1985

Multidimensional Timestamp Protocols for Concurrency Control

Pei-Jyun Leu

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Report Number:
85-521

MULTIDIMENSIONAL TIMESTAMP PROTOCOLS FOR CONCURRENCY CONTROL

Pei-Jyun Leu
Bharat Bhargava

CSD-TR-521
June 1985
Revised October 1986

MULTIDIMENSIONAL TIMESTAMP PROTOCOLS FOR CONCURRENCY CONTROL†

Pei-Jyun Leu -- Bharat Bhargava

CSD-TR-521
June 1985

revised Oct. 1986

ARPA: leu@purdue.edu
bb@purdue.edu

---

† This work was supported by the David Ross fellowship and partially supported by Sperry Corporation

Abstract–We propose multidimensional timestamp protocols for concurrency control in database systems where each transaction is assigned a timestamp vector containing multiple elements. The timestamp vectors for two transactions can be equal if timestamp elements are assigned the same values. The serializability order among the transactions is determined by a topological sort of the corresponding timestamp vectors. The timestamp in our protocols is assigned dynamically and is not just based on the starting/finishing time as in conservative and optimistic timestamp methods. The concurrency control can be enforced based on more precise dependency information derived dynamically from the operations of the transactions. Several classes of logs have been identified based on the degree of concurrency or the number of logs accepted by a concurrency controller. The class recognized by our protocols is within D-serializable (DSR), and is different from all previously known classes such as two phase locking (2PL), strictly serializable (SSR), timestamp ordering (TO), which have been defined in literature. The protocols have been analyzed to study the complexity of recognition of logs. We briefly discuss the implementation of the concurrency control algorithm for the new class, and give a timestamp vector processing mechanism. The extension of the protocols for nested transaction and distributed database models has also been included.

Index Terms–Database systems, transactions, logs, degree of concurrency, concurrency control algorithms, serializability, k-dimensional timestamp ordering, parallel processing.

I. INTRODUCTION

There are two basic concurrency control approaches for transaction processing in database systems. The first is based on locking. An example is the *two phase locking* protocol [9], which requires that no two transactions hold conflicting locks at the same time, and that no transaction obtain a lock after it has released one. The second approach is based on time stamps associated with transactions or data items. In the conventional *timestamp ordering*, each transaction has a unique timestamp which is the starting time of that transaction. All the conflicting operations are required to occur in the timestamp order [2, 4, 21]. There are several variations of timestamp ordering. Multiple versions of item values have been used to increase the degree of concurrency [3, 18, 19]. The conventional timestamp ordering tends to prematurely determine the serializability order, which may not fit in with the subsequent operation sequence, forcing some operations to abort. In contrast, the optimistic approach [13] waits till the end of the transaction to make a commit/abort decision. Thomas [20] describes the situation when some writes can be simply ignored instead of being aborted.

In this paper, we present the multidimensional timestamp protocols MT(k), which provide higher degree of concurrency than a single dimensional timestamp protocol. These protocols allow a transaction to have a timestamp vector of up to k elements. The maximum value of k is limited by twice the maximum number of operations in a single transaction. Each operation may set up a new dependency relationship between two transactions. We encode the relationship by making one vector less than the other. A single timestamp element is used to bear this information. Earlier assigned elements are more significant in the sense that subsequent dependency relationships cannot conflict with the previously encoded relationships. In such a way, we can decide to accept or abort an operation based on the dependency information derived from all the preceding operations. In other words, we use the approach of dynamic timestamp vector generation for each transaction and dynamic validation of conflicting transactions to increase the degree of concurrency.

We start by giving an example that distinguishes our approach. Section II contains the notation and definitions. In Section III, we present the protocol MT(k). In this protocol, two timestamp vectors are compared according to lexicographic order. We go through corresponding

elements in the two vectors from left to right until we reach two unequal or undefined elements. If both elements are defined, their order determines whether to accept or abort an incoming operation; otherwise we need to encode a newly discovered dependency relationship by making one element less than the other and accept the operation. Next, the timestamp vector processing mechanism is given. In Section IV, we present the composite protocol MT(k^+) that recognizes the union of the classes recognized by the protocols MT(1), MT(2),..., MT(k). The protocol MT(k^+) is guaranteed to allow higher concurrency as the vector size increases. In Section V, we present the protocol MT(k_1, k_2) suitable for concurrency control of nested transactions [15]. This protocol is a variation of MT(k). We partition the transactions or actions into groups which are formed either based on transaction types as in SDD-1 [4] or based on hierarchical levels as in nested transactions [15]. Serializability is assured at two levels. The technique of MT(k) is used at each level. We next design the decentralized concurrency control protocol DMT(k). In Section VI, a comparison between our approach and some related work [1] is discussed. Next, guidelines to choose an appropriate timestamp vector size are studied. Finally, we investigate two rollback approaches for timestamp protocols. Section VII gives the conclusions.

A. An Example to Illustrate Advantages of the Protocol MT(k):

Please see Section II for the definition of *log*. To simplify the example, we do not require a transaction to have each read followed by a write. We show that the degree of concurrency achieved by our approach can be higher than if only one-dimensional timestamp is used. Let a log L have the sequence of read/write operations of several transactions T_1, T_2, T_3 as follows.

Example 1: Suppose

$$L = W_1[x]W_1[y]R_3[x]R_2[y]\cdots$$

where $W_1[x]$ denotes a write operation of T_1 on item x etc.

The dependency digraph for the log is shown in Fig. 1(a). In the conventional timestamp ordering (as in the protocol P4 in [4]), we observe that the dependency between T_2 and T_3 will be assumed to be $T_3 \rightarrow T_2$ according to the starting timestamps, even though the two transactions do not have any real dependency since $R_3[x]$ and $R_2[y]$ do not conflict. The disadvantage of

creating this dependency at this stage in the log is that if we find that actually T_3 should depend on T_2 (i.e., $T_2 \rightarrow T_3$) due to some conflict in the future, we may have to abort and restart T_3. It is preferable to assign equal time stamps to T_2 and T_3 when only limited information is available in the log. If the timestamp of a transaction is a scalar (one-dimensional), we cannot have two transactions with the same time stamp. However, if the time stamp is a vector (multidimensional), then the current elements of the vectors for T_2 and T_3 can be the same as shown in Fig. 1(b). The timestamp vectors for the transactions T_1, T_2, T_3 at this stage of the log are:

T_1: <1, *>, * is an undefined element.

T_2: <2, *>

T_3: <2, *>

These vectors enforce precisely the partial order $T_1 \rightarrow T_2$, $T_1 \rightarrow T_3$ depicted in Fig. 1(b). Let us say the log at some later time is

$$L = W_1[x]W_1[y]R_3[x]R_2[y]W_3[y]$$

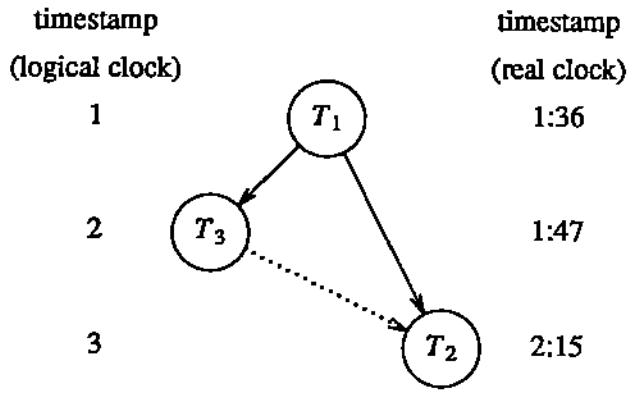
Now, we know that T_3 depends on T_2 because $R_2[y]$ precedes and conflicts with $W_3[y]$. We use the 2nd dimension of timestamps to "encode" the dependency as shown in Fig. 1(c). The resulting timestamp vectors are:

T_1: <1, *>, * is an undefined element.

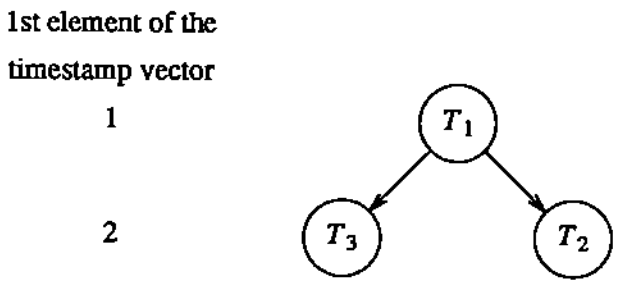
T_2: <2, 1>

T_3: <2, 2>

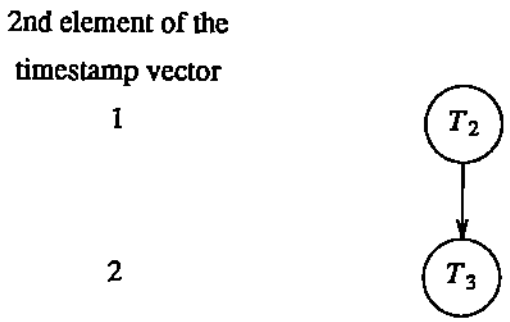
The serializability order for log L will be $T_1T_2T_3$ without a need to abort T_3. This example shows more logs may be acceptable to the concurrency controller. But we will further show in Section III-C that some serializable logs are acceptable to the single-valued timestamp protocol but not a multidimensional timestamp protocol.



(a)



(b)



(c)

Figure 1. Dependency digraph for example 1

As a second example, suppose

$$T_i: \langle 2, 1, * \rangle,$$

$$T_j: \langle 2, *, * \rangle.$$

To encode a dependency between T_i and T_j, we compare the two vectors. We find the 1st element in both transaction timestamp vectors does not tell the order between T_i and T_j. However, we can set the 2nd element in the vector of T_j to either 2 (if $T_i \rightarrow T_j$) or 0 (if $T_j \rightarrow T_i$) such that the order of the vectors is consistent with either dependency order.

