A UNIFIED CONCURRENCY CONTROL ALGORITHM FOR DISTRIBUTED DATABASE SYSTEMS *

C. P. Wang and Victor O. K. Li
Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089-0272

ABSTRACT

We present a unified concurrency control algorithm for distributed database systems in which each transaction may choose its own concurrency control protocol. Specifically, we integrate Two-Phase Locking, Timestamp Ordering, and Precedence Agreement into one unified concurrency control scheme. We show the correctness of the new scheme and study the problem of selecting the best protocol for each transaction in order to optimize system performance.

1 INTRODUCTION

Many concurrency control algorithms have been proposed for distributed database systems[3]. Most of them fall into two categories, namely, Two-Phase Locking (2PL)[8,19,22] and Timestamp Ordering (T/O)[2,4,5,9,17]. 2PL, which was first designed for centralized systems and later applied to distributed ones, has the merit of simplicity but suffers from the distributed deadlock problem [1,6,11]. On the other hand, T/O avoids deadlocks by enforcing a special execution order (timestamp order) among transactions, which may cause transaction restarts and degradation of system performance. In [24], the Precedence Agreement (PA) algorithm is proposed. This concurrency control algorithm is free from deadlocks and restarts. However, communication cost increases as the system load increases.

The best concurrency control algorithm for a particular application depends on the system parameters [10,14,16,20]. For example, in an environment where each transaction only accesses one data item through a write operation, 2PL outperforms T/O since no deadlocks may occur. On the other hand, when system load is heavy and transaction size (in terms of the number of data items accessed) is small (but bigger than one), T/O is superior to 2PL. Many parameters may affect the performance of a concurrency control algorithm. Some of them are (1) transaction arrival rate, (2) the number of read operations versus the number of write operations, (3) transmission delay, (4) the number of data items requested by each transaction, (5) the cost of restarts, (6) deadlock detection time and cost, etc.

To implement a distributed database system, the system designers may estimate the system parameters beforehand and pick the best concurrency control algorithm for that system. From then on, every transaction in the database must follow that algorithm. We call this type of concurrency control *static concurrency control*. Most existing concurrency control methods are static. The major drawback is inflexibility. The originally chosen algorithm may not always be the best as the system parameters change. In addition, the term '*the best concurrency control algorithm*' is really transaction dependent. Static concurrency control can only capture the average behavior but fails to reflect the individual differences among transactions.

The alternative is *dynamic concurrency control*. In this case, depending on the system states and the transaction types, different concurrency control methods are used for different transactions in an attempt to improve performance. There are several design problems associated with dynamic concurrency control. These include: (1) What are the candidate concurrency control algorithms? (2) How does one integrate these algorithms together and still preserve their correctness? (3) What are the system parameters relevant to choosing the best concurrency control algorithm and how does one determine them? (4) Given the relevant system parameters, how does one choose the best concurrency control method?

This paper presents the design of a dynamic concurrency control system which integrates 2PL, T/O, and PA. Our design consists of two steps:

1. Algorithm Integration: The goal is to design an integrated system such that each transaction can choose one of the above three algorithms. Our approach includes defining a unified model for distributed concurrency control algorithms, called the Precedence-Assignment Model (PAM)[24]. Based on this model, the problem of unifying 2PL, T/O and PA is reduced to the distributed computation of two functions. Then we solve the distributed computation problems and prove the correctness of our unified algorithm.

2. Algorithm Selection: A cost function is defined to measure the loss of system throughput due to the requests issued by a transaction under its concurrency control method. The concurrency control algorithm will be chosen to minimize this cost function. We will discuss the evaluation of the cost function.

This paper is organized as follows. We describe the distributed database model and the precedence assignment model in sections 2 and 3, respectively. In section 4, the integration

*This work was supported in part by the United States Department of Defense Joint Services Electronics Program under Contract No. F49620-85-C-0071, and by the National Science Foundation under Grant No. DCI-8519101

of concurrency control algorithms is presented. The selection of the best concurrency control algorithm is discussed in section 5. In section 6, we conclude with a list of future research tasks.

2 SYSTEM MODEL

A database consists of a finite set of logical data items D = { D_1, D_2, ...}. A transaction is a sequence of logical read/write operations requested by the user. We assume that a legal transaction consists of three phases, namely, the read phase, the local computing phase and the write phase. The read phase must be executed first and consists of operations which read from the database to the user's local memory. The local computing phase, consisting of all local computations, is then executed. Finally, during the write phase, data is copied from the user's memory back to the database. For database correctness, a transaction must either be completely executed or not executed at all.

Each logical data item D_i can be stored redundantly at different computer sites in the network. The physical copies of D_i are denoted $D_{i,j}$. Operations which access physical data item $D_{i,j}$ are physical reads $r(D_{i,j})$, or physical writes $w(D_{i,j})$. To execute a transaction, the system first translates all the logical operations into their corresponding physical operations. The physical operations are sent to the sites where the physical data items are located and implemented there.

Let T = { t_1, t_2, ...} be a set of transactions. An execution of T is the implementation of all operations associated with all the transactions in T on the database. An execution E is serial if no two transactions are executed concurrently at any time. An execution is said to be serializable [12] if the effect of the execution is equivalent to some serial execution.

Two logical (physical) operations conflict if they access the same logical (physical) data item and one of them is a write operation. We model the execution of transactions by a set of logs. There is one log associated with each physical data item. The log indicates the order in which physical operations are implemented on that data item.

