

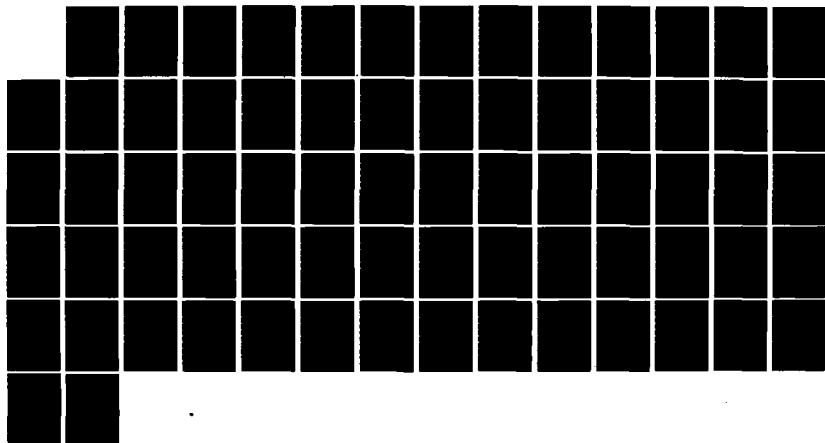
AD-A134 356

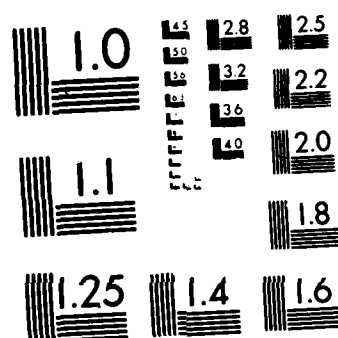
HIERARCHICAL DATABASE DECOMPOSITION: A TECHNIQUE FOR  
DATABASE CONCURRENCY. (U) ALFRED P SLOAN SCHOOL OF  
MANAGEMENT CAMBRIDGE MA CENTER FOR I. M HSU DEC 82  
CISR-M010-8212-12 N00339-81-C-0663 F/G 5/2

1/1

UNCLASSIFIED

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

AD-A134356



12

Hierarchical Database  
Decomposition:  
A Technique For Data-  
base Concurrency Control

Meichun Hsu

INFOPLEX Technical Report  
# 12

December 1982

DTIC

COLLECTED

NOV 3 1983

A

This document has been approved  
for public release and sale; its  
distribution is unlimited.

Center for Information Systems Research

Massachusetts Institute of Technology  
Sloan School of Management  
77 Massachusetts Avenue  
Cambridge, Massachusetts, 02139

DTIC FILE COPY

83 11 03 086

Contract Number N0039-81-C-0663  
Internal Report Number M010-8212-12  
Deliverable Number 001

Hierarchical Database  
Decomposition:  
A Technique For Data-  
base Concurrency Control

Meichun Hsu

INFOPLEX Technical Report  
# 12

December 1982

NOV 3 1983

Principal Investigator:  
Professor S.E. Madnick

Prepared for:  
Naval Electronic Systems Command  
Washington, D.C.

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Technical Report #12	2. GOVT ACCESSION NO. AD-A134 356	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Hierarchical Database Decomposition - A Technique for Database Concurrency Control		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER M010-8212-12
7. AUTHOR(s) Meichun Hsu		8. CONTRACT OR GRANT NUMBER(s)  N0039-81-C-0663
		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
9. PERFORMING ORGANIZATION NAME AND ADDRESS  Sloan School of Management, MIT 50 Memorial Drive, Cambridge MA 02139		12. REPORT DATE December 1982
		13. NUMBER OF PAGES 61
11. CONTROLLING OFFICE NAME AND ADDRESS		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)  Database systems, database concurrency control, timestamp algorithm		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)  A generally accepted criterion for correctness of a concurrency control algorithm is the criterion of serializability of trans- actions. 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 time- stamp 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, and at the same time achieving the goal of serializability.

To this end, a hierarchical structure is proposed here as the means for analyzing opportunities of reducing concurrency overhead in a database application. A theorem is discovered which indicates that in a hierarchically decomposed database one can construct a type of partial order of all transactions which is different from the simple commit order and the simple initiation order, called the activity link order, which enables a new algorithm for concurrency control to be devised which is believed to have the potential of effectively reducing the overhead of read access synchronization in a database system that endorses a hierarchical decomposition of database.

Accession For  
NTIS  
DTIC  
Unannounced  
Justification



ABSTRACT

Increasing demand for information system capacity has prompted researchers to find ways to improve the computer systems used in information processing. Database management systems (DBMS's) represent one such effort to provide better information services at lower costs. In order to minimize response time and maximize throughput, it is desirable that a DBMS supports multiple users at the same time, allowing multiple transactions to run in parallel. However, for the purpose of maintaining database consistency and integrity, such parallelism must be properly controlled. For example, suppose two transactions that transfer money into the same bank account are run in parallel. Without proper coordination between the two transactions, it is possible that both of them would read the same old balance, modify it independently and write these independently-modified balances back to the database. If this happens, the final balance will reflect the result of only one, but not both, of the transactions, causing one transaction to be lost. To prevent such violation of database consistency and integrity from taking place, the concurrency control facility is an indispensable component of the database management system.

A generally accepted criterion for correctness of a concurrency control algorithm is the criterion of *serializability* of transactions. The classical

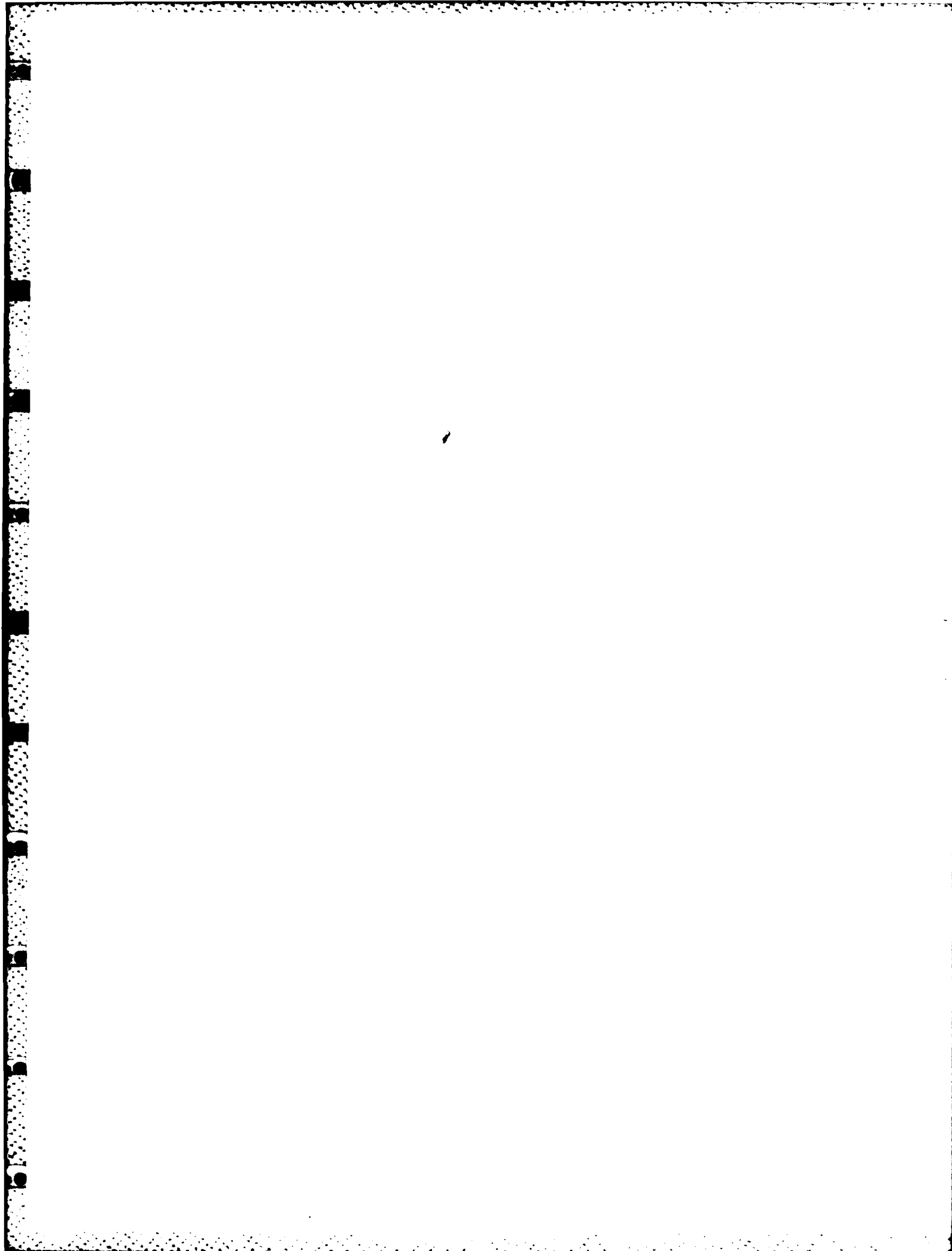
approaches to enforcing serializability are the *two-phase locking* technique and the *timestamp ordering* technique. The first technique ensures serializability by imposing a partial order on all transactions based on their commit order, while the second on their initiation order. Either approach requires 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, and at the same time achieving the goal of serializability.

