Survey Paper for Concurrency Control in Distributed Database Management Systems

Zhongtian Ouyang
Department of Computer Science
University of Toronto
Toronto, Canada
leo.ouyang@mail.utoronto.ca

Yihao Ni
Department of Computer Science
University of Toronto
Toronto, Canada
ricky.ni@mail.utoronto.ca

Justin Lai
Department of Computer Science
University of Toronto
Toronto, Canada
justinholam.lai@mail.utoronto.ca

***Abstract*—Distributed Database Management System is widely used across the world nowadays. In general, data is distributed as fragmentation or replicated at various locations across the database system. In case of replication of copy of the database, the distributed database management system must always update all the database copies, such that the data is consistent across all sites. In case of fragmentation of distributed database, efficient transaction is the key for optimal performance. In this survey paper, we will discuss our research on consistency concurrency control algorithm to resolve the consistency problem in replicated copies of distributed database, and effective algorithm and protocols to perform transactions more efficiently.**

Keywords—Survey paper, Concurrency control, Distributed Database Management Systems

I. INTRODUCTION

This survey paper is aimed at audiences with basic understanding of DBMS and are interested in the concurrency control problem. To understand this paper, one need to know background knowledges including how basic concurrency control algorithms, such as Two-Phase Locking (2PL) and Optimistic Concurrency Control (OCC), works.

Unlike in Centralized Database Management Systems, in Distributed Database Management Systems, the storage devices are not attached to a common processor, instead they form an interconnected network while appeared as a unit to users. Some basic structure of distributing data are Fragmentation, Replication, Non-replicated & Non-fragmented. Notice that the storage devices in a DDBMS don't have to locate in the same physical location.

There are many advantages in implementing a distributed Database, for instance: protection of valuable data, performance improvement, higher reliability, easier to scale, etc. Easier to scale is possibly the most desired aspect. With Centralized DBMS, we need to develop better hardware to achieve that. With DDBMS, we can simply add more computer to the network.

We choose to discuss concurrency control in distributed database management system because we believe it is an important area in database. With better concurrency control algorithms, we could encourage more parallelism in execution and increase the throughput of databases. As DDBMS are widely used nowadays, any improvement of performance on the concurrency control protocols in DDBMS could bring large impacts. Some examples of popular distributed database include MongoDB, Hadoop and more. The idea of distributed ledger in Blockchain is also an implementation of DDBMS.

Although distributed database management systems are widely used, we are still facing many difficult challenges in designing them and implementing them. Concurrency control is one of the major problems that a distributed database management system environment has. Most of these issues only exist in distributed database environment, but not in a centralized database environment, so we need specialized algorithms to tackle these problems. For example: Failure of Individual Sites, Failure of Communication Links, Distributed Deadlock, Distributed commit, Dealing with multiple copies of the data items, sync between different sites.

In this survey paper, we discuss three different approaches proposed in the three papers to solve various problems in the concurrency control in DDBMS: Speculation-Based Locking Protocols, Adaptive and Speculative Concurrency Control, Efficient Concurrency control mechanism. The first two algorithms focus on improving the throughput and performance in DDBMS without replications. The third one focus on solving the synchronization problem among different storage devices with replications.

The classification schema of our survey paper is shown in figure 1. In section II, III, IV, we will present each of the three approaches respectively and discuss our opinion on the approaches. Section V will be a conclusion of the survey paper.

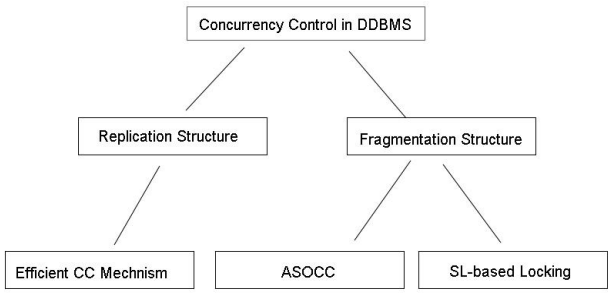

Fig. 1. Classification Schema

II. SPECULATION-BASED LOCKING PROTOCOLS

A. Terminology in this section

read-only transactions (ROT): transactions that only read data record
UT: update transactions
2PL: two phase locking
2PC: two phase commit in distributed database system
SI-based: Snapshot-Isolation based
SL-based: Speculation-based