The properties of serializable execution have been extensively studied [3,12]. An important result is :

Theorem 1: [12,13,18] Let T be a set of transactions, An execution E of T is serializable if there exists a total order on T such that if O_i and O_j are conflicting operations from distinct transactions t_i and t_j, O_i is implemented before O_j in any log L_1, L_2, ... L_m if and only if $t_i < t_j$ in the total order.

An execution is conflict serializable if it satisfies the condition stated in Theorem 1. The total order defined in Theorem 1 is called the serialization order. All existing concurrency control algorithms use conflict serializability as their correctness criterion.

3 THE PRECEDENCE-ASSIGNMENT MODEL

PAM was developed to model concurrency control algorithms in distributed databases. It is shown that 2PL, T/O and many other existing concurrency control algorithms may be modeled by PAM. In this section, we will describe PAM.

3.1 THE CONCURRENCY CONTROL SUBSYSTEM

The concurrency control subsystem is modeled as follows:

1. Request Issuer (RI): Located at each user site is a request issuer. RI takes user transactions as its input and sends requests to data sites.
2. Data Queue (Queue): There is a data queue for each physical data item. Each entry in the queue is a user request. The request at the head of the queue has the right to access the data.
3. Data Queue Manager (QM): There is a data queue manager for each data queue. The data queue manager communicates with the RI's and enforces the order of execution of requests on the data queue.

Each transaction t_i is sent to one of the RI's, say r_i. Let o_1, o_2, ... , o_n be the read or write operations of t_i. For each o_k, r_i sends a read or write request to the corresponding data queue manager accessed by o_k. o_k cannot be implemented until its request is granted by that data queue manager.

3.2 PRECEDENCE ASSIGNMENT AND ENFORCEMENT FUNCTIONS

The concurrency control subsystem must serve two major functions:

1. Precedence Assignment: Assigns a precedence to each request accessing the same data. That is, for each data D_j, there is a non-empty set SP_j together with a total ordering $<_j$ on SP_j. (SP_j, $<_j$) is called the precedence space for D_j. Let O_j be the set of all operations accessing D_j. There is a one-to-one function AS_j: $O_j \rightarrow SP_j$ computed by the concurrency control subsystem which assigns a precedence (an element of SP_j) to each operation accessing D_j.
2. Precedence Enforcement: Controls the implementation of operations on each data item to satisfy the following two conditions:

   E1: Suppose o_1 and o_2 are conflicting operations accessing D_j. If $AS_j(o_1) <_j AS_j(o_2)$, then o_1 is implemented before o_2; else o_2 is implemented before o_1.

   E2: There exists a serialization order on the set of transactions such that the precedence order of two conflicting operations follows the corresponding transactions' serialization order.

3.3 MODELING 2PL AND T/O

In this section, we illustrate how to use PAM to model 2PL and T/O.

2PL

Static 2PL is considered in this paper. For 2PL, the requests are handled in a first-come-first-served (FCFS) manner at each data queue. The precedences are assigned to operations according to their order of arrival. (E1) is satisfied by the assignment function itself as long as the data queue is FCFS and conflicting operations are not allowed to be implemented concurrently. (E2) is satisfied by the following locking protocol:

1. A request is granted a lock if all conflicting operations with lower precedences have been implemented (locks have been released).

2. The operations in a transaction can be implemented only if all the requests for that transaction are granted.

Some transactions may be blocked by the locking protocol forever, i.e., there is a deadlock. Ignoring deadlocks, the locking protocol together with (E1) implies (E2).

To sum up, 2PL can be specified by

1. Precedence Assignment: the arrival order at the data queue.
2. Precedence Enforcement: (E1) is implied by the assignment function; (E2) is satisfied by the locking protocol.

T/O

There are several versions of the T/O algorithm. Here we consider the Basic T/O algorithm [3] only. In Basic T/O, all the data items have the same precedence space, which is the set of timestamps. Operations from the same transaction are assigned the same precedence (or timestamp). This assignment function satisfies (E2) automatically as the serialization order can be defined to be the timestamp order of the transactions. (E1) is enforced by rejecting requests which arrive out of timestamp order.

In summary, T/O can be specified as:

1. Precedence Assignment: Assign the transaction's timestamp to each operation.
2. Precedence Enforcement: (E2) is automatically satisfied, (E1) is enforced by transaction restarts.

3.4 THE PRECEDENCE AGREEMENT ALGORITHM BASED ON TIMESTAMPS

PA decides the precedence for each operation through negotiation with the participating RI's and QM's, and is free from deadlocks and restarts. The following is a description of one version of PA based on timestamps.

Each transaction t_i is assigned a timestamp tuple (TS_i, INT_i), where TS_i is the timestamp for t_i and INT_i is the backoff interval associated with t_i. This algorithm works just like T/O except when a request arrives at the data queue and finds some conflicting operation with a later timestamp has already been granted. Instead of rejecting the newly arrived request, the data queue for data item j will backoff the new request's timestamp, i.e., find the minimum TS'_{ij} which is acceptable by the T/O protocol, with $TS'_{ij} = TS_i + k \cdot INT_i$, $k \in N$ (the natural numbers). Then TS'_{ij} is sent back to the request issuer r_i. r_i will either receive grants from all the data queues accessed by t_i (corresponding to the no rejection case in T/O) or receive some backoff timestamps, TS'_{ij}'s. In the former case, the transaction can be executed. In the latter case, the transaction will backoff its timestamp to $\max_j TS'_{ij}$. This new timestamp will be sent to all the data queues accessed by t_i to update each request's timestamp. Now t_i can wait for the grants and begin execution.

