A Distributed Architecture for Transactions Synchronization in Distributed Database Systems

Arun Kumar Yadav
Associate Professor & Head, Department of Computer Science & Engineering
Nikhil Institute of Engineering & Management
Mathura (U.P.), India

Dr. Ajay Agarwal
Professor & Head, Department of MCA
Krishna Institute of Engg. & Technology
Ghaziabad (U.P.), India

***Abstract*-Various concurrency control algorithms have been proposed for use in distributed database systems. But, the number of algorithms available for the distributed concurrency control, come into one of three basic classes: locking algorithms, Timestamp algorithms and optimistic (or certification) algorithms. In this paper we are presenting a Distributed Transaction Processing Model and an approach for concurrency control in distributed database systems. The analysis of our approach is a decomposition of the concurrency control problem into two major sub-problems: *read-write* and *write-write* synchronization. We describe a series of synchronization techniques for solving each sub-problem and will show how to combine these techniques into algorithms for solving the entire concurrency control problem. Such algorithms are called "concurrency control methods". Our approach concentrates on the structure and correctness of concurrency control methods and also the performance of such methods up to some extent.**

Keywords: Distributed Database Management System, Transaction Manager, Data Manager

I. INTRODUCTION

Concurrency control is the activity of coordinating concurrent accesses to a database in a multi-user Database Management System (DBMS). Concurrency control permits users to access a database in a multi-programmed manner provided each user is executing alone on a dedicated system. The main technical difficulty in attaining this goal is to prevent database updates performed by one user from interfering by another user. The concurrency control is very complex problem in Distributed DBMS (DDBMS) because (1) users may access data stored in many different computers in a distributed system, and (2) a concurrency control mechanism at one computer cannot instantaneously know about the interactions at other computers.

Concurrency control has been actively investigated for the past several years, and the problem for non-distributed DBMSs is well understood. A mathematical theory has been developed to analyze the problem, and one approach, called two-phase locking, has been accepted as a standard solution. Recent research on non-distributed concurrency control is mainly focused on improvements to two-phase locking, detailed performance analysis and optimization. Distributed concurrency control, by contrast, focuses on consistency, degree of concurrency and abort ratios. Numbers of concurrency control algorithms [1, 2, 3] have been proposed for DDBMS**,** and several have been implemented. These algorithms are usually complex, hard to understand, and difficult to prove correct because they are described in different terminologies and make different assumptions about the DDBMS environment.

Most of the concurrency control algorithms proposed previously were concerned with ensuring the atomicity property of the transactions in Distributed Database. We are proposing a standard model for the DDBMS environment. For analysis purpose we decompose the concurrency control problem into two major sub-problems, called *read-write* (rw) and *write-write* (ww) synchronization. Every concurrency control algorithm must include a sub-algorithm to solve each sub-problem. The first step towards understanding a concurrency control algorithm is to isolate the sub-algorithm employed for each sub-problem.

After studying relevant proposed algorithms [4, 5], we find that they are composition of only a few sub-algorithms. In fact, the sub-algorithms used by all practical DDBMS concurrency control algorithms are variations of just two basic techniques: two-phase locking and time-stamp ordering.

In the remainder of this paper, section 2 describes the concurrency control problems. The structure and functionality of our Distributed Transaction Processing model are described in section 3 and implementation of transaction synchronization and testing on our model in section 4. Finally, section 5 summarizes the main conclusion of this study and raises questions that we plan to address in the future.

II. CONCURRENCY CONTROL PROBLEM

A. *Concurrency Control Anomalies*

The goal of concurrency control is to prevent interference among users who are simultaneously accessing the same database. Let us illustrate the problem by

presenting two examples of inter-user interference. Both examples are based on online electronics funds transfer system accessed via remote automated teller machines (ATMs). In response to customer requests, ATMs retrieve data from a database, perform computations, and store results back into the database.

Anomaly 1: *lost updates*. Suppose two customers simultaneously try to deposit money into the same account. In the absence of concurrency control, these two activities could interfere (see Figure 1). The two ATMs handling the two customers could read the account balance at approximately same time, compute new balances in parallel, and then store the new balances back into the database. The net effect is incorrect. Although two customers deposited money, the database only reflects one activity; the other deposit is lost by the system.

