# TIMESTAMP-BASED ALGORITHMS FOR CONCURRENCY CONTROL IN DISTRIBUTED DATABASE SYSTEMS*

Philip A. Bernstein**
Nathan Goodman**

Computer Corporation of America
and Harvard University

## Abstract

We decompose the problem of concurrency control into the sub-problems of read-write and write-write synchronization. We present a series of timestamp-based algorithms (called synchronization techniques) that achieve read-write and/or write-write synchronization. And we show how to combine any read-write technique with any write-write technique to yield a complete concurrency control algorithm (called a method). Using this framework we describe 12 "principal" concurrency control methods in detail. Each principal method can be modified by refinements described in the paper, leading to more than 50 distinct concurrency control algorithms.

## 1. Introduction

In this paper we present a framework for the design and analysis of concurrency control algorithms for distributed database management systems (DDBMS). This framework permits us to describe a large number of concurrency control algorithms in concise terms and guides us in the discovery of new algorithms. Using this framework we describe 12 "principal" concurrency control algorithms in detail and show how these principal algorithms can be refined to yield more than 50 distinct algorithms. These algorithms subsume about half of the literature on DDBMS concurrency control [BG1,2]; in addition these algorithms extend the state-of-the-art in DDBMS concurrency control, because most of them are new.

* This work was supported by Rome Air Development Center under contract no. F30602-79-C-0191. The views expressed are those of the authors and do not necessarily represent the opinion of Rome Air Development Center or the U.S. Government.

** Author's address: Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138.

We begin by defining in (Section 2) a standard terminology for describing DDBMS concurrency control algorithms and a standard model of the DDBMS environment. Using this terminology and model as a foundation, we decompose the problem of concurrency control into the sub-problems of read-write and write-write synchronization (in Section 3). In Section 4 we present a series of timestamp-based algorithms (called synchronization techniques) that achieve read-write and/or write-write synchronization. Finally in Section 5 we show how each read-write technique can be integrated with each write-write technique to form a complete and correct concurrency control algorithm.

This work is part of a larger study of concurrency control [BG2] that considers locking-based synchronization techniques in addition to timestamp-based ones.

## 2. Transaction Processing Model

To understand how a concurrency control algorithm operates, one must understand how the algorithm fits into an overall DDBMS. In this section we present a simple model of a DDBMS, emphasizing how the DDBMS processes transactions.

### 2.1 Preliminary Definitions

A distributed database management system (DDBMS) is a collection of sites interconnected by a network. Each site is a computer running one or both of the following software modules: a transaction manager (TM) or a data manager (DM). Briefly, TMs supervise user interactions with the DDBMS while DMs manage the actual database. A network is a computer-to-computer communication system. The network is assumed to be perfectly reliable -- if site A sends a message to site B, site B is guaranteed to receive the message without error. In addition, we assume that between any pair of sites the network delivers messages in the order they were sent.

From a user's perspective, a database consists of a collection of logical data items, denoted X,Y,Z,... We leave the granularity of logical data items unspecified. In practice, logical data items may be files, records, etc. A logical database state is

an assignment of values to the logical data items comprising a database. Each logical data item may be stored at any DM in the system or redundantly at several DMs. A stored copy of a logical data item is called a stored data item; $x_1, \ldots, x_m$ denote the stored copies of logical data item X. When no confusion is possible we use the term data item for stored data item. A stored database state is an assignment of values to the stored data items of a database.

Users interact with the DDBMS by executing transactions. Transactions may be on-line queries expressed in a self-contained query language; application programs written in a general-purpose programming language; etc. The concurrency control algorithms we study pay no attention to the computations performed by transactions. Instead these algorithms make all of their decisions based on the data items a transaction reads and writes, and so the detailed form of transactions is unimportant in our analysis. However we do assume that transactions represent complete and correct computations; i.e. each transaction if executed alone on an initially consistent database would terminate, output correct results, and leave the database consistent. The logical readset (resp. writeset) of a transaction is the set of logical data items the transaction reads (resp. writes). Stored readsets and stored writesets are defined analogously. Two transactions are said to conflict if the stored readset or writeset of one intersects the stored writeset of the other.

