

Hierarchical Database Decomposition - A Technique for Database Concurrency Control

Meichun Hsu
Stuart E. Madnick

Massachusetts Institute of Technology

ABSTRACT

The classical approaches to enforcing serializability are the *two-phase locking* technique and the *timestamp ordering* technique. Either approach requires that a read operation from a transaction be *registered* (in the form of either a read timestamp or a read lock), so that a write operation from a concurrent transaction will not interfere improperly with the read operation. However, setting a lock or leaving a timestamp with a data element is an expensive operation. The purpose of the current research is to seek ways to reduce the overhead of synchronizing certain types of read accesses while achieving the goal of serializability.

To this end, a new technique of concurrency control for database management systems has been proposed. The technique makes use of a hierarchical database decomposition, a procedure which decomposes the entire database into data segments based on the access pattern of the update transactions to be run in the system. A corresponding classification of the update transactions is derived where each transaction class is 'rooted' in one of the data segments. The technique requires a timestamp ordering protocol to be observed for accesses within an update transaction's own root segment, but enables read accesses to other data segments to proceed without ever having to wait or to leave any trace of these accesses, thereby reducing the overhead of concurrency control. An algorithm for handling ad-hoc read-only transactions in this environment is also devised, which does not require read-only transactions to wait or set any read timestamp.

1.0 INTRODUCTION

A generally accepted criterion for correctness of a concurrency control algorithm is the criterion of serializability of transactions. The classical approach to enforcing serializability are the

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

two-phase locking technique and the timestamp ordering technique. Either approach requires that a read operation from a transaction be *registered* in the form of either a read timestamp or a read lock. Setting a lock or leaving a timestamp with a data element is an expensive operation. It not only incurs a write operation in the database (in the form of setting the read lock or writing the timestamp), but also potentially causes unnecessary delays for concurrent transactions.

The purpose of the current research is to seek ways to reduce overhead of synchronizing certain types of read accesses while maintaining serializability. The bases for our technique are transaction analysis and the maintenance of a multi-version database. The transaction analysis decomposes the database into hierarchically related data segments, such that transactions that write into one segment will only read from the same data segment or segments of higher levels. The technique enables read accesses to higher-level data segments to proceed without ever having to wait; it requires no read locks or read timestamps be set for such accesses.

The structure of the paper is as follows. A brief overview of other relevant research is presented in the next section, followed by a review of basic concepts of multi-version consistency in Section 3. Sections 4 to 6 contain a description of our concurrency control technique. Section 4 introduces hierarchical database decomposition. Given such a decomposition, concurrency control algorithms for update transactions and for read-only transactions are presented in Section 5 and 6. Section 7 concludes the paper.

2.0 REVIEW OF RELEVANT RESEARCH

Algorithms for database concurrency control abound in the literature. Most algorithms are considered variations, extensions and/or combinations of the two basic techniques for concurrency control - two-phase locking and time stamp ordering. The two-phase locking algorithm ensures consistency by imposing a partial order on all transactions based on their lock points. (A lock point of a transaction is the point in time when the locking phase of the transaction reaches its peak.) The timestamp ordering algorithm ensures consistency by imposing a partial order on all transactions based on the initiation times of the transactions.

One of the recent developments in concurrency control algorithms centers around the identification of techniques that increase level of concurrency and/or reduce synchronization overhead, while preserving the correctness of the algorithm. One approach to these goals involves the use of a multi-version database. It has been observed that keeping multiple versions of database elements will improve concurrency of the database <Papadimitriou82>. The concept of a timestamp-based multi-version database system was first proposed in <Reed78>. One-previous-version concurrency control methods are discussed in <Bayer80, Garcia-Molina82, Viemont82>, while multiple-previous-version methods are presented in <Stearns81, Chan82>. In particular, Chan's method is based on two-phase locking but allows the read-only transaction to receive special treatment - they do not have to set read locks. Our technique, however, is one which is timestamp based and strives to reduce the need for leaving read timestamps for not just read-only transactions, but update transactions as well.

Another approach to reducing synchronization overhead is conflict analysis <Bernstein80b>. In the research on concurrency control for SDD-1, conflict analysis was proposed to exploit a priori knowledge of the nature of the transactions to be run in the system. The approach reported in the present paper is different from that of SDD-1 because it is not oriented towards distributed databases, and, because of the special structure of applications that our approach exploits, together with the fact that multiple version technique is employed, the protocols are much less restricted. These new protocols are likely to allow for a higher level of concurrency.

3.0 BASIC CONCEPTS OF MULTI-VERSION CONSISTENCY

The following material concerning multi-version consistency is mostly taken from <Papadimitriou82, Bernstein82>, with some notational differences, and is included here for notational purposes.

Definition A schedule of a set of transactions T , denoted as $S(T)$, is a sequence of steps, each of which is denoted as a tuple of the form <transaction id, action, version of a data granule>.

The action can be read (r) or write (w). The version of a data granule is denoted as d^v , where d indicates the data granule and v indicates the version. If the action is write, then the version of the data granule included in the step is created by the transaction. If the action is read, then the transaction reads the version of the data granule indicated in the tuple.

An example of a schedule is < t_1, w, a^1 >, < t_2, r, a^1 >, < t_2, w, b^1 >, < t_3, r, b^1 >.

Definition A version j of a data element d is the predecessor of a version k of d if < t_1, w, d^j > is before < t_2, w, d^k > in $S(T)$ where $t_1, t_2 \in T$, and there exists no $t \in T$ and i such that < t, w, d^i > is between < t_1, w, d^j > and < t_2, w, d^k > in $S(T)$.

Definition A transaction dependency graph of a schedule $S(T)$ is a digraph, denoted as $TG(S(T))$, where the nodes are the transactions in T and the

arcs, representing *direct dependencies* between transactions, exist according to the following rules:

- $t_2 \rightarrow t_1 \in A$ iff
- (1) < t_1, w, d^v > and < t_2, r, d^v > are in $S(T)$ for some d^v , or
 - (2) < t_1, r, d^j > and < t_2, w, d^k > are in $S(T)$ for some d^j, d^k where d^j is the predecessor of d^k .

In other words, the transaction dependency graph represents a relation \rightarrow (depends on) of transactions such that $t_2 \rightarrow t_1$ if t_2 reads a version of a data granule created by t_1 or if t_2 creates a version of a data granule whose predecessor has been read by t_1 .