II. NOTATION AND DEFINITIONS

In the *two-step transaction model*, a transaction T_i consists of a read operation R_i followed by a write operation W_i. We choose this basic model mainly for analysis purposes. Our protocols, however, are not restricted to this model. The sequence of operations produced by a set of transactions is captured by a *log*. A log is a quintuple $L = <D, T, \Sigma, S, \pi>$, where D is the *database item set*, T is the *transaction set*, Σ is the *atomic operation set*, S is the *access function* which gives the set of items accessed by an atomic operation, and π is the *permutation function* which gives the sequence number of an operation. For example, if a log L is the sequence $\alpha\beta\gamma\cdots$, then $\pi(\alpha) = 1$, $\pi(\beta) = 2$, ... An atomic operation is represented by $A_i[x]$, where i is a unique identification for a transaction, A is either R or W representing a read or a write operation, and x is a single database item. $S(R_i)$ is the *read set* of transaction T_i, and $S(W_i)$ the *write set*. This model has also been used in [16].

Notation.

MT(k)	k-dimensional timestamp protocol.
TO(k)	the set of logs recognized by MT(k).
$TS(i)$	the timestamp vector of transaction T_i, $TS(i) = <t_1, t_2,...,t_m,...,t_k>$, t_m is an integer or undefined.
$TS(i, m)$	the m-th element of $TS(i)$.
$T_{RT(x)}$	the most recent transaction that reads data item x where $RT(x)$ is a transaction index.
$T_{WT(x)}$	the most recent transaction that writes data item x where $WT(x)$ is a transaction index.
$TS(RT(x))$	the most recent read timestamp of a data item x.
$TS(WT(x))$	the most recent write timestamp of a data item x.

The timestamp table is shown in Fig. 2. The columns represent timestamp order based on operations, and the rows represent timestamp vectors. $TS(RT(x))$ or $TS(WT(x))$ changes every time x is read or written because of an operation on x. The protocol will derive these elements from a logical clock (instead of a real clock) according to the dependency relationships among

the transactions.

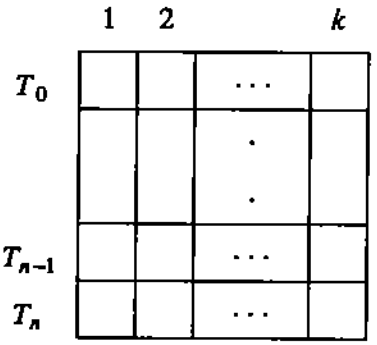


Figure 2. Timestamp table of MT(k)

Definition 1. Two atomic operations O_i and O_j conflict if 1) they belong to different transactions; 2) both access the same database item; 3) at least one of them is a write operation.

Definition 2. [16] A log $L = <D, T, \Sigma, S, \pi>$ is D-serializable (DSR) iff there exist real numbers $s_1, \ldots, s_n$ for the n transactions $T_1, T_2, \ldots, T_n$ such that

i) If $S(W_i) \cap S(R_j) \neq \emptyset, i \neq j$ and $\pi(W_i) < \pi(R_j)$, then $s_i < s_j$.

ii) If $S(R_i) \cap S(W_j) \neq \emptyset, i \neq j$ and $\pi(R_i) < \pi(W_j)$, then $s_i < s_j$.

iii) If $S(W_i) \cap S(W_j) \neq \emptyset, i \neq j$ and $\pi(W_i) < \pi(W_j)$, then $s_i < s_j$.

We add one more condition to describe the class TO(k) as follows.

Definition 3. A k-dimensional timestamp ordering (TO(k)) log has the following necessary condition for serializability: There exist real numbers $s_1, \ldots, s_n$ for the n transactions $T_1, T_2, \ldots, T_n$ such that

i)-iii) the same as in Definition 2.

iv) If $S(R_i) \cap S(R_j) \neq \emptyset, i \neq j$ and $\pi(R_i) < \pi(R_j)$, then $s_i < s_j$.

Considering the two-step transaction model, we add restriction iv) so that the protocol MT(k) can keep track of the latest transaction that reads a data item, and enforce correct dependencies due to read-write conflicts. In the two-step transaction model, R_j is the first operation of T_j. Once T_j issues $R_j[x]$ after T_i has issued $R_i[x]$, $TS(j)$ becomes the most recent read timestamp of item x. This implies the order $s_i < s_j$. There can be several transactions that read x before it is written by some other transaction T_l. Those read-write dependencies can be easily enforced by only enforcing the order between the most recent read timestamp and $TS(l)$. Considering the multi-

step transaction model, a read may not be the first operation of a transaction. The most recent read timestamp of item x is the timestamp of the transaction that reads x and also has the largest timestamp value (derived from a logical clock rather than a real clock). We can see that TO(k) is a proper subset of DSR because the conditions for TO(k) are more restrictive. Moreover, each TO(k) class has a different restriction on the range of s_i. For example,

Definition 4. A log is 1-dimensional timestamp ordering (TO(1)) iff

i)-iv) the same as in Definition 3.

v) $s_i = \pi(R_i)$.

For simplicity, we do not define the exact range of s_i for each class TO(k), $k \geq 2$. However, we define a necessary condition for the range of s_i for all TO(k), $k \geq 2$ as a comparison to Definition 4. Basically, for $k \geq 2$, s_i is restricted to a certain range after the first operation of T_i occurs. As more operations occurs, the range of s_i will be further restricted because more dependencies are created. These dependencies imply a partial order of s_i's.

Definition 5. A necessary condition for a TO(k) log, $k \geq 2$, is that there exist real numbers $s_1, \ldots, s_n$ for the n transactions $T_1, T_2, \ldots, T_n$ such that

i)-iv) the same as in Definition 3.

v) $t_i - 1 < s_i < t_i$, where t_i is the first element of the timestamp vector of T_i.

Note that if $t_i = t_j$, $i \neq j$, then the ranges of s_i and s_j overlap. Since timestamp elements are assigned integer values based on Algorithm 1 in Section III-A, the serializability number s_i is not an integer based on our definition. The order of two timestamp vectors is defined as follows.

Definition 6.

$TS(i) < TS(j)$ if there exists $1 \leq m \leq k$ such that $TS(i, m) < TS(j, m)$, and $TS(i, h) = TS(j, h), 1 \leq h < m$.

$TS(i) > TS(j)$ if there exists $1 \leq m \leq k$ such that $TS(i, m) > TS(j, m)$, and $TS(i, h) = TS(j, h), 1 \leq h < m$.

$TS(i) = TS(j)$ if there exists $1 \leq m \leq k$ such that both $TS(i, m)$ and $TS(j, m)$ are undefined, and $TS(i, h) = TS(j, h), 1 \leq h < m$.

$TS(i)$? $TS(j)$ if there exists $1 \leq m \leq k$ such that either $TS(i, m)$ or $TS(j, m)$ is undefined (but not both), and $TS(i, h) = TS(j, h)$, $1 \leq h < m$.

We assume that an undefined element is not equal to any integer.

III. THE PROTOCOL MT(k)

In this section, the algorithm, proof of correctness, the degree of concurrency, the complexity analysis, the optimized encoding of dependencies, and the implementation issues including vector processing will be presented.

A. The Algorithm and the Example for the Protocol MT(k)

Now we describe the algorithm in detail. Our algorithm is applicable to the multi-step transaction model. Let T_i attempt to issue an operation O on a data item x. Let j be either $RT(x)$ or $WT(x)$ such that $TS(j)$ is the larger timestamp. j is used to locate the timestamp vector of the latest transaction that accesses x. The design of this protocol is based on the idea that we compare $TS(j)$ with $TS(i)$. The comparison is done by going through corresponding elements in the two timestamp vectors from left to right until we reach two unequal or undefined elements. This gives the two timestamp elements that determine the order of T_j and T_i based on Definition 6. Then if both elements are defined, their order determines whether to accept or abort an incoming operation; otherwise we need to "encode" a new dependency relationship by making one element less than the other and accept the operation. The symbol '*' represents an undefined timestamp element. Each vector has k elements. Two counters *lcount* and *ucount* are used to set the k-th elements to distinct values. *lcount* is the current lower bound of the k-th elements, and *ucount* the current upper bound. Procedure $Set(j, i)$ is used to compare the vectors and encode the dependency $T_j \rightarrow T_i$ by setting $TS(j) < TS(i)$.

Algorithm 1: (the algorithm for the protocol MT(k))

Initialization:
1. $TS(i) := <*,*,...,*>$ for $i > 0$;
2. $TS(0) := <0,*,...,*>$;
3. $RT(x) := WT(x) := 0$ for *any* x;
4. $lcount := 0$; $ucount := 1$;
   /* counters for the k-th elements */

procedure *Scheduler*;
begin
  integer j;

  /* pick up the larger one */
  if $TS(RT(x)) < TS(WT(x))$ **then**
5. $j := WT(x)$;
  else
6. $j := RT(x)$
  endif;

  case(O) {
   read:
    if $Set(j, i)$ **then**
7. $RT(x) := i$;
8. *do the operation*;
9. **elseif** $j = RT(x)$ **and** $TS(WT(x)) < TS(i)$ **then**
10. *do the operation*;
    else
11. *abort* T_i
    endif;
   write:
    if $Set(j, i)$ **then**
12. $WT(x) := i$;
13. *do the operation*;
    else
14. *abort* T_i
    endif;
  }
end *Scheduler*;

boolean procedure $Set(j, i)$; **integer** j, i;
/* set $TS(j) < TS(i)$ */
begin
  integer m;

15. **if** $j = i$ **then**
    return(*true*)

```
        endif;

16.     m := compare(TS(j), TS(i));
        /* based on Definition 6 */

        case(TS(j) : TS(i)) {
17.        < : return(true);
18.        > : return(false);
        /* otherwise set TS(j, m) < TS(i, m) */
19.        = : if m = k then
                    TS(j, k) := ucount;
                    TS(i, k) := ucount + 1;
                    ucount := ucount + 2;
                else
                    TS(j, m) := 1;
                    TS(i, m) := 2
                endif;
                return(true);

20.        ? : if TS(i, m) = '*' then
                    if m = k then
                        TS(i, k) := ucount;
                        ucount := ucount + 1;
                    else
                        TS(i, m) := TS(j, m) + 1
                    endif;
                else
                    if m = k then
                        TS(j, k) := lcount;
                        lcount := lcount - 1;
                    else
                        TS(j, m) := TS(i, m) - 1
                    endif;
                endif;
                return(true);
            }
    end Set;
```

We think of T_0 as a virtual transaction which reads and writes all the data items before any other transactions. This is characterized by lines 2 and 3. As a result, there exists a dependency $T_0 \rightarrow T_i$ for any other transaction T_i, and $TS(0)$ never changes after initialization. The two counters *ucount* and *lcount* are used to set the k-th elements to distinct values. So any two timestamp vectors, after all k elements are set, are always distinguishable. Otherwise, we cannot enforce any further dependency between the two transactions. We give an example to show how

the algorithm works:

Example 2: Let $k = 2$, and three transactions T_1, T_2, T_3. The log L is as follows:

$$L = \begin{cases} T_1: & R_1[x] & & & W_1[y]\ W_1[z] \\ T_2: & & R_2[y] & & \\ T_3: & & & R_3[z] & \end{cases}$$

$\longrightarrow$ real time

The corresponding dependency digraph for log L is shown in Fig. 3. The dependencies are established in the sequence a, b, c, d, e. Edge d is created due to the conflicting operations $R_2[y]$ and $W_1[y]$. Edge e is created due to the conflicting operations $R_3[z]$ and $W_1[z]$. Table I shows how each dependency edge is encoded in the associated vectors. For example, edge a is represented by the relationship $TS(0) < TS(1)$. The second row records the initial value of each vector. A blank entry in the table means that the vector remains unchanged from the entry just above it.