We state the algorithm formally:

1. For each t_i, r_i performs the following:
   (a) Generate $Q_i=(TS_i, INT_i)$.
   (b) Send requests together with Q_i to each data site D_j accessed by the current transaction.
   (c) Wait until either a grant or TS'_{ij} is received from each QM(j).
   (d) If grants are received from each QM(j), jump to (g).
   (e) Let TS'_i be $\max_j TS'_{ij}$. Send TS'_i to each QM(j).
   (f) Wait until lock grants arrive.
   (g) Perform local computing (the data read are attached to the corresponding lock grant).
   (h) Send lock release to each QM(j).
2. The data queue manager QM(j) for D_j performs the following:
   (a) Keep track of two variables, R_TS(j) and W_TS(j), which are the biggest timestamp of granted read re-
   (b) Receive request q_i and timestamp tuple Q_i from r_i.
   (c) Depending on whether q_i is a read or write request do:
      read: If $TS_i > W\_TS_j$, let TS_i be q_i's timestamp and mark q_i 'accepted'; insert q_i in QUEUE(j). Otherwise, calculate $TS'_{ij} = TS_i + k \cdot INT_i$, $k \in N$, such that $TS'_{ij} > W\_TS(j)$; let TS'_{ij} be q_i's timestamp and mark q_i 'blocked'; insert q_i in QUEUE(j); send TS'_{ij} to r_i.
      write: If $TS_i > W\_TS_j$ and $TS_i > R\_TS_j$, let TS_i be q_i's timestamp and mark q_i 'accepted'; insert q_i in QUEUE(j). Otherwise, calculate $TS'_{ij} = TS_i + k \cdot INT_i$, $k \in N$, such that $TS'_{ij} > \max\{W\_TS(j), R\_TS(j)\}$; let TS'_{ij} be q_i's timestamp and mark q_i 'blocked'; insert q_i in QUEUE(j); send TS'_{ij} to r_i.
   (d) When TS'_i is received from r_i, update the timestamp of q_i and mark q_i 'accepted'; re-insert q_i into the proper position in QUEUE(j).
   (e) Queue(j) works as follows:
      i. All entries are sorted in increasing timestamp order.
      ii. Define HD(j) to be the request in QUEUE(j) such that all requests in QUEUE(j) with smaller timestamps have already been granted.
         A. If HD(j) is marked 'blocked', wait.
         B. If HD(j) is a read request marked 'accepted', and if all previously granted write requests have been released, then set R_TS(j) to the timestamp of HD(j). Send grant to the transaction issuing HD(j).
         C. If HD(j) is a write request marked 'accepted', and if all previously granted requests have been released, then set W_TS(j) to the timestamp of HD(j). Send grant to the transaction issuing HD(j).
         D. Otherwise, wait.

PA can be modeled by PAM as follows:

1. Precedence Assignment: Through the PA protocol, a single precedence is assigned to each request of a transaction.
2. Precedence Enforcement: The precedence assignment function guarantees both (E1) and (E2).

The correctness of PA will be shown in the next section.

4 THE UNIFIED CONCURRENCY CONTROL SYSTEM

We have already demonstrated the precedence assignment and enforcement functions for 2PL, T/O, and PA. To unify these algorithms, we have to unify their assignment functions and then their precedence enforcement functions.

4.1 UNIFIED PRECEDENCE ASSIGNMENT FUNCTION

The assignment function for 2PL resides at each data site, and assigns the next available precedence to an incoming request. The assignment function for T/O is at the request issuers, and assigns the timestamp of each transaction to all of its requests. The assignment function for PA is more complicated: The initial timestamp is decided at the request issuer; then each data site decides a feasible timestamp; finally, collecting all the feasible timestamps, the request issuer decides the timestamp for all requests of the current transaction. To unify these three algorithms, we first define the unified precedence space (UPS) at each data queue. UPS is the timestamp space and the precedence is assigned as follows: The precedence for T/O and PA requests are their respective timestamps. The precedence for a 2PL request q accessing data item j is the biggest timestamp which has ever appeared in data queue j before q's arrival. This ensures the new request is inserted at the tail of the queue and all 2PL requests follow the FCFS rule. The precedence order is defined as follows:

1. Compare the value of the timestamps first.
2. If it is a tie, compare the site id's of the transactions. A 2PL controlled transaction is regarded as having the biggest site id.
3. If still tied, then either both requests are 2PL or both are not. If both are 2PL transactions, compare their arrival order at the data queue; otherwise, compare their transaction id's.

4.2 UNIFIED PRECEDENCE ENFORCEMENT FUNCTION

In the unified system, many types of requests may appear in each data queue. The enforcement function must enforce both (E1) and (E2) under all possible situations. Some requests which may be granted by individual enforcement functions cannot be granted by the unified enforcement function. For example, in pure T/O, a granted read request never prevents write requests with bigger timestamps from accessing the data. But the following example shows that if T/O is used together with 2PL, then sometimes read requests must lock the data to preserve (E2).