To this end, a hierarchical structure is proposed here as the means for analyzing opportunities of reducing concurrency overhead in a database application. A theorem is discovered which indicates that in a hierarchically decomposed database one can construct yet a third type of partial order of all transactions which is different from the simple commit order and the simple initiation order that have been used in the existing algorithms. This new type of partial order, called the *activity link order*, enables a new algorithm for concurrency control to be devised which is believed to have the ability of effectively reducing the overhead of read access synchronization in a database system that endorses a hierarchical decomposition of the database.



## TABLE OF CONTENTS

1.0	INTRODUCTION AND LITERATURE OVERVIEW . . . . .	1
1.1	Introduction To The Concurrency Control Problem . . . . .	1
1.2	The Scope of Current Research . . . . .	2
1.2.1	A Motivating Example . . . . .	4
1.2.2	The Goal of the Current Reserarch . . . . .	7
1.3	Overview of Relevant Literature . . . . .	9
2.0	BASIC CONCEPTS OF MULTI-VERSION CONSISTENCY . . . . .	13
3.0	HIERARCHICAL DATABASE DECOMPOSITION . . . . .	16
3.1	Some Graphic-Theoretic Definitions . . . . .	16
3.2	Database Partition . . . . .	17
4.0	SYNCHRONIZING UPDATE TRANSACTIONS . . . . .	21
4.1	The Activity Link Function . . . . .	22
4.2	Concurrency Control Algorithm for Update Transactions . . . . .	24
4.3	Proof of correctness . . . . .	25
5.0	SYNCHRONIZING READ-ONLY TRANSACTIONS . . . . .	30
5.1	The Extended Activity Link Function . . . . .	30
5.2	Concurrency Control Protocol for Read-Only Transactions . . . . .	36
6.0	SUMMARY . . . . .	39
7.0	FUTURE RESEARCH DIRECTIONS . . . . .	39
7.1.1	Dynamic Restructuring of Database Decomposition . . . . .	40
7.2	Hierarchical Database Decomposition Methodology . . . . .	41
7.2.1	Handling Acyclic Decomposition . . . . .	41
7.2.2	Database Decomposition Methodology Via Data Analysis . . . . .	42
7.3	Implementation of the Concurrency Control Protocols . . . . .	42
7.4	The Efficacy of The HDD Approach . . . . .	42
7.5	Database Computer Applications . . . . .	42
	Bibliography . . . . .	44
	Appendix . . . . .	49



## 1.0 INTRODUCTION AND LITERATURE OVERVIEW

### 1.1 INTRODUCTION TO THE CONCURRENCY CONTROL PROBLEM

Increasing demand for information system capacity has prompted researchers to find ways to improve the computer systems used in information processing. Database management systems (DBMS's) represent one such effort to provide better information services at lower costs. In order to minimize response time and maximize throughput, it is desirable that a DBMS supports multiple users at the same time, allowing multiple transactions to run in parallel. However, for the purpose of maintaining database consistency and integrity, such parallelism must be properly controlled. Therefore the concurrency control facility is an indispensable component of the database management system.

The role of a concurrency control mechanism is to preserve the atomicity of a user transaction, i.e., it will prevent the processing of a transaction from being *erroneously interleaved* with other concurrent transactions, so that each transaction sees a consistent database state and, if necessary, can be recovered or backed out as a single unit.

A typical example of database inconsistency induced by improper interleaving of steps of concurrent transactions is shown in Figure 1. As shown in the figure, two transactions simultaneously accessing the same piece of data may result in lost update, leaving the database in an incorrect state. This is due to the fact that the database access steps from these two transactions

Currently there is \$100 in Smith's account.

$t_1$ : Deposit \$50 in Smith's account.

$t_2$ : Withdraw \$50 from Smith's account.

Schedule of steps of $t_1$ and $t_2$	Result of Smith's account
$t_1$ reads Smith's balance	\$100
$t_2$ reads Smith's balance	\$100
$t_1$ computes new balance = \$150	\$100
$t_2$ computes new balance = \$50	\$100
$t_1$ writes new balance	\$150
$t_2$ writes new balance	\$50

Figure 1. An example of database inconsistency induced by concurrent processing.

---

are not properly interleaved. By requiring that the database management system exercise control over such interleaving of concurrent transactions, the undesirable effects can be eliminated.

## 1.2 THE SCOPE OF CURRENT RESEARCH

A generally accepted criterion for correctness of a concurrency control algorithm is the criterion of serializability of transactions, which means that interleaving is harmless so long as one can show that the net effect of such interleaving is *equivalent to some serialized processing* (i.e., one after another) of all the transactions involved. In the above example, it is apparent that the steps of the two transactions are scheduled in such a way that

there exists no serialized schedule (i.e., either  $t_1$  after  $t_2$  or  $t_2$  after  $t_1$ ) that would have generated the same net effect. Therefore the schedule of these steps is not serializable, and therefore incorrect.

The classical approach to enforcing serializability is the *two-phase locking* technique. This technique locks up data elements being accessed by one transaction and blocks other transactions from operating on these data elements until the first transaction is finished. Another approach to dealing with this problem is that of the *timestamp ordering* technique, which stamps the data elements with the timestamps of the transactions that have operated on them so as to prevent violation of a pre-determined order from taking place.

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. 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 delays for concurrent transactions.

The purpose of the current research is to seek ways to reduce overhead of synchronizing certain types of read accesses, and at the same time achieving the goal of serializability. A more comprehensive overview will be given in

the next subsection of the literature in concurrency control, including efforts up to now that have aimed at similar goals. Here we motivate this research further through the following simple example.

#### 1.2.1 A MOTIVATING EXAMPLE

Consider an inventory database application of a retail business with a database shown in Figure 2. There are several types of transactions that operate on this database. A type 1 transaction inserts a sales, sales-modification, or a merchandise-arrival record into the database when the event of a sales, sales modification or arrival of certain merchandise occurs. A type 2 transaction is generated periodically for each item in the inventory to compute the current inventory level of that item. This transaction visits the sales, sales-modification and merchandise-arrival records to compute the net change ever since the last computation of the inventory level of that item. A new inventory level for that item is then posted in the inventory record.

To control these transactions using two-phase locking, a type 2 transaction would have set read locks on every read access it has generated to the sales, sales-modification and merchandise-arrival records. Using the timestamp ordering approach, the type 2 transaction would have left read timestamp for every such record it has read. However, a closer look at the application reveals that this overhead of read-locking or read-timestamping is not neces-

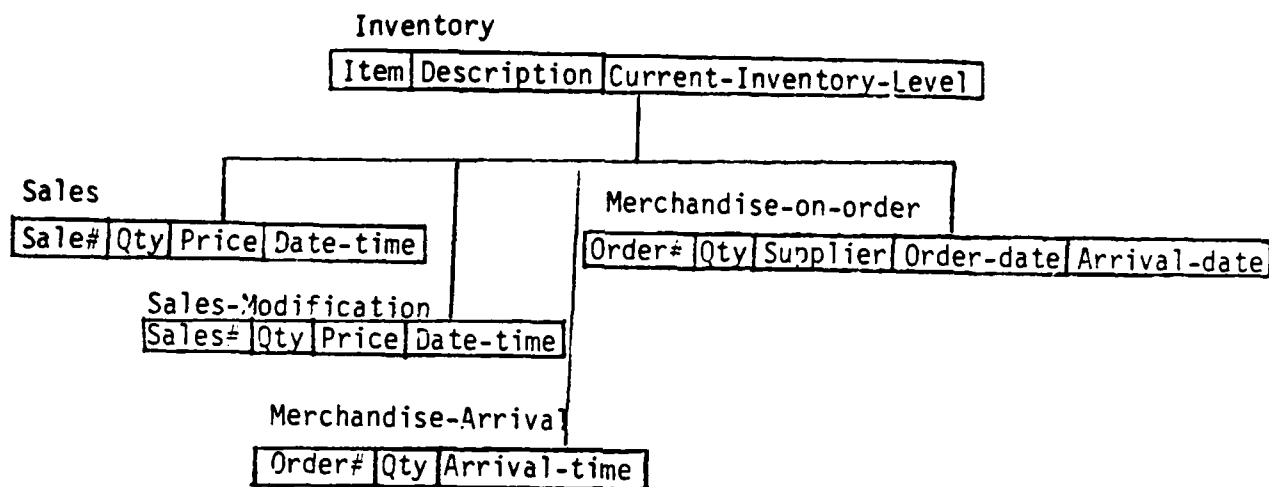


Figure 2. An example inventory database.

sary. The sales records, for example, once committed, will not be modified. In other words, with the exception of being deleted for garbage collection purposes, the sales records, once committed, have become 'read-only' records. Therefore the function served by setting read locks, etc., is no longer needed.