Definition. Two schedules $S_1(T)$ and $S_2(T)$ of the same set of transaction set T is said to be *equivalent* iff $TG(S_1(T)) = TG(S_2(T))$.

Definition A schedule $S(T)$ is *serializable* if there exists an equivalent schedule $S_s(T)$ where all transactions in $S_s(T)$ are serialized. (i.e., no steps of one transaction are interleaved with steps from another transaction.)

In <Bernstein82> the following theorem has been shown: a schedule $S(T)$ is serializable iff $TG(S(T))$ is acyclic.

4.0 HIERARCHICAL DATABASE DECOMPOSITION

4.1 SOME GRAPHIC-THEORETIC DEFINITIONS

We first briefly introduce the concept of a digraph called a transitive semi-tree. This concept will then be used to describe the desirable database partition to which our concurrency control technique can be applied. Informally, a semi-tree is a digraph such that, if the directions of the arcs in the graph are ignored, the graph appears to be a spanning tree. A transitive semi-tree is a digraph whose transitive reduction is a semi-tree, i.e., it is a semi-tree with an arbitrary number of additional transitively induced arcs.

Definition A *semi-tree* is a digraph such that there exists at most one undirected path between any pair of nodes in the graph. Every arc in a semi-tree is called a *critical arc*.

Definition A digraph G is a *transitive semi-tree* iff its transitive reduction is a semi-tree.

An example of a transitive semi-tree is shown in Figure 1. It can be seen that the definition of a transitive semi-tree is more relaxed than a directed tree, but is more restricted than an acyclic directed graph. The following two properties are associated with the transitive semi-tree.

Property A path in a transitive semi-tree is a critical path iff it is composed of critical arcs alone.

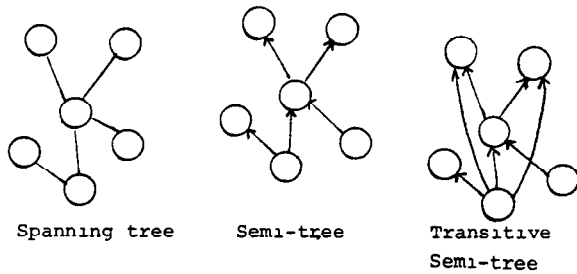


Figure 1. Illustration of a transitive semi-tree.

Property There exists at most one critical path between any pair of nodes in a transitive-semi tree.

4.2 DATABASE PARTITION

We will use the concept of a data hierarchy graph (DHG), constructed by means of transaction analysis, to characterize the relationship between a database partition scheme and database transactions. As will be shown later, the topology of the DHG of a particular database partition scheme will indicate whether or not our concurrency control technique can be applied to that partition scheme. Informally, let a database be partitioned into data segments. A DHG is a digraph with nodes corresponding to the data segments and arcs constructed in such a way that there is an arc from a data segment D_i to another data segment D_j if and only if one can find a potential transaction in the database system that updates data elements in D_i and accesses (i.e., reads or writes) data elements in D_j . In other words, $D_i \rightarrow D_j, i \neq j$, indicates that there exist transactions in the system that would link updates of data elements in D_i to the content of data elements in D_j .

Definition Let T^u be a set of update transactions to be performed on a database D . Let P be a partition of D into data segments D_1, D_2, \dots, D_n . A data hierarchy graph of P w.r.t. T^u is a digraph denoted as $DHG(P, T^u)$ with nodes corresponding to the data segments of P and a set of directed arcs joining these nodes such that, for $i \neq j, D_i \rightarrow D_j$ iff there exist $t \in T^u$ s.t. $w(t) \cap D_i \neq \emptyset$ and $a(t) \cap D_j \neq \emptyset$, where t is a transaction, $w(t)$, $r(t)$ and $a(t)$ the write set, the read set and the access set of transaction t . (The access set $a(t)$ is the union of $r(t)$ and $w(t)$.)

The kind of database partition to which our concurrency control technique can be applied is one such that its data hierarchy graph satisfies the topological requirement that it be a transitive semi-tree.

Definition. A partition P of a database D is *TST-hierarchical* with respect to T^u iff $DHG(P, T^u)$ is a transitive semi-tree.

Property Let p be a TST-hierarchical partition w.r.t. T^u . Then $t \in T^u$ writes in one and only data segment in P .

Proof Suppose t writes in two distinct data segments D_i and D_j , then according to our rule of construction of $DHG(P, T^u)$, $D_i \rightarrow D_j, D_j \rightarrow D_i \in DHG(P, T^u)$, therefore $DHG(P, T^u)$ is no longer a transitive semi-tree, which means that P is not TST-hierarchical w.r.t. T^u , and contradicts the assumption.

Based on the above property, a TST-hierarchical database partition P also defines a transaction classification as follows.

Definition A transaction classification of a database partition which is TST-hierarchical w.r.t. T^u , is a partition of the set T^u of all update transactions into *transaction classes* T_1, T_2, \dots, T_n , such that a transaction $t \in T_i$ iff t writes in data segment D_i .

Therefore a transaction classification partitions the set of update transactions into classes, each of which corresponds to a data segment in the data partition. We define the image of the data hierarchy graph for the transaction classification as follows:

Definition A transaction hierarchy graph $THG(P, T^u)$ of a database partition P , TST-hierarchical w.r.t. T^u , is a digraph where the nodes are transaction classes T_i 's based on transaction classification defined above, and arcs connecting these nodes such that $T_i \rightarrow T_j, i \neq j$ iff $D_i \rightarrow D_j$ exists in the corresponding $DHG(P, T^u)$.

Given definitions of DHG and THG above, we shall denote a *critical path* from i to j in THG or DHG as $CP_{i,j}$. Therefore, $T_i \rightarrow T_k \rightarrow \dots \rightarrow T_j = CP_{i,j}$ iff every arc is a critical arc. In addition, we give the following definition:

Definition We define *higher than* (denoted as \uparrow) as a partial ordering of nodes in a THG or a DHG. Specifically, we say that T_j *higher than* T_i (or $T_j \uparrow T_i$) iff $CP_{i,j}$ exists in the graph.