B. Purpose of the Protocol

In reference [1], the SL-based protocol is presented and a way to accommodate it to distributed database management system is proposed by Mohit Goyal, T. Ragunathan and P. Krishna Reddy. The proposal aims at providing a solution that could improve the performance and accuracy when executing read-only transactions in DDBMS. The authors choose to tackle only ROTs because of the large portion of information access requests in modern DDBMS. They suggest that current existing methods have drawbacks. 2PL's performance is not ideal with data contention while SI-based protocol sacrifice accuracy for better performance. The following protocols are a paraphrase of protocols presented in reference [1].

C. Basic SL-based ROT protocol

1) Synchronous SL-based protocol for ROTs:

In this protocol, instead of usual R and W locks in 2PL, there are four different type of locks. The lock compatibility matrix is shown in Figure 2. When a ROT wants to read a data record, it requests an RR lock on the object. When a UT wants to wants to read a data record, it requests an RU lock on the object. Executive write(EW) lock and speculative write(SPW) lock are for writing. When a UT wants to write a data object x, it first requests an EW lock of that object. After the write is done, the EW lock is converted into SPW lock. The SPW lock is hold by UT until it commits/aborts.

When a ROT have a conflict with a UT, that is, the ROT requests a RR lock on an data object that has its EW lock or SPW lock hold by an UT, it should follows the following steps:

1. If the UT holds the EW lock, ROT wait for the lock to be converted to SPW lock.
2. If the UT holds the SPW lock, ROT accesses both the before image and after image of the data object.
3. ROT synchronously carries out speculative executions based on both the before image and the after image of the data object respectively.

Lock requested by T_j	**Lock held by T_i**			
	RR	RU	EW	SPW
RR	yes	yes	no	ssp_yes
RU	yes	yes	no	no
EW	no	no	no	no

Fig. 2. SSLR Lock Compatibility Matrix[1]

Identify applicable funding agency here. If none, delete this text box.

Lock requested by T_j	**Lock held by T_i**			
	RR	RU	EW	SPW
RR	yes	yes	asp_yes	asp_yes
RU	yes	yes	no	no
EW	no	no	no	no

Fig. 3. ASLR Lock Compatibility Matrix[1]

4. When the ROT commit, the choice of appropriate speculative execution depends on the state of conflicting UT. If the UT have committed successfully, retain the speculative execution based on the after image of the data object. If the UT is active or have aborted, retain the speculative execution based on the before image of the data object.

2) Asynchronous SL-based protocol for ROTs:

The locks used with ASLR are the same as SSLR. The lock compatibility matrix is shown in Figure 3. We can see that in ASLR, difference from in SSLR, RR lock are compatible with both EW lock and SPW lock. So in ASLR, when a ROT have a conflict with a UT, it should follows the following steps:

1. ROT accesses the before image of data object and carries out the corresponding speculative execution.
2. When UT's EW lock on the data object is converted to a SPW lock, ROT accesses the after image of data object and carries out the corresponding speculative execution.
3. When one of the speculative executions finished, check whether any of the completed speculative executions is valid depending on the state of the conflicting UT. If it does, ROT retain this speculative execution, abort all other speculative executions and commit. Else, keep going until we get a valid speculative execution.

3) Comparison between SSLR and ASLR with examples:

The examples in Figure 4 and Figure 5 demonstrate different behaviours between SSLR and ASLR under the same situation. In the examples, T1 and T3 are UTs and T2 is ROT. T21 and T22 are speculative executions of T2. Obviously, T21 is retained as the appropriate speculative execution in both examples because when T2 want to commit, T1 is not yet finished.