While the above case is very simple, a more complicated case may confuse the situation and make it not so obvious as to whether or not the read accesses should be registered. Suppose there are type 3 transactions which are also generated periodically to check for the need of reordering certain merchandise items. This type of transaction reads the merchandise-arrival records and adjust the merchandise-on-order records (by setting the arrival-date field of such records.) It then computes a gross inventory level

by summing the current-inventory-level and the quantities indicated by the non-arrived merchandise-on-order records. Based on this gross inventory level a decision is made as to whether to reorder this item. If it decides to do so, an order request is printed and a merchandise-on-order record is generated and inserted.

Under this situation, whether or not type 3 transactions can avoid setting read locks or read timestamps for the inventory and merchandise-arrival records it retrieves is not so obvious. In fact, if accesses to such records by type 3 transactions are not controlled, one can construct situations in which serializability is violated. Suppose two-phase locking is used. Then we can construct a case in which a merchandise-arrival record y for item x is seen by a type 2 transaction to compute the current level of inventory. This current level of inventory is in turn seen by a type 3 transaction to figure out whether to reorder. However, this type 3 transaction may *not* have seen the merchandise-arrival record y. As shown in Figure 3, a timing of these three transactions can be found such that if the type 3 transaction does not set read locks, violation of serializability occurs. This violation would generate adverse effect on the performance and integrity of the database, and is clearly undesirable. A similar case is constructed in Figure 4 which shows that if the timestamp ordering technique is used and if the type 3 transaction does not leave read timestamps, violation of serializability is also possible.



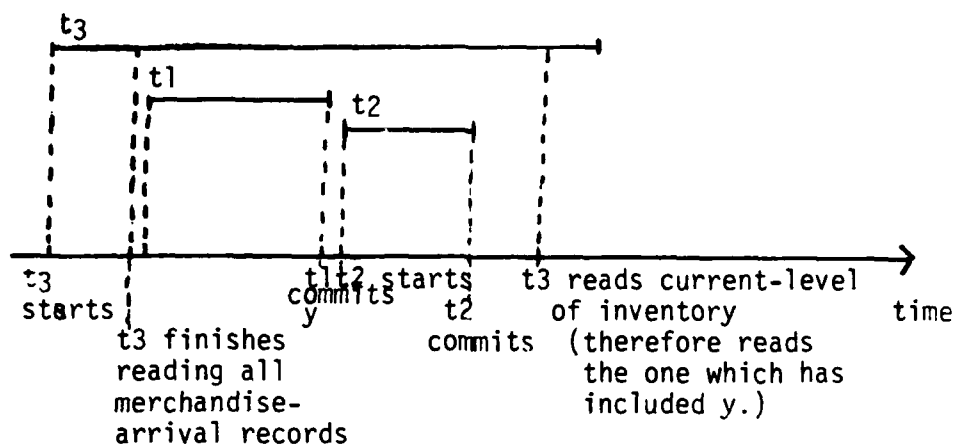


Figure 3. If read locks are not used, an anomaly may occur.

Because of the seeming unpredictability of the situation, most concurrency control algorithms, for simplicity, choose to ignore such opportunity for reducing synchronization overhead, and complies strictly to the assumption that all transactions may read and write any part of the database and therefore every access has to be controlled.

#### 1.2.2 THE GOAL OF THE CURRENT RESERARCH

The goal of the current research is to seek for a *systematic exploitation* of the kind of opportunity for reducing concurrency control overhead indicated in the case above.

Theoretically speaking, the above case would not exist if all the derived values of the entire database were computed at the time of the entry of the

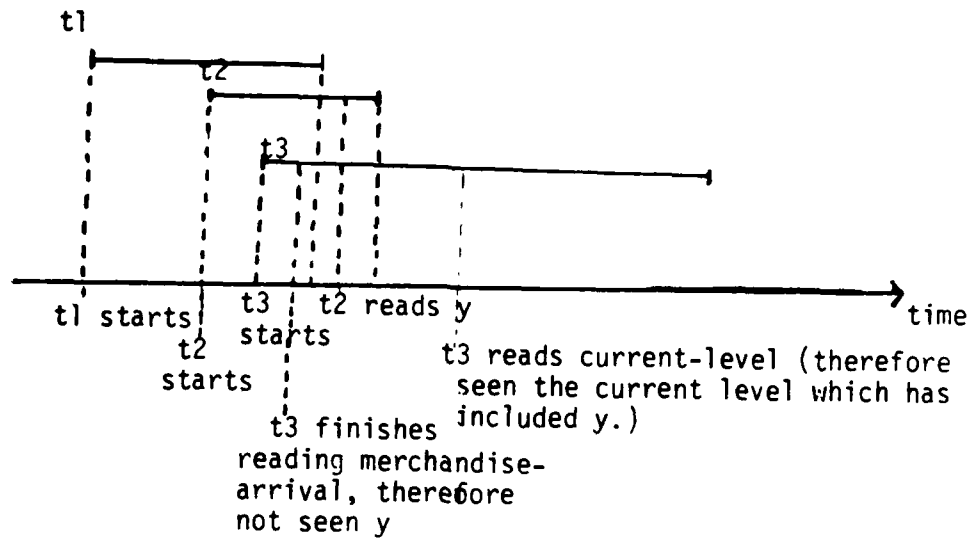


Figure 4. If read timestamps are not used, an anomaly may occur.

event. However, it is our belief that such case do exist widely in the real-life applications of database systems, because the instant computation of all the derived values at the time when an event is captured is both costly and unnecessary, and does not comply with how the organizations that use a database system are accustomed to operate. Delayed computations also supports certain method of operation that an organization may have purposely imposed. (e.g., the organizational unit that is responsible for the transaction that logs merchandise arrival should not have the knowledge of the current inventory level or the details of the merchandise ordering process.) In fact, the above case can be generalized further by, for example, adding to it applica-

tions that read from the reordering records and the merchandise-arrival records to generate supplier profile records in the database. As the list goes on, it is believed that a hierarchy of applications exists such that at each level of the hierarchy the transaction only reads from but does not write into data produced by applications of an earlier level. This structure presents a definite opportunity for concurrency control algorithms to explore.

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. This methodology may also bear implications on database and transaction design activities.

### 1.3 OVERVIEW OF RELEVANT LITERATURE

Concurrency control in a centralized or a distributed database system has been an active research area. The concept of database consistency has been formally analyzed in <Eswaran76, Gray76>, in which set-theoretic notions are used to formulate the concept. A consistent schedule of concurrent transactions has been defined to be one which is equivalent to a serialized schedule. Two-phase locking has been proposed in their papers as a mechanism which

preserves serializability. This notion of serializability has been further explored in <Bernstein79>.

Algorithms for database concurrency control abound in the literature. The distributed database present a challenge for the consistency problem which has encouraged development of many new algorithms. <For example, Ellis77, Lamport78, Rosenkrantz78, Thomas79, Bernstein80b.> A survey and comparison of theories and algorithms of concurrency control can be found in <Bernstein80b, Badal80>. 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. These two techniques have been used as the basis for further explorations.

One of the foci of recent development in concurrency control algorithms centers around the identification of techniques that increase level of concurrency and/or reduce synchronization overhead, at the same time preserving the correctness of the algorithm. One dimension of such effort is the multi-version database. It's been observed that keeping multiple versions of database elements will improve concurrency of the database <Papadimitriou82>.

In Papadimitriou's paper, it is shown that there exists an infinite hierarchy of multi-version serializability, and proven that the more versions a DBMS keeps, the higher is the level of concurrency it may achieve.

The concept of a timestamp-based multi-version database system was first proposed in <Reed78>. It is a general scheme in which the identifier of a data element consists of two components: the name of the data and the version of the data. In Reed's scheme, retrieval of an arbitrary time slice of the database is allowed.

A more limited multi-version concept was developed in <Bayer80>. In his scheme, one previous version of a data element, which has been saved for recovery purposes when the data element is going through changes made by uncommitted transactions, is utilized to allow read accesses to proceed without having to wait for the commitment of the update transaction. In <Garcia-Molina82>, a framework of strategies for processing read-only transactions is presented. In <Viemont82>, an interesting method for concurrency control is devised which also makes use of this one extra copy of data elements to synchronize transactions by order of commit time. In essence his technique is one which blends timestamp ordering and two-phase locking in one and chooses to switch to one or the other at the most opportune time so as to increase level of concurrency. In <Stearns81, Chan82> the one-previous-version method was extended to accommodate multiple previous versions (but does not allow for access of an arbitrary time slice of the

database from a user.) 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.