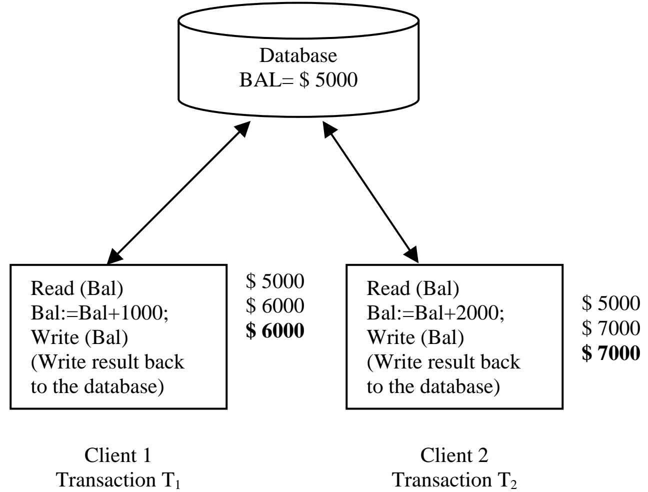

Figure 1: Lost update Anomaly

Anomaly 2: Inconsistent retrievals.

Suppose two customers simultaneously execute following transactions.

Customer 1: moving $1000 from saving account S to its checking account C.

Customer 2: print the total balance of S and C.

In the absence of concurrency control, these two transactions can interfere (see Figure 2). The first transaction might read the savings account balance, subtract $1000, and store the result back in the database. Then the second transaction might read the savings and checking accounts balances and print the total. Then the first transaction might finish the funds transfer by reading the checking account balance, adding $1000, and finally storing the result in the database. Unlike Anomaly 1, the final values placed into the database by this execution are correct. Still, the execution is incorrect because the balance printed by Customer 2 is $1000 short. These two examples are typical of the concurrency control problems that arise in DBMSs.

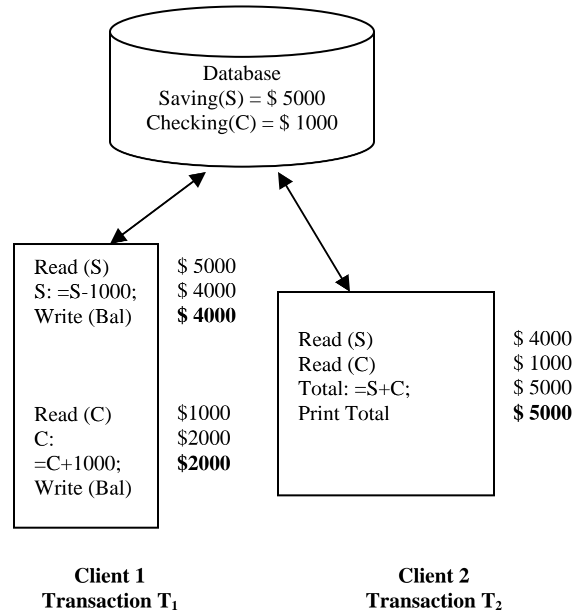

Figure 2: Inconsistent Retrieval Anomaly

B. *Comparison to Mutual Exclusion Problems*

The concurrency control problem in database system is similar in some respects of mutual exclusion in operating systems. The latter problem is concerned with coordinating access by concurrent processes to system resources such as memory, I/O devices, and CPU. Many solution techniques have been developed, including locks, semaphores, monitors, and serializers [6, 7, 8, 9].

Mutual Exclusion is one of the necessary and sufficient conditions for concurrency control methods to control concurrent accesses to shared resources. However, mutual exclusion that works for operating system does not necessarily work for DBMS, as illustrated by the following example. Suppose processes P_1 and P_2 require access to resources R_1 and R_2 at different points in their execution. In an operating system, the following interleaved execution of these processes is perfectly acceptable:

P_1 uses R_1
P_2 uses R_1
P_2 uses R_2
P_1 uses R_2

In a database, however, this execution is not always acceptable. Assume, for example, that P_2 transfer funds by debiting one account (R_1), then crediting another (R_2). If P_2 checks both balances, it will see R_1 *after* it has been debited, but see R_2 *before* it has been credited. Other differences between concurrency control and mutual exclusion are discussed in [10, 11, 12].

III. DISTRIBUTED TRANSACTION-PROCESSING MODEL

To understand how a concurrency control algorithm operates, one must understand how the algorithm fits into an overall DDBMS. In this section we propose a simple model for DDBMS, emphasizing how the DDBMS processes user interactions. Later we will explain how concurrency control algorithms operate in the context of this model. Our distributed transaction processing model differs from the centralized model in two areas: handling private workspaces and implementing two-phase commit.

DDBMS is a collection of sites interconnected by a network. It contains four components (see Figure 3):

- Transactions
- Transaction Managers (TMs)
- Data Managers (DMs)
- Data

Transactions communicate with TMs, TMs in turn communicate with DMs, and DMs manage the data. It is important to not that neither TMs communicate with other TMs, nor DMs communicate with other DMs.