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

## 2.2 DDBMS Architecture

A DDBMS contains four components (see fig. 2.1): transactions, TMs, DMs, and data. Transactions communicate with TMs, TMs communicate with DMs, and DMs manage the data. (TMs do not communicate with other TMs, nor do DMs communicate with other DMs.) The interface between transactions and TMs is the external interface of the DDBMS; the interface between TMs and DMs is its internal interface.

TMs supervise transactions. Each transaction executed in the DDBMS is supervised by a single TM, meaning that the transaction issues all of its database operations to that TM. Any distributed computation that is needed to execute the transaction is managed by the TM. Therefore, each transaction believes the system consists of a single TM and multiple DMs.

Four operations are defined at the external interface. Let X be any logical data item. READ(X) returns the value of X in the current logical database state. WRITE(X, new-value) creates a new logical database state in which X has the specified new value. Since transactions are assumed to represent complete computations, we use BEGIN and END operations to bracket transaction executions.

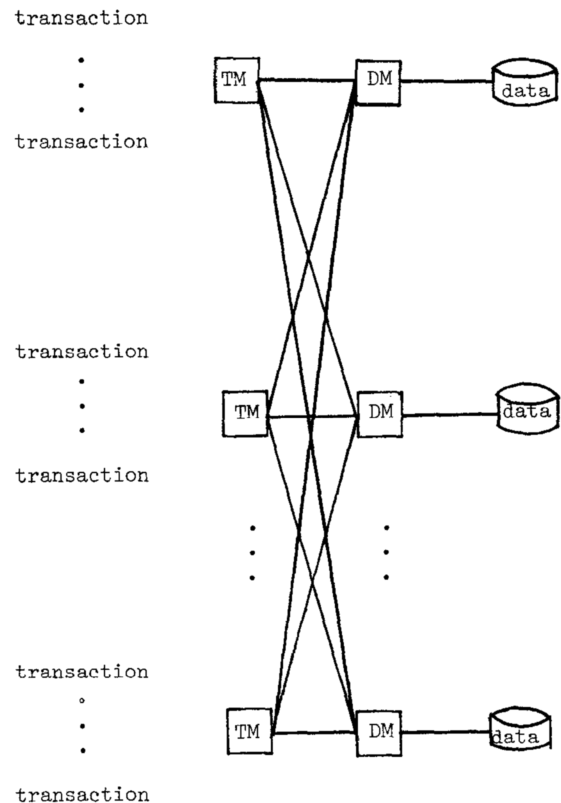
DMs manage the stored database, functioning essentially as back-end database processors. In response to commands from transactions, TMs issue commands to DMs specifying stored data items to be read or written. The details of the TM-DM interface constitute the core of our transaction processing model and are discussed in Sections 2.3 and 2.4. Section 2.3 describes the TM-DM interaction in a centralized database environment. Section 2.4 extends the discussion to a distributed database setting.

## 2.3 Centralized Transaction Processing Model

A centralized DBMS consists of one TM and one DM executing at the same site. A transaction, T, accesses the DBMS by issuing BEGIN, READ, WRITE, and END operations, which are processed as follows.

BEGIN: The TM initializes a private workspace for T. The private workspace functions as a temporary buffer for values that T writes into the database, and as a cache for values that T reads from the database.

Figure 2.1 DDBMS System Architecture



READ(X): The TM looks for a copy of X in T's private workspace. If the copy exists, its value is returned to T. Otherwise the TM issues a command to the DM asking it to retrieve a stored copy of X from the database. This operation is denoted dm-read(x). The value retrieved by the DM is given to T and put into T's private workspace.

WRITE(X, new-value): The TM again checks the private workspace. If the workspace has a copy of X, its value is updated to new-value; otherwise a copy of X with that value is created in the workspace. The new value of X is <u>not</u> stored in the database at this time.

END: The TM issues an operation denoted dm-write(x) for each logical data item X updated by T. Each dm-write(x) requests that the DM update the value of X in the stored database to the value of X in T's local workspace. When all dm-writes are processed, T is finished executing, and its private workspace is discarded.

The DBMS may <u>restart</u> T any time before a dm-write has been processed. The effect of restarting T is to obliterate its private workspace and to re-execute T from the beginning. As we will see, many concurrency control algorithms use transaction restarts as a tactic for attaining correct executions. However, once a single dm-write has been processed, T cannot be restarted. This is because each dm-write permanently installs an update into the database, and we cannot permit the database to reflect partial effects of transactions.