The above list of research bears resemblance to the research to be reported here. However, our technique 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 dimension of efforts of reducing synchronization overhead is that of conflict analysis <Bernstein80b>. In the research on concurrency control for SDD-1, conflict analysis was proposed which exploits a priori knowledge of the nature of the transactions to be run in the system. (To some extent, some of the research listed in the previous paragraphs concerning providing special protocols for read-only transactions fall into this category too, as it exploits the knowledge, albeit a limited one, of the nature of the transactions, namely, whether they are read-only or not.) The approach reported in the present paper is different from that of SDD-1 because it is not oriented towards distributed database systems, 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 therefore more practical to implement. A comparison between the approach proposed here and that of SDD-1 is included in the summary.

## 2.0 BASIC CONCEPTS OF MULTI-VERSION CONSISTENCY

In this chapter, we present the basic concept of multi-version consistency in graph-theoretic terms. This material is mostly taken from <Papadimitriou82, Bernstein82>, except for some notational differences. We wish to establish the fact that, if serializability of transactions is taken as the criterion for correctness of concurrency control, then acyclicity of a transaction dependency graph, constructed as a result of a multi-version scheduling algorithm, is the evidence of correctness of the algorithm, as it preserves serializability of transactions.

We present the following definitions and theorem.

*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

$\langle \text{transaction id, action, version of a data granule} \rangle$ .

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  $\langle t_1, w, d^1 \rangle, \langle t_2, r, d^1 \rangle, \langle t_2, w, d^2 \rangle, \langle t_3, r, d^2 \rangle$ .

**Definition.** A version  $j$  of a data element  $d$  is the *predecessor* of a version  $k$  of  $d$  if  $\langle t_1, w, d^j \rangle$  is before  $\langle t_2, w, d^k \rangle$  in  $S(T)$  where  $t_1, t_2 \in T$ , and there exists no  $t \in T$  and  $i$  such that  $\langle t, w, d^i \rangle$  is between  $\langle t_1, w, d^j \rangle$  and  $\langle t_2, w, d^k \rangle$  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)  $\langle t_1, w, d^v \rangle$  and  $\langle t_2, r, d^v \rangle$  are in  $S(T)$  for some  $d^v$ , or

(2)  $\langle t_1, r, d^j \rangle$  and  $\langle t_2, w, d^k \rangle$  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.

### 3.0 HIERARCHICAL DATABASE DECOMPOSITION

#### 3.1 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 5. 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.

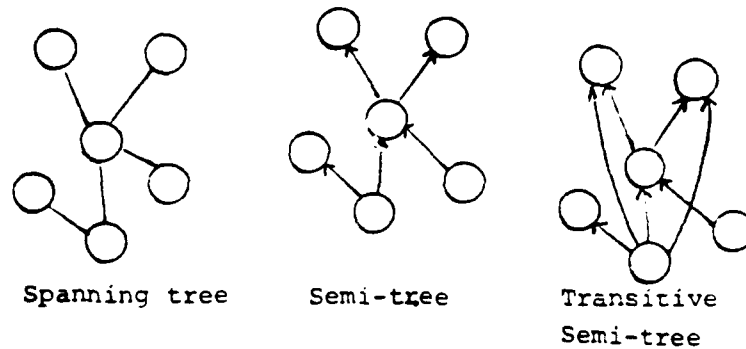


Figure 5. Illustration of a transitive semi-tree.

---

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

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

### 3.2 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  $T_i \rightarrow T_j \in A$  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  $CP_{ij}$ . Therefore,  $T_i \rightarrow T_k \rightarrow \dots \rightarrow T_j = CP_{ij}$  iff every arc is a critical arc. In addition, we give the following definition:

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

#### 4.0 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.)

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

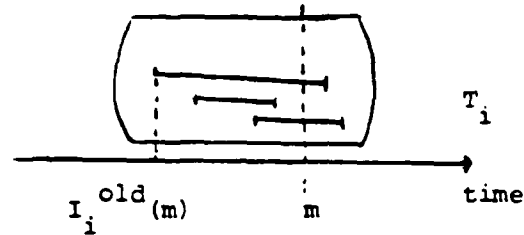
$$I_i^{old}(m) = \begin{cases} m & \text{if there exists no } t \in T_i \text{ active at time } m, \\ \text{Min } (I(t)) & \text{otherwise, where } t \in T_i, I(t) < m \text{ and } C(t) > m. \end{cases}$$

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

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



$I_i^{old}(m)$  = init. time of the oldest  
active trans. in class  
 $T_i$  at time  $m$ :



$A_i^j(m) = I_j^{old}(I_k^{old}(m))$ , if  $CP_i^j =$   
 $T_i \rightarrow T_k \rightarrow T_j$

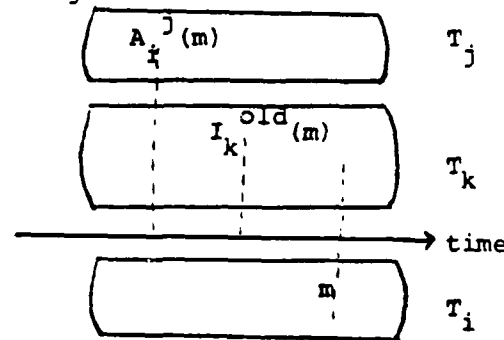


Figure 6. Graphical representation of the A function.

That is, the function A maps a time  $m$  for a transaction from class  $T_i$  to the initiation time,  $A_i^j(m)$ , 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  $T_i$  and  $T_j$  is  $T_i \rightarrow T_k \rightarrow T_j$ , then  $A_i^j(m) = I_j^{old}(I_k^{old}(m))$ . This is exemplified in Figure 6.

#### 4.2 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.

##### *Concurrency control algorithm for update transactions:*

For every database access request from an update transaction  $t \in T$ , 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  $d^0$  of  $d$  such that

$$TS(d^0) = \text{Max}(TS(d^v)) \text{ for all } v \text{ such that}$$

$$TS(d^v) < A_i^j(I(t)).$$

(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>.

#### 4.3 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 &multi.) 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 transactions  $t_1, t_2$ , where  $t_1 \in T_i, t_2 \in T_j$ ,  $T_i$  and  $T_j$  are connected by a critical path in THG,  $i$  and  $j$  not necessarily distinct. We say that  $t_1$  *topologically follows*  $t_2$  (or  $t_1 \Rightarrow t_2$ ) iff

- (1) if  $T_i = T_j$  then  $I(t_1) > I(t_2)$ .
- (2) If  $T_i \uparrow > T_j$  then  $I(t_1) \geq A_j^{-1}(I(t_2))$ .
- (3) If  $T_j \uparrow > T_i$  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

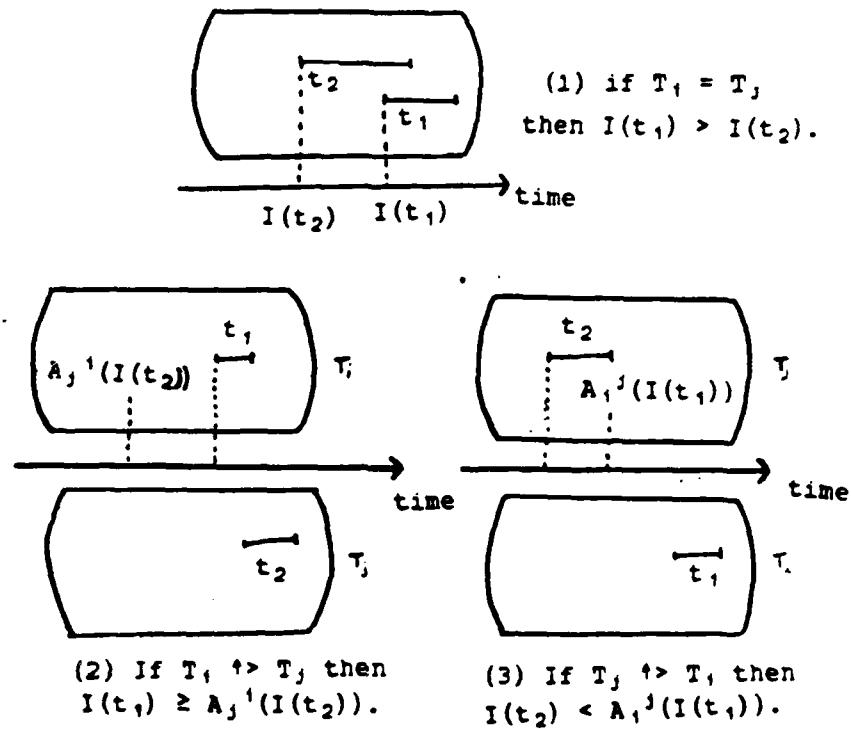


Figure 7. Graphical representation of the relation  $t_1 \Rightarrow t_2$ .

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 7. 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  $t_1 \in T_1$ ,  $t_2 \in T_k$ ,  $t_3 \in T_j$ , such that  $t_1 \Rightarrow t_2$ ,  $t_2 \Rightarrow t_3$  and  $T_1$ ,  $T_k$  and  $T_j$  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  $t_1, t_2 \in T^u$ ,  $t_1 \sim t_2 \in TG(S(T^u))$  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  $T_i$  of a lower level while  $t_2$  belongs to a class  $T_j$  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  $T_i$  of a higher level while  $t_2$  belongs to a class  $T_j$  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  $T_i$  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)

## 5.0 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  $CP_j$  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  $CP_j$  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 8.