5.0 SYNCHRONIZING UPDATE TRANSACTIONS

Given a TST-hierarchical database partition, the key to our concurrency control technique is the recognition that, if a transaction t belongs to a class T_i that writes data segment D_i and reads data segment D_j , and D_j is *higher than* D_i in the Data Hierarchy Graph, then this transaction would appear to be a read only transaction so far as D_j is concerned. Therefore when a request to read a data element d in D_j is issued by t , there may exist a proper committed version of d that is *safe* to be given to t without the need of leaving a read timestamp with d . However, the way this proper version is computed must be such that the overall serializability is enforced. In other words, the introduction of transaction dependency of t on t' , where t' is the transaction in class T_j which created the version of d that t is allowed to read, must never induce cycles in the transaction dependency graph as defined in Section 3. To this end, a function called the activity link function is devised to compute versions that cross-class read accesses may be granted, and a theorem which testifies to the correctness of this computation is

presented. Based on this theorem, a concurrency control algorithm is also presented.

Notations

- (1) $I(t)$ = the initiation time of a transaction t .
- (2) $C(t)$ = the commit time of a transaction t
- (3) $TS(d^v)$ = the initiation time of the transaction that creates the version v of a data granule d , i.e., the write timestamp of d^v . (A data granule is the smallest unit that concerns the concurrency control component of the database system, and is the smallest unit of accesses so far as concurrency control is concerned.)

5.1 THE ACTIVITY LINK FUNCTION

The following definitions and properties apply to a database with a partition P which is TST-hierarchical w.r.t T^u and has a corresponding transaction classification.

Definition A function I_1^{old} defined for a transaction class T_1 is a function which maps a time m to another time m' such that $m' = I_1^{old}(m)$, where m' is the initiation time of the oldest active (i.e., uncommitted and un-aborted) transaction in the transaction class T_1 at time m . Formally,

$$I_1^{old}(m) = \begin{cases} m & \text{if there exists no } t \in T_1 \text{ active at time } m, \\ \text{Min} (I(t)) & \text{otherwise, where } t \in T_1, \\ & I(t) \leq m \text{ and } C(t) > m. \end{cases}$$

Definition Let the *activity link function* A_1^j be a function defined for a pair of transaction classes T_1 and T_j , where T_1 and T_j are transaction classes such that $T_j \uparrow T_1$. A_1^j recursively maps a time m to another time as follows.

$$A_1^j(m) = \begin{cases} I_j^{old}(m) & \text{if } T_1 \rightarrow T_j = CP_1^j \\ A_k^j(A_1^k(m)) & \text{otherwise, where } T_1 \rightarrow T_k \rightarrow \dots \rightarrow T_j = CP_1^j. \end{cases}$$

That is, the function A maps a time m for a transaction from class T_1 to the initiation time, $A_1^j(m)$, of successively (i.e., along the critical path of THG) the oldest active transaction in the class T_j . For example, if the critical path between T_1 and T_j is $T_1 \rightarrow T_k \rightarrow T_j$, then $A_1^j(m) = I_j^{old}(I_k^{old}(m))$. This is exemplified in Figure 2.

5.2 CONCURRENCY CONTROL ALGORITHM FOR UPDATE TRANSACTIONS

Based on the definitions given above, we describe in this subsection the concurrency control algorithm for update transactions under the hierarchical decomposition approach, and prove its correctness. For the purpose of concurrency control, we assume that every data segment is controlled by a *segment controller* which supervises accesses to data granules within that segment.

$I_1^{old}(m)$ = init time of the oldest active trans in class T_1 at time m

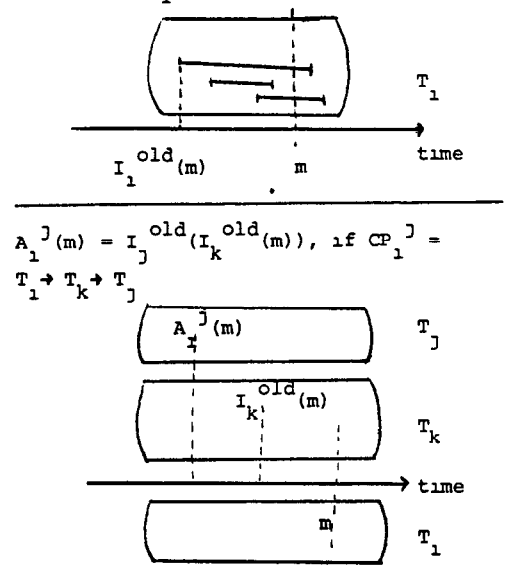


Figure 2. Graphical representation of the A function.

Concurrency control algorithm for update transactions

For every database access request from an update transaction $t \in T_1$ for a data granule $d \in D_j$, the following protocol is observed:

Protocol A

If $i \neq j$, then the segment controller of D_j provides the version d^o of d such that $TS(d^o) = \text{Max}(TS(d^v))$ for all v such that $TS(d^v) < A_1^j(I(t))$.

(Note that no trace of this access needs to be registered in any form for the purpose of concurrency control by the segment controller.)

Protocol B

If $i = j$, then use the *basic timestamp ordering protocol* <Bernstein80> or the *multi-version timestamp ordering protocol* <Reed78>.

5.3 PROOF OF CORRECTNESS

To show that the above algorithm is correct, one must show that serializability is enforced. In order to do this, we define a relation \Rightarrow between a pair of transactions and show that the above algorithm allows a transaction t_1 to directly depend on a transaction t_2 only if $t_1 \Rightarrow t_2$. (Direct dependency is defined in Section 3) We then show that properties of the relation \Rightarrow lead to Theorem 1, which concludes that the above algorithm preserves serializability.

Definition A relation *topologically follows* (denoted as \Rightarrow) is defined for a pair of trans-

actions t_1, t_2 , where $t_1 \in T_1, t_2 \in T_2, T_1$ and T_2 are connected by a critical path in THG, 1 and 2 not necessarily distinct. We say that t_1 topologically follows t_2 (or $t_1 \Rightarrow t_2$) iff

- (1) if $T_1 = T_2$ then $I(t_1) > I(t_2)$.
- (2) If $T_1 \uparrow T_2$ then $I(t_1) \geq A_j^{-1}(I(t_2))$.
- (3) If $T_2 \uparrow T_1$ then $I(t_2) < A_i^{-1}(I(t_1))$.

Intuitively, \Rightarrow is a relation between transactions based on both the timing of the transactions and the hierarchical levels in the THG of the transaction classes that the transactions belong to. To be more specific, ' $t_1 \Rightarrow t_2$ ' always means that t_1 is 'later' than t_2 . However, this 'later' is not only based on the initiation times of the two transactions involved, but also on the relative levels of the transaction classes to which t_1 and t_2 belong: Given a fixed t_2 , the lower the level of t_1 , the later t_1 's initiation time has to be in order for $t_1 \Rightarrow t_2$ to hold. Clearly, \Rightarrow is defined only between transactions that belong to classes that are on a critical path in THG, because otherwise the A function is undefined. This relation is exemplified in Figure 3. Two interesting properties concerning the relation \Rightarrow are presented below:

Property 1.1 The relation \Rightarrow is anti-symmetric. (This directly follows from the definition of the relation.)