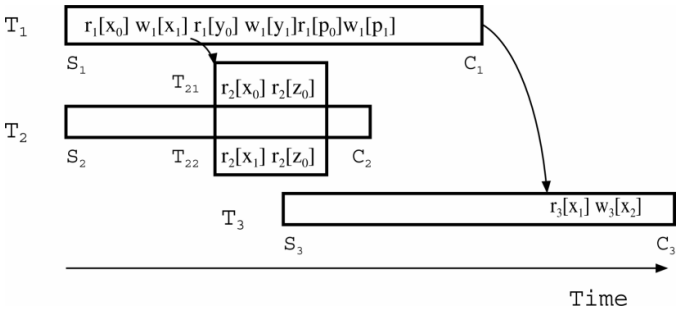

Fig. 4. SSLR Example[1]

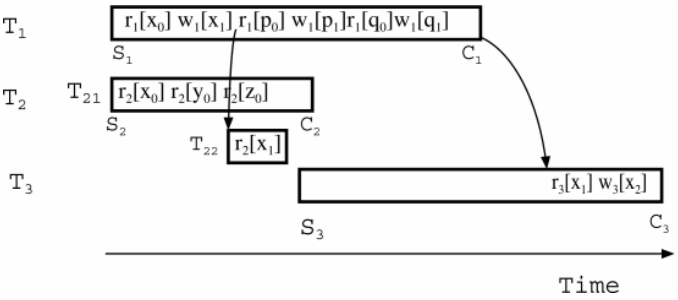

Fig. 5. ASLR Example[1]

D. Extend SL-based ROT protocol to DDBMS

In the DDBMS scenario, the processing of transaction is divided into two phases, execution phase and commit phase.

i) Protocol for UTs:

Execution phase: Distributed 2PL is used with the modification of locks as we described in the previous section. Also, whenever after-image is produced, it is communicated to the object site and target sites where ROTs are waiting.

Commit phase: 2PC. In first phase, it sends PREPARE message to all the participant sites. If all sites respond with "yes" then it issues GLOBAL COMMIT message. If any one of the sites respond negatively then Ti is aborted and GLOBAL ABORT message is sent to all participants sites.[1]

ii) Protocols for ROTs:

Like basic SL-based protocols, there are two different protocols for ROTs, the distributed synchronous speculative locking protocol for ROTs (DSSLR) and the distributed asynchronous speculative locking protocol for ROTs (DASLR). In both DSSLR and DASLR, two additional data structures are added. A "dependent_set" is maintained for each ROT, which contains IDs of the conflicting UTs. A "dependent_list" is maintained for each speculative execution, which contains IDs of UTs from which it has used an after-image when performing the speculative execution.

1) DSSLR:

Execution phase steps:

1. ROT send lock requests to object sites. For each requests, do step 2 to 4.
2. If the lock request conflict with UTs, ROT wait for the after-image of the data object to be produced.
3. Both before image and after image of the data object are sent to ROT's home site.
4. ROT synchronously carries out speculative executions based on both the before image and after image of the data object. Add UT's ID to ROT's "dependent_set". Add UT's ID to "dependent_list" of speculative executions which used the after image.
5. When all the requested locks are granted, ROT complete the executions and turn into commit phase.

Commit phase steps:

1. When all the speculative executions complete, ROT asks the home sites of the UTs in the "dependent_set" for their status to see whether they are committed or aborted/active
2. Base on the statuses of UTs, ROT select an appropriate speculative execution using "dependent_list".

2) DASLR:

Execution phase steps:

1. ROT send lock requests to object sites. For each requests, do step 2 to 4.
2. If the lock request conflict with UTs, ROT gets the before-image of the data object and carries out corresponding speculative execution. Add UT's ID to ROT's "dependent_set".
3. When after image is ready, it will be sent to ROT. ROT carries out additional speculative executions based on the after image of the data object. Add UT's ID to "dependent_list" of speculative executions which used the after image.
4. When all the requested locks are granted, ROT complete the executions and turn into commit phase.

Commit phase steps:

6. Whenever one of the speculative executions complete, ROT asks the home sites of the UTs in the "dependent_set" for their status to see whether they are committed or aborted/active.
7. Base on the statuses of UTs, ROT decide if this is an appropriate speculative execution using "dependent_list". If it is, the transaction commit. If it is not, wait for the next speculative execution to complete and repeat the steps.