Example: Consider three data items x, y, z, and three transactions t_1, t_2, t_3. t_1 and t_2 are controlled by T/O and t_3 is controlled by 2PL. The operations for the transactions are:

$t_1 : r_1(x), w_1(y)$.
$t_2 : r_2(y), w_2(z)$.
$t_3 : r_3(z), w_3(x)$.

The precedence order of these requests in the data queue may be:

Queue(x): $r_1 < w_3$.
Queue(y): $r_2 < w_1$.
Queue(z): $r_3 < w_2$.

Both r_2 and w_1 are T/O type requests. Therefore, w_1 will be implemented before t_2 is executed. But then all three transactions will be executed and the execution is not serializable.

One solution to the unified enforcement function is to use locking for all requests, i.e., before a transaction leaves the system, lock releases must be sent to each data queue accessed by it and a request cannot be granted until all previously granted conflicting requests have been released. This method, while sacrificing the degree of concurrency for T/O transactions, guarantees (E2).

We now introduce a more sophisticated approach, called **semi-locks**. This type of locks preserve (E2) without reducing the degree of concurrency for T/O. Intuitively, a data is semi-locked if it is considered unlocked by the T/O protocol but must be treated as locked by 2PL and PA. The following semi-lock protocol decides, before a data item is unlocked, when the lock can become a semi-lock, i.e., ready for T/O type requests:

1. There are four possible locks: read (RL), write (WL), semi-read (SRL), and semi-write (SWL).
2. Two locks conflict if they lock the same data item and at least one is a WL or SWL. A lock is pre-scheduled if at least one conflicting lock is granted earlier but not yet released. All other locks are called normal locks. The following rules describe when HD(j) may be granted and what kind of locks are granted:
   (a) If HD(j) is a read request issued by a 2PL or PA transaction, then it can be granted a RL if all previously granted WL's and SWL's on the data j have been released.
   (b) If HD(j) is a write request issued by a 2PL or PA transaction, then it can be granted a WL if all previously granted locks on data j have been released.
   (c) If HD(j) is a read request issued by a T/O transaction, then it can be granted a SRL if all previously granted WL's on data j have been released.
   (d) IF HD(j) is a write request issued by a T/O transaction, then it can be granted a WL if all previously granted RL's and WL's on data j have been released.
   (e) If HD(j) was once granted a pre-scheduled lock, which later became normal, then a normal lock grant will be issued for HD(j).
3. Before execution, 2PL and PA transactions must hold proper RL's or WL's. After execution, lock releases are sent to the locked data item.
4. Before execution, a T/O transaction must hold a SRL on each data item read by it, and a WL on each data item written by it. After execution, it sends lock releases to the locked data if it did not get any pre-scheduled locks. Otherwise, it transforms all its locks into semi-locks (RL $\rightarrow$ SRL, WL $\rightarrow$ SWL). At this point, the transaction is considered executed. But the request issuer will continue to collect lock grants for this transaction until there is one normal lock grant received from each data item accessed by this transaction. Then lock releases are sent to the data locked by that transaction.

The semi-lock protocol preserves the enforcement functions for 2PL and PA in the following sense: If the system runs 2PL (PA) transactions only, then the unified enforcement function

works in the same way as the enforcement function for 2PL (PA). On the other hand, when the system runs T/O only, the unified enforcement function results in lock release messages which are not generated by the T/O enforcement function. But these messages are irrelevant in the sense that no T/O transactions need these messages for synchronization.

4.3 CORRECTNESS

We will show the correctness of the unified concurrency control system. By doing so, the correctness of PA is also proved since it can be regarded as a special instance of the unified scheme where all transactions use PA as their concurrency control algorithm.

First define when an operation is implemented: Operations of 2PL and PA type transactions are implemented when the corresponding locks are released; a T/O operation is implemented when the operation changes a lock to a semi-lock on the data or, if the lock is released.

Let $<_S$ be a binary relation on the set of executed transactions defined as follows: Let t_i and t_j be two executed transactions, $t_i <_S t_j$ iff (1) $t_i = t_j$ or (2) there is a pair of conflicting operations o_i and o_j issued by t_i and t_j respectively, and o_i is implemented before o_j. An execution is conflict serializable iff the conflict graph induced by $<_S$ is acyclic[12,18], i.e., $<_S$ is a partial order.

Theorem 2: Let T be a set of transactions executed under the unified concurrency control algorithm. $<_S$ must be a partial order on T. In other words, the execution is conflict serializable.

Proof: Suppose $<_S$ is not a partial order on T. Then there is a cycle $t_1 <_S t_2 <_S \ldots <_S t_n <_S t_1$. In addition, $t_1, \ldots, t_n$ cannot all be T/O type transactions since the semi-lock protocol implements T/O type operations according to their timestamp order. Without loss of generality, assume t_n is a 2PL or PA type transaction. The semi-lock protocol requires t_n to get all normal locks on all data items accessed by it. This implies t_{n-1} has to receive all normal locks and to release all of these locks, either semi or normal, before t_n can release any locks. By induction, t_1 has to release all the locks before t_n releases any lock. But t_n is of 2PL or PA type and $t_n <_S t_1$ implies t_n releases some locks before t_1. This is a contradiction.

Q. E. D.

Theorem 2 shows that if the unified system allows a set of transactions to be executed, the execution must be conflict serializable. Now we want to consider the situation when the system is blocked. It will be shown that if the system is blocked, there must be at least one 2PL type transaction in the wait-for cycle, that is, T/O and PA never block the system by themselves.

