COMP90020: Distributed Algorithms – Term Project

Multi-version Concurrency Control Algorithm

Group Number: 4

Name	Student ID	Email
Xinnan SHEN	1051380	xinnan.shen@student.unimelb.edu.au
Xiguang Li	944558	xiguangl@student.unimelb.edu.au
Chaoxian Zhou	1096367	chaoxianz@student.unimelb.edu.au
Huidu Lu	717910	huidul@student.unimelb.edu.au

Abstract

With the increasing number of users, the ways of correctly and efficiently handling the concurrent access to the shared resources would significantly influence the stability and performance of a system. A typical system which has a high concurrency requirement is database. Great amounts of transactions could result in a heavy concurrent executed operation, leading to the unexpected read and write conflicts. There have been many mature concurrency control algorithms applied on the modern application, such as 2-phase locking, timestamp ordering and serial validation. The goal is to improve the system performance while still keeping the system's data consistency. Multiversion concurrency control provides a good support in both these two desires. In this project, we have designed and implemented a database management system demo and applying timestamp ordering based multi-version concurrency control algorithm on it. By applying this algorithm on the database management system, the database system management will not be likely to lead to inconsistent states in some scenarios. The results have shown that after using the algorithm, the database system can correctly deal with the conflicted concurrent transactions and tolerate some crash failures of client processes.

Keywords MVCC, Concurrency Control, Database Management System

1. Introduction

1.1 Background

Multi-version Concurrency Control (MVCC) Algorithm is used in many applications, especially database systems. At present, most widely-used database systems have used multi-version concurrency control algorithms, such as MySQL, Oracle and SQL Server. In this project, a timestamp ordering based MVCC algorithm has also been implemented to control the concurrency in a simulated database management system so that the data in the system will be consistent in some scenarios.

To demonstrate the effect of MVCC algorithm, a client part and a server part are designed in the project. The MVCC algorithm runs on the server part, and the client part is used to manipulate the simulated database management system. The whole system allows different clients to access the database, and the MVCC algorithm can guarantee the data consistency in some scenarios when same data are accessed simultaneously. The operations on the client part are similar to the real database management system, including:

- Insert data
- Update data
- Select data

- Delete data
- Commit
- Rollback

To demonstrate the logic of this algorithm, the real database management system is not adopted since these mature databases have implemented this algorithm internally. Instead, we have designed a specific data format, stored the data in server memory and apply the MVCC algorithm on the data to simulate a real database management system.

1.2 System Structure

The function of the system includes: concurrency control, failure handling, and timeout handling. Firstly, the concurrency control is aimed at preventing the database from leading to an inconsistent state. For example, if one transaction is updating the data, while the other wants to update the same data, the problems may occur. To try to solve these problems, the concurrency control strategy has been implemented so that the conflicting issues can be dealt with properly. In this project, we have chosen the isolation level of reading committed data, so that the problem of dirty read can be easily solved by implementing the MVCC algorithm.

Apart from concurrency control, the system in the project can also deal with crashes properly. In some scenarios, one client might have updated some data and crashes without committed, which might lead to other clients unable to update or delete the same data even after it crashes, so there should be a crash handling method to deal with this issue. When the client crashes, the system will operate as usual and other clients can modify the same data held by the client which has crashed, so that it will not lead to an undesirable state. Moreover, it is necessary to deal with server crashes as well. All the data stored in the server will be lost after server restarts without a tolerating server shutdown. In this project, the server crash problem is also handled. When the server crashes, the clients will be notified and they will need to reconnect to the server when the server restarts.

Furthermore, we have also implemented the timeout handling in this system. Sometimes, the client may not crash but it cannot make any transactions due to some problems in the computer, like out of memory. In such case, the client might hold some data for a very long time and other clients might wait for them to release the data. If this situation cannot be handled properly, other clients may wait for the client indefinitely, which is a big issue. Therefore, a timeline threshold is set in this project. If a client has not made any action in a certain amount of time, the system would assume it has timed out and let it disconnect with the server. After it makes another transaction, the client is required to reconnect to the server.

