IJCSC International Journal of Computer Science & Communication Vol. 1, No. 1, January-June 2010, pp. 137-141

An Approach for Concurrency Control in Distributed Database System

Arun Kumar Yadav[1] & Ajay Agarwal[2]

[1]Asst. Professor, Nikhil institute of Engineering & Management, Mathura (UP),
[2]Professor & Head, Krishna Institute of Engg. & Technology, Ghaziabad (UP),
[1]aky26977@yahoo.in, [2]ajay.aagar@gmail.com

ABSTRACT

Various concurrency control algorithms have been proposed for use in distributed database systems. Even, the large number of available algorithms, but the fact that distributed database systems are becoming a commercial reality, distributed concurrency control performance tradeoffs are still not well understood. In this paper, we focus on some of the important issues by studying four representative algorithms - Distributed 2PL, Wound-Wait, Basic Timestamp ordering and a Distributed optimistic algorithm - using a detailed model of a distributed DBMS.
Keywords: Master Process (M), Cohort Process (Ci)

1. INTRODUCTION

From the past years, Distributed Databases have taken attention in the database research community. Data distribution and replication offer opportunities for improving performance through parallel query execution and load balancing as well as increasing the availability of data. In fact, these opportunities have played a major role in motivating the design of the current generation of database machines (e.g., [1], [2]). This paper addresses some of the important performance issues related to these algorithms.

Most of the distributed concurrency control algorithms come into one of three basic classes: locking algorithms [3,4,5,6,7], Timestamp algorithms, [8,9,1], and optimistic (or certification) algorithms [10,11,12, 13]. Many proposed algorithms reviewed [14] and describe how additional algorithms may be synthesized by combining basic mechanisms from the locking and timestamp classes [14].

Given the many proposed distributed concurrency control algorithms, a number of researchers have undertaken studies of their performance. For example, the behavior of various distributed locking algorithms was investigated in [15, 4, 16, 17]. Where algorithms with varying degrees of centralization of locking and approaches to deadlock handling have been studied and compared with one another. Several distributed Timestamp based algorithms were examined in [18]. A qualitative study addressing performance issues for a number of distributed locking and timestamp algorithms was presented in [14]. The performance of locking was compared with that of basic timestamp ordering in [19], with basic and multi-version timestamp ordering in [20].

While the distributed concurrency control performance studies to date have been useful but a number of important questions remain unanswered. These include:

1. How do the performance characteristics of the various basic algorithm classes compare under alternative assumptions about the nature of the database, the workload, and the computational environment?
2. How does the distributed nature of transactions affect the behavior of the various classes of concurrency control algorithms?
3. How much of a performance penalty must be incur for synchronization and updates when data is replicated for availability or query performance reasons?

We examine four concurrency control algorithms in this study, including two locking algorithms, a timestamp algorithm and an optimistic algorithm. The algorithms considered span a wide range of characteristics in terms of how conflicts are detected and resolved. Section 2 describes our choice of concurrency control algorithms. The structure and characteristics of our model are described in Section 3 and section 4 describes the testing of algorithms. Finally, Section 5 summarizes the main conclusions of this study and raises questions that we plan to address in the future.

2. DISTRIBUTED CONCURRENCY CONTROL ALGORITHMS

For this study, we have chosen to examine four algorithms that we consider to be representative of the basic design space for distributed concurrency control

mechanisms. We summarize the salient aspects of these four algorithms in this section. In order to do this, we must first explain the structure that we have assumed for distributed transactions.

2.1. The Structure of Distributed Transactions

Figure 1 shows a general distributed transaction in terms of the processes involved in its execution. Each transaction has a master process (M) that runs at its site of origination. The master process in turn sets up a collection of Cohort processes (Ci) to perform the actual processing involved in running the transaction. Since virtually all query processing strategies for distributed database systems involve accessing data at the site(s) where it resides, rather than accessing it remotely. There is at least one such cohort for each site where data is accessed by the transaction. In general, data may be replicated, in which case each cohort that update any data items is assumed to have one or more update (Uij) processes associated with it at other sites. In particular, a cohort will have an update process at each remote site that stores a copy of the data items that it updates. It communicates with its update processes for concurrency control purposes, and it also sends them copies of the relevant updates during the first phase of the commit protocol described below.