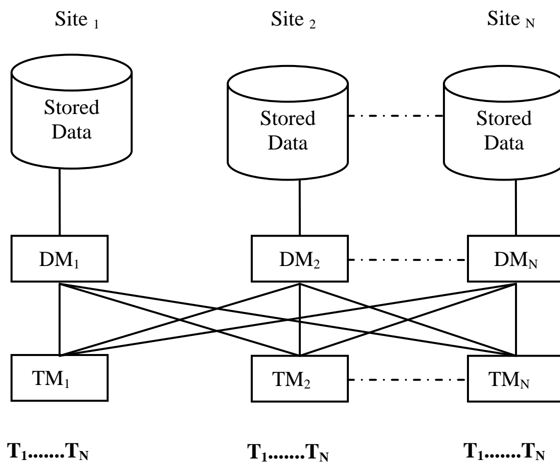

Figure 3: DDBMS Transaction processing Model

Each transaction executed in the DDBMS is supervised by a single TM, meaning that the transaction issues all of its database operations to that TM. Further, any distributed computation that is needed to execute transaction is managed by TM. TMs supervise interactions between users and the DDBMS while DMs manages the actual database.

Given *computer network* is assumed to be perfectly reliable i.e. if site A sends a message to site B, site B is guaranteed to receive the message without error or vice versa. In addition, we assume that between any pair of sites the network delivers messages in the order they were sent.

From a user's perspective, a database consists of a collection of logical data items, denoted X, Y, Z. A logical database state is an assignment of values to the logical data items composing a database. Each logical data item may be stored at any DM in the system or redundantly at several DMs. A stored copy of a logical data item is called a *stored* data item. In this paper, we use the term data item for stored data item. The stored copies of logical data item X are denoted X_l . . . . . X_m. We typically use X to denote an arbitrary stored data item. A stored database state is an assignment of values to the stored data items in a database. Users interact with the DDBMS by executing transactions. Transactions may be on-line queries expressed in a self-contained query language, or application programs written in a general-purpose programming language.

The concurrency control algorithms we study pay no attention to the computations performed by transactions. Instead, these algorithms make all of their decisions on the basis of the data items a transaction reads and writes, and so details of the form of transactions are unimportant in our analysis. However we do assume that transactions represent complete and correct computations; each transaction, if executed alone on an initially consistent database, would terminate, produce correct results, and leave the database consistent. The logical *readset* (correspondingly, *writeset*) of a transaction is the set of logical data items the transaction reads (or writes). Similarly, stored *readsets* and stored *writesets* are the stored data items that a transaction reads and writes.

The correctness of a concurrency control algorithm is defined relative to users' expectations regarding transaction execution. There are two correctness criteria: (1) users expect that each transaction submitted to the system will eventually be executed; (2) users expect the computation performed by each transaction to be the same whether it executes alone in a dedicated system or in parallel with other transactions in a multi-programmed system. Realizing this expectation is the principal issue in concurrency control.

Four operations are defined at the transaction TM interface:

- READ(X) returns the value of X (a logical data item) in the current logical database state
- WRITE(X, new-value) creates a new logical database state in which X has the specified new value.

Since transactions are assumed to represent complete computations, we use

- BEGIN and
- END operations to bracket transaction executions.

DMs manage stored database, functioning as back-end database processors. In response to commands from transactions, TMs issue commands to DMs specifying stored data items to be read or written. The details of the TM-DM interface constitute the core of our transaction-processing model.

In a centralized DBMS we assumed that (1) private workspaces were part of the TM, and (2) data could freely move between a transaction and its workspace, and between a workspace and the DM. These assumptions are not appropriate in a DDBMS because TMs and DMs may run at different sites and the movement of data between a TM and

a DM can be expensive. To reduce this cost, many DDBMSs employ query optimization procedures which regulate and reduce the flow of data between sites.

IV. TRANSACTIONS SYNCHRONIZATION BASED ON TWO-PHASE LOCKING

In this section, we have presented two-phase locking as the most widely used technique for Transaction Synchronization and implementation of two-phase locking (2PL) in Distributed Database. Two-phase locking (2PL) synchronizes reads and writes by explicitly detecting and preventing conflicts between concurrent operations. Before reading data item x, a transaction must "own" a read-lock on x. Before writing into x, it must "own" a write-lock on x. The locks are granted to transaction by two rules: (1) different transactions cannot simultaneously own conflicting locks; and (2) once a transaction releases the lock, it may never obtain additional locks.