1.3 Report Structure

The report will focus on the design and implementation details of the MVCC algorithms. It consists of 11 sections. In section 2, we will briefly review the past literature and works related to the MVCC algorithms. In section 3, we will show the details of three different types of MVCC algorithms. We compare them with its single-version counterpart in section 4. In section 5, we will state some assumptions of the algorithm and system model before we start implementing them. In section 6 and 7, we will show the specific details of system design and implementation. Furthermore, in section 8, we will describe some of the shortcomings of the algorithm and make a summary of what we have done in our system and analyze the existing drawbacks in our system in section 9. Finally, in section 10 and 11, we will share our thinking about the possible future works that our implementation and the MVCC algorithms.

2. Literature Review

With the increasing number of users, concurrent access to the part of the system becomes the main trouble spot and bottleneck. Researchers have deeply investigated the concurrency control theory for the past several decades. The goal of the concurrency control is to prevent interference caused by multiple users simultaneously accessing the same shared object resource. In database management systems, the interleaving read and write operations executed by the concurrent transactions could lead to this undesired interference. To solve the problems, the concurrent interleaving executions need to be serialized. This serialization theory gives a guideline to design a concurrency control protocol (Eswaran et al., 1976; Bernstein et al., 1979; Papadimitriou et al., 1976).

There are many approaches to reach serializability. The basic idea of multi-version concurrency control is to maintain one or more old versions of an object and allow concurrent access to different versions by pre-defined rules. The idea of applying multi-versions of data items to solve synchronization problems was first introduced by Reed (1978) in his dissertation. He tries to provide a standard set of mechanisms for organizing and controlling concurrent computations in a decentralized system environment.

Traditional ways to solve conflicts caused by concurrent access relies on the concept of mutual exclusion and critical section so that we can control the shared data objects. Within a certain time period, only one transaction could access the shared object and do its tasks. Others need to wait until the on-going transaction completes. Instead of directly dealing with dynamic real-time conflict operations, Reed (1978) includes the history of operations applied on the object to eliminate the conflicts. Each object has a history recording its previous states. All the histories of each independent object together form a system history. This system history works like a "snapshot". The pseudo-time notion is used to relate the system history into a consistent part. In practice, the implementation of pseudo-time could be an incremented timestamp count. By allowing different transactions accessing its related snapshot of the system, the read and write conflicts are solved without using locks.

Bayer et al. (1980) came up with a similar idea which utilizes the implicit versions of data that exist in the general 2-phase lock concurrency control algorithm. It is possible that a bad transaction needs to rollback. For recovery reasons, there might be two values of one object. In the old convention, concurrency is controlled by locks. A write operation has to wait for all the exclusive read locks on that object to be released. This is not efficient. It is feasible to take advantage of the old/new version of that object so that a read access could always be granted, either to the old or new version of that object. This attempt does have consistency problems if we just simply grant read access to the newest version, so several rules need to be followed. The detail of this algorithm is included in section 3 of this report. It is worth pointing out that in their design, an object would only have no more than two versions of value, old and new versions or only the new values if the old version is released after all readers of it have completed and left. Stearns and Daniel (1981) had the similar idea. This is the raw form of the multi-version concurrency control algorithm.

Bernstein and Nathan (1981) expand and consolidate the idea of multi-version from Reed and Bayer et al. They provide a formal definition of the multi-version concurrency control algorithm in their research. They categorize concurrency control algorithms into two big groups, Two-Phase Locking and Timestamp Ordering. Basic MVCC algorithm is in the Timestamp Ordering group since each transaction is assigned a unique timestamp in MVCC algorithms and read/write operations updates the data object's timestamp from the issued transaction. The serial equivalent is ensured by the timestamp order. In addition to multi-version timestamp ordering concurrency control algorithm, they also generalize Bayer et al and Stearns et al. idea of two-version with lock to multi-versions locking (1983). Another multi-version algorithm based on 2-phase locking is designed by Chan and Gray (1982). The main improvement they introduced is the version pool, which is used to maintain and clean up the old versions.