The centralized two-phase commit protocol will be used in conjunction with each of the concurrency control algorithms examined. The protocol works as follows [5]:

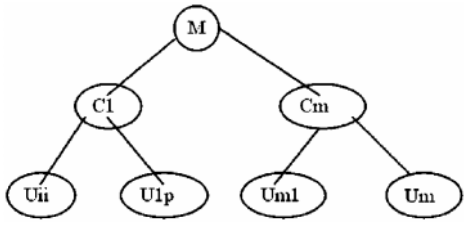


Fig 1: Distributed Transaction Structure

When a cohort finishes executing its Portion of a query, its sends an "execution complete" message to the master. When the master has received such a message from each cohort, it will initiate the commit protocol by sending "prepare to commit" messages to all sites. Assuming that a cohort wishes to commit, it sends a "prepared" message back to the master, and the master will send "commit" messages to each cohort after receiving prepared messages from all cohorts. The protocol ends with the master receiving "committed" messages from each of the cohorts. If any cohort is unable to commit, it will return a "cannot commit" message instead of a "prepared" message in the first phase, causing the master to send "abort" instead of "commit" messages in the second phase of the protocol.

When replica update processes are present, the commit protocol becomes a nested two-phase commit protocol. Messages flow between the master and the cohorts, and the cohorts in turn interact with their updaters. That is, each cohort sends "prepare to commit" messages to its updaters after receiving such a message from the master, and it gathers the responses from its updaters before sending a "prepared" message back to the master; phase two of the protocol is similarly modified.

2.2. Distributed Two-Phase Locking (2PL)

The first algorithm is the distributed "read any, write all" two-phase locking algorithm described in [15]. Transactions set read locks on items that they read, and they convert their read locks to write locks on items that need to be updated. To read an item, it suffices to set a read lock on any copy of the item, so the local copy is locked; to update an item, write locks are required on all copies. Write locks are obtained as the transaction executes, with the transaction blocking on a write request until all of the copies of the item to be updated have been successfully locked. All locks are held until the transaction has successfully committed or aborted.

Deadlock is a possibility, Local deadlocks are checked for any time a transaction blocks, and are resolved when necessary by restarting the transaction with the most recent initial startup time among those involved in the deadlock cycle. (A cohort is restarted by aborting it locally and sending an "abort" message to its master, which in turn notifies all of the processes involved in the transaction.) Global deadlock detection is handled by a "Snoop" process, which periodically requests waits-for information from all sites and then checks for and resolves any global deadlocks (using the same victim selection criteria as for local deadlocks). We do not associate the "Snoop" responsibility with any particular site. Instead, each site takes a turn being the "Snoop" site and then hands this task over to the next site. The "Snoop" responsibility thus rotates among the sites in a round-robin fashion, ensuring that no one site will become a bottleneck due to global deadlock detection costs.

2.3. Wound-Wait (WW)

The second algorithm is the distributed wound-wait locking algorithm, again with the "read any, write all" rule. It differs from 2PL in its handling of the deadlock problem: Rather than maintaining waits-for information and then checking for local and global deadlocks, deadlocks are prevented via the use of timestamps. Each transaction is numbered according to its initial startup time, and younger transactions are prevented from making older ones wait. If an older transaction requests a lock, and if the request would lead to the older transaction waiting for a younger transaction, the

younger transaction is "wounded" – it is restarted unless it is already in the second phase of its commit protocol (in which case the "wound" is not fatal, and is simply ignored). Younger transactions can wait for older transactions so that the possibility of deadlocks is eliminated.

2.4. Basic Timestamp Ordering (BTO)

The third algorithm is the basic timestamp ordering algorithm of [9, 14]. Like wound-wait, it employs transaction startup timestamps, but it uses them differently. Rather than using a locking approach, BTO associates timestamps with all recently accessed data items and requires that conflicting data accesses by transactions be performed in timestamp order. Transactions that attempt to perform out-of-order accesses are restarted. When a read request is received for an item, it is permitted if the timestamp of the requester exceeds the item's write timestamp. When a write request is received, it is permitted if the requester's timestamp exceeds the read timestamp of the item; in the event that the timestamp of the requester is less than the write timestamp of the item, the update is simply ignored (by the Thomas write rule [14]).

For replicated data, the "read any, write all" approach is used, so a read request may be sent to any copy while a write request must be sent to (and approved by) all copies. Integration of the algorithm with two-phase commit is accomplished as follows: Writers keep their updates in a private workspace until commit time.