The definition of conflicting lock depends on the type of synchronization being performed. The *rw synchronization* states that if a transaction has read-lock on x then no other transaction can have write-lock on x at the same time. For *rw synchronization* two locks conflicts exist if: (a) both locks are on the same data item, and (b) one is a read-lock and the other is a write-lock.

On the other hand, the *ww synchronization* states that if a transaction has write-lock on x then no other transaction can have write-lock on x at the same time. For *ww synchronization* two locks conflicts if: (a) both locks are on the same data item, and (b) both are write-locks.

The ownership rule of lock states that every transaction is to obtain locks in a two-phase manner. During the *growing phase* the transaction obtains locks without releasing any locks. By releasing a lock the transaction enters the *shrinking phase (see Figure 4).* During this phase the transaction releases locks, and is prohibited from obtaining additional locks.

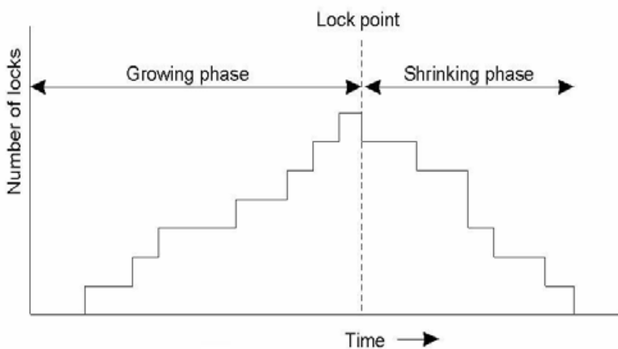

Figure 4: Two-phase Locking

Another variant of two-phase locking (2PL) is the strict-two phase locking, which requires not only the locking be two phases, but also all locks taken by a transaction be held until that transaction commits. When the transaction terminates (or aborts), all remaining locks are automatically released.

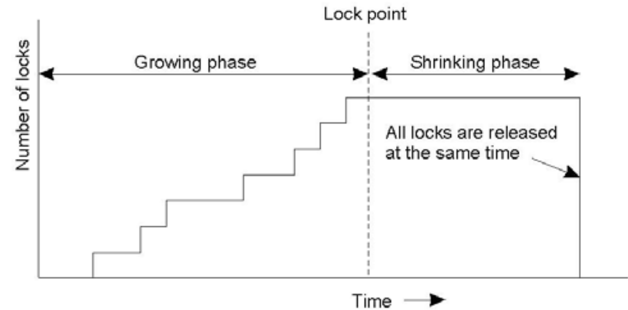

Figure 5: Strict two-phase locking

A common variation requires that transactions obtain all locks before beginning the main execution. This variation is called *pre-declaration*. Some systems also require that transactions hold all locks until termination.

Two-phase locking is a correct synchronization technique, meaning that 2PL attains an acyclic $\rightarrow_{RWR}$ ($\rightarrow_{WW}$) relation when used for rw (ww) synchronization [13, 14, 15]. The serialization order attained by 2PL is determined by the order in which transactions obtain locks. The end point of the growing phase, when a transaction owns its final lock, is called the locked point of the transaction [13] (see Figure 4 & 5). Let E be an execution in which 2PL is used for rw (ww) synchronization. The $\rightarrow_{RWR}$ ($\rightarrow_{WW}$) relation induced by E is identical to the relation induced by a serial execution E' in which every transaction executes at its locked point. Thus the locked points of E determine a serialization order for E.

A. Implementation of 2PL

The 2PL implementation is a 2PL scheduler, a software module that receives lock requests and locks release request, processes them according to the 2PL specification.

To implement 2PL in a distributed database is to distribute the schedulers along with the database, placing the scheduler for data item X at the DM where X is stored. In this implementation read-locks may be implicitly requested by dm-reads and write-locks may be implicitly requested by pre-writes. If the requested lock cannot be granted, the operation is placed on a waiting queue for the desired data item. This may produce deadlock, as discussed in Section *E*. Write-locks are implicitly released by dm-writes. However, to release read-locks, special lock-release operations are required. These lock releases may be transmitted in parallel with the dm-writes, since the dm-writes signal the start of the shrinking phase. When a lock is released, the operations on the waiting queue of that data item are processed in first-in/ first-out (FIFO) order.

Note that this implementation "automatically" handles redundant data correctly. Suppose logical data item X has m copies viz. X_l, . . . , X_m. If basic 2PL is used for rw synchronization, a transaction may read any copy and need

only obtain a read-lock on the copy of X it actually reads. However, if a transaction updates X, then it must update all m copies of X, and so must obtain write-locks on all m copies of X.