Property 1.2 (The property of transitivity). The relation \Rightarrow is critical-path transitive, i.e., if there exists $t_1 \in T_1, t_2 \in T_k, t_3 \in T_2$, such

that $t_1 \Rightarrow t_2, t_2 \Rightarrow t_3$ and T_1, T_k and T_2 are on a critical path in THG, then $t_1 \Rightarrow t_3$.

Proof (See Appendix)

We now define the following synchronization rule and show that our concurrency control algorithm enforces this rule.

Definition We say that the *partition synchronization rule* (abbreviated as PSR) is enforced in a schedule $S(T^U)$ if, for any $t_1, t_2 \in T^U, t_1 \rightarrow t_2 \in TG(S(T^U))$ implies that $t_1 \Rightarrow t_2$.

A concurrency control algorithm enforces the partition synchronization rule if it allows direct dependencies to occur between transactions t_1 and t_2 only if $t_1 \Rightarrow t_2$. This is translated into the following three cases:

- (1) If t_1 and t_2 are in the same transaction class, the algorithm must allow t_1 to read a version v of a data granule d created by t_2 , or to create a new version of a data granule d whose latest version d^v was created by t_2 , only if t_2 has an initiation time that is less than that of t_1 . (i.e., only if $TS(d^v) < I(t_1)$.)

Protocol B of our algorithm satisfies this requirement.

- (2) If t_1 belongs to a class T_1 of a lower level while t_2 belongs to a class T_2 of a higher level, then the algorithm must allow t_1 to read d^v created by t_2 only if t_2 has an initiation time less than $A_j^{-1}(I(t_1))$. (i.e., only if $TS(d^v) < A_j^{-1}(I(t_1))$.)

Protocol A of our algorithm satisfies this requirement.

- (3) If t_1 belongs to a class T_1 of a higher level while t_2 belongs to a class T_2 of a lower level, then the algorithm must allow t_1 to create, at time m , a new version of a data granule whose predecessor d^v has been read by t_2 , only if t_1 has an initiation time greater than or equal to $A_j^{-1}(I(t_2))$.

This, however, is always true because, by the very fact that t_1 is active at time m and $I(t_2) < m$, and that $A_j^{-1}(I(t_2))$ yields a time value which is definitely smaller than the initiation time of the oldest active transaction in class T_1 at time m , $A_j^{-1}(I(t_2))$ must be less than $I(t_1)$.

Therefore we conclude that our algorithm enforces PSR. What is left to do in proving the correctness of our algorithm is to show that a schedule that enforces PSR is also correct. The following theorem therefore completes our proof.

Theorem 1 Let $TG(S(T^U))$ be a transaction dependency graph of a set of update transactions T^U run on a database with a TST-hierarchical partition P , and the schedule S observes the partition synchronization rule with respect to the transaction classification corresponding to P , then $TG(S(T^U))$ has no cycles.

Proof (See Appendix)

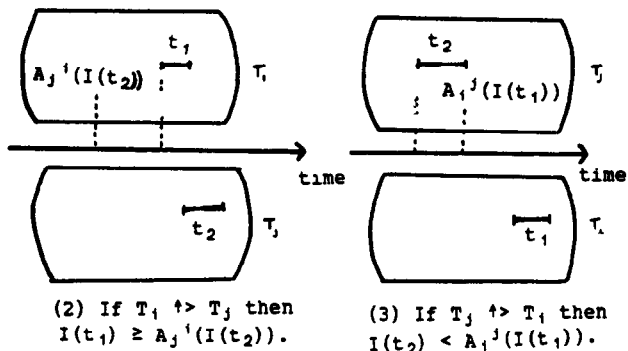
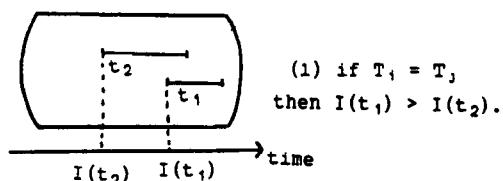


Figure 3. Graphical representation of the relation $t_1 \Rightarrow t_2$.

6.0 SYNCHRONIZING READ-ONLY TRANSACTIONS

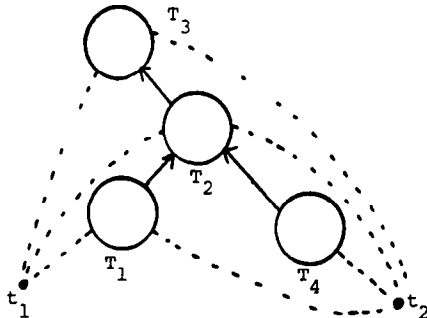
What has been discussed is the algorithm for controlling concurrent update transactions. Now we turn to the read-only transactions.

For a read-only transaction t that reads from data segments that lie on one critical path $CP_{i,j}$ of the DHG, the protocol that should be observed is the same as that observed by the update transactions in a class immediately below the lowest class of the critical path $CP_{i,j}$ in THG, namely, a class right below class T_i . (If there exists no class below T_i in THG, then a fictitious class can be created to 'host' this read-only transaction.) Therefore read-only transactions will have to obey protocol A alone and will not cause any read timestamp or read lock to be generated. This is graphically presented by transaction t_1 in Figure 4

What we are concerned with here are those read-only transactions that read from any combination of data segments that do not lie on a critical path in DHG, as illustrated by transaction t_2 in Figure 4. To handle these transactions, we first introduce the *extended activity link function* in the following subsection.

6.1 THE EXTENDED ACTIVITY LINK FUNCTION

In the previous section we have introduced the activity link function which centers around the linkage between transactions in classes that are on a critical path in the transaction hierarchy graph. The extended function, on the other hand, specifies how transactions in a transaction class are linked



- t reads from class T_i
(t is a read-only transaction)

Figure 4. Read-only transactions that read from one critical path.

to transactions in another transaction class when there is not necessarily any critical path that connects the two. This function is used to provide a way of computing a *consistent database state* that can be accessed by a read only transaction that reads from any combination of data segments in the database.