Definition 1: At time u, if there are requests not implemented yet, $p_{small}(u)$ is the smallest precedence of such requests; else $p_{small}(u)$ is undefined.

Let P be the set of all possible precedences, i.e., extended timestamp space, and let T be the set of all transactions. Let P_T be the set of precedences assigned to requests associated with the transactions in T. Our proof starts with a special case of P_T and will be generalized.

Lemma 1: If P_T is isomorphic to the order of N, then the system is blocked if and only if $\exists$ u such that $\forall$ x >u, $p_{small}(x)$ = C where C is a constant.

Proof: ($\Leftarrow$) In any data queue, different requests have different precedences. Therefore, the number of requests whose precedence is C is finite. At least one of these requests, say rq, cannot be implemented in finite time, else $p_{small}()$ will be changed to a value different from C. Therefore, the transaction issuing rq is blocked.

($\Rightarrow$) If no such C exists, there are two cases: (a) $\exists$ u such that $\forall$ x >u, $p_{small}(x)$ is undefined or (b) $\forall$ u, $\exists$ x >u such that $p_{small}(u) \neq p_{small}(x)$.

Case (a) implies the system becomes empty after finite time. Therefore, no transaction is blocked. Case (b) implies that $p_{small}()$ is changed for infinitely many times. Since the number of requests with the same precedence are finite, $p_{small}()$ cannot be changed to the same value for infinitely many times. This implies the range of $p_{small}()$ is infinite. Therefore, if at time u, a request rq with precedence p is not implemented yet, there is time x >u and $p_{small}(x) > p$. This implies at time x, (1) rq has been implemented, or (2) rq is of T/O type and has been rejected, or (3) rq is of PA type and it has been assigned a new backoff precedence. It is clear that if case (1) or (2) occurs, the transaction issuing rq is either executed or restarted. If (3) occurs, rq's precedence is backed off to a bigger one. But eventually $p_{small}()$ will be greater than this precedence because PA can backoff the precedence of rq at most once. Then rq will be implemented and the transaction issuing rq will be executed. Thus we conclude that all transactions will be either executed or restarted after finite time, i.e., no blocking occurs.

Q. E. D.

Theorem 3: Assume the precedence order on P_T is isomorphic to the order of N, then the system is blocked only if the blocked request with the smallest precedence is from a 2PL type transaction.

Proof: From Lemma 1, the system is blocked if and only if $\exists$ u, such that $\forall$ x > u, $p_{small}(x)$ = C. There are three cases:

1. $p_{small}()$ is owned by a request issued by a T/O type transaction t: $p_{small}()$ is the timestamp of t. If t is rejected, then $p_{small}()$ will be updated to another value. If t is not rejected, all of its requests have precedence $p_{small}()$ which is the smallest precedences not yet implemented. In short, t can be executed and p_{small} is updated to another value. We reach contradictions in both cases, thus $p_{small}()$ cannot be owned by T/O type requests.

2. $p_{small}()$ is owned by a request issued by a PA type transaction t. There are two cases:

   (a) $p_{small}()$ is the original timestamp of t: t can receive either lock grants or back-off precedences from all the data accessed by it. Then either t can be executed or t will back off its precedence to the biggest of the back-off precedences received.

   (b) $p_{small}()$ is the back-off precedence: t can receive back-off precedences from all the data accessed by it. Similarly, t can be executed or t will back off its precedence to the biggest of the back-off precedences received.

   In either case, $p_{small}()$ will be changed. Thus $p_{small}()$ cannot be owned by a request issued by PA type transactions.

3. $p_{small}()$ is owned by a request issued by a 2PL type transaction t.

We conclude that case (3) must be true.

Q. E. D.

Corollary 1: PA is correct and free from deadlocks and rejections.

Proof: Theorem 2 shows PA will output a conflict serializable execution. Theorem 3 shows every transaction will be executed eventually. In addition, from the definition of PA, no transactions will be restarted.

Q. E. D.

Corollary 2: Any deadlock cycle must contain at least one 2PL type transaction.

Proof: This is also a consequence of Theorem 3. The rationale is as follows. Our unified precedence assignment functions have the property that the removal of 2PL transactions will not affect the precedences assigned to T/O and PA type requests. Thus the removal of 2PL transactions will not destroy any deadlock cycles which contain T/O and PA transactions only. But the removal of 2PL transactions does guarantee freedom from blocked transactions and hence deadlock freedom. Thus we conclude there must be at least one 2PL transaction in each deadlock cycle.

Q. E. D.

The assumption that P_T is isomorphic to N can be relaxed if, during any finite period of time, only finitely many transactions may arrive. Ignoring 2PL type transactions, the timestamp space is isomorphic to N since the values of timestamps are from N and only finitely many T/O or PA type transactions can have the same timestamp value. The key problem which prevents P_T from being isomorphic to N is that there are potentially infinitely many 2PL requests assigned the same timestamp. But this problem can be fixed. Conceptually, we can modify the precedence assignment function of 2PL to prevent infinitely many requests assigned the same timestamp and still preserve the precedence order of requests in the data queue. This modification will make P_T isomorphic to N without changing any implementation order of operations.