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 8. To handle these transactions, we first introduce the *extended activity link function* in the following subsection.

### 5.1 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 crit-



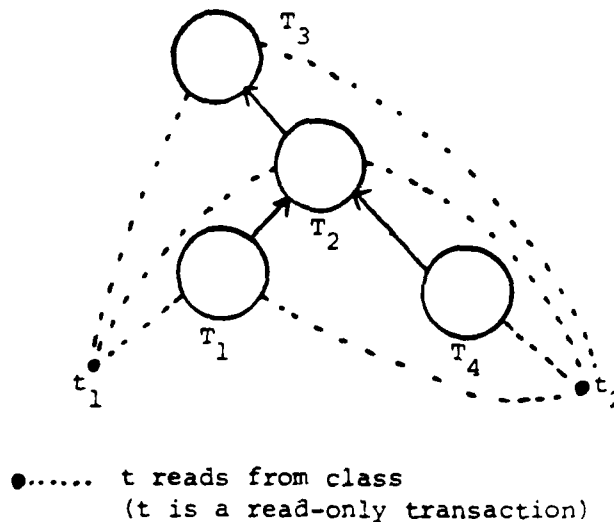


Figure 8. Read-only transactions that read from one critical path.

ical path in the transaction hierarchy graph. The extended function, on the other hand, specifies how transactions in a transaction class are linked 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  $C_{state}$  and  $B_j$  that can be considered conceptually the *inverse* of functions  $I_{old}$  and  $A_j$ . Then two properties of

the relationship between the functions  $A_j$  and  $B_j$  are derived. The extended activity link function  $E_j$  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  $C_j^{late}: m \rightarrow m'$  be a function which maps time  $m$  to another  $m'$  where  $T_j$  is a transaction class and  $C_j^{late}(m)$  is determined as follows.

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

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

**Definition.**  $C_j^{late}(m)$  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  $B_j^{-1}(m)$ , is a function which maps a time value  $m$  to another such that

$$B_j^{-1}(m) = \begin{cases} C_j^{\text{late}}(m) & \text{if } T_i \rightarrow T_j = CP_i^j \\ B_k^{-1}(B_j^k(m)) & \text{otherwise,} \\ & \text{where } T_i \rightarrow \dots \rightarrow T_k \rightarrow T_j = CP_i^j. \end{cases}$$

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.*  $A_i^j(B_j^{-1}(m)) \geq m$ , 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  $\epsilon$ ,  $A_i^j(B_j^{-1}(m) - \epsilon) < m$ , 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  $UCP_i^j = \langle i,$

$i_1, i_2, \dots, i_n, j$  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

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

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)

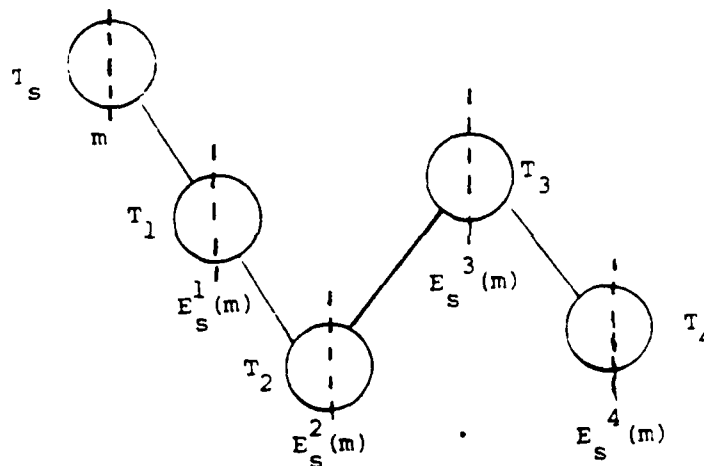
Intuitively, the  $E$  function provides a way of computing a *time wall* for all transaction classes in the database system across which no direct dependency from the 'older side' of the wall to the 'newer side' of the wall can occur. A time wall  $TW(m,s)$  is the set of all times  $E_s^i(m)$  where  $m$  is a time,  $D_s$  is a chosen data segment, and  $D_i$  is any data segment. This concept is graphically presented in Figure 9. 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  $d^0$  such that

$$TS(d^0) = \text{Max} (TS(d^v)) \text{ where } TS(d^v) < E_s^i(m),$$

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

*Proof.* (See Appendix)



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.e.,  $I(t) < E_s^i(m)$ ) and that on the right side of the dotted line (i.e.,  $I(t) > E_s^j(m)$ ).

Figure 9. The E function used as a 'time wall.'

In other words, if a read-only transaction reads the latest versions of data granules of data segment  $D$ , 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.

## 5.2 CONCURRENTLY 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)  $E_s'(m)$  (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  $E_s'(m')$  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  $E_s'(m)$  is computable at  $I(t)$ , therefore  $t$  potentially does not have to wait until a later time to access from certain segment. (If some  $E_s'(m)$  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  $C_i$  late 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  $d \in D_1$ . The segment controller of  $D_1$  provides the version  $d^0$  of  $d$  such that

$$TS(d^0) = \text{Max}(TS(d^v)) \text{ for all } v \text{ such that}$$

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

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



## 6.0 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 10.

## 7.0 FUTURE RESEARCH DIRECTIONS

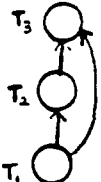
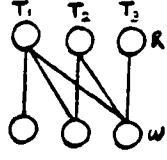

	HDD	SDD-1	MV2PL
Trans Analysis	Hier-archival	General*	None
Representation			
Inter-Class Synch.	Never reject or block a read req.	May cause read req. to be rejected or blocked	N.A.
Intra-Class Synch.	Timestamp Ordering	Serialized Pipelining	2-phase locking
Read-only Trans.	Similar to Inter-class Synch.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

Figure 10. A comparison of the HDD, SDD-1 and MV2PL approaches.

#### 7.1.1 DYNAMIC RESTRUCTURING OF DATABASE DECOMPOSITION

We will develop a scheme for handling ad hoc transactions that introduce access patterns which cause an original partition to become illegal. This aspect of the technique is important for two reasons. First, any database

partition and transaction classification based on an a priori analysis of all possible transactions to be run in the system is not likely to be fault-proof. Second, some transactions that are not frequently run but demand an access pattern which causes a finer partition to become illegal may be left out of the pre-analysis intentionally, so that for the majority of the time the system can operate under a finer partition while a special handling is adopted to take care of this type of transactions when they enter the system. We will try to achieve a scheme which can *dynamically* restructure the database partition. That is, it should be a scheme which does not require a quiescence of the database activity in order to perform the restructuring or the restoration.

## 7.2 HIERARCHICAL DATABASE DECOMPOSITION METHODOLOGY

### 7.2.1 HANDLING ACYCLIC DECOMPOSITION

The present technique centers around the transitive semi-tree form of hierarchical database decomposition. The transitive semi-tree, while much more relaxed than a directed tree, may still be too restricted for the technique to be useful in some applications. Based on the theories developed for the current technique, we propose to find an algorithm that will transform a database partition whose data hierarchy graph is of the form of an acyclic graph to a legal partition, while preserving the granularity of the original partition as much as possible.

#### 7.2.2 DATABASE DECOMPOSITION METHODOLOGY VIA DATA ANALYSIS

We propose to study in detail graph-theoretic methodologies that can be used to cluster data elements of a database to arrive at a legal or an acyclic decomposition of the database.

#### 7.3 IMPLEMENTATION OF THE CONCURRENCY CONTROL PROTOCOLS

Studies will be made of the complexity of implementation of protocols developed in the current and future research. Algorithms for (1) computing the version numbers to be accessed by a transaction, (2) maintaining multiple-versions of the database, and (3) garbage collection will be addressed.

#### 7.4 THE EFFICACY OF THE HDD APPROACH

In order to substantiate the underlying assumptions used in the HDD approach of the hierarchical organization of databases, we propose to undertake a case study of operations of some real organizations. These cases will also be used to validate the claimed advantages of the HDD approach over conventional approaches to concurrency control.

#### 7.5 DATABASE COMPUTER APPLICATIONS

One of the motivations for the current research is to find a way to optimize the concurrency control activities inside of a multi-processor based database computer that employs a hierarchical decomposition of the DBMS functionalities. The potential of the current technique in reducing inter-level synchronization communications will be explored and algorithms for concurrency control in such an environment will be proposed.