Bernstein and Nathan also analyze the correctness of the MVCC algorithms and provide a mathematical proof to support the algorithm's serializability in their following research (1983). The proof process is based on the serializability theory. Transaction executions could be described by logs. A transaction log is a partially ordered set representing a read/write operation sequence of that transaction. The partial order used here is the happens-before relation. A log over a set of transactions gives the partial order of all the operations of these transactions. To form a log in multi-version database system, we need to have an extra step of mapping each item operation to version operation. Then, they define a serial log as a totally ordered log, that is, for every pair of transactions, all the operations of one transaction is before or after another one. This definition implies that a serial log must result in a correct execution of transactions because by definition there is no concurrency at all. By having defined serial log, they convert the proof of serial equivalent of transaction to log equivalence. The whole proof process is complicated. They finally found that to decide whether a multi-version log of sets of transactions is equivalent to a serial log is NP-complete.

Concurrency control algorithms could be divided into two groups, pessimistic approach and optimistic approach. Locking or Timestamp Ordering are common pessimistic methods. The above MVCC algorithms are based on these methods. There is also an MVCC algorithm based on optimistic methods, which is designed and published by Carey (1983). It is based on the single-version optimistic concurrency control algorithm (Kung and Robinson, 1981). In the optimistic method, a read set and write set is kept for each transaction. Transactions use a temporary copy of the object during the working phase and no blocks occur. A validation check is performed when a transaction is about to commit. The invalid transaction might be aborted or restarted. Carey includes the timestamp into the optimistic approach and uses the multi-versions of data to improve the concurrency. The timestamp is used to facilitate the validate check instead of simply doing the readset/writeset intersections test.

At this point, all three design schemas of MVCC algorithms have been published: Timestamp Ordering based, Locking based and optimistic based. The concurrency control bases used in MVCC algorithms have not had significant changes. In the following years, some of the research focuses on the improvement of version control mechanisms, that is, how to efficiently maintain versions and clean up old versions (Divyakant and Soumitra, 1989). Others try to present a performance model for MVCC and compare it with the single-version counterpart (Michael and Waleed, 1986; Sanzo et al., 2008). David et al. (2012) point out that the current implementation of MVCC algorithms on any database system only reaches snapshot isolation. They do not achieve full serializability and as a result, the integrity constraints could be violated. This flaw has been found in production code. They came up with a novel timestamp conflict manager to solve this problem by managing a range of possible timestamps for each transaction instead of only the startup timestamp in the old convention.

3. Algorithms Details

In this section, we are going to explain in detail the three different types of multiversion concurrency control algorithm: Timestamp Ordering based, Locking based and Serial Validation (Optimistic method) based.

To restate the goals, we want to improve the system performance by allowing concurrent access from different transactions to the shared data item while keeping the data system in a consistent state. Concurrent access involves concurrent read and write operations. A set of operations in a single transaction is serialized but concurrent read and write operations from different transactions usually interleave with each other. If the concurrent operations are only read operations then everything works fine. However, once write operations are involved, the system data consistency might be violated. This is called the read-write conflict.

Therefore, concurrent accesses need to be carefully controlled. The serialization theory tells that if we could control the execution of concurrent operations in a serialized order as if these transactions are serialized, then we could ensure the data consistency and have a good performance by allowing concurrent access at the same time. In other words, all the concurrency control algorithms aim at serializing the interleaving operations from different transactions by controlling those operations' execution order in some rules.

The core idea of all MVCC algorithms could be briefly summarized into two points:

- Each write operation on a data object produces a new copy of that object. Old versions are kept until it is expiry.
- Every read operation on a data object will always be allowed after it is issued. The read operation would be granted the "correct" version of the data to ensure the serialization of concurrent transactions.

Controlling of version access gives the control of concurrent operations' execution order.

3.1 MVCC Timestamp Ordering Base.

The core concept of MVCC-TO algorithm is to use timestamps to control the correct version of data objects an issued operation could access to control the concurrent operation's execution order. Every transaction is issued a unique timestamp at its start. The timestamp could be a centralized server clock time or an incrementing number. It defines the happens-before order between each transaction.