We will first introduce the functions C_i^{late} and B_j^i that can be considered conceptually the *inverse* of functions I_i^{old} and A_i^j . Then two properties of the relationship between the functions A_i^j and B_j^i are derived. The extended activity link function E_i^j is then defined in terms of functions A and B, and its usefulness is indicated in a lemma that follows. The existence and derivation of a consistent database state is given in theorem 2, which makes use of the extended activity link function.

Definition Let $C_i^{late} : m \rightarrow m'$ be a function which maps a time m to another m' where T_i is a transaction class and $C_i^{late}(m)$ is determined as follows.

$$C_i^{late}(m) = \begin{cases} m & \text{if there exists no } t \in T_i \\ & \text{active at time } m, \\ \text{Max } (C(t)) & \text{otherwise, where } t \in \\ & T_i, I(t) \leq m \text{ and } C(t) > m. \end{cases}$$

That is, $C_i^{late}(m)$ is the *latest* commit time of all transactions in class T_i that started before or at time m . However, to make $C_i^{late}(m)$ computable, all such transactions must have committed at the time of computation of $C_i^{late}(m)$. We give the following definition concerning the *computability* of $C_i^{late}(m)$.

Definition $C_i^{late}(m)$ is *computable at time* m^0 iff there exists no transaction started before or at time m that is still active at time m^0 .

Now we introduce a function which is conceptually the inverse of the function A. While the A function maps a time in a lower level class to the initiation time of some transaction in a higher level class, the B function maps a time in a higher level class to the commit time of some transaction in a lower level class:

Definition The *Backward activity link function*, defined for a pair of transaction classes T_i and T_j , where $T_j \uparrow T_i$, denoted as $B_j^i(m)$, is a function which maps a time value m to another such that

$$B_j^i(m) = \begin{cases} C_i^{late}(m) & \text{if } T_i \rightarrow T_j = CP_{i,j} \\ B_k^i(B_j^k(m)) & \text{otherwise,} \\ & \text{where } T_i \rightarrow \dots \rightarrow T_k \rightarrow T_j = CP_{i,j}. \end{cases}$$

The following two properties bind the functions A and B together and formally describe how they are the inverse of each other.

Property 2.1. $A_i^j(B_j^i(m)) \geq m$, where $T_i \rightarrow \dots \rightarrow T_j = CP_{i,j}$ in the transaction hierarchy graph.

Proof (See Appendix)

Property 2.2. For every positive ϵ , $A_i^j(B_j^i(m) - \epsilon) < m$, where $T_i \rightarrow \dots \rightarrow T_j = CP_{i,j}$ in the transaction hierarchy graph.

Proof (See Appendix)

Definition An *undirected critical path*, denoted as $UCP_{i,j}$, is an ordered set of *distinct* indices of transaction classes in THG such that $UCP_{i,j} = \langle i, i_1, i_2, \dots, i_n, j \rangle$ where for any two indices h, k adjacent in the set, either $T_h \rightarrow T_k$ or $T_k \rightarrow T_h$ is a critical arc in THG.

It is obvious that for a TST-hierarchical partition there exists one and only one UCP in THG between any pair of transaction classes. While the activity link function A is defined for any pair of transaction classes that lie on a critical path, the extended activity link function, using the concept of UCP, is defined for any pair of transaction classes.

Definition The extended activity link function defined for a pair of transaction classes T_i and T_j , denoted as $E_i^j(m)$, is a function which maps a time value m to another such that

$$E_i^j(m) = \begin{cases} m & \text{if } i = j, \\ C_i^{\text{late}}(m) & \text{if } i \neq j \text{ and } T_j \rightarrow T_i \text{ is a} \\ & \text{critical arc in THG,} \\ I_j^{\text{old}}(m - \text{adj}(m, i)) & \text{if } i \neq j \text{ and } T_i \rightarrow \\ & T_j \text{ is a critical arc in THG,} \\ E_k^j(E_i^k(m)) & \text{otherwise, where} \\ & \langle i, k, \dots, j \rangle = \text{UCP}_{i,j}^c \end{cases}$$

where the value of $\text{adj}(m, i)$ is defined as follows

$$\text{adj}(m, i) = \begin{cases} 0 & \text{if there exists } t \in T_i \text{ such that} \\ & I(t) = m \text{ or there exists no} \\ & \text{transaction in } T_i \text{ active at } m \\ m - I(t') & \text{otherwise, where } t' \text{ is} \\ & \text{such that } C_i^{\text{late}}(m) = C(t'). \end{cases}$$

The following lemma illustrates the usefulness of the extended activity link function.

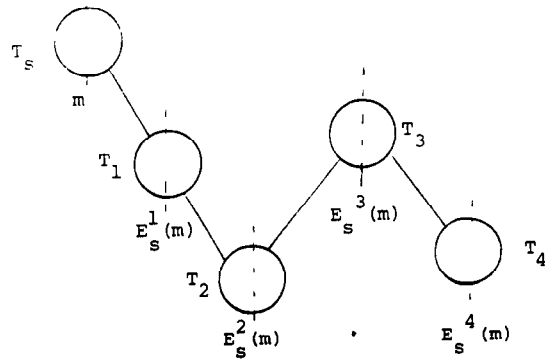
Lemma 2.1 Let T_k , T_i and T_j be transaction classes in a THG of a TST-hierarchical database partition, and T_i and T_j are on one critical path. Then for any time value m and $t_1 \in T_i$, $t_2 \in T_j$, if $I(t_1) < E_k^i(m)$ and $I(t_2) \geq E_k^j(m)$ then there exists no $t_1 \rightarrow t_2$ in the transaction dependency graph $TG(S(T^u))$ where the schedule S enforces the PSR's.

Proof (See Appendix)

Intuitively, the E function provides a way of computing a *time wall* for all transaction classes in the database system across which no direct dependency from the 'older side' of the wall to the 'newer side' of the wall can occur. A time wall $TW(m, s)$ is the set of all times $E_s^i(m)$ where m is a time, D_s is a chosen data segment, and D_i is any data segment. This concept is graphically presented in Figure 5. The significance of this concept is that if a read-only transaction reads the latest versions of data granules of data segment D_i which are right before the time indicated by the time wall component $E_s^i(m)$ of certain time wall $TW(m, s)$, then it is accessing a consistent database state and will in no way induce cycles into the transaction dependency graph. This discussion is formally presented in the following theorem.