BIBLIOGRAPHY

Astrahan76:

Astrahan, M.M. et al. System R: Relational approach to database management. ACM Trans. Database Syst. 1, 2 (June 1976) 25-35

Badal80:

Badal, D.Z. The analysis of the effects of concurrency control on distributed database system performance. Proc. VLDB (1980)

Baer80:

Baer, J.L. et al. The two step commitment protocol: modelling, specification and proof methodology. Proc. 5th Intern. Conf. on Software Eng. (March 1980)

Bayer80:

Bayer, R., Heller, H., and Reiser, A. Parallelism and recovery in database systems. ACM Trans. Database Syst. 5, 2 (June 1980)

Bernstein79:

Bernstein, P.A., Shipman, D.W., and Wong, W.S. Formal aspects of serializability in database concurrency control. IEEE Trans. Software Eng. SE-5, 3 (May 1979)

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

Chamberlin73:

Chamberlin, D.D., Boyce, R.F., and Traiger, I.L. A deadlock-free scheme for resource locking in a data-base environment. IBM RJ 1329 (#20657) (Dec. 1973)

Chamberlin81:

Chamberlin, D.D. et al. A history and evaluation of system R. Comm. ACM 24, 10 (Oct. 1981)

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)

Coffman71:

Coffman, Jr. E.G., Elphick, M.J., and Shoshani, A. System deadlocks. ACM Comput. Surv. 3, 2 (June 1971)

DeWitt79:

Dewitt, D.J. Direct - A multiprocessor organization for supporting relational database management systems. IEEE Trans. Computers C-28, 6 (June 1979)

Garcia-Molina82:

Garcia-Molina, H. and Wiederhold, G. Read-only transactions in a distributed database. ACM Trans. Database Syst. 7, 2 (June 1982)

Ellis77:

Ellis, C.S. Consistency and correctness of duplicate database systems. In Proc. 6th Symposium on Operating System Principles, West Lafayette, Ind. (1977) 67-84

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

Gardarin80:

Gardarin, G. An introduction to SABRE, a multiprocessor database computer. Publications Sabre (1980)

Goodman79:

Goodman, N., Bernstein, P.A., Wong, E., Christopher, R., and Rothnie, J.B. Query processing in SDD-1: a system for distributed databases. Computer Corporation of America, TR CCA-79-06, Cambridge, MA (Oct. 1979)

Gray75:

Gray, J.N., Lorie, R.A., and Putzolu, G.R. Granularity of locks in a shared data base. Proc. VLDB (1975)

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)

Hsiao77:

Hsiao, D.K., and Madnick, S.E. Database machine architecture in the context of information technology evolution. Proc. VLDB (1977)

Hsiao79:

Hsiao, D.K., Ed. Collected readings on a database computer (DBCI). Department of computer and Information Science, The Ohio State Univ., Columbus, Ohio (March 1979)

Hsiao80:

Hsiao, D.K. Database computers. In Advances in Computers, Academic Press, Vol. 19 (1980)

King73:

King, P.F., and Collmeyer, A.J. Database sharing - An efficient mechanism for supporting concurrent processes. NCC (1973)

Lamport78:

Lamport, L. Time, clocks, and the ordering of events in a distributed system. Commun. ACM, 21, 7 (July 1978) 558-565

Mager80:

Mager, P.S., Goldber, R.P. A survey of some approaches to distributed data base and distributed file system architecture. BGS Systems, INC. Waltham, Mass. TR-80-001 (Jan 1980)



Maryanski80:

Maryanski, F.J. Backend database systems. ACM Comput. Surv. 12, 1 (March 1980)

Menasce80:

Menasce, D.A., Popek, G.J., and Muntz, R.R. A locking protocol for resource coordination in distributed databases. ACM Trans. Database Syst. 5, 2 (June 1980)

Mohan79:

Mohan, C. An analysis of the design of SDD-1: A system for distributed data bases. SDBEG-11, Univ. of Texas at Austin (April 1979)

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)

Rozenkrantz78:

Rozenkrantz, D.J., Stears, R.E., and Lewis II, P.M. System level concurrency control for distributed database systems. ACM Trans. Database Syst. 3, 2 (June 1978)

Rypka79:

Rypka, D.J., and Lucido, A.P. Deadlock detection and avoidance for shared logical resources. IEEE Trans. Software Eng. SE-5, 5 (Sept. 1979)

Silberschatz80:

Silberschatz, A., and Kedem, Z. Consistency in hierarchical database systems. Journal of ACM 27, 1 (Jan 1980), 72-80

Stearns81:

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

Stonebraker80:

Stonebraker, M. Retrospection on a database system. ACM Trans. Database Syst. 5, 2 (June 1980) , 225-240

Thomas79:

Thomas, R.H. A majority consensus approach to concurrency control for multiple copy databases. ACM Trans. Database Syst. 4, 2 (June 1979) ,180-209

Viemont82:

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

Verhofstad78:

Verhofstad. Recovery techniques for database systems. ACM Comput. Surv. (June 1978)

APPENDIX

(I) PROOF OF PROPERTY 1.2

*Property 1.2 (The property of transitivity).*

*Proof.* We consider the following 5 groups of cases:

- (1)  $T_1 = T_k = T_j$ . By definition of  $\Rightarrow$  we have  $I(t_1) > I(t_2) > I(t_3)$ . Therefore  $t_1 \Rightarrow t_3$ .
- (2)  $T_1 = T_k \neq T_j$ . Two cases are considered:
  - (2.1)  $T_1 \uparrow T_j$ . Then  $t_2 \Rightarrow t_3$  implies  $I(t_2) \geq A_j^{-1}(I(t_3))$ .  $t_1 \Rightarrow t_2$  implies  $I(t_1) > I(t_2)$ . Therefore  $I(t_1) > A_j^{-1}(I(t_3))$ . Therefore  $t_1 \Rightarrow t_3$ .
  - (2.2)  $T_j \uparrow T_1$ . Then  $t_1 \Rightarrow t_2$  implies  $I(t_1) > I(t_2)$ . By Property 0.2 we have  $A_j^{-1}(I(t_1)) \geq A_j^{-1}(I(t_2))$ .  $t_2 \Rightarrow t_3$  implies  $A_j^{-1}(I(t_2)) > I(t_3)$ . Therefore  $A_j^{-1}(I(t_1)) > I(t_3)$ . Therefore  $t_1 \Rightarrow t_3$ .
- (3)  $T_1 \neq T_k = T_j$ . Two cases are considered:
  - (3.1)  $T_1 \uparrow T_j$ . Then  $t_2 \Rightarrow t_3$  implies  $I(t_2) > I(t_3)$ . By Property 0.2 we have  $A_j^{-1}(I(t_2)) \geq A_j^{-1}(I(t_3))$ .  $t_1 \Rightarrow t_2$  implies  $I(t_1) > A_j^{-1}(I(t_2))$ . Therefore  $I(t_1) \geq A_j^{-1}(I(t_3))$ . Therefore  $t_1 \Rightarrow t_3$ .
  - (3.2)  $T_j \uparrow T_1$ . Then  $t_2 \Rightarrow t_3$  implies  $I(t_2) > I(t_3)$ .  $t_1 \Rightarrow t_2$  implies  $A_j^{-1}(I(t_1)) > I(t_2)$ . Therefore  $A_j^{-1}(I(t_1)) > I(t_3)$ . Therefore  $t_1 \Rightarrow t_3$ .
- (4)  $T_1 = T_j \neq T_k$ . Two cases are considered:
  - (4.1)  $T_1 \uparrow T_k$ . Then  $t_1 \Rightarrow t_2$  implies  $I(t_1) > A_k^{-1}(I(t_2))$ .  $t_2 \Rightarrow t_3$  implies  $A_k^{-1}(I(t_2)) > I(t_3)$ . Therefore  $I(t_1) > I(t_3)$ . Therefore  $t_1 \Rightarrow t_3$ .

(4.2)  $T_k \uparrow T_1$ . Then  $t_1 \Rightarrow t_2$  implies  $A_1^k(I(t_1)) > I(t_2)$ .  $t_2 \Rightarrow t_3$  implies  $I(t_2) \geq A_1^k(I(t_3))$ . Therefore  $A_1^k(I(t_1)) > A_1^k(I(t_3))$ . By Property 0.2 we have  $I(t_1) > I(t_3)$ . Therefore  $t_1 \Rightarrow t_3$ .

(5)  $T_1 \neq T_k \neq T_j$ ,  $T_1 \neq T_j$ . Six cases are considered:

(5.1)  $T_j \uparrow T_k \uparrow T_1$ . Then  $t_1 \Rightarrow t_2$  implies  $A_1^k(I(t_1)) > I(t_2)$ . From Property 0.1 and 0.2 we have  $A_1^j(I(t_1)) = A_k^j(A_1^k(I(t_1))) \geq A_k^j(I(t_2))$ . Therefore  $A_1^j(I(t_1)) \geq A_k^j(I(t_2))$ .  $t_2 \Rightarrow t_3$  implies  $A_k^j(I(t_2)) > I(t_3)$ . Therefore  $A_1^j(I(t_1)) > I(t_3)$ . Therefore  $t_1 \Rightarrow t_3$ .