We need to define some terms to help our explanation:

Term Definition:

- T_i: The ith transaction, with the assigned startup timestamp i.
- TS(x): the most recently committed write timestamp.
- x: The kth version of the data object x.
- $r_i[x_k]$: A read operation on the version-k of data object x issued by the transaction $T_{i.}$
- $w_i[x_i]$: A write operation on the data object x issued by the transaction T_i, resulting in a new version of x, x_i. Note the i-notation of object x in here denotes a side effect on x, not a version before write operation starting.
- <: A happens-before relation defines the order.

Timestamp assign rules works as follows:

- Each transaction T is assigned a unique startup transaction: T_i
- Each operation issued by T_i carries its startup timestamp: r_i, w_i.
- The most recent version of object x has a read timestamp R_TS(x) from the latest read operation and a write timestamp W_TS(x) from the latest write operation.

MVCC-TO Algorithm rules:

For write operation:

```
A transaction $T_i$ wants to write to the object $x_k$, a write operation $w_i$ is issued:
    If (R_TS(x) < i and W_TS(x) < i) :
        $w_i$ success, a new version, $x_i$ is created by $T_i$. W_TS(x) = i
    else:
        $w_i$ fail, the transaction $T_i$ might be aborted or restarted

A transaction $T_i$ wants to read to the object x, a read operation $r_i$ is issued:
    $r_i$ read the most recent version of x where W_TS(x) < i
```

By following the above rules, concurrent operations would be processed as if it follows the order denoted by the transaction timestamp. Good concurrency is reached while data consistency is met. This algorithm is the one we chose to implement in this project. In order to follow these two rules, the actual implementation is much more complicated. The implementation detail could be found in section 5 and 6.

3.2 MVCC Locking Base:

MVCC Locking Base is similar to the two-phase Locking single-version algorithm, except each object has multiple versions. A commit timestamp is assigned to the object. Read and write operations both carry the timestamp of the issued transaction. The newest version of the object x is locked when a write operation is issued. When the transaction commits, the new version's commit timestamp is updated. Read operation only sees the committed latest version of the object that is earlier than its carried timestamp.

3.3 MVCC + Serial Validation (Optimistic methods):

Original serial validation checks the interception of reads and writes of a ready-to-commit transaction with that of the past or future transactions. The transaction is aborted or restarted if there is some interception between headset and writeset. The multiversion used in the validation step to facilitate the test. Each transaction has a start-timestamp and a commit-timestamp. Each data object also has a timestamp which is updated by the latest committed write operation's issued transaction.

For a transaction T, with readset R_Xs and writeset W_Xs, at the validation step, if:

- for all objects x in its R_Xs, TS(x) < TS(T)

The transaction would pass the validation and commit its change. After it completes commit, a commit-timestamp is created and all the objects in its W_Xs would have a new-version with the timestamp setting to be this commit-timestamp.

The implementation of these three types of algorithm might be variant with what we describe in this report. Bernstein and Nathan (1983) have mathematically proved that the simplified version of all the above three types of MVCC algorithms could reach the serial equivalence requirement of concurrency control.

4. Comparative Analysis

Compared with the single-version concurrency control algorithm, the main benefit gained from the multiversion is that read operations are never blocked. Read-write conflict is solved by allowing read operation access to old versions of objects and pay more penalties on invalid write operations. As a result, the concurrency ability of the system is significantly improved. In the scenario of large amounts of read-only transactions, MVCC would achieve much better performance than the conventional locking-based concurrency control algorithms.

However, the extra costs are also added in the MVCC algorithms. MVCC needs to maintain a chain of versions. To find the suitable version for a specific read operation, we need to compare the timestamp of the object with the issued transaction. Following the chain to do the comparison would be a problem with the efficiency of O(nlog(n)) in the best case, which exposes an obvious bottleneck of the algorithm.

Garbage collection of the old-versions' data is another additional cost. The old versions with the write timestamp TS_old that does not have any on-going transactions with the timestamp between the TS_old and TS_newest of that object issuing any read operations are regarded as garbage and could be removed. The process would not only add more storage overhead but also cause the extra CPU cost to do the cleaning task.