Theorem 2 If the schedule S enforces the PSR on T^u , and for every $d \in D_i$ that a read-only transaction t_R reads, S allows it to read the version d^o such that

$$TS(d^o) = \text{Max} (TS(d^v)) \text{ where } TS(d^v) < E_s^i(m),$$



A time wall $TW(s, m)$ is such that no direct dependencies occur between a transaction on the left side of the dotted line ($i \in T_i$, $I(t) < E_s^i(m)$) and that on the right side of the dotted line ($i \in T_i$, $I(t) > E_s^j(m)$).

Figure 5. The E function used as a time wall.

for some time m and some transaction class index s , then $TG(S(T^u \cup T_R))$ has no cycle.

Proof (See Appendix)

In other words, if a read-only transaction reads the latest versions of data granules of data segment D_i which are right before the time indicated by the time wall component $E_s^i(m)$ of certain time wall $TW(m, s)$, then it is accessing a consistent database state and will not induce cycles into the transaction dependency graph.

6.2 CONCURRENCY CONTROL PROTOCOL FOR READ-ONLY TRANSACTIONS

Making use of Theorem 2, a read-only transaction t that reads from data segments that do not lie on one critical path in DHG should be given versions that are the latest before certain time wall. However, to compute the time wall the system has to determine the starting transaction class T_s and a starting time value m . While the choice can be arbitrary, it is theoretically desirable that the following criteria are met:

- (1) $E_s^i(m)$ (for all T_i in the THG) is computable at $I(t)$, the initiation time of the read-only transaction.
- (2) There exists no $m' > m$ such that $E_s^i(m')$ is computable at $I(t)$ for all T_i in the THG.

The first criterion stipulates that m should be *small enough* so that all $E_s^i(m)$ is computable at $I(t)$, therefore t potentially does not have to wait until a later time to access from certain segment. (If some $E_s^j(m)$ is not computable at $I(t)$, t would have to wait till a later time when it is computable before accessing data from data segment D_j .) The second criterion strives to achieve reading of the *newest possible* database state.

A compromise is struck here in devising our protocol for read-only transactions. First, to save computation time, a new time wall is computed by the system at certain intervals and the new time wall is 'released' to all read-only transactions

that start before the next *version* of the time wall is released by the system (That is, there is no need to compute a time wall for every read-only transaction.) In computing the next version of the time wall, the system can choose arbitrarily a starting class T_s which is of one of the lowest levels and choose m to be the current time. If it encounters any C_i^{state} function that it cannot compute, it waits until it becomes computable. Eventually enough time will elapse such that $E_s^{-1}(m)$ becomes computable for all T_i 's. Then a newly constructed time wall is released.

Let the release time of a time wall $TW(m,s)$ be denoted as $RT(TW(m,s))$. Now we provide the formal definition of the read-only transaction synchronization protocol.

Concurrency Control Algorithm for Read-Only Transaction

For every database read request from a read-only transaction t for a data granule d , the following protocol is observed:

Protocol C

Let $d \in D_i$. The segment controller of D_i provides the version d^v of d such that

$$TS(d^v) = \text{Max}(TS(d^v)) \text{ for all } v \text{ such that}$$

$$TS(d^v) < E_s^{-1}(m)$$

where $RT(TW(m,s)) = \text{Max}(RT(TW))$ for all TW such that $RT(TW) < I(t)$.

7.0 SUMMARY

A new technique of concurrency control for database management systems has been proposed. The technique makes use of a hierarchical database decomposition, a procedure which decomposes the entire database into data segments based on the access pattern of the update transactions to be run in the system. A corresponding classification of the update transactions is derived where each transaction class is 'rooted' in one of the data segments.

The technique requires a timestamp ordering protocol be observed for accesses within an update transaction's own root segment, but enables read accesses to other data segments to proceed without ever having to wait or to leave any trace of these accesses, thereby reducing the overhead of concurrency control. An algorithm for handling ad-hoc read-only transactions in this environment is also devised, which does not require read-only transactions to wait or set any read timestamp. The proof of correctness of these algorithms in terms of their preservation of serializability is provided through a set of eight properties, three lemmas and two theorems. A comparison of the SDD-1 approach, the multi-version two-phase locking approach (MV2PL) and the Hierarchical Database Decomposition (HDD) approach proposed here is given in Figure 6.

Acknowledgements The authors would like to thank W. Frank of Sloan School, M.I.T. and A. Chan of Computer Corporation of America for their helpful suggestions. Work reported herein has been supported, in part, by the Naval Electronic Systems Command through contract N0039-81-c-0663.

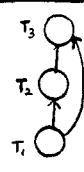
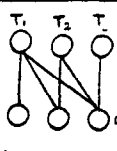

	HDD	SDD-1	MV2PL
Trans Analysis	Hierarchical	General	None
Representation			
Inter-Class Synch	Never reject or block a read req	May cause read req to be rejected or blocked	N A
Intra-Class Synch	Timestamp Ordering	Serialized Pipelining	2-phase locking
Read-only Trans	Similar to Inter-class Synch	No special handling	Never block or reject

Figure 6 A comparison of the HDD, SDD-1 and MV2PL approaches.

BIBLIOGRAPHY

- Bayer80 Bayer, R., Heller, H., and Reiser, A. Parallelism and recovery in database systems. ACM Trans Database Syst 5, 2 (June 1980)
- Bernstein80 Bernstein, P.A., and Goodman, N. Fundamental algorithms for concurrency control in distributed database systems. Computer Corporation of America, TR CCA-8-05 (Feb 1980)
- Bernstein80b Bernstein, P.A., Shipman, D.W., and Rothnie, J.B. Concurrency control in a System for Distributed Databases (SDD-1). ACM Trans. on Database Syst, 5, 1 (March 1980)
- Bernstein82. Bernstein, P.A., Goodman, N., and Hadzilacous. Distributed database control and allocation. CCA Semi-annual technical report (July 1982)
- Chan82: Chan, A. et. al. The implementation of an integrated concurrency control and recovery scheme. Technical report CCA-82-01, Computer Corporation of America, Cambridge, Mass (1982)
- Garcia-Molina82. Garcia-Molina, H and Wiederhold, G. Read-only transactions in a distributed database. ACM Trans Database Syst 7, 2 (June 1982)
- Eswaran76: Eswaran, K P, Gray, J N., Lorie, R.A., and Traiger, I L. The notions of consistency and predicate locks in a database systems. Comm. ACM 19, 11 (Nov. 1976), 624-634
- Gray76 Gray, J.N., Lorie, R.A., Putzolu, G.R., and Traiger, I L. Granularity of locks and degrees of consistency in a shared data base. In Modelling in Data Base Management Systems, G.M. Nijssen. (ed) North Holland Publishing Company (1976)