(5.2)  $T_1 \uparrow T_k \uparrow T_j$ . Then  $t_2 \Rightarrow t_3$  implies  $I(t_2) \geq A_j^k(I(t_3))$ . From Property 0.1 and 0.2 we have  $A_k^1(I(t_2)) \geq A_k^1(A_j^k(I(t_3))) = A_j^1(I(t_3))$ .  $t_1 \Rightarrow t_2$  implies  $I(t_1) \geq A_k^1(I(t_2))$ . Therefore  $I(t_1) \geq A_j^1(I(t_3))$ . Therefore  $t_1 \Rightarrow t_3$ .

(5.3)  $T_j \uparrow T_1 \uparrow T_k$ . Then  $t_1 \Rightarrow t_2$  implies  $I(t_1) \geq A_k^1(I(t_2))$ . From Property 0.1 and 0.2 we have  $A_1^j(I(t_1)) \geq A_1^j(A_k^1(I(t_2))) = A_k^j(I(t_2))$ .  $t_2 \Rightarrow t_3$  implies  $A_k^j(I(t_2)) > I(t_3)$ . Therefore  $A_1^j(I(t_1)) > I(t_3)$ . Therefore  $t_1 \Rightarrow t_3$ .

(5.4)  $T_k \uparrow T_1 \uparrow T_j$ . Then  $t_1 \Rightarrow t_2$  implies  $A_1^k(I(t_1)) > I(t_2)$ .  $t_2 \Rightarrow t_3$  implies  $I(t_2) \geq A_j^k(I(t_3))$ . From Property 0.1 and 0.2 we have  $A_1^k(I(t_1)) > A_j^k(I(t_3)) = A_1^k(A_j^1(I(t_3)))$ . Therefore  $I(t_1) > A_j^1(I(t_3))$ . Therefore  $t_1 \Rightarrow t_3$ .

(5.5)  $T_k \uparrow T_j \uparrow T_1$ . Then  $t_1 \Rightarrow t_2$  implies  $A_1^k(I(t_1)) > I(t_2)$ .  $t_2 \Rightarrow t_3$  implies  $I(t_2) \geq A_j^k(I(t_3))$ . Therefore  $A_1^k(I(t_1)) > A_j^k(I(t_3))$ . However,  $A_1^k(I(t_1)) = A_j^k(A_1^j(I(t_1)))$ . Therefore  $A_1^j(I(t_1)) > I(t_3)$ . Therefore  $t_1 \Rightarrow t_3$ .

(5.6)  $T_i \uparrow > T_j \uparrow > T_k$ . Then  $t_1 \Rightarrow t_2$  implies  $I(t_1) \geq A_k^{-1}(I(t_2))$ . But  $A_k^{-1}(I(t_2)) = A_j^{-1}(A_k^{-1}(I(t_2)))$ . And  $t_2 \Rightarrow t_3$  implies  $A_k^{-1}(I(t_2)) > I(t_3)$ . Therefore  $A_k^{-1}(I(t_2)) \geq A_j^{-1}(I(t_3))$ . Therefore  $I(t_1) \geq A_j^{-1}(I(t_3))$ . Therefore  $t_1 \Rightarrow t_3$ .

In each group, we have permuted 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.

## (II) 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 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  $T_1$ ,  $T_j$  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  $T_1$ ,  $T_j$  and  $T_k$  are *not* on one critical path.

*Property.* If  $BCD(t_1, t_4)$ , where  $t_1 \in T_1$  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  $T_1, T_j, T_k$  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  $\ell$  be the length (in number of arcs, i.e., direct dependencies) of a critical path dependency. Then  $\ell$  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  $\ell(CD(t_1, t_2)) = 1$  then  $t_1 \Rightarrow t_2$ .
- (2) If  $\ell(CD(t_1, t_2)) = g$  and if  $t_a \Rightarrow t_b$  for all  $t_a, t_b$  s.t. there exists  $CD(t_a, t_b)$  and  $\ell(CD(t_a, t_b)) < g$ , 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  $t_4 \in T_l$  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  $\ell(CD(t_1, t_2)) < g$  therefore  $t_1 \Rightarrow t_2$ . And by the definition of  $CD$ ,  $T_i$ ,  $T_j$ ,  $T_k$  and  $T_l$  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  $\ell(CD(t_1, t_5)) < g$ , 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.*

(III) PROOF OF PROPERTY 2.1

**Property 2.1.**  $A_i^j(B_j^i(m)) \geq m$ , where  $T_1 \rightarrow T_{1,1} \rightarrow \dots \rightarrow T_{1,(n-1)} \rightarrow T_{1,n} \rightarrow T_j$   
 $= CP_i^j$  in the transaction hierarchy graph.

**Proof.**  $A_i^j(B_j^i(m)) = A_i^j(C_{1,1}(\dots(C_{1,n}(C_j(m))))\dots)$ . ( $C_j$  is an abbreviated expression for  $C_j^{late}$  and  $I_j$  is an abbreviated expression for  $I_j^{old}$ .) Let  $m_j = C_j(m)$ . Then  $m_j = C(t_j^0)$  if there exists  $t \in T_j$  active at time  $m$  and  $C_j(m) = C(t_j^0)$ ,

and  $m_j = m$  if there exists no  $t \in T_j$  active at time  $m$ . Therefore  $A_i^j(B_j^i(m)) = A_i^j(C_{1,1}(\dots(C_{1,n}(m_j))))$ . Continue substitution of the L function in the expression with similarly defined  $m_{1,n}, \dots, m_{1,1}$ , we get  $A_i^j(B_j^i(m)) = A_i^j(m_{1,1})$ . Now we start spelling out the function  $A_i^j$ :  $A_i^j(m_{1,1}) = A_{i,1}^j(I_{1,1}(m_{1,1}))$ .

Consider the following two cases:

- (1) If there exists no  $t \in T_{1,1}$  active at  $m_{1,1}$ , then  $I_{1,1}(m_{1,1}) = m_{1,1}$ . Since  $m_{1,1} = C_{1,1}(m_{1,2}) \geq m_{1,2}$ , we have  $I_{1,1}(m_{1,1}) \geq m_{1,2}$ .
- (2) If there exists  $t \in T_{1,1}$  active at  $m_{1,1}$ , then  $I_{1,1}(m_{1,1}) = I(t_{1,1}')$ , where  $t_{1,1}' \neq t_{1,1}^0$  (since  $t_{1,1}'$  is active at  $m_{1,1}$  while  $t_{1,1}^0$  commits at  $m_{1,1}$ ), and  $I(t_{1,1}') \geq m_{1,2}$  (since if  $I(t_{1,1}') < m_{1,2}$  then during the previous application of  $C_{1,1}$ ,  $C_{1,1}(m_{1,2})$  should be equal to  $C(t_{1,1}')$  and not  $C(t_{1,1}^0)$ , and contradicts the assumption.) Therefore  $I_{1,1}(m_{1,1}) \geq m_{1,2}$ .

Therefor we conclude  $I_{1,1}(m_{1,1}) \geq m_{1,2}$ . By the same reasoning we continue spelling out the A function to arrive at the following:  $A_i^j(B_j^i(m)) = A_{i,n}^j(I_{1,n}(m_{1,n}))$ .

Since  $I_{1,n}(m_{1,n}) \geq m_j = C_j(m)$ , we have  $A_i^j(B_j^i(m)) \geq A_{i,n}^j(C_j(m))$ . Since  $A_{i,n}^j(C_j(m)) = I_j(C_j(m)) \geq m$ , we have  $A_i^j(B_j^i(m)) \geq m$ . *Q.E.D.*



(IV) PROOF OF PROPERTY 2.2

**Property 2.2.**  $A_i^j(B_j^i(m) - \epsilon) < m$ , where  $T_i \rightarrow T_{i1} \rightarrow \dots \rightarrow T_{i(n-1)} \rightarrow T_{in} \rightarrow T_j = CP_i^j$  in the transaction hierarchy graph, and  $\epsilon$  a small value.

**Proof.** Let  $m_j, m_{in}, \dots, m_i$  be defined in the same way as in the proof of Property 2.1. We have  $A_i^j(B_j^i(m) - \epsilon) = A_i^j(m_{i1} - \epsilon) = A_{i1}^j(I_{i1}(m_{i1} - \epsilon))$ . Now we show that  $I_{i1}(m_{i1} - \epsilon) < m_{i2}$ . Consider the following two cases:

(1) If there exists no  $t \in T_{i1}$  active at  $m_{i2}$ , then  $m_{i1} = C_{i1}(m_{i2}) = m_{i2}$ .

Therefore  $I_{i1}(m_{i1} - \epsilon) = I_{i1}(m_{i2} - \epsilon) \leq m_{i2} - \epsilon < m_{i2}$ .

(2) If there exists  $t \in T_{i1}$  active at  $m_{i2}$ , then  $m_{i1} = C_{i1}(m_{i2}) = C(t_{i1}^0)$

where  $I(t_{i1}^0) < m_{i2}$ . Therefore  $I_{i1}(m_{i1} - \epsilon) = I_{i1}(C(t_{i1}^0) - \epsilon) \leq I(t_{i1}^0) < m_{i2}$ .