2.5. Distributed Certification (OPT)

The fourth algorithm is the distributed, timestamp-based, optimistic concurrency control algorithm from [13], which operates by exchanging certification information during the commit protocol. For each data item, a read timestamp and a write timestamp are maintained. Transactions may read and update data items freely, storing any updates into a local workspace until commit time. For each read, the transaction must remember the version identifier (i.e., write timestamp) associated with the item when it was read. Then, when all of the transaction's cohorts have completed their work, and have reported back to the master, the transaction is assigned a globally unique timestamp. This timestamp is sent to each cohort in the "prepare to commit" message, and it is used to locally certify all of its reads and writes as follows: A read request is certified if (i) the version that was read is still the current version of the item, and (ii) no write with a newer timestamp has already been locally certified. A write request is certified if (i) no later reads have been certified and subsequently committed, and (ii) no later reads have been locally certified already.

3. MODELING A DISTRIBUTED DBMS

Figure 2 shows the general structure of the model. Each site in the model has four components: a source, which generates transactions and also maintains transaction-level performance information for the site, a transaction manager, which models the execution behavior of transactions, a concurrency control manager, which implements the details of a particular concurrency control algorithm, and a resource manager, which models the CPU and I/O resources of the site. In addition to these per-site components, the model also has a network manager, which models the behavior of the communications network. Figure 3 presents a slightly more detailed view of these components and their key interactions.

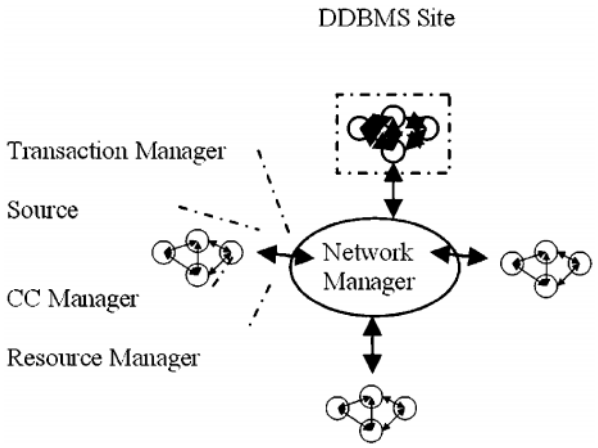


Fig 2: Distributed DBMS Model Structure

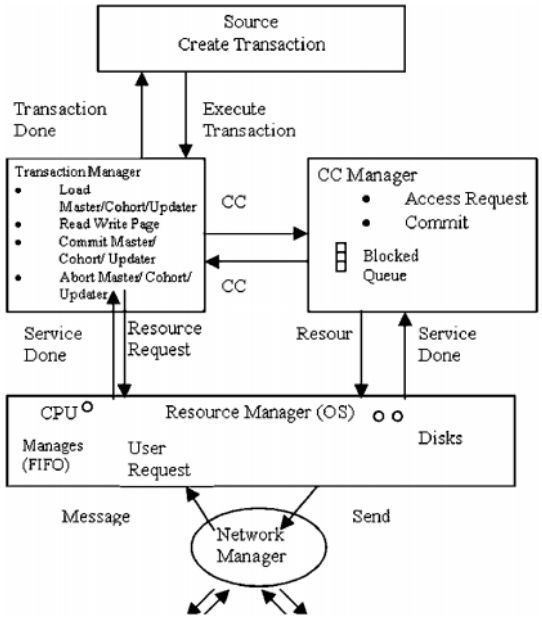


Fig 3: A Closer Look at the Model

3.1. The Transaction Manager

Each transaction in the workload will have a master process, a number of cohorts, and possibly a number of

updaters. As described earlier, the master resides at the site where the transaction was submitted. Each cohort makes a sequence of read and writes requests to one or more files that are stored at its site; a transaction has one cohort at each site where it needs to access data. Cohorts communicate with their updaters when remote write access permission is needed for replicated data, and the updaters then make the required write requests for local copies of the data on behalf of their cohorts. A transaction can execute in either a sequential or parallel fashion, depending on the execution pattern of the transaction class.