E. Commentary on this approach

This proposal is important because it improve the performance of the ROTs while not sacrificing the accuracy of data. It does so by interpret all the possible outcomes of ROTs with conflicting UTs and choose the appropriate one at the time of commit. It reduces the amount of time a ROT needs to wait for UT and use that time to precompute the results. The improvement of performance can be observed on the experiment results in the paper shown in Figure 6. However, in this approach, the CPU usage would be very inefficient if there are multiple conflicting UTs for a ROT. The number of speculative executions will be growing exponentially. Another problem is that this protocol won't work with replicated database; it is designed to work with DDBMS with non-replicated structure, and it can't handle synchronization problem among storage devices, so in the future, extending the protocol to work replicated database structure could be a direction to develop. In addition, there is a lot of messages communicated in the process. Since the latency in network is not negligible, it is time-consuming to exchange these messages. It would be beneficial to optimize the number of messages need during the execution process.

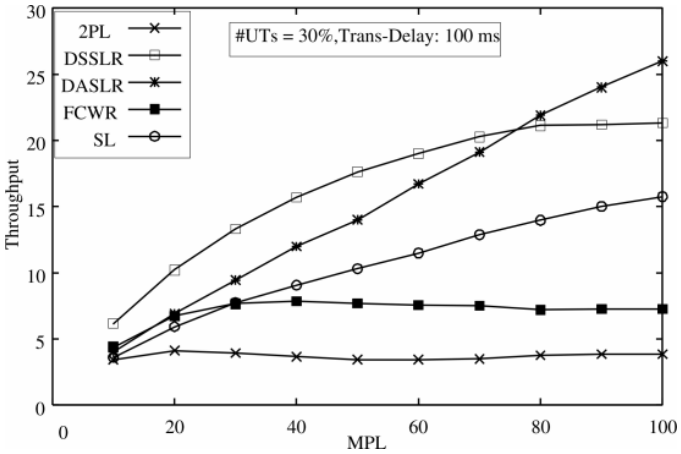

Fig. 6. Performance of different protocols[1]

III. ADAPTIVE AND SPECULATIVE CONCURRENCY CONTROL

A. *Terminology in this section*

OCC: optimistic concurrency control
2PL: two-phase locking (the 2PL adopted here is the strong strict 2PL with Wait-Die policy)
ASOCC: Adaptive and Speculative Optimistic Concurrency Control

B. *Purpose of the Protocol*

This algorithm described in reference [2] aims at providing a solution to improve the performance of concurrency control. The authors mention that when the workload is dynamic, existing concurrency control protocols often do not scale well. They suggest that it is necessary to take into account insights from the workload at runtime to tune the concurrency control scheme autonomously.

C. *Adaptive and Speculative Optimistic Concurrency Control*

The ASOCC protocol is a combination of multiple protocols widely used. It switches between different concurrency control strategies with respect to the data access pattern. There are three types of data classified under ASOCC: cold, hot and warm. Whether data is hot or cold is based on the frequency of data accesses, while classification of warm data is based on the correlation of data accesses. In detail, for data records r1 and r2, r1 is hot but r2 is not. If the access to r2 is correlated with the access to r1, then r2 is considered to be warm with r1 [2]. And the definition of data access correlation is as follows: For data records r1 and r2 if the access to r1 accompanies the access to r2 in the same transaction with high probability, then the access to r1 is deemed to be correlated with the access to r2 [2].

In ASOCC, there are three cases of abort or restart (as illustrated in Fig. 7):

- Abort due to failed validation: occurs when cold or warm data validation fails.

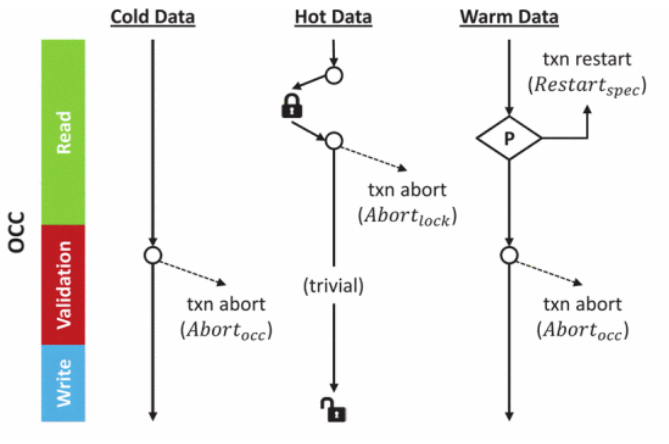

Fig. 7. The ASOCC protocol [2]

- Abort due to failed lock acquisition: occurs when lock acquisition for hot data access being rejected.
- Speculative restart: follows the speculation strategy, warm data only.