B. *Primary Copy 2PL*

Primary copy 2PL is a 2PL technique that handles the data redundancy. One copy of each logical data item is designated the primary copy; before accessing any copy of the logical data item, the appropriate lock must be obtained on the primary copy.

For read-locks, this technique requires more communication than basic 2PL. Suppose X_l is the primary copy of logical data item X, and suppose transaction T wishes to read some other copy X_i, of X. To read X_i, T must communicate with two DMs, the DM where X_l is stored (so T can lock X_l) and the DM where X_l. By contrast, under basic 2PL, T would only communicate with X_l's DM. For write-locks, however, primary copy 2PL does not incur extra communication. Suppose T wishes to update X. Under basic 2PL, T would issue pre-writes to all copies of X (thereby requesting write-locks on these data items) and then issue dm-writes to all copies. Under primary copy 2PL the same operations would be required, but only the pre-write (X_l) would request a write-lock. That is, pre-writes would be sent for X_l . . . X_m, but the pre-writes for X_2 . . . . . X_m would not implicitly request write-locks.

C. *2PL Voting Protocol*

Voting 2PL (or majority consensus 2PL) is another 2PL implementation that exploits data redundancy. Voting 2PL is derived from the majority consensus technique in and is only suitable for ww synchronization.

To understand voting, we must examine it in the context of two-phase commit. Suppose transaction T wants to write into X. Its TM sends pre-writes to each DM holding a copy of X. For the voting protocol, the DM always responds immediately. It acknowledges receipt of the pre-write and says "lock-set" or "lock blocked". After the TM receives acknowledgments from the DMs, it counts the number of "lock-set" responses: if the number constitutes a majority, then the TM behaves as all locks were set. Otherwise, it waits for "lockset" operations from DMs that originally said "lock blocked". It will eventually receive enough "lockset" operations to proceed.

Since only one transaction can hold a majority of locks on X at a time, only one transaction writing into X can be in its second commit phase at any time. All copies of X thereby have the same sequence of writes applied to them. A transaction's locked point occurs when it has obtained a majority of its write-locks on each data item in its write-set. When updating many data items, a transaction must obtain a majority of locks on every data item before it issues any dm-writes.

In principle, voting 2PL could be adapted for rw synchronization. Before reading any copy of X transaction requests read-locks on all copies of X; when a majority of locks are set, the transaction may read any copy. This technique works but is overly strong: Correctness only requires that a single copy of X be locked-namely, the copy that is read-yet this technique requests locks on all copies. For this reason we consider that voting 2PL to be inappropriate for rw synchronization.

D. *Centralized 2PL*

Instead of distributing the 2PL schedulers, one can centralize the scheduler at a single site [16, 17]. Before accessing data at any site, appropriate locks must be obtained from the central 2PL scheduler. So, for example, to perform dm-read(X) where X is not stored at the central site, the TM must first request a read-lock on X from the central site, wait for the central site to acknowledge that the lock has been granted, then send dm-read(X) to the DM that holds X. (To save some communication, one can have the TM send both the lock request and dm-read (X) to the central site and let the central site directly forward dm-read(X) to X's DM; the DM then responds to the TM when dm-read (X) has been processed.) Like primary copy 2PL, this approach tends to require more communication than basic 2PL, since dm-reads and pre-writes usually cannot implicitly request locks.

E. *Deadlock Detection and Prevention*

The implementations of 2PL force transactions to wait if locks are not granted. If this waiting is uncontrolled then deadlock may arises as shown in Figure 6.

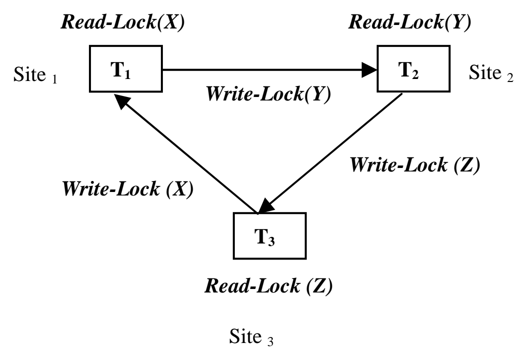

Figure 6: Deadlock

Deadlock situations can be characterized by waits-for graphs [18, 3], and directed graphs that indicate which transactions are waiting for which other transactions to release the locks. Nodes of the graph represent transactions, and edges represent the "waiting-for" relationship: an edge is drawn from transaction T_i, to transaction T_j if T_i, is waiting for a lock currently owned by T_j. The deadlock exists in a system if and only if the waits-for graph contains a cycle**.** Two general techniques can be used for deadlock resolution: deadlock prevention and deadlock detection.