5. Assumption and Implications

As stated in section 2, the concept of multi-version concurrency control is firstly proposed by Reed in 1978. This algorithm is currently applied in many database management systems. Reed (1978) states that for each logical entity, the system simultaneously maintains multiple physical versions in the database. When a transaction is going to write to an entity, this system will make a copy version of this entity and the transaction would write to the copy version. When a transaction executes a read operation on an entity, it reads the latest committed.

The algorithm we implemented in the project is based on basic timestamp order protocol, with multiple versions concurrency control added. To achieve this algorithm, several assumptions about the nature of multi-version entities are made. These assumptions are shown in the following:

1. Each transaction should have a unique identifier, and each entity should keep the transaction unique identifier which issues it.
2. An entity should have multiple copies and these copy versions of entity should be maintained at the same time.

3. Each entity also should have an ability to ‘inform’ systems that whether it is still valid, in other words, whether this entity is still ‘visible’ to any one transaction.
4. To guarantee multiple versions of entity, the database management system would make a copy version of an entity when a transaction wants to write to this entity.
5. For each transaction, it should have an action log, since this log is critical when this transaction is aborted.
6. When there is a write-read conflict, the write operation would be failed.

The first assumption is the important one and the implementation of the next few assumptions are all based on assumption one, so it will be explained first. For a single server service, two numbering system, incrementing number and timestamp, could be used to create a unique identifier for each transaction. Incrementing number is the easiest way to number transactions, but the prerequisite of implementing this method is a stable running server or a mechanism to record the latest issued number, otherwise when the system crashes and restarts, the numbering system would also restart from the original initial number.

According to assumption 4, the original entity would not be locked when there is a read operation in another transaction, since this transaction would write on a copy of this entity, and the read operation access on the latest committed version, these two operations do conflict. The new update operation implemented in the algorithm in this paper would be different from the traditional method which performs updates directly on the original entity, so a new update method should be considered, and this method would not conflict other operations. To guarantee the achievement of assumption 6, each entity needs to store more additional information.

6. The Assumptions Impacts on the Implementation

For the implementation of this algorithm in this paper, the second numbering system is adopted considering that there is only one single server in the implementation. Although a backup mechanism is designed and deployed to tolerate server crash and it provides the change to attach a number recording approach, the timestamp numbering system is still chosen due to the single responsibility principle. The benefit of choosing this numbering system is that timestamp would be always unique, and there is no need for a numbering recording mechanism even if the system restarted. Another reason for choosing this approach is the easy accessing property of timestamp and the timestamp can help to achieve assumption 6.

In terms of entity storage, it is not only storing the entity itself but also some additional information, including an identifier of the issued transaction and an identifier of transaction which make it invalid. Through observing these two identifiers and the states of transactions, the visibility of each entity to transactions can be judged. Additionally, an entity should also have a read-timestamp field and a write-timestamp filed to keep track of the identifier of the last transaction that read it and the last transaction that wrote it. The sample entities are shown in the following Figure:

ID	Name	Created_Tid	Expired_Tid	LastRead_Tid	LastWrite_Tid
1	Bob	1591330375	1591330380	1591330375	1591330380
1	Aaron	1591330380	0	1591330380	1591330380

