.html`.

[5] M. Atif. Analysis and verification of two-phase commit & three-phase commit protocols. In *2009 International Conference on Emerging Technologies*, pages 326–331, 2009.

[6] V. Bhupathiraju and R. P. Ravuri. The dawn of big data - hbase. In *2014 Conference on IT in Business, Industry and Government (CSIBIG)*, pages 1–4, 2014.

[7] J. Camacho-Rodríguez, A. Chauhan, A. Gates, E. Koifman, O. O'Malley, V. Garg, Z. Haindrich, S. Shelukhin, P. Jayachandran, S. Seth, D. Jaiswal, S. Bouguerra, N. Bangarwa, S. Hariappan, A. Agarwal, J. Dere, D. Dai, T. Nair, N. Dembla, G. Vijayaraghavan, and G. Hagleitner. Apache hive: From mapreduce to enterprise-grade big data warehousing. In *Proceedings of the 2019 International Conference on Management of Data*, page 1773–1786, New York, NY, USA, 2019. Association for Computing Machinery.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[10] D. Ganesh Chandra. Base analysis of nosql database. *Future Generation Computer Systems*, 52:13–21, 2015. Special Section: Cloud Computing: Security, Privacy and Practice.

[11] Z. Guo, K. Wu, C. Yan, and X. Yu. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking (extended version). *CoRR*, abs/2103.09906, 2021.

[12] T. Ivanov and M. Pergolesi. The impact of columnar file formats on sql-on-hadoop engine performance: A study on orc and parquet. *Concurrency and Computation: Practice and Experience*, 32(5):e5523, 2020.

[13] Jenpen. consistency models. `https://jepsen.io/consistency/`.

[14] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256, 2011.

[15] A. Khetrapal and V. Ganesh. Hbase and hypertable for large scale distributed storage systems. *Dept. of Computer Science, Purdue University*, 10(1376616.1376726), 2006.

[16] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, jun 1981.

[17] V. O. K. Li. Performance models of timestamp-ordering concurrency control algorithms in distributed databases. *IEEE Transactions on Computers*, C-36(9):1041–1051, 1987.

[18] A. Pavlo. Concurrency control theory. `https://15445.courses.cs.cmu.edu/fall2021/slides/15-concurrencycontrol.pdf`.

[19] A. Pavlo. Multiversion concurrency control. `https://15445.courses.cs.cmu.edu/fall2021/slides/18-multiversioning.pdf`.

[20] A. Pavlo. Time stamp ordering concurrency control theory. `https://15445.courses.cs.cmu.edu/fall2021/slides/17-timestampordering.pdf`.

[21] A. Pavlo. Two phase locking concurrency control. `https://15445.courses.cs.cmu.edu/fall2021/slides/16-twophaselocking.pdf`.

[22] V. Sajwan, V. Yadav, and M. Haider. The hadoop distributed file system: Architecture and internals. *International Journal of Combined Research & Development (IJCRD)*, pISSN:2321–2241, 05 2015.

[23] K. Shvachko, H. Kuang, S. Radia, and R. Chansler.

The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[24] C. Strauch, U.-L. S. Sites, and W. Kriha. Nosql databases. *Lecture Notes, Stuttgart Media University*, 20(24):79, 2011.

[25] tsunanet. Class row lock. `https://data-flair.training/blogs/hbase-operations/`.

[26] Tsunanet. Class row lock. `https://tsunanet.net/~tsuna/asynchbase/api/org/hbase/async/RowLock.html`.

[27] M. N. Vora. Hadoop-hbase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 1, pages 601–605, 2011.

[28] Wikipedia. Isolation level. `https://en.wikipedia.org/wiki/Isolation_(database_systems)`.

[29] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending acid semantics to the file system. *ACM Transactions on Storage (TOS)*, 3(2):4–es, 2007.

[30] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, mar 2017.

[31] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance acid via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 279–294, New York, NY, USA, 2015. Association for Computing Machinery.