3.2. The Resource Manager

The resource manager can be viewed as a model of the operating system for a site; it manages the physical resources of the site, including its CPU and its disks. The resource manager provides CPU and I/O service to the transaction manager and concurrency control manager, and it also provides message sending services (which involve using the CPU resource). The transaction manager uses CPU and I/O resources for reading and writing disk pages, and it also sends messages. The concurrency control manager uses the CPU resource for processing requests, and it too sends messages.

3.3. The Network Manager

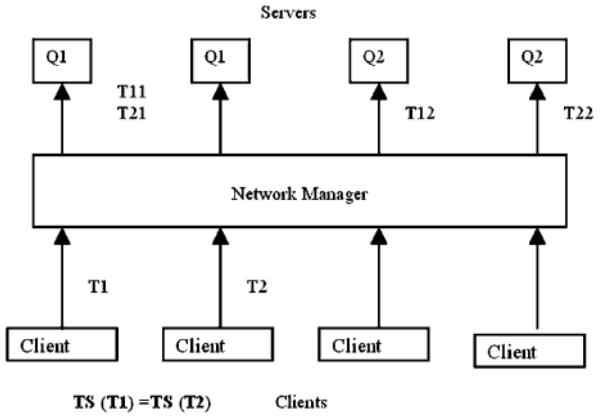
The network manager encapsulates the model of the communications network. Our network model is currently quite simplistic, acting just as a switch for routing messages from site to site. The characteristics of the network are isolated in this module; it would be a simple matter to replace our current model with a more sophisticated one in the future.

3.4. The Concurrency Control Manager

The concurrency control manager captures the semantics of a given concurrency control algorithm, and it is the only module that must be changed from algorithm to algorithm. As was illustrated in Figure 3, it is responsible for handling concurrency control requests made by the transaction manager, including read and write access requests, requests to get permission to commit a transaction, and several types of master and cohort management requests to initialize and terminate master and cohort processes.

4. TESTING OF ALGORITHMS

Suppose two transactions with the same time-stamp are submitted from the client 1 &2. The cohort processes (T11 & T12) of transaction T1 and T21 and T22 of transaction T2 are executing on server 1 & 3 and server 1 & 4 respectively.



In the first case, where the cohort processes (T11& T21) of transaction T1 and T2 updating the same data content Q1.

Two phase locking Protocol (2PL) will ensure that T11 and T21 update the data content in sequential manner. Both the cohorts at first make the request to lock the data content Q1. Database Management System will grant the lock either to the T11 or to the T21 depending on their arrival at server.

Wound-Wait (WW) protocol will uses to ensure the consistency of the database. If the TS (T11) <TS (T21) and the suppose Q1 is presently locked by the T21 then T21 will wounded to wait and Q1 will be allocated to the T11. When T11 will finish the execution then Q1 will given to the T21 for further execution.

Basic Time-Stamp Ordering (BTO) is uses to ensure that whether all the cohorts reading and updating the correct value version of the contents and ensuring the consistency of the database or not. If T11 have locked the data content and updating the data content then ***BTO*** protocol will ensure T11 is allowed to update only if TS(T11)>RTS(Q1) otherwise the updates will be ignored.

If T11 reading the data contents then it will ensure that if TS (T11)>WTS (Q1) then only T11 is permitted otherwise simply ignored.

Distribution Certification (OPT) protocol is uses to ensure the correct version of data contents read and update by the transactions. This protocol uses the read certification where it ensure that the read version of the data content is still current version and no write with newer time-stamp has been locally or globally certified. In the case of write request certification it ensures that no later read have been certified and subsequently commit and no later read have been locally or globally already certified.

In the Second case, where cohort processes (T12 & T22) of transactions T1 and T2 are updating the data content Q2 on server3 &4 respectively.

Two Phase locking Protocol (2PL) will ensure that data content Q2 should be updated in sequential manner.

T12 and T22 will make request to lock the Q2 data content. If the TS (T12) <TS (T22) then Q2 locked by T12 otherwise locked by the T22. If TS (T12) =TS (T22), both the cohorts want to lock all the copies of the Q2 data content but due to the same time-stamp lock can not be grant to the transaction T12 and T22. In this situation, there may be a global deadlock. This global deadlock handled by the "snoop" process. In this situation, one of the cohort either T12 or T22 will be selected as victim and restart again. So, lock can be granted to either T12 or to the T22.