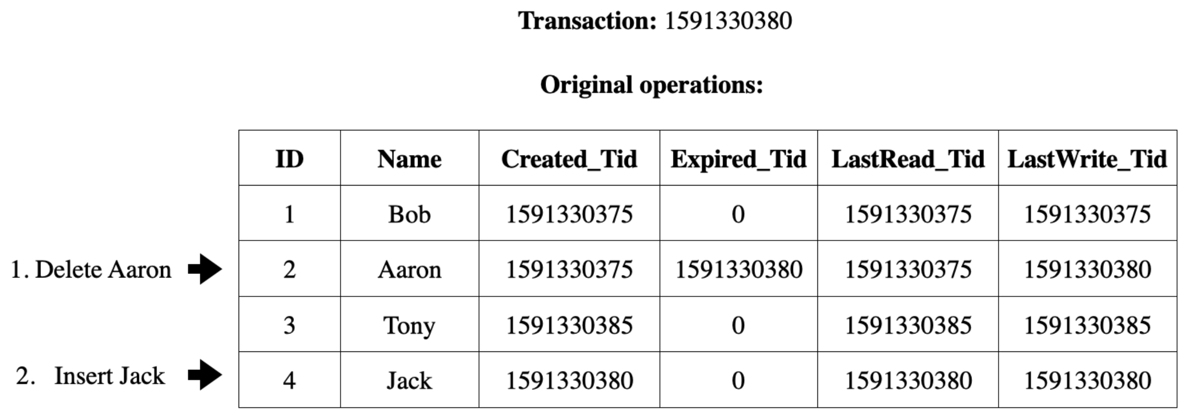
Figure 1 Sample Entities

To make a new update operation without interference other read and write operations, the general way is making this operation based on the existed operations of the system, so the algorithm discussed in this paper combine creation and deletion together to make the new update operation. Importantly, the deletion operations are different from the traditional deletion operation, it would not actually delete an entity, but making the original entity into a state of being temporarily expired, when this transaction is successful committed, the temporary expired state of this entity would be transformed into the permanently expired state. Temporary expired state can guarantee the accessibility of this entity by other transactions. In an update operation, an entity would be made into the temporary expired state, and an updated copy entity would be written into the system.

Furthermore, simultaneous transactions cannot make a modification to the same entity. If this happens there are two ways to handle this:

1. Rollback the transaction and abort all the changes made by this transaction.
2. Make the second transaction waiting until the entity released by the first transaction and becomes available. This has some special challenges with performance and potential reread errors.

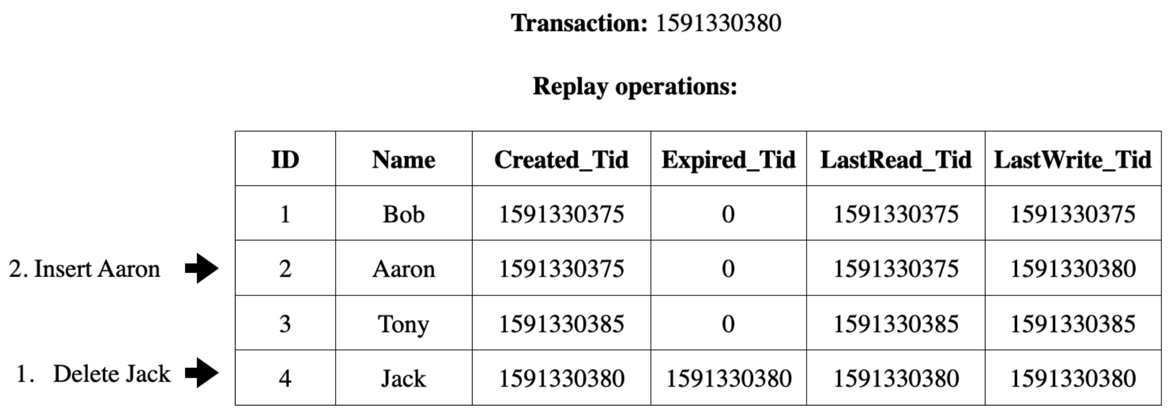
The second method would have some special challenges and may lead to some potential reread errors, so the method one is chosen and applied to the algorithm in this paper. To achieve this functionality, all the changes made in a transaction should be recorded into the corresponding log, when rollback a transaction the database management system replays all the changed in reverse. This requires that all rollbacks actions would be replayed correctly even if the application is shut down forcefully. The rollback processes are shown in follow:



Transaction: 1591330380

Original operations:

	ID	Name	Created_Tid	Expired_Tid	LastRead_Tid	LastWrite_Tid
	1	Bob	1591330375	0	1591330375	1591330375
1. Delete Aaron	2	Aaron	1591330375	1591330380	1591330375	1591330380
	3	Tony	1591330385	0	1591330385	1591330385
2. Insert Jack	4	Jack	1591330380	0	1591330380	1591330380


Transaction: 1591330380

Replay operations:

ID	Name	Created_Tid	Expired_Tid	LastRead_Tid	LastWrite_Tid
1	Bob	1591330375	0	1591330375	1591330375
2	Aaron	1591330375	0	1591330375	1591330380
3	Tony	1591330385	0	1591330385	1591330385
4	Jack	1591330380	1591330380	1591330380	1591330380