The 2PL precedence assignment function AS_j for data j is modified as follows: If there are infinitely many T/O or PA type requests entering Queue(j), AS_j does not need to be changed since the arrival of these requests will stop AS_j from assigning the same precedence. If, on the other hand, there are only finitely many T/O or PA type requests entering Queue(j), AS_j remains unchanged until the T/O (or PA) request o_{max} with the largest timestamp TS_{max} arrives. Then AS_j changes and assigns TS_{max}+m to the mth arriving 2PL requests after the arrival of o_{max}. The new AS_j will only assign a timestamp, if ever, finitely many times.

5 DYNAMIC CONCURRENCY CONTROL

In the previous section, the unified concurrency control system is presented. We now discuss how to select the most profitable concurrency control algorithm for each transaction.

Before developing the selection method, we define our performance measure to be the average transaction system time S and study how the transaction arrival rate, λ, and the number of data items accessed by each transaction s_t, affect S. Our simulation results [23] show that 2PL performs well when λ is low. When λ is high, although the number of transactions directly involved in deadlocks does not increase very much, S goes up dramatically since more transactions are blocked by deadlocked transactions. For T/O, S grows steadily as λ increases. It outperforms 2PL when λ is high. However, as is also shown in [10], T/O becomes worse than 2PL and PA as s_t increases. Apparently, this is due to the significant increase of restart probability. PA is a compromise between 2PL and T/O. It performs like 2PL when λ is low and like T/O while λ is high. When λ is moderate, it outperforms both 2PL and T/O.

5.1 THE SYSTEM THROUGHPUT LOSS FUNCTION

One possible method for selecting concurrency control algorithms is as follows: Each request issuer keeps a table of average system time versus transaction types and concurrency control method chosen. For each newly arrived transaction, the request issuer picks the concurrency control algorithm which minimizes the average system time. However, we believe this method does not perform well for the following reasons:

1. Minimizing the system time for individual transactions is not equivalent to optimizing S.
2. This method is biased towards 2PL since once deadlocks occur, not only is the system time of deadlocked transactions prolonged, but also the system time of all blocked transactions. Thus 2PL transactions tend to shorten their own system time by degrading other transactions' performance.

The method we have adopted, instead of considering average system time, is to estimate the system throughput loss, STL. Our method picks the concurrency control algorithm for each transaction to minimize STL. Before discussing STL any further, let us introduce the following notations and definitions:

1. The current transaction t has $m(t)$ read requests, $r_1, r_2, \cdots, r_{m(t)}$, and $n(t)$ write requests, $q_1, q_2, \cdots, q_{n(t)}$.
2. Let $D(rq)$ denote the data item accessed by request rq.
3. The write throughput of a data queue, which is the average number of write locks granted per unit time, is denoted $\lambda_w(j)$. Similarly, the read throughput is denoted $\lambda_r(j)$. Let λ_w (λ_r) denotes the throughput averaged over all $\lambda_w(j)$'s ($\lambda_r(j)$'s).
4. λ_A, the system throughput, is the summation of all $\lambda_r(j)$'s and $\lambda_w(j)$'s.
5. The fraction of read requests among all requests is Q_r.
6. K is the average number of requests per transaction.

We also make the following assumptions and approximations:

1. In computing STL, we assume a transaction gets all its locks at the same time, and all its locks are released at the same time.
2. The output process at each data queue (the process of lock releases) is Poisson.

Now we consider $STL(t)$, the STL due to t, under our assumptions. Assume a request rq locks a data for U time units, what is the throughput loss on that data? This depends on what kind of lock rq has. If it is a read lock, then during this period of time, no write locks can be granted. Potentially, the throughput

loss is $\lambda_w D(rq) \cdot U$. On the other hand, if it is a write lock, the loss is $(\lambda_w D(rq) + \lambda_r D(rq)) \cdot U$. But there are other losses due to rq: Suppose that after rq gets its lock, another request rq' gets a lock on data k, but the transaction issuing rq' also issues a request blocked by rq, or blocked by some request blocked by rq, then rq' is blocked until rq releases the locks. Therefore, $STL(t)$ is difficult to calculate or measure in general. Thus we use the following approximation.

Suppose transaction t starts to hold all $(m(t) + n(t))$ locks at time 0, the throughput loss, λ_t, is given by

$$\lambda_t = \sum_{i=1}^{m(t)} \lambda_w D(r_i) + \sum_{i=1}^{n(t)} \lambda_w D(q_i) + \lambda_r D(q_i)$$

Now, the rate at which requests get locks is $\lambda_A - \lambda_t$. A request to obtain a lock will be blocked if the transaction issuing this request also issues a blocked request. A random transaction, on the average, issues K requests. Each request is blocked with probability $\frac{\lambda_t}{\lambda_A}$. Assuming that the probability of blocking at different data site are independent, the rate at which requests obtaining locks is blocked is

$$(\lambda_A - \lambda_t) \cdot (1 - (1 - \frac{\lambda_t}{\lambda_A})^{K-1})$$

If a request gets a lock and blocks the data, the new throughput loss depends on whether it gets a read lock or a write lock. Again, we approximate it by taking the average, that is, the new throughput loss is $\lambda_t + \lambda_w + (1 - Q_r) \cdot \lambda_r$.

Now we are ready to solve $STL(t)$ recursively by defining the following functions. Let $STL'(\lambda_{loss}, u)$ represents the STL during period of u time units where the initial throughput loss is λ_{loss}. There are two cases:

1. During this period, no other new lock grants block any data queue; then the STL is $\lambda_{loss} \cdot u$.
2. At time x, a new lock grant blocks a data queue; then the STL is $\lambda_{loss} \cdot x + STL'(\lambda_{loss} + \lambda_w + (1 - Q_r) \cdot \lambda_r, u - x)$.