1) Deadlock Prevention: Deadlock prevention scheme protect the system from occurring the deadlock, a

transaction is restarted when the system assume that deadlock may occur. To implement deadlock prevention, 2PL schedulers are modified as follows. When a lock request is denied, the scheduler tests the requesting transaction (say $T_{i,}$) and the transaction that currently owns the lock (say T_j). If T_i and T_j pass the test then T_i is permitted to wait for T_j as usual. Otherwise, one of the transactions is aborted. If T_i is restarted, the deadlock prevention algorithm is called non-preemptive; if T_j is restarted, the algorithm is called preemptive. The test applied by the scheduler must guarantee that if T_i, waits for T_j, then deadlock cannot result. One simple approach is never to let T_i wait for T_j. This trivially prevents deadlock but may forces many restarts.

A better approach is to assign priorities to transactions and to test priorities to decide whether T_i, can wait for T_j. For example, we could let T_i, wait for T_j if T_i, has lower priority than T_j (if T_i and T_j have equal priorities, T_i, cannot wait for T_j, or vice versa). This test prevents deadlock because, for every edge (T_i, T_j) in the waits-for graph, T_i, has lower priority than T_j. Since a cycle is a path from a node to itself and since T_i, cannot have lower priority than itself, no cycle can exist.

One problem with the previous approach is that there may be cyclic restart of the transactions because some transactions could be continually restarted without ever finishing. To avoid this problem, [19] proposes "timestamps" as priorities. A transaction's timestamp is the time at which it begins executing, so old transactions have higher priority than new ones transaction. This technique requires that a unique time-stamp should be assigned to each transaction by its TM.

Two timestamp-based deadlock prevention schemes are proposed in [18]. Wait-Die is the non-preemptive technique. Suppose transaction T_i, tries to wait for T_j. If T_i, has lower priority than T_j (i.e., T_i, is younger than T_j), then T_i, is permitted to wait. Otherwise, it is aborted ("dies") and must be restart. It is important that new time-stamp should not be assigned to $T_{i,}$ when it restarts. Wound-Wait is the preemptive and opposite to Wait-Die. If T_i has higher priority than T_j, then T_i waits; otherwise T_j is aborted.

Both Wait-Die and Wound-Wait avoid cyclic restart. However, in Wound-Wait an old transaction may be restarted many times, while in Wait-Die old transactions never restart. It is suggested in [20] that Wound-Wait induces fewer restarts in total.

If we use preemptive deadlock prevention with two-phase commit then a transaction must not be aborted once the second phase of two-phase commit has started. If a preemptive technique wishes to abort T_j, it checks with T_j's TM and cancels the abort if T_j has entered in second phase. No deadlock can result because if Tj is in the second phase, it cannot be waiting for any transactions.

Deadlock avoidance technique is based on preordering of resources which avoids restarts altogether. This technique requires pre-declaration of locks (each transaction obtains all its locks before execution). Data items are numbered and each transaction requests locks one at a time in numeric order. The priority of a transaction is the number of the highest numbered lock it owns. Since a transaction can only wait for transactions with higher priority, no deadlocks can occur. In addition to requiring pre-declaration, a principal disadvantage of this technique is that it forces locks to be obtained sequentially, which increases the response time.

2) Deadlock Detection in DDBMS: In deadlock detection technique, transactions wait for each other in an uncontrolled manner and are aborted when deadlock actually occurs. Deadlocks are detected by explicitly constructing the waits-for graph and searching it for cycles. Cycles in a graph can be found efficiently proposed in [21]. If a cycle is found, one transaction on the cycle, called the victim, is aborted, thereby breaking the deadlock. To minimize the cost of restarting the victim, victim selection is usually based on the amount of resources used by each transaction on the cycle.

The principal difficulty in implementing deadlock detection in a distributed database is constructing the waits-for graph efficiently. Each 2PL scheduler can easily construct the waits-for graph based on the waits-for relationships local to that scheduler. However, these local waits-for graphs are not sufficient to characterize all deadlocks in the distributed system (see Figure 7 & 8). Instead, local waits-for graphs must be combined into a more "global" waits-for graph. Centralized 2PL does not have this problem, since there is only one scheduler. We describe two techniques for constructing global waits-for graphs: centralized and hierarchical deadlock detection.