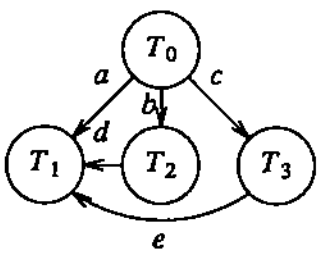


Figure 3. Dependency digraph for example 2

	$TS(0)$	$TS(1)$	$TS(2)$	$TS(3)$
initialization	<0, *>	<*, *>	<*, *>	<*, *>
$a : T_0 \rightarrow T_1$		<1, *>		
$b : T_0 \rightarrow T_2$			<1, *>	
$c : T_0 \rightarrow T_3$				<1, *>
$d : T_2 \rightarrow T_1$		<1, 2>	<1, 1>	
$e : T_3 \rightarrow T_1$				<1, 0>
resulting vectors	<0, *>	<1, 2>	<1, 1>	<1, 0>

Table I. Recording of timestamp vectors for example 2

The edge e is represented by the relationship <1, 0> < <1, 2> where the 2nd element of $TS(3)$ has been set to 0 rather than 1 because we need to distinguish $TS(2)$ and $TS(3)$. Otherwise we cannot enforce any further dependency between T_2 and T_3 because we run out of timestamp elements. The log L is equivalent to the serial log $T_3T_2T_1$ or $T_2T_3T_1$.

B. Correctness Proof of the Protocol

Lemma 1. The relation < is transitive. That is, if $TS(i) < TS(j)$ and $TS(j) < TS(l)$, then $TS(i) < TS(l)$.

Proof:

Suppose $TS(i) < TS(j)$ and $TS(j) < TS(l)$.

By Definition 6, there exist $1 \leq m_1 \leq k$, and $1 \leq m_2 \leq k$ such that

i) $TS(i, h) = TS(j, h)$, $1 \le h < m_1$,

ii) $TS(i, m_1) < TS(j, m_1)$,

iii) $TS(j, h) = TS(l, h)$, $1 \le h < m_2$,

iv) $TS(j, m_2) < TS(l, m_2)$.

Then, we have either $m_1 < m_2$ or $m_2 < m_1$ or $m_1 = m_2$.

If $m_1 < m_2$, then

$TS(i, h) = TS(j, h) = TS(l, h)$, $1 \le h < m_1$ by i) and iii),

$TS(i, m_1) < TS(j, m_1)$ by ii),

$TS(j, m_1) = TS(l, m_1)$ by iii).

Thus there exists $1 \le m_1 \le k$ such that

$TS(i, h) = TS(l, h)$, $1 \le h < m_1$,

$TS(i, m_1) < TS(l, m_1)$.

That is, $TS(i) < TS(l)$.

Similarly, $TS(i) < TS(l)$, if $m_2 < m_1$ or $m_1 = m_2$. □

Lemma 2. The relation $<$ is irreflexive. That is, there exists no T_i such that $TS(i) < TS(i)$.

Proof: By Definition 6, $TS(i) = TS(i)$. Also there exists no T_j such that $TS(i) < TS(j)$ and $TS(j) < TS(i)$, and thus $TS(i) < TS(i)$ by transitivity. □

Definition 7. Transactions T_i and T_j are dependent (denoted by $T_i \rightarrow T_j$) in a log if

i) Some operation O_i of T_i precedes and conflicts with some operation O_j of T_j, $i \neq j$; or

ii) there exists T_l such that $T_i \rightarrow T_l$ and $T_l \rightarrow T_j$.

Theorem 1. A log is D-serializable (DSR) iff its dependency relation ($\rightarrow$) defines a partial order[1].

□

This result has been discussed in [2, 5, 16]. To get a total order, we do a topological sort on the partial order.

Theorem 2. MT(k) assures serializability.

Proof: From the algorithm, we observe that if $T_i \rightarrow T_j$ then $TS(i) < TS(j)$, and that once $TS(i) < TS(j)$ is determined, the relationship will not change afterwards. Also from Lemma 1 and Lemma 2, we know that < defines a partial order. The relation $\rightarrow$ must also be a partial order. Then, by Theorem 1, it implies serializability.

□

C. Degree of Concurrency

Papadimitriou [16] defines the degree of concurrency provided by a scheduler as the number of serializable logs accepted by the scheduler. If a scheduler allows more serializable logs, it will spend less time in rearranging the execution order of operations. The hierarchy of the degree of concurrency for the classes 2PL, SSR, DSR, and SR can be found in [5, 16]. Also the hierarchy of the degree of concurrency for the classes 2PL, TO(1), SSR, and SR for distributed databases has been given in [11]. Based on the *two-step transaction model*, we add TO(k) where $k \geq 3$ and form the extended hierarchy depicted in Fig. 4. TO(k) is the class of logs recognized by the protocol MT(k).

We observe that TO(3) and TO(1) are distinct classes. But for $k > 3$, we have TO(k) = TO(3). We will show further in Theorem 3 that for the *q-step transaction model*, we have $TO(2q - 1) = TO(k)$ for all $k \geq 2q - 1$. In Fig. 4, we have $q = 2$. Generally, when $2 \leq k \leq 2q - 1$, $TO(k - 1)$ is not a subset of TO(k) because, excluding undefined elements, column $k - 1$ of the timestamp table of the protocol $MT(k - 1)$ contains only distinct elements but column $k - 1$ of the table of the protocol MT(k) may contain equal elements. Thus values in

1. A relation θ is a partial order if θ is transitive ($a \theta b$, $b \theta c$ implies $a \theta c$), and irreflexive ($\forall a$, $a \sim\theta a$).

the timestamp table of MT(k – 1) is not the same as the values in the prefix of the table of MT(k). That is, the dependency information stored in the table of MT(k – 1) is not fully covered by that in the table of MT(k).

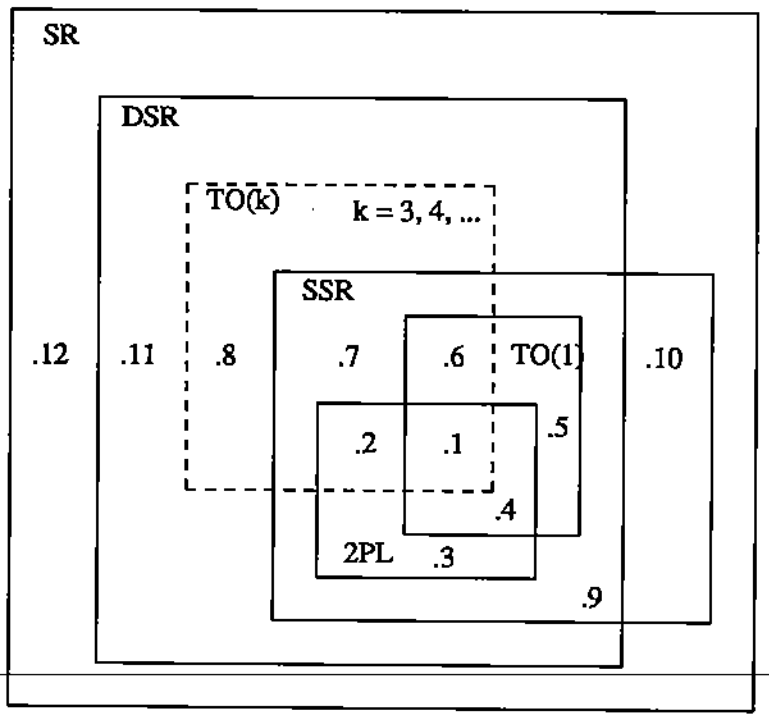


Figure 4. Classes of serializable logs for the two-step transaction model

The graph is partitioned into 12 regions. .*i* denotes region *i* in the graph. TO(k) is placed inside DSR because any TO(k) is a subset of DSR as mentioned in Definition 3 in Section II. Further, log L_i implies the existence of the region *i* in the hierarchy graph. Each simple log, not a concatenation of several logs, is equivalent to the serial log $T_1T_2T_3$.

$L_1 = R_1[x]W_1[x]R_2[x]W_2[x]R_3[x]W_3[x]$

$L_2 = R_2[y]R_1[z]R_3[z]W_1[x]W_2[x]W_3[y]$

$L_3 = R_2[x]R_1[x]R_3[z]W_1[y]W_2[y]W_3[x]$

$L_4 = R_1[x]W_1[y]R_2[x]R_3[z]W_2[x]W_3[x]$

$L_5 = L_4 \cdot L_6$ ($\cdot$ means concatenation of two logs [16])

$L_6 = R_1[z]R_2[y]R_3[y]W_3[y]W_1[x]W_2[x]$

$L_7 = L_2 \cdot L_6$

$$L_8 = R_2[y]R_3[y]W_3[y]R_1[z]W_1[x]W_2[x]$$

$$L_9 = L_4 \cdot L_7$$

$$L_{10} = R_1[y]R_2[y]W_2[z]R_3[z]W_1[z]W_3[z]$$

$$L_{11} = L_8 \cdot L_9$$

$$L_{12} = L_{10} \cdot L_{11}$$

The membership of each composite log is a little tricky to understand. We clarify this by showing

i) $L_7 = L_2 \cdot L_6 \in$ region 7.

ii) $L_9 = L_4 \cdot L_7 \in$ region 9.