Thus STL' can be defined recursively as follows:

If $\lambda_{loss} \geq \lambda_A$,

$$STL'(\lambda_{loss}, u) = \lambda_A \cdot u;$$

else

$$STL'(\lambda_{loss}, u) = (1 - exp(-\lambda_{block} \cdot u)) \cdot \lambda_{loss} \cdot u + \int_0^u \lambda_{block} \cdot exp(-\lambda_{block} \cdot x) \cdot (\lambda_{loss} \cdot x + STL'(\lambda_{loss} + \lambda_{new}, u - x))dx$$

where $\lambda_{block} = (\lambda_A - \lambda_{loss}) \cdot (1 - (1 - \frac{\lambda_{loss}}{\lambda_A})^{K-1})$ and $\lambda_{new} = \lambda_{loss} + \lambda_w + (1 - Q_r) \cdot \lambda_r$.

Note that STL' can be evaluated efficiently through Dynamic Programming techniques [7].

5.2 SELECTION BASED ON *STL*

Now we describe the selection method based on STL. Our selection is based on the following parameters which can either be collected periodically or estimated through analytical methods[14,15,21,25]:

1. For 2PL transactions:
   (a) U_{2PL}: average lock time if a request is not aborted.
   (b) U'_{2PL}: average lock time of an aborted request.
   (c) P_A: probability of abortion of a transaction due to deadlocks.
2. For T/O transactions:
   (a) $U_{T/O}$: average lock time if a request is not aborted.
   (b) $U'_{T/O}$: average lock time if a request is aborted.
   (c) P_r: the probability of rejection of a read request.
   (d) P'_r: the probability of rejection of a write request.
3. For PA transactions:
   (a) U_{PA}: average lock time if a request is not backed off later.
   (b) U'_{PA}: average lock time if a request is backed off later.
   (c) P_B: the probability a read request is backed off.
   (d) P'_B: the probability a write request is backed off.

Let t be a transaction as described in the last section, we will estimate STL due to t for 2PL, T/O and PA, by the STL' function just defined.

2PL

The STL due to t under 2PL, denoted $STL_{2PL}(t)$ can be solved by the following equation:

$$STL_{2PL}(t) = (1 - P_A) \cdot STL_{2PL}(t|\ t \textit{ is not aborted}) + P_A \cdot STL_{2PL}(t|\ t \textit{ is aborted}),$$

or

$$STL_{2PL}(t) = (1 - P_A) \cdot STL'(\lambda_t, U_{2PL}) + P_A \cdot (STL_{2PL}(t) + STL'(\lambda_t, U'_{2PL}))$$

T/O

Similarly, $STL_{T/O}(t)$ can be solved by the follow equation:

$$STL_{T/O}(t) = (1 - P_r)^{m(t)} \cdot (1 - P'_r)^{n(t)} \cdot STL'(\lambda_t, U_{T/O}) + (1 - (1 - P_r)^{m(t)} \cdot (1 - P'_r)^{n(t)}) \cdot (STL'(\lambda^*_t, U'_{T/O}) + STL_{T/O}(t))$$

Here λ^*_t is the average system throughput loss due to blocking by t's requests given that at least one of t's requests is rejected. λ^*_t can be solved by the following equation:

$$(1 - P_r) \sum_{i=1}^{m(t)} \lambda_w D(r_i) + (1 - P'_r) \sum_{i=1}^{n(t)} \lambda_w D(q_i) + \lambda_r D(q_i) = (1 - (1 - P_r)^{m(t)}(1 - P'_r)^{n(t)})\lambda^*_t + (1 - P_r)^{m(t)}(1 - P'_r)^{n(t)}\lambda_t$$

In the above equation, The left hand side is the sum of the expected throughput loss due to each request of t. The right hand side calculates the average throughput loss by conditioning on whether t is rejected or not. Thus λ^*_t is given by

$$\lambda^*_t = \frac{1}{1 - (1 - P_r)^{m(t)} \cdot (1 - P'_r)^{n(t)}} \cdot ((1 - P_r) \cdot \sum_{i=1}^{m(t)} \lambda_w D(r_i) + (1 - P'_r) \cdot \sum_{i=1}^{n(t)} \lambda_w D(q_i) + \lambda_r D(q_i) - \lambda_t \cdot (1 - P_r)^{m(t)} \cdot (1 - P'_r)^{n(t)})$$

PA

Finally, $STL_{PA}(t)$ can be derived as follows

$$STL_{PA}(t) = (1-P_B)^{m(t)}(1-P'_B)^{n(t)}STL'(\lambda_t, U_{PA}) + (1-(1-P_B)^{m(t)}(1-P'_B)^{n(t)})(STL'(\lambda_t^+, U'_{PA}) + STL'(\lambda_t, U_{PA}))$$

where λ_t^+ is

$$\lambda_t^+ = \frac{1}{1-(1-P_B)^{m(t)}\cdot(1-P'_B)^{n(t)}} \cdot ((1-P_B) \cdot \sum_{i=1}^{m(t)} \lambda_w D(r_i) + (1-P'_B)\cdot \sum_{i=1}^{n(t)} \lambda_w D(q_i) + \lambda_r D(q_i) - \lambda_t \cdot (1-P_B)^{m(t)} \cdot (1-P'_B)^{n(t)})$$

In our selection method, for a new transaction t, $STL_{2PL}(t)$, $STL_{T/O}(t)$, and $STL_{PA}(t)$ are calculated separately. The algorithm with the smallest STL value is chosen to be t's concurrency control algorithm. To speed up the performance of the selection processes, transactions may be categorized into different classes and the STL for each class may be calculated in advance.