Wound-Wait (WW) protocol will check if the TS (T12) < TS (22) then Q2 locked by T12 otherwise by the T22. In this case we are assuming that T22 was selected as a victim and submitted again. ***Basic Time-Stamp Ordering (BTO)*** is uses to ensure that whether all the cohorts reading and updating the correct value version of the contents and ensuring the consistency of the database or not. If T12 have locked the data content and updating the data content then ***BTO*** protocol will ensure T12 is allowed to update only if TS(T12)>RTS(Q2) otherwise the updates will be ignored. If T12 reading the data contents then it will ensure that if TS (T12)>WTS (Q2) then only T12 is permitted otherwise simply ignored. ***Distribution Certification (OPT)*** protocol will use the read and write certification to maintain the consistency of the data content.

In this testing, we find that in all the cases our algorithm is maintaining the consistency of the data contents an also avoiding and resolving the deadlock.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have tried to get rid of on distributed concurrency control performance tradeoffs by studying the performance of four representative algorithms - Distributed 2PL, wound-wait, basic timestamp ordering, and a distributed optimistic algorithm - using a common performance framework. We examined the performance of these algorithms under various degrees of contention, data replication, and workload "Distributions."

The combination of these results suggests that "optimistic locking," where transactions lock remote copies of data only as they enter into the commit protocol (at the risk of end-of-transaction deadlocks), may actually be the best performer in replicated databases where messages are costly, We plan to investigate this assumption in the future.

REFERENCES

[1] "Implementing Atomic Actions on Decentralized Data," *ACM Trans. on Comp. Sys. 1*, **1**, Feb. 1994.

[2] "A High Performance Backend Database Machine," *Proc. 12th VLDB Conf.*, Kyoto. Japan. Aug. 1996.

[3] "Locking and Deadlock Detection in Distributed Databases," *Proc. 3rd Berkeley Workshop on Dist. Data Mgmt. and Camp. Networks*, Aug, 2000.

[4] "The Effects of Concurrency Control on the Performance of a Distributed Data Management System," *Proc. 4th Berkeley Workshop on Dist. Data Mgmt. and Comp. Networks*, Aug. 2000.

[5] "Distributed Concurrency Control Performance: A Study of Algorithm, Distribution, Replication", *Comp. Scien.* Deptt. Madison, 2002.

[6] "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Trans. on Software Engineering* SE-5.3. May 2004.

[7] "Transactions and Consistency in Distributed Database Systems," *ACM Trans. on Database Sys.* 7.3. Sept. 2004.

[8] "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Trans.on Database Sys.* 4. 2 June 2003.

[9] "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems," Proc. 6th VLDB Cot& Mexico City, Mexico, Oct. 2003.

[10] "Correctness of Concurrency Control and Implications in Distributed Databases," Proc. COMPSAC '04 Conf. Chicago, IL, Nov. 2004.

[11] "Optimistic Methods for Concurrency Control in Distributed Database Systems," Proc. 7th VLDB Conf. Cannes, France, Sept. 2005.

[12] "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases," Proc. 6th Berkeley Workshop on Dist. Data Mgmt. and Comp. Networks, Feb. 2000.

[13] "Timestamp Based Certification Schemes for Transactions in Distributed Database Systems," Proc. ACM SIGMOD Conf., Austin, TX, May 2000.

[14] "Concurrency Control in Distributed Database Systems," ACM Comp. Surveys 13. 2, June 2001.

[15] "Performance of Update Algorithm for Replicated Data in a Distributed Database", Ph.D. Thesis, Comp. Sci. Dept., Stanford Univ., June 2006.

[16] "Performance of Two Phase Locking," Proc. 6th Berkeley Workshop on Dist. Data Mgmt. and Comp. Network, Feb. 2005.

[17] "Measured Performance of Time Interval Concurrency Control Techniques," Proc. 13th VLDB Conf., Brighton, England, Sept. 2005.

[18] "Performance Model of Timestamp-ordering Concurrency Control Algorithms in Distributed Databases" IEEE Trans. on Camp. C-36.9, Sept. 2000.

[19] "Concurrency Control Performance Issues", Ph.D. Thesis, Comp. Sci. Dept., Univ. of Toronto, Sept.2006.

[20] "Basic Timestamp, Multiple Version Timestamp, and Two-Phase Locking," *Proc. 9th VLDB Conf. Florence*, Italy, Nov. 2004.