Proof:

i)

Suppose $L_2 \in$ region 2 and $L_6 \in$ region 6.

Then,

$L_2 \in \text{TO(3)} \cap \text{SSR}$,

$L_6 \in \text{TO(3)} \cap \text{SSR}$,

$L_2 \notin \text{TO(1)}$, and

$L_6 \notin \text{2PL}$.

Thus,

$$L_2 \cdot L_6 \in \text{TO(3)} \cap \text{SSR} - \text{TO(1)} - \text{2PL} = \text{region 7.}$$

ii)

Suppose $L_4 \in$ region 4 and $L_7 \in$ region 7.

Then,

$L_4 \in \text{DSR} \cap \text{SSR}$,

$L_7 \in \text{DSR} \cap \text{SSR}$,

$L_4 \notin \text{TO(3)}$, and

$L_7 \notin \text{2PL} \cup \text{TO(1)}$.

Thus,

$L_4 \cdot L_7 \in \text{DSR} \cap \text{SSR} - \text{TO}(3) - 2\text{PL} - \text{TO}(1) = \text{region 9}.$

□

D. Analysis of the Protocol MT(k)

We now present several observations, theorems, the timestamp complexity of the protocol, optimized encoding rules, and briefly the implementation issues.

1. Observations

Suppose transaction T_i has q operations. Based on Algorithm 1, we have:

i) $Set(j, i)$ may set at most two elements $TS(j, m)$ and $TS(i, m)$. Only one element may be set for T_i.

ii) $Set(j, i)$ will be called exactly q times for different j's. Once for each operation of T_i. As a result, $TS(i)$ can be the most recent read or write timestamp of at most q different items.

iii) $Set(i, l)$ may be called and return a true value at most q times for different l's while $TS(i)$ is used as the most recent read or write timestamp with respect to transaction T_l. If $Set(i, l)$ returns true, $TS(l)$ will take over from $TS(i)$ the most recent read or write timestamp of some item to be accessed by T_l.

iv) At most $2q$ elements in $TS(i)$ may be set either by $Set(j, i)$ or by $Set(i, l)$ for some j, l. To obtain $2q$, we apply i) to ii) and iii) and count the total elements that may be set in $TS(i)$.

2. Theorems

Lemma 3. TO(2q) = TO(k) for all k ≥ 2q, where q is the maximum number of operations in a single transaction, and k is the vector size.

Proof: By Observation iv), the $(2q + 1)$-th to the k-th elements of each vector will remain undefined throughout the log. So, MT(k) will still recognize the same set of logs as MT(2q) when k ≥ 2q + 1.

□

The result can be refined one step further.

Lemma 4. If $k = 2q$ and a single transaction has at most q operations, $TS(i, 2q)$ will not be set by MT(k) for any transaction T_i in the log.

Proof: We assume the contrary. That is, the $2q$-th element of some timestamp vector is set. Then there exists $TS(i, 2q)$ that is set by executing $Set(j, i)$ for some j. T_i will issue $q - 1$ operations before $TS(i, 2q)$ is set.

By Observation i), an element of $TS(i)$ can be set either a) by executing $Set(j, i)$ for some j, or b) by executing $Set(i, l)$ for some l. We next count how many times cases a) and b) can occur before $TS(i, 2q)$ is set.

By Observation ii), case a) can occur $q - 1$ times before $TS(i, 2q)$ is set. Further, $TS(i)$ can be the most recent read or write timestamp of at most $q - 1$ data items before $TS(i, 2q)$ is set. So, case b) can occur at most $q - 1$ times before $TS(i, 2q)$ is set. Since cases a) and b) as a whole can occur at most $2q - 2$ times, we have $2q - 2$ elements to the left of $TS(i, 2q)$ can be set. But there must be $2q - 1$ assigned elements to the left of $TS(i, 2q)$ in the vector $TS(i)$. Thus, our assumption does not hold. □

Theorem 3. TO(2q − 1) = TO(k) for all k ≥ 2q − 1, where q is the maximum number of operations in a single transaction. □

This is obtained by Lemma 3 and Lemma 4. Considering the extreme case when q = 1, we have TO(1) = TO(k) for all k ≥ 1. Note that at line 9 Algorithm 1 we may change the condition $TS(WT(x)) < TS(i)$ to $Set(WT(x), i)$ to allow higher concurrency. In this case, Observations ii)-iv) will not hold.

3. Time Complexity

We consider the complexity of an *on-line* algorithm[2] where serializability is validated once an operation is generated. We find MT(k) can recognize a TO(k) log of n transactions in

2. This is different from an *off-line* algorithm where serializability can be validated after the succeeding operations in a log have also been generated.

$O(nqk)$ time, since a log has $O(nq)$ operations and it takes $O(k)$ time to schedule an operation. The major cost to schedule one operation comes from the comparison of two vectors of size k. The time complexity is comparable to other schedulers since $O(n^2q)$ is currently known [16] to recognize either a DSR log or a 2PL log. Actually, our scheduler favors smaller transactions in the sense that transactions with fewer operations tend to get faster response time.

4. The Starvation Case

A transaction may be repeatedly aborted by the protocol MT(k) without being committed as illustrated by the following log as an example.

$$L = W_1[x]W_2[x]R_3[y]W_3[x]$$

The dependency digraph for log L is shown in Fig. 5. The dependencies are established in the sequence a, b, c, d. T_3 will be aborted once it issues $W_3[x]$ because $TS(2) > TS(3)$ and thus the dependency edge d is not allowed. Even if T_3 restarts, the same situation repeats. This starvation problem can be solved as follows. Suppose T_i is aborted due to the order $TS(i) < TS(j)$, we flush out $TS(i)$, set $TS(i, 1)$ to $TS(j, 1) + 1$, and only then abort T_i. Thus, we can ensure $TS(j) < TS(i)$ when T_i restarts in the future if T_j is not aborted before T_i restarts. So, in the above example, just before T_3 is aborted, $TS(3)$ is set to <3, *,...>. When T_3 restarts, it is allowed to proceed to its end. Note that we are not dealing with "cascading" aborts of several transactions, which may repeatedly occur and may not be solved easily.

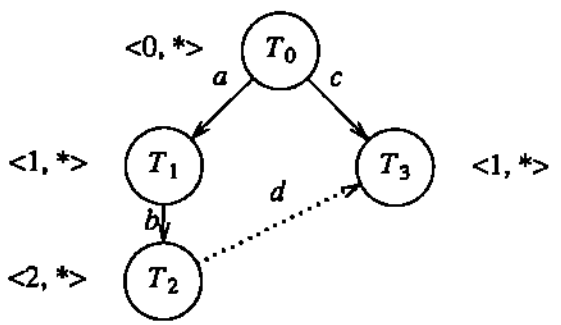


Figure 5. Dependency digraph for the starvation case

5. Optimized Encoding of Dependencies

We note that the protocol MT(k) does not necessarily generate a total order but a partial order among the transactions. It yields more freedom in determining the order based on subsequent dependency relationships. We can increase the degree of partial order by increasing k. A larger k provides higher degree of concurrency up to a certain limit based on Theorem 3.

In this section, we study a variation of the dependency encoding rules presented in the procedure *Set* in Algorithm 1 in order to increase the degree of concurrency. We observe that if a data item is frequently accessed, the original encoding rules are more likely to generate a total order as shown in the following example:

Example 3: Suppose

$$L = \cdots R_1[x]W_2[x]W_3[x]\cdots$$

Table II focuses on the middle operating of the log, and shows how each dependency is encoded in the associated vectors.

	$TS(0)$	$TS(1)$	$TS(2)$	$TS(3)$	$TS(4)$
vectors just before the middle operating	<0, *>	<*, *>	<*, *>	<*, *>	<1, 4>
$T_0 \rightarrow T_1$		<1, *>			
$T_1 \rightarrow T_2$			<2, *>		
$T_2 \rightarrow T_3$				<3, *>	
resulting vectors	<0, *>	<1, *>	<2, *>	<3, *>	<1, 4>

Table II. Recording of timestamp vectors for example 3

x is a frequently accessed data item. Table II shows that the order between T_2 and T_4 (or T_3 and T_4) has also been enforced. That is, accesses of the item x tend to create a total order among the

vectors. This may limit some potential concurrency in the future. We can relax this ordering a little by the following approach. Suppose we have the vectors

$$T_1 : \langle 1, 3, *, * \rangle,$$
$$T_2 : \langle *, *, *, * \rangle.$$

To encode the dependency $T_1 \rightarrow T_2$, the normal way is to set the 1st element of $TS(2)$ to 2. Suppose the dependency is created due to an access of a frequently accessed item. We can instead encode the dependency in the elements closer to the right end of the vectors. One possible way is to copy the prefix of $TS(1)$ to $TS(2)$, and encode the dependency using the 3rd elements:

$$T_1 : \langle 1, 3, 1, * \rangle,$$
$$T_2 : \langle 1, 3, 2, * \rangle.$$

instead of the encoding

$$T_1 : \langle 1, 3, *, * \rangle,$$
$$T_2 : \langle 2, *, *, * \rangle.$$

Then for any other vectors with the value <1, *, *, *> or <1, 3, *, *>, we will not create any total order with T_2 at this instance. Thus, higher concurrency among them can be allowed in the future. However, if a dependency is created due to an access of a less often accessed item, we still encode the dependency in the normal position.

This approach is based on the following observation. Setting an element near the left end of a vector can better distinguish the vector from other vectors than setting an element near the right end of the vector. If an item is frequently accessed, the encoding of the created dependency in the normal position may distinguish the vectors faster. So, we can encode this kind of dependency near the right end of the vector to allow higher concurrency. This observation is further discussed in Section VI-A.

To apply this approach, we need to know the access rate of each item, which is either static information or dynamic data measured during the scheduling. We also need to decide how close to the right end of the vector a dependency should be encoded as opposed to the normal encoding position. This can be an adaptable decision as the access rate is dynamically changed.