Therefore we conclude  $I_{i1}(m_{i1} - \epsilon) < m_{i2}$ . Let  $m_{i1}' = I_{i1}(m_{i1} - \epsilon)$ . Then  $m_{i1}'$

$< m_{i2}$ , and  $A_i^j(B_j^i(m) - \epsilon) = A_{i1}^j(m_{i1}')$ . Continue the process of substitution we have  $A_i^j(B_j^i(m) - \epsilon) = A_{in}^j(m_{in}') = I_j(m_{in}')$  where  $m_{in}' < m_j$ . But  $I_j(m_{in}') \leq I_j(m_j - \epsilon) = I_j(C_j(m) - \epsilon) < m$ . Therefore  $A_i^j(B_j^i(m) - \epsilon) < m$ .

**Q.E.D.**

(V) PROOF OF LEMMA 2.1

**Lemma 2.1.** Let  $T_k, T_i$  and  $T_j$  be transaction classes in a THG of a legal database partition, and  $T_i$  and  $T_j$  are on one critical path. Then for any time value  $m$ , 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.* Let  $T_{k_1}$  be the class such that  $k_1$  is the first index in  $UCP_{k_1}$  where  $T_{k_1}$  and  $T_1, T_j$  are on one critical path. ( $k$  and  $k_1$  are not necessarily distinct.) Then  $k_1$  will also be the first such index in  $UCP_k$ , and the subset of the ordered set  $UCP_{k_1}$  up to  $k_1$  and that of  $UCP_k$  up to  $k_1$  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)  $i = j \neq k_1$  or  $i = j = k_1$ .

In this case,  $E_k^i(m) = E_k^j(m)$ . Since  $t_1$  and  $t_2$  are in the same class, by intra-class-synchronization rule we have  $I(t_1) < I(t_2)$ , which implies that there exists no  $t_1 \rightarrow t_2$ .

- (2)  $i = k_1 \neq j$ . Two cases are considered:

- (2.1)  $T_1 \uparrow T_j$ .

$I(t_2) \geq E_k^j(m)$  implies that  $A_j^i(I(t_2)) \geq A_j^i(E_k^j(m)) = A_j^{k_1}(E_{k_1}^j(E_k^{k_1}(m)))$ . From Property 2.1 we have  $A_j^{k_1}(E_{k_1}^j(E_k^{k_1}(m))) \geq E_k^{k_1}(m) = E_k^i(m) > I(t_1)$ . Therefore  $A_j^i(I(t_2)) > I(t_1)$ , which implies that there exists no  $t_1 \rightarrow t_2$ .

- (2.2)  $T_j \uparrow T_1$ .

$I(t_1) < E_k^i(m)$  implies  $A_1^j(I(t_1)) \leq A_1^j(E_k^i(m)) = E_k^j(m) \leq I(t_2)$ . Therefore  $A_1^j(I(t_1)) \leq I(t_2)$ , which implies that there exists no  $t_1 \rightarrow t_2$ .

- (3)  $j = k_1 \neq i$ . Two cases are considered:

- (3.1)  $T_1 \uparrow T_j$ .

$I(t_2) \geq E_k^j(m)$  implies that  $A_j^i(I(t_2)) \geq A_j^i(E_k^j(m)) = E_k^i(m) > I(t_1)$ . Therefore  $A_j^i(I(t_2)) > I(t_1)$ , which implies that there exists no  $t_1 \rightarrow t_2$ .

(3.2)  $T_j \uparrow > T_i$ .

$I(t_1) < E_k^i(m)$  implies  $I(t_1) \leq E_k^i(m) - \epsilon$ , which implies  $A_i^j(I(t_1)) \leq A_i^j(E_k^i(m) - \epsilon)$ . From property 2.2 we have  $A_i^j(E_k^i(m) - \epsilon) < E_k^j(m)$ . Since  $I(t_2) \geq E_k^j(m)$ , therefore  $A_i^j(I(t_1)) < I(t_2)$ , which implies that there exists no  $t_1 \rightarrow t_2$ .

(4)  $i \neq j \neq k1$ . Six cases are considered:

(4.1)  $T_i \uparrow > T_{k1} \uparrow > T_j$ .

$I(t_2) \geq E_k^j(m)$  implies that  $A_j^i(I(t_2)) \geq A_j^i(E_k^j(m)) = A_j^i(B_{k1}^j(E_k^{k1}(m))) = A_{k1}^i(A_j^{k1}(B_{k1}^j(E_k^{k1}(m))))$ . By property 2.1,  $A_{k1}^i(A_j^{k1}(B_{k1}^j(E_k^{k1}(m)))) \geq A_{k1}^i(E_k^{k1}(m)) = E_k^i(m)$ . Since  $E_k^i(m) > I(t_1)$ , we have  $A_j^i(I(t_2)) > I(t_1)$ , which implies that there exists no  $t_1 \rightarrow t_2$ .

(4.2)  $T_j \uparrow > T_{k1} \uparrow > T_i$ .

$I(t_1) < E_k^i(m)$  implies  $I(t_1) < E_k^i(m) - \epsilon$ , which implies  $A_i^j(I(t_1)) \leq A_i^j(E_k^i(m) - \epsilon) = A_{k1}^j(A_i^{k1}(B_{k1}^i(E_k^{k1}(m)) - \epsilon))$ . Let  $m' = A_i^{k1}(B_{k1}^i(E_k^{k1}(m)) - \epsilon)$ . By property 2.2 we have  $m' < E_k^{k1}(m)$ .

Therefore  $A_{k1}^j(A_i^{k1}(B_{k1}^i(E_k^{k1}(m)) - \epsilon)) = A_{k1}^j(m') \leq A_{k1}^j(E_k^{k1}(m)) = E_k^j(m) \leq I(t_2)$ . Therefore  $A_i^j(I(t_1)) \leq I(t_2)$  which implies there exists no  $t_1 \rightarrow t_2$ .

(4.3)  $T_{k1} > T_i > T_j$ .

$I(t_2) \geq E_k^j(m)$  implies  $A_j^j(I(t_2)) \geq A_j^j(E_k^j(m)) = A_j^j(B_j^j(E_k^j(m))) \geq E_k^j(m) > I(t_1)$ . Therefore  $A_j^j(I(t_2)) > I(t_1)$ , which means that there exists no  $t_1 \rightarrow t_2$ .

(4.4)  $T_j \uparrow > T_1 \uparrow > T_{k1}$ .

$I(t_1) < E_k^j(m)$  implies  $A_1^j(I(t_1)) \leq A_1^j(E_k^j(m)) = A_1^j(A_{k1}^j(E_k^j(m))) = A_{k1}^j(E_k^j(m)) = E_k^j(m) \leq I(t_2)$ . That is,  $A_1^j(I(t_1)) \leq I(t_2)$  which means there exists no  $t_1 \rightarrow t_2$ .

(4.5)  $T_{k1} \uparrow > T_j \uparrow > T_1$ .

$I(t_1) < E_k^j(m)$  implies  $I(t_1) \leq E_k^j(m) - \epsilon$ , which means  $A_1^j(I(t_1)) \leq A_1^j(E_k^j(m) - \epsilon) = A_1^j(B_j^j(E_k^j(m)) - \epsilon) < E_k^j(m) \leq I(t_2)$ . That is,  $A_1^j(I(t_1)) < I(t_2)$ , which means that there exists no  $t_1 \rightarrow t_2$ .

(4.6)  $T_1 \uparrow > T_j \uparrow > T_{k1}$ .

$I(t_2) \geq E_k^j(m)$  implies  $A_j^j(I(t_2)) \geq A_j^j(E_k^j(m)) = A_j^j(A_{k1}^j(E_k^j(m))) = E_k^j(m) > I(t_1)$ . That is  $A_j^j(I(t_2)) > I(t_1)$ , which means that there exists no  $t_1 \rightarrow t_2$ .

For each of the group above we have permuted the level of the distinct classes and for a total of 11 cases we have shown that it is impossible to have  $t_1 \rightarrow t_2$ . Therefore we prove that there exists no  $t_1 \rightarrow t_2$ . *Q.E.D.*

## (VI) 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  $T^{CS} \subseteq T$  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  $T$  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.* We want to show that the following two parts are true:

- (1) If  $T^{u1}$  is a CS then there exists no  $t_1 \rightarrow t_2$ .

By definition of a CS, if  $t_1 \in T^{u1}$  and  $t_1 \rightarrow t_2$ , then  $t_2$  must be also in  $T^{u1}$ , which contradicts the given. Therefore there exists no  $t_1 \rightarrow t_2$ .

- (2) If there exists no  $t_1 \rightarrow t_2$  for any  $t_1 \in T^{u1}$  and  $t_2 \in T^{u2}$ , then  $T^{u1}$  is a CS.

$T^{u1}$  is a CS because no transaction in  $T^{u1}$  can have a dependency in the transitive closure on a transaction which is not in  $T^{u1}$ .

Therefore we conclude that this property is true. *Q.E.D.*

*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 trans-

actions 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.*

END

FLAMER