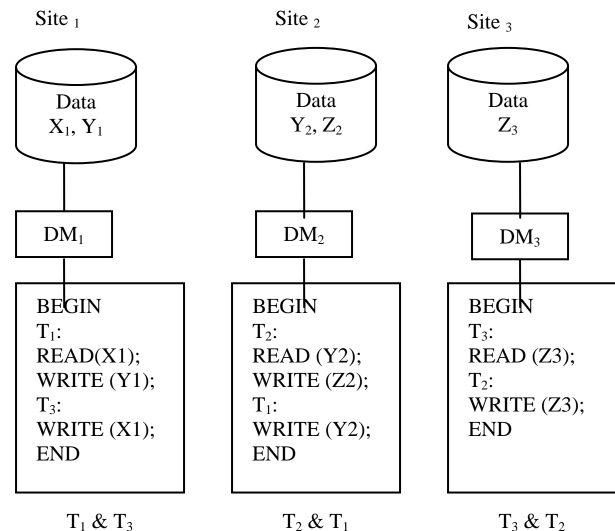

Figure 7: Locks are request by the transactions at DMs

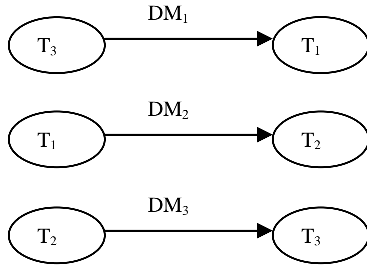

Figure 8: Deadlock on multi-sites

In the centralized approach, one site is designated the deadlock detector for the distributed system [22, 23]. Periodically (e.g., every few minutes) each scheduler sends its local waits-for graph to the deadlock detector. The deadlock detector combines the local graphs into a system wide waits-for graph by constructing the union of the local graphs.

In the hierarchical approach, the database sites are organized into a hierarchy (or tree), with a deadlock detector at each node of the hierarchy. For example, one might group sites by region, then by country, then by continent. Deadlocks that are local to a single site are detected at that site; deadlocks involving two or more sites of the same region are detected by the regional deadlock detector; and so on.

Although centralized and hierarchical deadlock detection differs in detail, both involve periodic transmission of local waits-for information to one or more deadlock detector sites. The periodic nature of the process introduces two problems. First, a deadlock may exist for several minutes without being detected, causing response-time degradation. The solution, executing the deadlock detector more frequently, increases the cost of deadlock detection. Second, a transaction T may be restarted for reasons other than concurrency control (e.g., its site crashed). Until T's restart propagates to the deadlock detector, the deadlock detector can find a cycle in the waits-for graph that includes T. Such a cycle is called a phantom deadlock. When the deadlock detector discovers a phantom deadlock, it may unnecessarily restart a transaction other than T. Special precautions are also needed to avoid unnecessary restarts for deadlocks in voting 2PL.

A major cost of deadlock detection is the restarting of partially executed transactions. Pre-declaration can be used to reduce this cost. By obtaining a transaction's locks before it executes, the system will only restart transactions that have not yet executed. Thus little work is wasted by the restart.

V. CONCLUSION

We have presented a transaction processing model for concurrency control in distributed database systems. Our approach has two main components: (1) a system model that provides common terminology and concepts used in a variety of concurrency control algorithms, and (2) a problem decomposition that decomposes concurrency control algorithms into *read-write* and write-write synchronization sub-algorithms.

We have considered synchronization sub-algorithms outside the context of specific concurrency control algorithms. Generally most of the database synchronization algorithms are variations of two basic techniques- two-phase locking (2PL) and timestamp ordering technique. We have described the principal variations for each technique, without these variations we cannot claims all possible variations. In addition, we have described deadlock problems and its resolution that must be solved to make each technique effective.

We have shown how to combine the described techniques to form complete concurrency control algorithms. We have considered almost all concurrency control algorithms described previously in the literature, plus several new ones.

The focal point of this paper has primarily been the structure and correctness of synchronization techniques and concurrency control algorithms. We have left one important issue as performance. The performance factor of concurrency control algorithms depends on system throughput and transaction response time. Four cost factors influence the performance: inter-site communication, local processing, transaction restarts, and transaction blocking. The impact of these cost factors on system throughput and response time varies from algorithm to algorithm, system to system, and application to application. The performance analysis of algorithm is remains and will be our future work. We hope, and really recommend, that future work on distributed concurrency control will concentrate on the performance of algorithms.

REFERENCES

[1] Stonebraker, M. "Concurrency control and consistency of multiple copies of data in distributed INGRES, IEEE Trans. Soflw. Eng. SE-5, 3 (2003), 188-194.

[2] Thomas, R.H. "A solution to the concurrency control problem for multiple copy databases," in Proc. 2004 COMPCON Conf. (IEEE), New York.