6. Implementation Issues

Some implementation issues are of practical interest. We now highlight the general ideas and avoid the details.

a) Based on the multiprogramming level analyzed in [6], normally there are 8 to 10 transactions which are currently active in the system. So, the size of the timestamp table can normally fit in main memory. b) Storage for a timestamp vector can be reclaimed as soon as the transaction is committed and it will not be used for the most recent read or write timestamp of a data item. c) Thomas [20] describes the situation when some writes can be simply ignored instead of being aborted. This can be easily incorporated into our protocols by not aborting T_i but ignoring the write operation if the condition $TS(RT(x)) < TS(i) < TS(WT(x))$ is true at line 14 in Algorithm 1. d) Reed [19] proposed a multiple version concurrency control mechanism using single-valued timestamps. The idea can be extended to timestamp vectors. e) Last but not least, we can use vector processors [12] to speed up the comparison of two timestamp vectors, and we can also store the timestamp table in cache memory to get faster access time. The vector processing mechanism is briefly discussed in next subsection.

E. The Timestamp Vector Processing Mechanism

As mention earlier, it takes $O(k)$ time to compare two vectors of size k. In this subsection, we show how vector processors can be used so that the comparison can be done in $O(\log k)$ time using the technique proposed in [12]. We illustrate the algorithm in Fig. 6. We have four rows of processors labeled a, b, c, d. a_i denotes the ith processor that holds the ith element of a vector. Similarly for b_i, c_i, and d_i. The input is two timestamp vectors. The output is the order between the two vectors. There are five processing phases. In Fig. 6, the third elements in the input vectors are the first pair of elements that are not equal. The parallel processing mechanism is designed to find this pair of elements faster.

input : $TS(1)$: <1, 3, 2, 2>

$TS(2)$: <1, 3, 5, 2>

output : the order of $TS(1)$ and $TS(2)$

Phases:

1. Load in the vector elements.

   (Each box stands for a processor.)

a:	1	3	2	2

b:	1	3	5	2

2. Subtraction.

   (c_i denotes the ith processor from the left.)

c:	0	0	1	0

$$c_i = \begin{cases} 0 & \text{if } a_i = b_i \\ 1 & \text{if } a_i \neq b_i \end{cases}$$

3. Partial **OR**.

   ($x \oplus y = 0$ if $x = y = 0$, and 1 otherwise.)

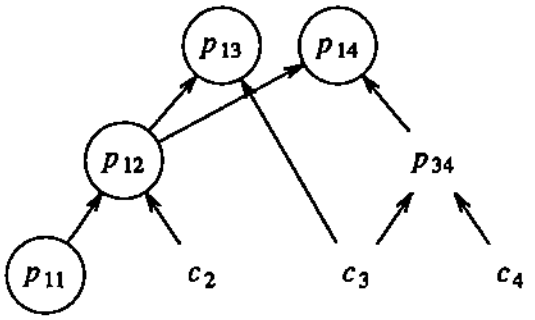
d:	0	0	1	1

$$d_i = c_1 \oplus c_2 \oplus \cdots \oplus c_i$$

4. d_i checks the value in d_{i-1} for all $i \geq 1$ (assuming d_0 holds 0). Only one processor will find that it holds the value 1 but its left neighbor holds the value 0. d_3 is the only processor for which this condition holds. This condition indicates that the 3rd elements in $TS(1)$ and $TS(2)$ are the first two corresponding elements that are not equal.

5. The order of $TS(1)$ and $TS(2)$ is determined by the order of a_3 and b_3.

Figure 6. Example for the vector comparison algorithm



$p_{12} = p_{11} \oplus c_2$

$p_{34} = c_3 \oplus c_4$

$p_{13} = p_{12} \oplus c_3$

$p_{14} = p_{12} \oplus p_{34}$

$\rightarrow$ indicates the data flow

Figure 7. Layout of vector processors

Phases 1, 2, 4, 5 can all be done in parallel in constant time. We now show that phase 3 can be done in parallel in O(log k) time where k is the vector size. Let p_{11} denote c_1. The processor layout to compute the values of $c_1 \oplus c_2 \oplus \cdots \oplus c_i$, $i \leq 4$ is shown in Fig. 7. At the end of phase 3, p_{1i} holds the value $c_1 \oplus c_2 \oplus \cdots \oplus c_i$. Without loss of generality, for timestamp vectors of size k, we can build up this kind of tree of height O(log$k$). The parallel processing time mainly depends on the height of the tree. Thus the time complexity is O(log$k$). Also each node of the tree need not correspond to a distinct processor. One processor can perform the function of two nodes if they have disjoint processing time. For example, p_{14} and p_{11} can be the same processor since their processing time do not overlap. In this algorithm, we have ignored the case when a vector may contain undefined elements. However, the algorithm can be easily refined without affecting the time complexity order if undefined elements are also considered. This leads to the following theorem.

Theorem 4. MT(k) can recognize a TO(k) log in O(nqlog$k$) time using O(k) processors, where n is the number of transactions in the log, q is the maximum number of operations in a single transaction, and k is the vector size. □

IV. THE COMPOSITE PROTOCOL MT(k^+)

Recall that in Fig. 4 TO(3) and TO(1) do not include each other. In general, the protocol MT(k) may not allow more concurrency than MT(h) for $h < k$. In this section, we construct the protocol MT(k^+) that recognizes the class

$$TO(k^+) = TO(1) \cup TO(2) \ldots \cup TO(k) \text{ for } k \geq 1$$

Obviously, the inclusivity property holds. That is

$$TO(1) = TO(1^+) \subset TO(2^+) \subset \ldots \subset TO(k^+)$$

Thus MT(k^+) is guaranteed to allow higher concurrency than MT(h^+) for $h < k$.

We first examine the simple case. We construct a combination of MT(k_1) and MT(k_2) such that the composite protocol recognizes the union of the classes TO(k_1) and TO(k_2). The simplest way is to run MT(k_1) and MT(k_2) independently using separate timestamp tables and data structures. Each operation is processed through both MT(k_1) and MT(k_2) before the next operation can be processed. Each scheduler updates its own timestamp table independently. If the operation is accepted by at least one of the schedulers, we can process the next operation. Otherwise we abort the transaction and rollback. If the operation is rejected by only one of the schedulers, say MT(k_1), we stop MT(k_1). The succeeding operations will be only processed through MT(k_2). The reason is that the log will not be in the class TO(k_1) once an operation of the log is rejected by MT(k_1). So, there is no need to use MT(k_1) to process the succeeding operations.

The above method can be more efficiently done. In practice, MT(k_1) and MT(k_2) can share the prefix part of each vector. We will show that the prefix part of the corresponding vectors generated by both schedulers are equal if the log is a member of both TO(k_1) and TO(k_2). So, we need only one table to store the prefix part of each vector, and build one shared module that updates the prefix part for both MT(k_1) and MT(k_2). We use the following additional notation for Fig. 8 and the discussion in rest of this section.

Notation.

$MT(k_1)$	the protocol MT using timestamp vectors of k_1 elements.
$MT(k_2)$	the protocol MT using timestamp vectors of k_2 elements.
$a(i)$	the timestamp vector of transaction T_i maintained by $MT(k_1)$.
$a(i, h)$	the h-th element of $a(i)$.
$a(i, [_1^l])$	the prefix of $a(i)$ containing $a(i, 1)$ to $a(i, l)$.
Set 1	the procedure *Set* used in the protocol $MT(k_1)$.
$b(i)$, $b(i, h)$, $b(i, [_1^l])$, and *Set* 2	: similar notation for $MT(k_2)$.
$a(i, [_1^l]) = b(i, [_1^l])$	denotes $a(i, h) = b(i, h)$ for $1 \le h \le l$. Note that two elements are said to be equal if they are either both undefined or both integers of the same value.

Set 1 is the same procedure as *Set* in Algorithm 1 in Section III-A except that each vector has k_1 element. Each of the protocols $MT(k_1)$ and $MT(k_2)$ has its own timestamp table and data structures. Theorem 5 shows that the prefix part of each two corresponding vectors will be always equal after any single operation is processed by both $MT(k_1)$ and $MT(k_2)$.

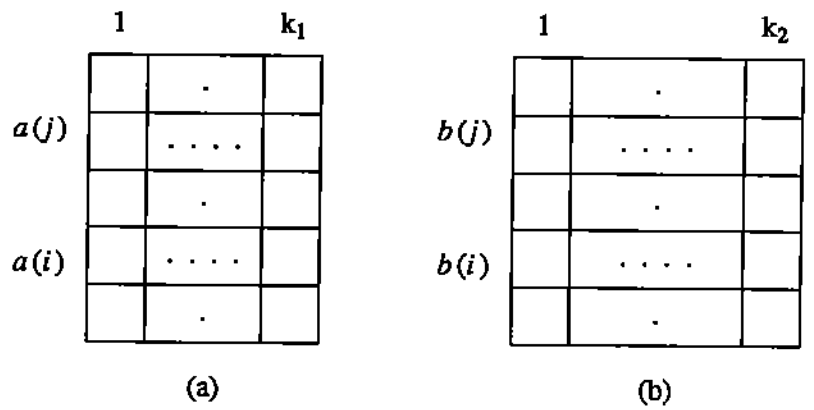


Figure 8. (a) Timestamp table of $MT(k_1)$

(b) Timestamp table of $MT(k_2)$

Lemma 5. Suppose $1 < k_1 \leq k_2$, and for any j, i

(5.1) $a(j, [_1^l]) = b(j, [_1^l])$, and

(5.2) $a(i, [_1^l]) = b(i, [_1^l])$, where $l = k_1 - 1$.

Then executing both $Set1(j, i)$ and $Set2(j, i)$ preserves equalities (5.1) and (5.2).

Proof: $Set1$ and $Set2$ update the corresponding two tables independently. Let m_1 and m_2 be the returned values at line 16 of Algorithm 1 for $Set1$ and $Set2$ respectively. Based on the algorithm, $Set1$ may only change the m_1-th element in a vector while $Set2$ may only change the m_2-th element. There are two cases:

i) $m_1 = m_2 \leq l$

$Set1$ and $Set2$ will either set no elements or set the corresponding elements to the same values according to the encoding rules. Thus equalities (5.1) and (5.2) in Lemma 5 are preserved.

ii) $m_1 = k_1$, and $m_2 \geq k_1$

$Set1$ and $Set2$ will not change any elements in the prefix parts $a(j, [_1^l])$, $a(i, [_1^l])$, $b(j, [_1^l])$, and $b(i, [_1^l])$ where $l = k_1 - 1$. Thus equalities (5.1) and (5.2) are preserved. □

Before we proceed to the next theorem, we assume that lines 9 and 10 are crossed out from Algorithm 1 in Section III-A to simplify our proof of the theorem. However, the theorem is not restricted to this assumption.