Access to cold data follows the OCC scheme, while for hot data accesses, the 2PL scheme is embedded into the OCC scheme (it acquires the corresponding data lock in the read phase and release the lock in write phase). This reason for doing this is that it is beneficial to abort transactions that are going to abort as early as possible. If a data record is accessed frequently, making it hot, then the transaction has higher probability to be destined to abort. In this case, abort due to failed lock acquisition is more efficient than abort due to failed validation regarding the CPU cycles. What is more, under low contention workload, a transaction is less likely to abort, in which case OCC performs better comparing to the pessimistic 2PL scheme. Access to warm data includes a speculative validation, if it detects any conflict, transaction restart is triggered. As illustrated in Fig. 7, transaction abort and restart for hot and warm data happens in read phase while transaction abort for cold data happens later in validation phase.

D. *Details about transaction restart*

The ASOCC protocol designs a speculative transaction restart for warm data. The algorithm only considers speculative transaction restart when warm data accesses precede the correlated hot data access in the same transaction. A key observation regarding warm data accesses is that accesses to warm data could be potentially contended. Instead of validating before commit, if any related lock request has experienced pending, then each transaction speculatively validates data accesses at the beginning. Transaction restart is triggered if the conjunctive validations of the warm data that are correlated with the accessing hot data fail at least once. By doing so, necessary early restart of transaction processing can be guaranteed by failed validations.

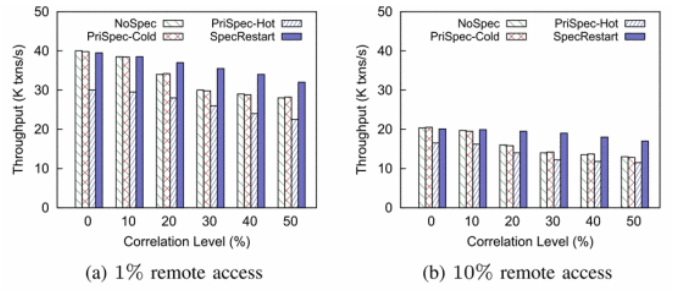

Fig. 8. Effectiveness of speculative concurrency control[2]

E. *Commentary on this approach*

This algorithm improves performance by saving CPU cycles wasted on transactions that will finally be aborted. According to the performance evaluation (Fig. 8), when data correlation level is at medium level, the performance is improved significantly. This is because the increased contention of warm data accesses tends to cause transaction abort and the speculative transaction early restart can help save the CPU cycles wasted on invalid processing. Aborts of distributed transactions are expensive, so saving those unnecessary aborts improves the performance a lot.

However, at the same time, the definitions in this algorithm are sometimes ambiguous. For example, it defines Data Access Correlation as two data records being in the same transaction with high probability, without noticing what is the exact boundary of two data records being correlated or not. And the authors wrote that classification for cold and hot data is based on the frequency of data accesses, but they did not give the actual boundary as well. This means anyone adopting this algorithm needs to tune those boundary before using it, and those optimal boundaries may change overtime. Besides, the authors mention that we need to tune the categorization algorithm to prevent from bringing in too may false positives of warm data. But tuning might be complicated and not always accurate. If the categorization is not accurate, although ASOCC guarantees that for warm data, if the validation will fail then early restart is triggered in read phase, this may lead to many unnecessary early restarts. As a result, the performance can be worse than not using this algorithm. Furthermore, according to the performance evaluation (Fig. 8), when correlation level is low or high, this algorithm does not make much difference compared to when the speculation strategy is disabled. At the same time, adopting this algorithm means that extra data (e.g. whether two data records are correlated or not, whether a record is hot/cold/warm) must be stored in the database. And these fields needs to be updated overtime.