2. Insert Aaron
1. Delete Jack

Figure 2 Rollback Process

7. The Implementation

To demonstrate how this algorithm works, the whole system is divided into client part and server part. The algorithm is running in the server part, and there is no real database in the server part and all the entities will be stored in the memory when the server is running. The backup system writes the memory data into a local file periodically. When the server part shuts down, the backup system could restore the previous data into the server memory from the backup file. For the client part, we adopt socket TCP protocol to connect to the server to guarantee the stability of connection and the integrity of message. When the begin message of transactions arrives in the server, the numbing system would serially assign a timestamp to each transaction. For each entity, the server would update each entity's read-timestamp and write-timestamp separately when a transaction executes the corresponding actions.

8. Shortcomings of multi-version concurrency control

The disadvantage of this algorithm is that each entity requires additional storage space. The traditional method to guarantee data consistency is using the lock, when an entity is accessed by a transaction, then this entity would be locked and no other transactions can access it until it is released. However, this method would diminish the concurrency ability of database management system. The nature of the algorithm in this paper is that each entity would have multiple versions at the same time, when transactions want to access the same entity, each transaction would access the corresponding visible version of the entity. For each entity, it would have many different versions, and some of these versions would be invalid. Invalid entities are the entities which would not be accessed anymore by any transactions. When the system experiences a heavy-traffic period, there would have plenty of invalid entities, and storing these expired entities would be a huge waste of storage. In addition, large amounts of invalid entities could also diminish the query efficiency of the system, so how to recycle these expired entities would be a critical issue for this algorithm.

According to assumption 2 and 3, a recycling approach should be applied into the algorithm discussed in this paper to improve the utilization of storage, and this method should not interference transactions' operations in this system, and guaranteeing data consistency would be the first priority for this method because valid data cannot be deleted by mistake.

The normal way is adopting a stop-wait protocol to achieve this approach, but the normal stop-wait method would stop any other transactions accessing entities, so this method is executed concurrently, and this method would record the system state as S_i when it begins to execute and the system state as S_e before it finished. Comparison is made between S_i and S_e, if these two states are not the same, this method would not be allowed to finish, since different states mean that some operations are made during the period of the recycling process, and fail the recycling process could guarantee no mistake deleting.

9.Conclusion

The database management system can be accessed by a large number of users simultaneously, therefore concurrency control becomes a significant issue for database systems. The major difference between multi-version concurrency control and other concurrency control algorithms is the ability to control concurrency. MVCC algorithm has a higher ability to help data management system to control concurrency. This algorithm handles data inconsistency though maintaining multiple versions of each entity. For every transaction, this algorithm assigns a one-way growing timestamp to the transaction and make a copy version of entity for each write operation. For a read operation, it can only get a snapshot of the committed entities in the database. As a result, when a read operation and a write operation conflicts, the multi-version can guarantee that these two operations execute on the corresponding version without conflictions, in other words, there is no need for blocks when read operations and write operations conflicts in logic, so the multi-version concurrency control algorithm could control concurrency in a higher level.

In this report, we state the application background of the multi-version concurrency control algorithm, introduce the research progress of this algorithm, and demonstrate how we implement this algorithm in a simulated database management system. In Addition, we also refer to the disadvantage of this algorithm and how to overcome this problem.

Additionally, to make the system tolerating more fails we also implement a crash handling method for each client part and server part, and a timeout handling approach to improve the ability of the system to tolerate errors.

In the following sections, we will talk about the possible improvements of our implementation in the future work. In addition, we will also discuss the future direction of MVCC algorithm, which may improve the efficiency of the algorithm.

10. Implement improvements

As we discussed above, we already implemented the MVCC algorithm in this project, and it is useful for handling some common issues in a real database system. However, there are still some aspects can be improved, and we will discuss some improvements that we can achieve in the further study.

At first, there is only one server in the system, and we do not have a standby server. Therefore, if there are some unexpected problems occurring in the server, the system will crash and cannot provide services until the server restarts. If this situation happens, it will have a bad influence on all users and lead to several problems, like data loss. Therefore, hot standby will be implemented in the future study. The principle of hot standby is to use two servers to back up each other and perform the same service together. When one server fails, the other server is responsible for the service task, so that the system can automatically guarantee the continuous service without human intervention. The standby server solves the problem that the service is interrupted when the primary server fails.