Theorem 5. If a log can be accepted by both MT(k_1) and MT(k_2) and $1 < k_1 \leq k_2$, then the equality

(5.3) $a(i, [_1^l]) = b(i, [_1^l])$ where $l = k_1 - 1$

holds for any transaction T_i in the log after an operation is processed by both MT(k_1) and MT(k_2).

Proof: (by induction on the number of operations)

i) Basis: (5.3) is true because vectors are all initialized to undefined values. Also $RT(x)$ and $WT(x)$ are all initialized to zero for any data item x by observing Algorithm 1.

ii) Induction hypothesis: Suppose (5.3) holds after a number of operations are accepted by both $\mathrm{MT}(k_1)$ and $\mathrm{MT}(k_2)$.

iii) Induction step: After any operation is accepted by both $\mathrm{MT}(k_1)$ and $\mathrm{MT}(k_2)$ in ii), the indices (say $RT(x)$ and $WT(x)$) to keep track of the most recent read timestamp and write timestamp in both $\mathrm{MT}(k_1)$ and $\mathrm{MT}(k_2)$ are still equal for any data item (say x) by observing lines 7 and 12 of Algorithm 1 in Section III-A. Thus when $\mathrm{MT}(k_1)$ and $\mathrm{MT}(k_2)$ schedule the next operation, procedures *Set* 1 and *Set* 2 will be given equal actual parameters. Then, by Lemma 5, equalities (5.1) and (5.2) are preserved. The theorem follows. □

By the theorem, $a(i, [^{l}_{1}])$ and $b(i, [^{l}_{1}])$ where $l = k_1 - 1$ are always equal. So, we need only one table to store the prefix. The revised composite protocol to recognize the class $\mathrm{TO}(k_1) \cup \mathrm{TO}(k_2)$ is as follows.

The timestamp tables for $\mathrm{MT}(k_1)$ and $\mathrm{MT}(k_2)$ with shared prefix are shown in Fig. 9. All the columns of M_0 and the one column of M_1 compose the timestamp table for $\mathrm{MT}(k_1)$. All the columns of M_0 and M_2 compose the timestamp table for $\mathrm{MT}(k_2)$.

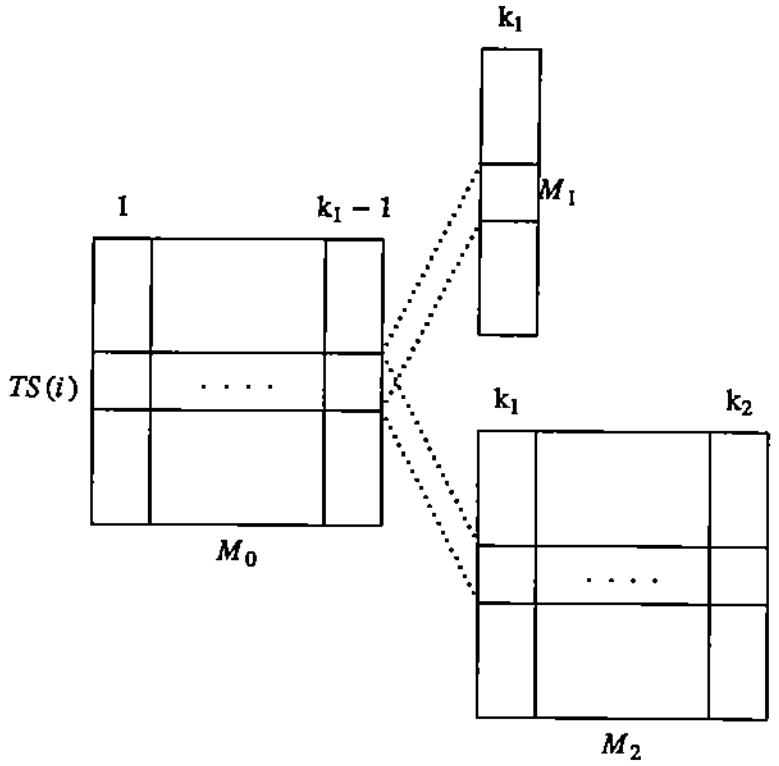


Figure 9. Timestamp tables of MT(k_1) and MT(k_2) with shared prefix

Since M_0 is shared by both MT(k_1) and MT(k_2), we need one module to update the prefix part for both MT(k_1) and MT(k_2). However, we need two tables to store the suffix part of each vector. Each column of M_0 can have many equal-valued timestamp elements. The elements in the last columns of M_1 and M_2 should be assigned distinct values to distinguish the two vectors. MT(k_1) and MT(k_2) use separate counters for the last columns. For each operation, if it creates a new dependency conflicting with the dependencies already encoded in M_0, we need to abort the associated transaction and rollback. For a newly-created dependency, if the prefix part of the two associated timestamp vectors are already set to equal, we use their suffix part stored in M_1 and M_2 to encode the dependency under the subprotocols MT(k_1) and MT(k_2) respectively. If the new dependency conflicts with the dependencies already encoded in M_1 or M_2, we stop MT(k_1) or MT(k_2) respectively for succeeding operations. If both MT(k_1) and MT(k_2) are stopped, we abort all the active transactions and rollback.

We can now construct the composite protocol MT(k^+) as follows. The timestamp tables *PREFIX* and *LASTCOL* are shown in Fig. 10. *PREFIX*(h) denotes column h of *PREFIX*, and *LASTCOL*(h) denotes column h of *LASTCOL*. *PREFIX*(1) to *PREFIX*(h − 1) and *LASTCOL*(h)

compose the timestamp table of MT(h). *PREFIX*(h) is shared by MT(h + 1), ..., MT(k). *LASTCOL*(h) serves as the last column of the timestamp table of MT(h). Each column of *PREFIX* can have many equal-valued elements. But the elements in each column of *LASTCOL* need to be assigned distinct values by using separate counters.

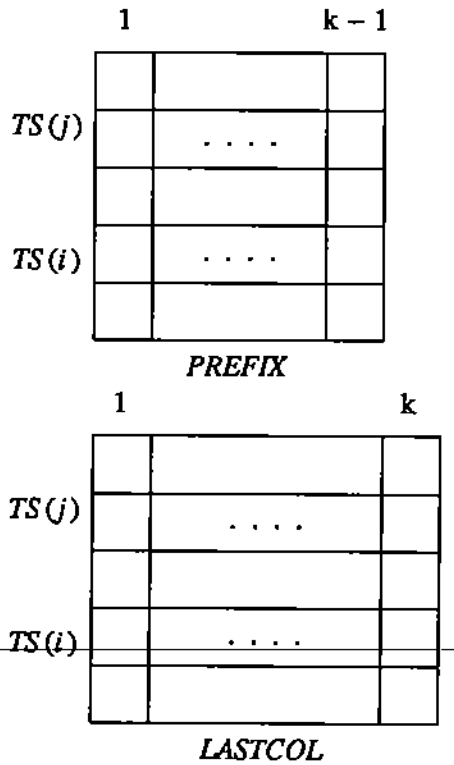


Figure 10. Timestamp tables of MT(k^+)

For a newly-created dependency $TS(j) \rightarrow TS(i)$, the composite protocol runs through *PREFIX*(h) and *LASTCOL*(h) (starting with h := 1), and encodes the dependency by setting $TS(j, h) < TS(i, h)$ in the column *LASTCOL*(h) for the subprotocol MT(h), and in the column *PREFIX*(h) for the subprotocols MT(h+1), MT(h+2), ..., MT(k). If the two timestamp elements are already set to equal values in *PREFIX*(h), MT(k^+) continues to check the next column *PREFIX*(h+1). If $TS(j, h) > TS(i, h)$ in the column *PREFIX*(h), we stop MT(h+1), MT(h+2), ..., MT(k). If $TS(j, h) > TS(i, h)$ in the column *LASTCOL*(h), we stop MT(h). If all the subprotocols are stopped, we abort all the active operations, and rollback. The scheduler can accept the current operation as long as the newly-created dependency can be encoded under at least one of the subprotocols. The algorithm is sketched as follows. The arrow "=>" indicates the actions

to be performed when the corresponding case condition is true.

Algorithm 2: (the algorithm for the protocol MT(k^+))

0. start all the subprotocols **MT**(i), $1 \le i \le k$;

   initialize data structures.

1. h := 1;

   wait for the operation to arrive.

2. (Comment: Check the column *LASTCOL*(h) for the subprotocol **MT**(h).)

   cases:

   i) The subprotocol **MT**(h) has been stopped.

      => go to 3.

   ii) The new dependency conflicts with the dependencies already encoded in *LASTCOL*(h).

      => stop **MT**(h);

         go to 3.

   iii) The new dependency can be (or has been) encoded in *LASTCOL*(h).

      => encode the dependency by using the counters of **MT**(h) (or do nothing);

         go to 3.

3. (Comment: Check the column *PREFIX*(h) for the subprotocols **MT**(h+1), **MT**(h+2), ..., **MT**(k).)

   cases:

   i) h = k.

      => go to 4.

   ii) All the subprotocols **MT**(h + 1), **MT**(h + 2), ..., **MT**(k) have been stopped.

      => go to 4.

   iii) The new dependency conflicts with the dependencies already encoded in *PREFIX*(h).

      (Comment: The prefix table for the subprotocols MT(i), $h < i \le k$, is invalid.)

      => stop **MT**(h + 1), **MT**(h + 2), ..., **MT**(k);

         go to 4.

iv) The new dependency can be (or has been) encoded in *PREFIX* (h).

=> encode the dependency under the subprotocols MT(i), $h < i \leq k$ (or do nothing);

go to 4.

v) The corresponding two elements in *PREFIX* (h) are already set to equal.

=> h := h + 1;

go to 2.

4. cases:

i) All the subprotocols MT(1), MT(2), ..., MT(k) have been stopped.

=> abort all the active transactions and rollback;

restart all the aborted transactions;

go to 0.

ii) Otherwise.

=> accept the current operation;

go to 1.

The time complexity of the worst case is the time taken to recognize a log in the intersection of TO(1), TO(2), ..., and TO(k). In that case, if we run MT(1), MT(2), ..., MT(k) independently, the total complexity will be $\sum_{h=1}^{h=k} O(nqh) = O(nqk^2)$, where n is the number of transactions in the log, and q is the maximum number of operations in a *single* transaction. However, by using the composite protocol MT(k^+) we eliminate some duplications in the processes of the subprotocols. The composite protocol will run through $O(k)$ elements to schedule each operation. The complexity is still $O(nqk)$, the same as that of MT(k). By using a similar timestamp vector processing mechanism described in Section III-E, the complexity can further reduce to $O(nq \log k)$.