IV. EFFICIENT CONCURRENCY CONTROL MECHANISM

A. *Terminiology in this Section*

AP: Application Process

DBMP: Database Managing Process

B. *Background of Replicated copies in Distributed Database*

Concurrency Control and Recovery is one of the major problems that are having in a distributed database management system environment. Most of these issues only exist in distributed database environment, but not in a centralized database environment. For example: Failure of Individual Sites, Failure of Communication Links, Distributed Deadlock, Distributed commit, Dealing with multiple copies of the data items.

In this surveyed paper, we are going to present one of the researched consistency concurrency control algorithms which can solve the above issues. There are some important terms/ protocols to be defined, before we present our research algorithm for concurrency control in Distributed Database Management Systems.

Wait: A conflicting transaction is waiting for the completion of the actions of other transaction

Timestamp: Each transaction has a unique timestamp value, and if there are conflicting actions of the transactions, the execution order will follow their corresponding timestamp values.

Rollback: Moving the database back to the original starting point of a transaction action

Locking: In the researched algorithm, a transaction will send a lock request to the lock manager, when it wants to update the data in this distributed database.

Voting Method: A lock request is sent to all the sites, including a copy of the data item. Each copy of data item has its own lock to the corresponding site, and the corresponding site can grant or deny the request. If a transaction gets more than half of the votes across all sites, it can hold the lock. Therefore, the decision of locking is dependent on every individual involved sites.

C. *Purpose of the Algorithm*

There are two major problems of adapting the original voting method in a distributed database management system. Firstly, all the individual sites across the distributed database might not be updating the data each time when a lock request arrives. Although majority protocol is used, the individual site has the authority to grant or deny the lock requesting updating its own data, such that the distributed database across all sites will not be consistent. Secondly, if a query request arrives on an individual site which has not been updated before, then the query request will return an incorrect old value of data.

To resolve the Consistency problems and Dirty Read problems from the original voting method in a typical distributed database management system, Parul Tomar and Suruchi from the YMCA University of Science and Technology proposed a research on Efficient Concurrency Control Mechanism for Distributed Databases. They proposed an algorithm to resolve the previously mentioned issues (Consistency Problem, Dirty Read [3]) from the Majority protocol in the voting method.

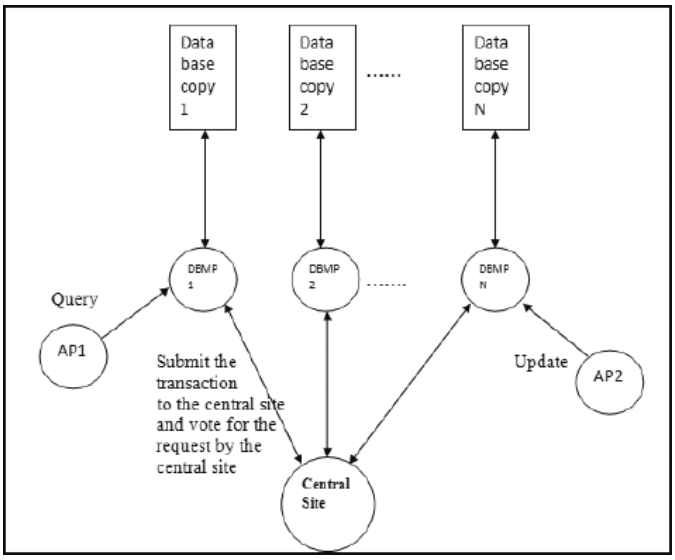

Fig. 9. Every Database copy site is only accessible through its local DBMP[3]

D. Environment Conditions for this Algorithm

The following database environment conditions must be satisfied, in order to allow the proposed algorithm to be ran successfully. A transaction will be worked as an execution tool for users to interact with the Distributed Database Management System. There are various sites across the Database Management System. Each site contains a replicated copy of the Database, and this database copy is only accessible through a Database Managing Process, which resides at the corresponding site (Figure 9). There is also a central site which monitors all the connected database sites. Therefore, this algorithm assumes this distributed database as a redundant database, that all logical data should be stored at every single site.