[3] King, P. P., and Collmeyer, A J. "Database sharing-an efficient method for supporting concurrent processes," in *Proc. 1974 Nat. Computer Conf.,* vol. 42, AFIPS Press, Arlington,Va., 2003.

[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. 2006.

[5] "Distributed Concurrency Control Performance: A Study of Algorithm, Distribution, Replication", *Comp. Scien.* Deptt. Madison, 2008.

[6] Brinch-Hansen, P. Operating system principles, Prentice-Hall, Englewood Cliffs, N. J., 2003

[7] Dijkstra, E.W. "Hierarchical orderingof sequential processes," Acta Inf. 1, 2 (2002), 115-138.

[8] Hewitt, C.E. "Protection and synchronization in actor systems," Working Paper No. 83, M.I.T. Artificial Intelligence Lab, Cambridge, Mass., Nov. 2006.

[9] Hoare, C. A.R. "Monitors. An operating system structuring concept," Communication. ACM 17, 10 (2004), 549-557.

[10] Chamberlin, D. D., Boyce, R. F., and Traiger, I.L. **"A** deadlock-free scheme for resource allocation in a database environment," *Info. Proc.* 74, North-Holland, Amsterdam, 2002.

[11] Deppe, M. E., and Fry, J. P. "Distributed databases' A summary of research," in Computer networks, vol. 10, no. 2, North-Holland, Amsterdam, 2005.

[12] Rothnie, J. B., and Goodman, N. "A survey of research and development in distributed databases systems," in *Proe 3rd Int. Conf. Very Large Data Bases* (IEEE), Tokyo, Japan, 2006.

[13] Bernstein, P. A., Shipman, D. W., and Wono, W.S. "Formal Aspects of Serializability in Database Concurrency Control," *IEEE Trans. Softw Eng.* SE-5, 3 (2002), 203-215.

[14] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I.L. "The notions of consistency and predicate locks in a database system." Communication. ACM 19, 11 (2003), 624-633.

[15] Papadimitriou, C. H. "Serializability of concurrent updates," J. ACM 26, 4 (2002), 631-653.

[16] Alsberg, P. A, and Day, J.D. "A principle for resilient sharing of distributed resources," in Proc. 2nd Int. Conf. Software Eng., 2002, pp. 562-570.

[17] Garcia-Molina, H. "Performance of update algorithms for replicated data in a distributed database," Ph.D. dissertation, Computer Science Dept., Stanford Univ., Stanford, Calif., 2002.

[18] Holt, R.C. "Some deadlock properties of computer systems," *Comput. Surv. 4, 3* (2002) 179-195.

[19] Rosenkrantz, D. J., Stearns, R E., and Lewis, P.M. "System level concurrency control for distributed database systems," *ACM Trans. Database Syst. 3, 2* (2002), 178-198.

[20] Reed, D.P. "Naming and synchronization in a decentralized computer system, Ph.D. dissertation, Dept. of Electrical Engineering, M.I.T., Cambridge, Mass., Sept., 2004.

[21] Aho, A. V., Hopcroft, E., and Ullman, J. D. The design and analysis of computer algorithms, Addison-Wesley, Reading, Mass., 2001.

[22] Gray, J.N. "Notes on database operating systems," in Operating Systems: An Advanced Course, vol. 60, Lecture Notes in Computer Science, Springer-Verlag, New York, 2002, pp. 393-481.

[23] Menasce, D. A., and Muntz, R. R. "Locking and deadlock detection in distributed databases," *IEEE Trans. Software Engineering.* SE-5, 3 (2004), 195-202.

AUTHORS PROFILE

Arun Kumar Yadav received the B.E. (Computer Science & Engineering) and M.Tech (Information Technology) degree in 2000 and 2004, respectively. Presently pursuing Doctorate Degree (Ph.D) in Computer Science from Singhania University, Rajasthan and working as Associate Professor & Head in the Department of Information Technology in Venkteshwar Institute of Technology, Indore (M.P.), India. His research interest includes Distributed Database Security, Data Structures and Algorithms. He is a member of IACSIT and IAENG.

Dr. Ajay Agrawal receive his B.Tech and M.E. Degree in Computer Science & Engineering and Ph.D. Degree in Computer Science & Engineering from IIT, Delhi and presently working as professor and head in the department of MCA in Krishna Institute of Engineering & Technology, Ghaziabad (U.P.), India. His Research interest includes Wireless Sensor Networks and Distributed Database Security. He has published more than 30 research papers in National and International journals and presented number of research papers in National/ International conferences.