We have found that the timestamp vector is a useful tool for switching between classes of concurrency algorithms such as MT(k_1) and MT(k_2). This work is being used for the design of adaptable concurrency control mechanisms [8].

V. PROTOCOLS FOR NESTED TRANSACTION AND DISTRIBUTED DATABASE MODELS

A. The Protocol MT(k_1, k_2) for Nested Transactions

In the *nested transaction model* [15], a *parent* transaction may have a set of subtransactions which can be concurrently executed. The protocol MT(k_1, k_2) is suitable for this kind of model. The transactions are partitioned into mutually disjoint groups $G_1, G_2, \ldots, G_m$ based on some partition rules. For example, a group may correspond to a level of parents, grandparents, or children of a nested transaction. The timestamp tables are shown in Fig. 11. The columns in the first table represent transaction timestamp order. The columns in the second table represent group timestamp order.

The serializability is assured at two levels. First, we enforce serializability among the groups based on the same algorithm of MT(k) except that groups instead of transactions are involved. Second, we enforce serializability among the transactions inside each group. Transaction indices $RT(x)$ or $WT(x)$ will be used to locate timestamps of a transaction as well as its associated group. The group timestamps will be involved if and only if two immediately dependent transactions are in two different groups. In such a case, we will use only the group timestamps to determine whether to accept or abort an operation.

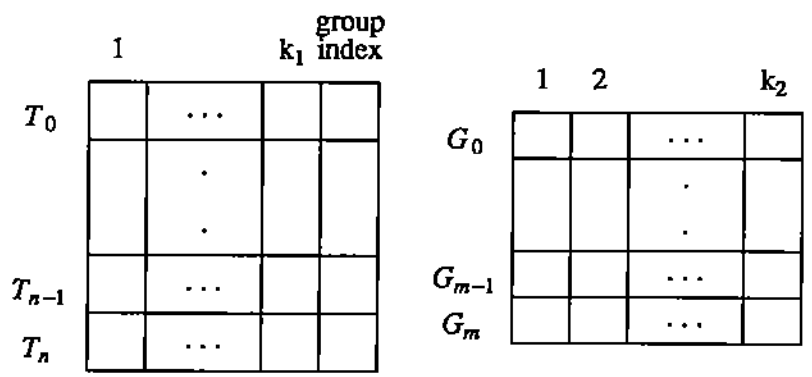


Figure 11. Timestamp tables of MT(k_1, k_2)

Example 4: Suppose $G_1 = \{ T_1, T_2 \}, G_2 = \{ T_3 \}, k_1 = k_2 = 2$, and

$$L = R_1[x]W_1[z]R_2[y]W_2[x]W_3[z].$$

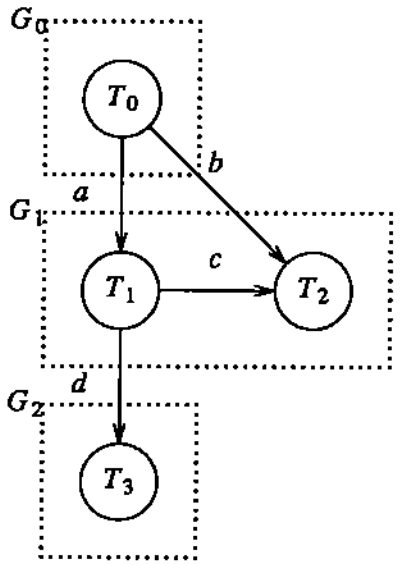


Figure 12. Dependency digraph for example 4.

The dependency digraph is drawn in Fig. 12. Each dotted-lined box denotes a group. T_0 is the virtual transaction and becomes a group by itself. The dependencies are established in the sequence a, b, c, d. MT(k_1, k_2) will encode each dependency edge by setting the associated timestamp vectors as shown in table III. A blank entry in the table means that the vector remains unchanged from the entry just above it.

$GS(i)$: group timestamp

$TS(i)$: transaction timestamp

	$GS(0)$	$TS(0)$	$GS(1)$	$TS(1)$	$TS(2)$	$GS(2)$	$TS(3)$
initialization	<0, *>	<0, *>	<*, *>	<*, *>	<*, *>	<*, *>	<*, *>
$a : G_0 \rightarrow G_1$			<1, *>				
$b : G_0 \rightarrow G_1$							
$c : T_1 \rightarrow T_2$				<1, *>	<2, *>		
$d : G_1 \rightarrow G_2$						<2, *>	
resulting vectors	<0, *>	<0, *>	<1, *>	<1, *>	<2, *>	<2, *>	<*, *>

Table III. Recording of timestamp vectors for example 4

The edges a, b or d connects two different groups. Group timestamps are used to encode every *group dependency*. For example, the dependency $G_0 \rightarrow G_1$ is represented by the order $GS(0) < GS(1)$. The group dependency a implies the group dependency b. So, we need not set any vectors for b. Edge c connects two transaction nodes within the same group. So, transaction timestamps are used to encode the *transaction dependency*. Based on MT(k_1, k_2), group dependency will be assured to be antisymmetric. That is, $G_1 \rightarrow G_2$ and $G_2 \rightarrow G_1$ cannot be both true. If in the future a new dependency $T_3 \rightarrow T_2$ is created due to some conflict, it is disallowed since it also implies the dependency $G_2 \rightarrow G_1$. It can be seen that $G_1, G_2, \ldots, G_m$ can be further grouped into supergroups, and the same idea applies. We can generalize MT(k_1, k_2) to MT(k_1, k_2, ..., k_l) for a hierarchy of groups with l levels. Each level then has an associated timestamp table.

If we let each group contain exactly one transaction or all the transactions belong to a single group, then MT(k_1, k_2) reduces to MT(k). Secondly, the group membership in MT(k_1, k_2) is *static* in the sense that a transaction may not migrate to another group during execution unless it restarts.

To partition transactions in the same group, they must share some common properties. We will give two examples to illustrate this. Each example gives a partition rule.

Example 5: Transactions initiated at the same site belong to the group associated with the site.

Example 6: Two groups G_1 and G_2 could be defined based on their read/write sets as shown in Table IV.

	x	y	z	w
G_1	R	W	R,W	
G_2	W	R		R,W

Table IV. Read/write set of groups G_1 and G_2

The read/write set of G_1 and G_2 is formally defined as follows:

$G_1 = \{ T_i \mid \text{read\_set}(T_i) = \{x,z\}, \text{write\_set}(T_i) = \{y,z\} \}$

$G_2 = \{ T_i \mid \text{read\_set}(T_i) = \{y,w\}, \text{write\_set}(T_i) = \{x,w\} \}$

The protocol assures the dependency relationship between G_1 and G_2 to be antisymmetric. This is not only for serializability but sometimes semantically required, especially when there exists a hierarchical relationship between the two groups.

B. The Decentralized Concurrency Controller: DMT(k)

In a distributed system, we can run MT(k) on each site. The timestamp vector of a transaction is stored on a *single* site. A local scheduler may need to access or update a vector on a remote site. To enforce serializability, local schedulers coordinate as follows.

1) *Assigning a globally unique value on the k-th element of a timestamp vector.* Recall that in Algorithm 1 in Section III-A two counters *ucount* and *lcount* are used to set the k-th elements to distinct values. Since each local scheduler has its own counters, two vectors may be assigned by different schedulers the same value on the k-th elements. To distinguish the two vectors, we concatenate the k-th element with the site number of the local scheduler that sets the k-th element. Note that this site should not be confused with the one where the vector is stored. The policy is fair to all the sites, if a) the site number is concatenated as low order bits, and b) the local counters *ucount* and *lcount* used for high order bits are synchronized periodically among all the local schedulers. This is necessary when the local schedulers have unbalanced input load. Then one local counter may increase faster than others. It is profitable that we let *ucount* equal the current value of a local real clock, and *lcount* be the negated value of the real clock. Then, as long as those local real clocks have been synchronized once they need not be synchronized frequently for the succeeding operations. Finally, we must point out that many distributed concurrency algorithms based on time stamps require the clocks to be synchronized. Thus this overhead is common to all those algorithms for performance purposes.

2) *Locking on timestamp vectors.* Each operation of a transaction implies a lock on its timestamp vector. To decide whether to accept the current operation, the associated local scheduler has to access the read timestamp or the write timestamp of a data item, and also lock that timestamp vector. When the operation is done, these locks are released. Because the two vectors and the data item may each reside on a different site, those sites need to exchange messages. Because *deadlock* may occur in this approach, we incorporate an efficient scheme to prevent deadlock. We require that locks be requested in a predefined linear order [10] on the objects. Once a scheduler locks an object, it can only lock another object that is later in the ordering. There are two significant advantages using this approach: a) There is no need to synchronize those lock requests to prevent deadlock. Thus the message overhead tends to be proportionate to the size of the vector. b) A local scheduler may lock at most three or four objects at a time to schedule each operation. Thus the availability of those objects is not restricted.

For either 1) or 2), we have proposed a method without introducing large message overhead, which is desirable in a distributed system. Further, there is an optimization for the protocol DMT(k). That is, if two adjacent operations in a log request locks on different objects, a

scheduler can schedule the next operation while waiting for the messages back from other sites for the earlier operation. On the other hand, if some objects are the same, a scheduler may retain the same lock for the next operation, and only the combined resulting value for the locked object is written back to its home site. In such a way, we save unnecessary messages, overlap the scheduling, and thus reduce the average response time.

VI. DISCUSSION

A. Comparison to Related Work

A concept similar to multidimensional timestamps is the work of Bayer *et al.* [1] which uses dynamic timestamp allocation and validation. Each transaction starts with a large time interval which is shrunk dynamically and *explicitly* as a dependency is found. The order of two disjoint intervals represents the dependency order of the two transactions. We present arguments to claim that our approach has more concrete and complete results.