6 CONCLUSIONS AND FUTURE RESEARCH

This paper describe a dynamic concurrency control method for distributed database systems. Our approach is to first define a unified model, PAM, and to show that 2PL, T/O, and PA fall in this model. Then, based on PAM, 2PL, T/O and PA are integrated to form a unified concurrency control system. Specifically, we unify the precedence space for these three algorithms and develop the semi-lock protocol, which serves to unify the precedence enforcement function.

To complete the design of the dynamic system, we also study the problem of algorithm selection. For this purpose, the STL criterion is proposed. We find efficient ways to estimate the STL for 2PL, T/O, and PA.

A number of problems remain to be investigated. Some of them are: (1) a detailed simulation of the proposed method, (2) integration of other concurrency control algorithms into our unified system, (3) development of more accurate criteria than STL, (4) allowing transactions to change their concurrency control methods, and (5) investigation of other applications of PAM.

References

[1] Baruch Awerbuch. Dynamic deadlock resolution protocols. In *Proc. 27th Symp. Foundations Computer Science (IEEE)*, pages 196–207, 1986.

[2] R. Bayer, K. Elhardt, J. Heigert, and A. Reiser. Dynamic timestamp allocation for transactions in database systems. In *Proc. 2nd Int. Symp. Distributed Databases*, 1982.

[3] P.A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[4] P.A. Bernstein, N. Goodman, J. B. Rothnie, and C. A. Papadimitriou. The correctness of concurrency mechanisms of SDD-1: a system for distributed database (the fully redundant case),. *IEEE Trans. on Software Engineering*, SE-4(3):154–168, May 1978.

[5] P.A. Bernstein, D. Shipman, and J. B. Rothnie. The correctness of concurrency mechanisms in a system for distributed databases. *ACM Trans. on Database Systems*, 5(1):52–68, March 1980.

[6] K. M. Chandy, L. M. Haas, and J. Misra. Distributed deadlock detection. *ACM Trans. on Computer Systems*, 1(2), 1983.

[7] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Rockville, MD: Computer Science Press, 1978.

[8] T. A. Joseph and K. P. Birman. Low cost management of replicated data in fault-tolerant distributed systems. *ACM Trans. on Computer Systems*, 1(4):54–70, February 1986.

[9] P. J. Leu and Bharat Bhargava. Multidimensional timestamp protocols for concurrency control. In *Proc. IEEE Data Engineering Conf.*, pages 482–489, February 1986.

[10] W. K. Lin and J. Nolte. Basic timestamp, multiple version timestamp, and two-phase locking. In *Proc. 9th VLDB*, pages 109–119, October 1983.

[11] R. Obermarck. Distributed deadlock detection algorithm. *ACM Trans. on Database Systems*, 2(7):187–208, June 1982.

[12] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of ACM*, 26(4):631–653, October 1979.

[13] C. H. Papadimitriou and P. A. Bernstein. Some computational problems related to database concurrency control. In *Proc. Conf. Theoretical Computer Science*, August 1977.

[14] K.C. Sevcik. Comparison of concurrency control methods using analytic models. In *Proc. IFIP 9th World Computer Congress*, pages 847–858, September 1983.

[15] S. C. Shyu and V. O. K. Li. Performance analysis of static locking in distributed database systems. submitted for publication.

[16] M. Singhal and A.K. Agrawala. Performance analysis of an algorithm for concurrency control in replicated database systems. In *Proc. SIGMETRICS Conf.*, pages 159–169, ACM, 1986.

[17] M. K. Sinha. Commutable transactions and the time-pad synchronization mechanism for distributed systems. *IEEE Trans. on Soft. Eng.*, SE-12(3):462–476, March 1986.

[18] R. E. Stearns, P. M. II Lewis, and D. J. Rosenkrantz. Concurrency controls for database systems. In *Proc. 17th Symp. Foundations Computer Science (IEEE)*, pages 19–32, 1976.

[19] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. on Software Engineering*, SE-5(3):188–192, May 1979.

[20] R. Sun and G. Thomas. Performance results on multiversion timestamp concurrency control with predeclared writesets. In *Proc. 6th Symp. Principles of Database Systems*, pages 177–184, ACM, March 1987.

[21] Y. C. Tay, R. Suri, and N. Goodman. A mean value performance model for locking in database: the no-waiting case. *JACM*, 32(3):618–651, July 1985.

[22] R.H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems*, 4(2):180–209, June 1979.

[23] C. P. Wang and Victor O. K. Li. The precedence-agreement concurrency control algorithm for distributed database systems. 1987. submitted for publication.

[24] C.P. Wang and Victor O.K. Li. The precedence-assignment model for distributed database concurrency control algorithms. In *Proc. 6th Symp. Principles of Database Systems*, pages 119–128, ACM, March 1987.

[25] C.P. Wang and Victor O.K. Li. Queueing analysis of the conservative timestamp-ordering concurrency control algorithm. In *Proc. International Computer Symposium*, pages 1450–1455, IEEE, December 1986.