An application process (AP) is a process tool to initiate the read and write operation in the database. Each Database Managing Process (DBMP) is working like an application process (AP) to initiate the transaction operations such as query and write in the database. Each site will also maintain information about the timestamp of the last transaction that was performed in the corresponding database copy. And the central site will hold all the key information about the data and the transactions.

Any site can initiate a transaction request to the database, and all transactions will be forwarded to the central site, to maintain a unique timestamp value for each transaction across the Distributed Database. Then, the central site will forward the new transaction request to all the connected sites, in order to maintain the consistency of data across the disturbed database system. Each site can have their own authority to grant or deny the transaction request to their corresponding replicated database copy. And the central site will maintain data information regarding which site has voted YES OR NO on certain transaction request.

Each site should also maintain a consistent data structure from (Figure 10).

Transaction ID: Transactions ids to differentiate different transactions

Time Stamp: Time Stamp to store when the data item has updated

Data Item: Name of the Data Item for which the transaction request is made

Data Value: The Data value which has been updated at the given timestamp

Status: Status of whether this transaction request has been approved or rejected on this corresponding site

E. Workthrough of the Algorithm

When all the environment conditions are satisfied across all sites within the Distributed Database system, we can break down the proposed algorithm to resolve the problems in Concurrency Control from Parul Tomar and Suruchi into 5 different steps, when a transaction is to be executed in the database.

Step1: A Transaction request Ti for modifying a data item in the database is forwarded to the central site. The monitoring site will then execute the request and send the metadata to all the other sites. All the other sites should check for acceptance for this transaction request for Data item Qi, and return the approval status to the central site.

Step2: After receiving the metadata and transaction request from the central site, site 1 will check whether Transaction Ti can lock its requested data or not from its corresponding replicated database copy. If site 1 is able to lock its data copy for data item Qi by its Database Managing Process (Local Lock manger), site 1 will then send a message to the central site to indicate its approval status for Transaction Request Tj.

Step 3: The central site will count the received votes for Transaction request Ti. If more than half of the sites approved this transaction Ti, then Transaction Ti can be ran on those sites which approved this transaction.

Step 4: If a site sends a REJECT approval status to the central site, but the majority is still claimed, then then transaction Ti will not be performed at this site. The central site will then resend to those sites who reject this transaction. Step 1 and Step 4 are then repeated for these sites.

Step 5: If the old request is still not performed at the rejected site when a new transaction request Tj arrives at this site, then the old request will be rejected automatically. Then, Step 1 to 5 will be executed according to the order of the time stamp value from all the transaction requests to this site.

Transaction ID	Timestamp	Data Item	Data Value	Status

Fig. 10. Data Sructure of a metadata in each site[3]

F. *Commentary on this Algorithm*

The use of timestamp values in the data structure of the metadata in each individual site helps promote an approach to consist the data across the database. This algorithm certainly solves the Consistency problem and Dirty Read problem from the original approach of the Voting method in a replicated Distributed Database. Replications of Database from the central sites helps improved reliability of the data, increased overall performance, and lower communication costs.

Because of the dynamic nature of this distributed database management system, inconsistency and complexity are major obstacles to overcome for a complete reliable and efficient DBMS. Therefore, concurrency control in distributed database management system will continuously require more researchers to develop a completely sustainable and effective solution.

V. CONCLUSION

In this paper we presented several algorithms to solve various problems in the concurrency control in DDBMS. The algorithms focus on either improving the throughput and performance in DDBMS without replications or solving the synchronization problem among different storage devices with replications. Overall, we can observe those algorithms improves the performance and accuracy of DDBMS in different ways. However, these works are still immature and need more development. Moreover, some of them only works in certain circumstances. Finding more general solutions is an open research problem.

REFERENCES

[1] M. Goyal, T. Ragunathan, and P. K. Reddy, "Extending Speculation-Based Protocols for Processing Read-Only Transactions in Distributed Database Systems," 2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC), 2010.

[2] Q. Lin, G. Chen, and M. Zhang, "On the Design of Adaptive and Speculative Concurrency Control in Distributed Databases," 2018 IEEE 34th International Conference on Data Engineering (ICDE), 2018.

[3] P. Tomar, and Suruchi, "Efficient concurrency control mechanism for distributed databases," 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom), 2016.