1) A timestamp vector can also be thought of as a timestamp interval which is shrunk *implicitly* in our protocols. For example, the vector <3, 2, *, *> represents the interval [3200 – 44, 3255]. (For simplicity, we assume that a timestamp element is restricted within the range – 4 to 5.) Note that the left boundary is 3200 – 44 = 3156 instead of 3200, since an element can be negative. If we set a new element, say <3, 2, 1, *>, it represents the interval [3210 – 4, 3215]. It can be seen that this interval shrinks from "both ends". This *implicit* shrinking behavior is significantly different from that in [1] where a time interval shrinks from only "one end" at all times. Another *implicit* shrinking property is that if two intervals need to be separated to represent a dependency, the larger interval (i.e., the vector with fewer assigned elements) will be chosen to shrink (i.e, one element will be set), and the smaller one (i.e., the vector with more assigned elements) is left intact. That is, the shrinking process in our protocols is performed in a fair, balanced way, which tends to have predictable, positive impact on the performance. Further, the vector representation allows a transaction to have an "interval" containing more *intermediate points* (from the hardware point of view) than the word pair representation of an interval in [1]. Thus, as the vector size increases, our protocols can allow an interval to shrink in many more possible ways than [1]. That implies higher degree of concurrency.

2) In [1], a dependency is assumed to have been found, and then one can apply the shrinking scheme for time intervals. The technique to find the dependencies was not addressed in the paper. However, in our protocols, two indices $RT(x)$ and $WT(x)$ are associated with a data item x. So, it is easy to locate the most recent read or write timestamp vectors and record the dependencies due to conflicting operations.

3) To represent a dependency in [1], we may have to choose a number c strictly within the overlapping region of two overlapping intervals, and shrink the two intervals. (For example, to make the interval [2, 61] precede another interval [1, 60], the two intervals are shrunk to [2, c) and [c, 60] respectively, where c is chosen from the overlapping region [2, 60].) It is evident that the choice of c is critical to the performance. However, the criteria to choose the number c was not given in [1]. We should expect that an interval after shrinking should still overlap with as many other intervals as possible. This can allow more dependency relationships in the future. In many cases, intervals may shrink exponentially in terms of the number of operations, and there tend to be fragmentation of intervals as more and more operations occur or restart. This restricts the degree of concurrency. In Section III-D-5, we have presented the rules to optimally represent dependencies in timestamp vectors.

4) If an aborted transaction always restarts with a fixed interval range as in [1], the starvation case identified in Section III-D-4 with a slight modification may also happen. That is, a transaction may be repeatedly aborted without being committed.

In fact, a timestamp vector, though it can be viewed as a kind of "interval", bears useful algebraic properties. It makes the *implicit* shrinking process simple, and allows more concurrency. The concurrency is allowed in a controllable way. That is, more dimensions implies more concurrency.

B. Guidelines to Choose the Timestamp Vector Size

We assume that the composite protocol MT(k^+) is used. The *maximum* vector size is determined based on two conditions: a) To recognize the largest class TO(k^+), based on Theorem 3, vector size (denoted by k) of $2q - 1$ is sufficient where q is the *maximum* number of operations in a single transaction. b) For efficient parallel processing, the timestamp vector size is limited by the number of vector processors available on the machine.

To determine the *optimal* vector size in the average case is theoretically difficult because the problem space is fairly large and the interaction of feature parameters is extremely complicated. However, one may choose an *appropriate* vector size for a specific application case based on the following guidelines: a) If the amount of conflict among transactions is large, most of the

vector elements tend to be set. Then a larger vector size is useful to record/enforce the large amount of dependencies for higher degree of concurrency. b) For efficient storage usage, the vector size is $2q - 1$ where q should be the *expected* number but possibly not the maximum number of operations in a single transaction. c) If most transactions are long-lived transactions in an application, it is desirable to use a larger vector size at the expense of vector processing time. This eliminates the disadvantage in most two-phase-type locking schemes where the availability of data items is restricted if they are locked by long-lived transactions. Also since larger vector size is used, the *abort rate* of transactions will be smaller.

C. Rollback Schemes

The protocol MT(k^+) provides a higher degree of concurrency than the protocol MT(k), but it is possible that more transactions may abort. We propose the following two approaches to reduce the rollback overhead:

1) *Partial rollback.* A transaction may be rollback to an earlier operation where serializability of the log is assured. The timestamp tables may also reset to some state so that only the dependency information at the restart point of the transaction is represented. In this way, the computation results up to the restart point of the transaction are preserved.

2) *Two-phase commit for each write operation.* In the first phase of a transaction, each write produces a temporary copy invisible to all the other transactions. In the commit phase, each write operation is validated by checking/setting the order of timestamp vectors. If *all* the writes of a transaction still preserve the serializability property, updated values are all written to the database. Otherwise, the transaction is aborted. For each read operation, validation of serializability is the same as described in Algorithm 1 in Section III-A. This approach has the following advantages. a) Since no temporary copy is visible to other transactions, the abort of a not-yet-committed transaction does not affect other transactions. b) Once a transaction is committed, it will never be aborted. c) Timestamp vector of an aborted transaction can be pruned from the timestamp table without affecting the other part of the table.

The optimistic concurrency control [13] also uses two-phase commit scheme for the write operations. Our approach in 2) is different from [13] in the following sense. a) Validation for a

read operation is performed at the same time when the read is requested. b) The timestamp of a transaction is dynamically created and partial order oriented. Thus validation of serializability tends to be less restricted.

VII. CONCLUSIONS

We have identified a hierarchy of a new family of concurrency protocols using multidimensional timestamp ordering within the known class DSR. The timestamp vector elements are assigned as the operations of the transaction become clear. Based on our mechanism, dependency information is more precisely represented by a timestamp vector than a single-valued timestamp. A new class $TO(k^+)$ has higher degree of concurrency than any other class $TO(h^+)$ where $h < k$. However, when $h < k \leq 2q - 1$ and q is the maximum number of operations in a single transaction, TO(h) is not a subset of TO(k) because, excluding undefined elements, column h of the timestamp table of MT(h) contains only distinct elements but column h of the table of MT(k) may contain equal elements. Thus dependency information in the table of MT(h) is not fully covered by that in the table of MT(k).

The protocols are generalized for concurrency control in centralized systems, decentralized systems, and the nested transaction model. For efficient implementation, parallel processing mechanism on timestamp vectors, guidelines to choose an appropriate vector size, efficient rollback techniques are also presented.

ACKNOWLEDGMENTS

We thank Professor Henri Tiri of Univ. of Helsinki, Finland, who, after reading an earlier version of the paper, pointed out some related work based on time-intervals [1]. We thank the anonymous referees for their thorough reading and helpful criticism. Due to referees' comments, we study three additional important issues in the final version: 1) the parallel processing mechanism for the timestamp vectors; 2) the construction of composite protocols that allow higher degree of concurrency as the vector size increases; and 3) guidelines to choose an appropriate timestamp vector size. Suggestions by John T. Riedl greatly improved our presentation.

REFERENCES

[1] R. Bayer, K. Elhardt, J. Heigert, A. Reiser, "Dynamic timestamp allocation for transactions in database systems," in *Distributed Databases*, H. J. Schneider, ed., North-Holland, 1982.

[2] P. A. Berstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys 13*, 2(June 1981), 185-221.

[3] P. A. Berstein and N. Goodman, "Multiversion concurrency control–theory and algorithms," *ACM Trans. Database Syst. 8*, 4(Dec. 1983), 465-483.

[4] P. A. Berstein, J. B. Rothnie, N. Goodman, and C. H. Papadimitriou, "The concurrency control mechanism of SDD-1: a system for distributed databases (the fully redundant case)," *IEEE Trans. Softw. Eng. SE-4*, 3(May 1978), 154-168.

[5] P. A. Berstein, D. W. Shipman, and W. S. Wong, "Formal aspects of serializability in database concurrency control," *IEEE Trans. Softw. Eng. SE-5*, 3(May 1979), 203-216.

[6] B. Bhargava, "Performance evaluation of the optimistic concurrency control approach to distributed database systems and its comparison to locking," in *Proc. 3rd IEEE Int. Conf. Distributed Computing Syst.*, Miami, FL, Oct. 1982.

[7] B. Bhargava and P. J. Leu, "Multidimensional timestamp processing," in *Proc. 10th IEEE COMPSAC*, Chicago, IL, Oct. 1986.

[8] B. Bhargava and J. Riedl, "A model for an adaptable concurrency control," *CSD-TR-609*, Dept. of Computer Sciences, Purdue University, West Lafayette, IN, June. 1986.

[9] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistence and predicate locks in a database system," *Commun. ACM 19*, 11(Nov. 1976), 624-633.

[10] J. W. Havender, "Avoiding deadlock in multitasking systems," *IBM Systems Journal 7*, 2(1968), 74-84.

[11] C. Hua and B. Bhargava, "Classes of serializable histories and synchronization algorithms in distributed database systems," in *Proc. 3rd IEEE Int. Conf. Distributed Computing Syst.*, Miami, FL, Oct. 1982.

[12] C. P. Kruskal, L. Rudolph, and M. Snir, "The power of parallel prefix," in *Proc. 10th IEEE Int. Conf. on Parallel Processing*, St. Charles, IL, Aug. 1985.

[13] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst. 6*, 2(June 1981), 213-226.

[14] P. J. Leu and B. Bhargava, "Multidimensional timestamp protocols for concurrency control," in *Proc. 2nd IEEE Int. Conf. Data Engineering*, Los Angeles, CA, Feb. 1986.

[15] J. E. Moss, "Nested transactions and reliable distributed computing," in *Proc. 2nd IEEE Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, PA, July 1982.

[16] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM 26*, 4(Oct 1979), 631-653.

[17] C. H. Papadimitriou, P. A. Berstein, and J. B. Rothnie, "Computational problems related to database concurrency control," in *Proc. Conf. Theoretical Computer Science*, Waterloo, Ont., Canada, Aug. 1977.

[18] C. H. Papadimitriou and P. C. Kanellakis, "On concurrency control by multiple versions," *ACM Trans. Database Syst. 9*, 1(March 1984), 89-99.

[19] D. P. Reed, "Naming and synchronization in a decentralized computer system," Ph.D. thesis, Dept. Elec. Eng. Comput. Sci., MIT, Sept. 1978.

[20] R. H. Thomas, "A majority consensus approach to concurrency control," *ACM Trans. Database Syst. 4*, 2(June 1979), 180-209.

[21] J. D. Ullman, *Principles of Database Systems*, Computer Science Press, Rockville, MD, 1982.