Hsu82 Hsu, M The Hierarchical database decomposition approach to concurrency control. INFOPLEX Tech. Report No 12, Center for Information Systems Research, M.I.T., Cambridge, MA. (Dec 1982)

Papadimitriou79 Papadimitriou, C.H. The serializability of concurrent database updates. Journal of ACM 26, 4 (Oct 1979)

Papadimitriou82 Papadimitriou, C.H. and Kanellakis, P.C. On concurrency control by multiple versions. Proc. 1982 ACM SIGACT-SIGMOD Symp. on Principles of Database Syst. (March 1982)

Reed78: Reed, D.P. Naming and synchronization in a decentralized computer system. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass. (September 1978)

Stearns81: Stearns, R. and Rosenkrantz, D. Distributed database concurrency control using before-values. ACM SIGMOD Conference Proceedings (1981)

Viemont82: Viemont, Y.H. and Gardarin, G.J. A distributed concurrency control algorithm based on transaction commit ordering. Proceedings of Fault Tolerance Computer Systems, Santa Monica, Cal. (June 1982)

a $CD(t_1, t_2)$ such that either or both of the following are true:

1. There exists $t_3 \in T_k$ such that $t_1 \rightarrow t_3 \in CD(t_1, t_2)$ and T_i, T_j and T_k are *not* on one critical path;
2. There exists $t_4 \in T_1$ such that $t_4 \rightarrow t_2 \in CD(t_1, t_2)$ and T_i, T_j and T_1 are *not* on one critical path.

Property If $BCD(t_1, t_4)$, where $t_1 \in T_i$ and $t_4 \in T_j$, then there exist $t_2 \in T_k$ and $t_3 \in T_1$, t_2, t_3 not necessarily distinct, such that $CD(t_1, t_2) \subset CD(t_1, t_4)$, $CD(t_2, t_3) \subset CD(t_1, t_4)$, $CD(t_3, t_4) \subset CD(t_1, t_4)$ and T_i, T_j, T_k and T_1 are on one critical path in THG. (This directly follows from the fact that THG is a transitive semi-tree.)

Lemma 1 If there exists a critical path dependency $CD(t_1, t_2)$ in a transaction dependency graph $TG(S(T))$ where the schedule S enforces the partition synchronization rule, then $t_1 \Rightarrow t_2$.

Proof Let l be the length (in number of arcs, i.e., direct dependencies) of a critical path dependency. Then l has a total order and is bounded from below by 1. By way of complete mathematical induction, to prove that if $CD(t_1, t_2)$ then $t_1 \Rightarrow t_2$, we have to show the following

- (1) If $l(CD(t_1, t_2)) = 1$ then $t_1 \Rightarrow t_2$.
- (2) If $l(CD(t_1, t_2)) = g$ and if $t_a \Rightarrow t_b$ for all t_a, t_b s.t. there exists $CD(t_a, t_b)$ and $l(CD(t_a, t_b)) < g$, then $t_1 \Rightarrow t_2$.

Now we prove the above two statements.

- (1) In this case, $CD(t_1, t_2) = t_1 \rightarrow t_2$. By property 1.3 we have $t_1 \Rightarrow t_2$
- (2) To prove the second statement, let $t_3 \in T_k$ and $t_4 \in T_1$ be such that $t_1 \rightarrow t_3 \in CD(t_1, t_2)$, $t_4 \rightarrow t_2 \in CD(t_1, t_2)$, and a path, denoted as $Path(t_3, t_4)$, from t_3 to t_4 such that $Path(t_3, t_4) \subset CD(t_1, t_2)$. Also let $t_1 \in T_i$ and $t_2 \in T_j$. Consider the following two cases.

(2.1) If $CD(t_1, t_2)$ is *not* a BCD, then $Path(t_3, t_4)$ is a $CD(t_3, t_4)$. Since $l(CD(t_1, t_2)) < g$ therefore $t_1 \Rightarrow t_2$. And by the definition of CD , T_i, T_j, T_k and T_1 must be on one critical path of THG. Therefore we have $t_1 \rightarrow t_3, t_4 \rightarrow t_2$ and $t_3 \Rightarrow t_4$. By property 1.2 (i.e., the property of critical path transitivity) we have $t_1 \Rightarrow t_2$.

(2.2) If $CD(t_1, t_2)$ is a BCD, then by the property above of a BCD we have that there exist $t_5 \in T_m$ and $t_6 \in T_n$ such that $CD(t_1, t_5) \subset CD(t_1, t_2)$, $CD(t_5, t_6) \subset CD(t_1, t_2)$, and $CD(t_6, t_2) \subset CD(t_1, t_2)$ where T_m, T_n, T_i and T_j are on one critical path of THG. Since $l(CD(t_1, t_5)) < g$, therefore $t_1 \Rightarrow t_5$. Similarly, $t_6 \Rightarrow t_2$ and $t_5 \Rightarrow t_6$. By property 1.2 we conclude $t_1 \Rightarrow t_2$ *Q.E.D.*

Theorem 1

Proof Suppose there exists a cycle. Then the cycle involves at least two transactions t_1 and t_2 that belong to transactions that are on one critical path. This means that there exist $CD(t_1, t_2)$ and $CD(t_2, t_1)$. By the above lemma, $CD(t_1, t_2)$ implies $t_1 \Rightarrow t_2$ and $CD(t_2, t_1)$ implies $t_2 \Rightarrow t_1$. However, \Rightarrow is anti-symmetric (by property 1.1). Therefore $t_1 \Rightarrow t_2$ and $t_2 \Rightarrow t_1$ cannot be true at the same time. Therefore there can be no cycle in this transaction dependency graph. *Q.E.D.*

APPENDIX

(Refer to Hsu82 for complete proofs whose abbreviated versions are given here.)

(1) PROOF OF PROPERTY 1.2

Property 1.2 (The property of transitivity)

Proof (Sketch) We consider the following 5 groups of cases: (1) $T_i = T_k = T_j$, (2) $T_i = T_k \neq T_j$, (3) $T_i \neq T_k = T_j$, (4) $T_i = T_j \neq T_k$, and (5) $T_i \neq T_k \neq T_j, T_i \neq T_j$. In each group, we permute the order of levels among the distinct transaction classes to arrive at a total 13 cases. These cases exhaust all the possible situations that govern t_1, t_2 and t_3 and for every situation, transitivity is shown to hold. Therefore we conclude that \Rightarrow is critical-path transitive.