Another enhancement that can be done is implementing a more efficient garbage collection method. There are many versions for each entity. However, some of them are not useful anymore, storing these expired entities would be a huge waste of storage. Therefore, if a more efficient garbage collection method is implemented, it can contribute to releasing the heavy burden of the server. In addition, if there are less useless entities, the query efficiency of the database can be improved.

Additionally, we should increase the maximum load of our server. In this project, we regard one laptop as the server of our system. However, its maximum load is not enough for a real database system, and we do not execute a stress testing of our system. The purpose of the stress testing is to improve the reliability and stability of the database system and reduce the downtime and loss of the system by performing repeatable load test. In the future work, we will try to increase the maximum load of our system and execute a necessary stress test.

11. Future direction

In this section, we will focus on discussing the future developments of MVCC algorithm. One possible improved method introduced by Prime is the mixed method. In this method, multiple certified versions of data items can be exploited rather than only the newest version. The main idea of its implementation is combining locking with timestamping. One new problem introduced by this method is consistent timestamping generation. Therefore, in order to solve this problem, and combine locking and timestamping, we must render their synchronization orders consistent. One significant advantage of this method is that queries never delay or cause the abort of updaters, and updaters never delay or cause the abort of queries. Prime's algorithm is most naturally implemented in a centralized DBS because of the need to totally order certify events.

Multiversion of each entity is the major feature of MVCC algorithm. However, it also brings an inevitable problem. In order to store these multi-versions of all the entities, more extra storage is required. In addition, more line checking and some extra maintenance work are needed. Therefore, one future direction can be reducing the need for extra storage, which can contribute to improving the efficiency of this algorithm.

12. Reference

Agrawal, D., & Sengupta, S. (1989, June). Modular synchronization in multiversion databases: Version control and concurrency control. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data* (pp. 408-417).

Bayer, R., Heller, H., & Reiser, A. (1980). Parallelism and recovery in database systems. *ACM Transactions on Database Systems (TODS)*, *5*(2), 139-156.

Bernstein, P. A., & Goodman, N. (1981). Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, *13*(2), 185-221.

Bernstein, P. A., & Goodman, N. (1983). Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, *8*(4), 465-483.

Bernstein, P. A., Shipman, D. W., & Wong, W. S. (1979). Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, (3), 203-216.

Carey, M. J. (1983). *Multiple versions and the performance of optimistic concurrency control*. University of Wisconsin-Madison Department of Computer Sciences.

Carey, M. J., & Muhanna, W. A. (1986). The performance of multiversion concurrency control algorithms. *ACM Transactions on Computer Systems (TOCS)*, *4*(4), 338-378.

Di Sanzo, P., Ciciani, B., Quaglia, F., & Romano, P. (2008, September). A performance model of multi-version concurrency control. In *2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems* (pp. 1-10). IEEE.

Eswaran, K. P., Gray, J. N., Lorie, R. A., & Traiger, I. L. (1976). The notions of consistency and predicate locks in a database system. *Communications of the ACM*, *19*(11), 624-633.

Kung, H. T., & Robinson, J. T. (1981). On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, *6*(2), 213-226.

Lomet, D., Fekete, A., Wang, R., & Ward, P. (2012, April). Multi-version concurrency via timestamp range conflict management. In *2012 IEEE 28th International Conference on Data Engineering* (pp. 714-725). IEEE.

Papadimitriou, C. H. (1979). The serializability of concurrent database updates. *Journal of the ACM (JACM)*, *26*(4), 631-653.

Reed, D. P. (1978). *Naming and synchronization in a decentralized computer system* (Doctoral dissertation, Massachusetts Institute of Technology).

Stearns, R. E., & Rosenkrantz, D. J. (1981, April). Distributed database concurrency controls using before-values. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data* (pp. 74-83).

DuBourdieux, D. (1982, February). Implementation of Distributed Transactions. In *Berkeley Workshop* (pp. 81-94).