(2) PROOF OF THEOREM 1

In order to prove Theorem 1, we first give the following two definitions and a lemma about the transaction dependency graph.

Definition A *critical path dependency*, between two distinct transactions $t_1 \in T_i$ and $t_2 \in T_j$, denoted as $CD(t_1, t_2)$, is a cycle-free dependency path from t_1 to t_2 in $TG(S(T^u))$ and T_i and T_j are on one critical path in THG, i and j not necessarily distinct.

Definition. A *boundary critical path dependency* in $TG(S(T^u))$ between two transactions $t_1 \in T_i$ and $t_2 \in T_j$, where $t_1 \neq t_2$, denoted as $BCD(t_1, t_2)$, is

(3) PROOF OF PROPERTY 2.1

Property 2 1

Proof (Sketch) Let $CP_1^j = T_1 \rightarrow T_{11} \rightarrow \dots \rightarrow T_{1(n-1)} \rightarrow T_{1n} \rightarrow T_j$. Replace the expression $B_j^i(m)$ in $A_i^j(B_j^i(m))$ by $B_j^i(m) = C_{11}(\dots(C_{1n}(C_j(m))))$. (C_j is an abbreviated expression for C_j^{late}). Substitute recursively the terms involved in $B_j^i(m)$ by time values while keeping track of the values substituted. Eventually we transform the expression $A_i^j(B_j^i(m))$ into $A_i^j(m_{11})$ where m_{11} is a proper time value derived from transaction activities in class T_{11} . Then start expanding the expression $A_i^j(m_{11})$ and substitute the terms involved in it with appropriate time values. Compare these time values with those used in substituting terms in $B_j^i(m)$ previously. Eventually $A_i^j(m_{11})$ is reduced to a simple time value to be compared with the original value m , and the inequality will be found to hold.

(4) PROOF OF PROPERTY 2.2

Property 2 2

Proof (Sketch)

The proof is similar to that of property 2.1. It involves carefully spelling out the terms involved in the expression and compare the values used during substitution.

(5) PROOF OF LEMMA 2.1

Proof (Sketch) Let T_{k1} be the class such that $k1$ is the first index in UCP_{k1} where T_{k1} and T_1, T_j are on one critical path. (k and $k1$ are not necessarily distinct.) Then $k1$ will also be the first such index in UCP_{k1} , and the subset of the ordered set UCP_{k1} up to $k1$ and that of UCP_k up to $k1$ are equivalent. (This is because between any pair of nodes there is one and only one UCP.) Consider the following four groups of cases. (1) $i = j \neq k1$ or $i = j = k1$, (2) $i = k1 \neq j$, (3) $j = k1 \neq i$, and (4) $i \neq j \neq k1$. For each of the group above we permute the level of the distinct classes and for a total of 11 cases we have shown that it is impossible to have $t_1 \rightarrow t_2$. (The proof makes extensive use of properties 2.1 and 2.2 concerning the relationship between the functions A and B which are used to construct the function E) Therefore we prove that there exists no $t_1 \rightarrow t_2$

(6) PROOF OF THEOREM 2

In order to prove Theorem 2, we first give the following definitions and a lemma (Lemma 2.2.)

Definition A consistent transaction set with respect to a schedule $S(T)$, abbreviated as a CS w.r.t. $S(T)$, is a set of transactions $T^{CS} \subset T$ such that if $t \in T^{CS}$ and if there exists $t_1 \in T$ such that $t \rightarrow \dots \rightarrow t_1 \in TG(S(T))$, (i.e., if t depends on t_1 in the transitive closure of \rightarrow), then $t_1 \in T^{CS}$.

Property 2 3 (The Property of a consistent transaction set) Partition T^U into T^{U1} and T^{U2} . Then T^{U1} is a consistent transaction set w.r.t $S(T)$ iff for any two transactions t_1, t_2 , such that $t_1 \in T^{U1}$ and $t_2 \in T^{U2}$, there exists no $t_1 \rightarrow t_2$ in the transaction dependency graph $TG(S(T))$. (Proof omitted)

Definition Given a time value m and a starting transaction class T_s , a designated consistent transaction set, denoted as $T^{CS}(m,s)$, is a consistent transaction set such that for all $t \in T_s, t \in T^{CS}(m,s)$ iff $I(t) < m$.

Lemma 2 2 partition T^U into T^{U1} and T^{U2} . Then T^{U1} is the designated consistent transaction set $T^{CS}(m,s)$ w.r.t. $S(T^U)$, where the schedule S enforces the PSR, if T^{U1} contains, for all i , all and only transactions t such that $I(t) < E_s^i(m)$ where $t \in T_i$.

Proof Construct a time wall $TW(m,s)$. Then by the previous lemma (Lemma 2.1) we know that for any j, k , if $t_1 \in T_j$ and $I(t_1) < E_s^j(m)$, and $t_2 \in T_k$ and $I(t_2) \geq E_s^k(m)$ then there exists no $t_1 \rightarrow t_2$. Therefore by Property 2.3 above we know that T^{U1} is a consistent transaction set if it contains for all i only transactions t such that $I(t) < E_s^i(m)$ where $t \in T_i$. And since $E_s^s(m) = m$, we have $I(t_1) < m$ if $t_1 \in T_s$. Therefore T^{U1} must be the designated consistent transaction set $T^{CS}(m,s)$. Q E D

Corollary Given a time value m and a starting transaction class T_s , there exists a designated consistent transaction set $T^{CS}(m,s)$.

Theorem 2

Proof Partition T^U into T^{U1} and T^{U2} such that for all $t \in T_i$, for all $i, t \in T^{U1}$ iff $I(t) < E_s^i(m)$. Then it is clear that dependencies induced by t_R must be arcs that go from t_R to transactions in T^{U1} , and arcs from transactions in T^{U2} to t_R . By Lemma 2.2, there exist no dependencies from transactions in T^{U1} to those in T^{U2} . Therefore arcs introduced by t_R will not introduce any cycle into the original $TG(S(T^U))$. Since $TG(S(T^U))$ has no cycle, therefore $TG(S(T^U \cup t_R))$ has no cycle Q E D