A DBMS can fail in many ways and a detailed treatment of reliability issues is beyond the scope of this paper. However, a reliability problem called <u>atomic commitment</u> has a major impact on concurrency control. Consider a transaction T that updates data items x,y,z,... and suppose the DBMS fails while processing T's END. If this occurs, some of T's updates may have been installed in the stored database while others have not, and the database may contain incorrect information (see fig. 2.2). To avoid this problem, the DBMS must ensure that <u>all</u> of a transaction's dm-writes are processed <u>or none</u> are.

The "standard" way to implement atomic commitment involves a procedure called <u>two-phase commit</u> [LS, Gray].* Again suppose T is updating x,y,z,... When T issues its END, the <u>first phase</u> of two-phase commit begins. During this phase the DM copies the values of x,y,z,... from T's private workspace onto secure storage. If the DBMS fails during the first phase, no harm is done, since <u>none</u> of T's updates have yet been applied to the stored database. During the <u>second phase</u>, the DBMS copies the values of x,y,z,... into the stored database. If the DBMS fails during the second phase, the database may contain incorrect information. However since the values of x,y,z,... are stored on secure storage, this inconsistency can be rectified when the system recovers: the recovery procedure reads the values of x,y,z,... from secure storage and resumes the commitment activity.

Figure 2.2 The Need for Atomic Commitment

- Consider a database of banking information
- Suppose Acme Corp.'s savings account has $2,000,000 and its checking account has $500,000. And suppose the DBMS fails while processing the following transaction.

  T: Move $1,000,000 from savings to checking

- In the absence of atomic commitment, the following incorrect execution could occur.

| <u>Execution of T</u> | | <u>Database</u> |
|---|---|---|
| | | S $2,000,000 |
| READ savings | S $2,000,000 | C 500,000 |
| READ checking | C 500,000 | |
| Subtract $1,000,000 from savings | | |
| Add $1,000,000 to checking | | |
| | S $1,000,000 | |
| | C 1,500,000 | |
| WRITE savings | | |
| | | $1,000,000 |
| | | 500,000 |
| ------SYSTEM CRASHES------ | | |
| WRITE checking --- never executed | | |

To model two-phase commit, it is convenient to add a third TM-DM operation, <u>pre-commit</u>, which instructs the DM to copy a data item from the private workspace to secure storage.

## 2.4 Distributed Transaction Processing Model

Our model of transaction processing in a distributed environment differs from the centralized case in two areas: how private workspaces are handled, and the implementation of two-phase commit.

### <u>Private Workspaces in a DDBMS</u>

In a centralized DBMS we assumed that private workspaces were part of the TM. We also assumed that data could freely move between a transaction and its workspace, and between a workspace and the DM. These assumptions are not appropriate in a DDBMS because TMs and DMs may run at different sites and the movement of data between a TM and a DM can be expensive. To reduce this cost, many DDBMSs employ <u>query optimization procedures</u> which regulate (and hopefully reduce) the flow of data between sites.

For example, in SDD-1 the private workspace for transaction T is distributed across all sites at which T accesses data [GBWRR]. The details of how T reads and writes data in these workspaces is a query optimization problem, and has no direct effect on concurrency control. Consequently, we factor this issue out of our model for distributed transaction processing.

In detail, our model of distributed transaction execution is as follows.

1. When transaction T issues its BEGIN operation, T's TM creates a private workspace for T. The location and organization of this workspace is left unspecified.
2. When T issues a READ(X) operation, the TM checks T's private workspace to see if a copy of X is present. If so, the value of that copy is made available to T. Otherwise the TM selects some stored copy of X, say $x_i$, and issues dm-read($x_i$) to the DM at which $x_i$ is stored. The DM responds by retrieving the stored value of $x_i$ from the database, placing this value in the private workspace. The TM then returns this value to T.
3. When T issues a WRITE(X, new-value) operation, the value of X in T's private workspace is updated to new-value, assuming the workspace contains a copy of X. Otherwise, a copy of X with the new value is created in the workspace.
4. When T issues its END operation, two-phase commit begins. For each X updated by T, and for each stored copy $x_i$ of X, the TM issues a pre-commit($x_i$) to the DM that stores $x_i$. The DM responds by copying the value of X from T's private workspace onto secure storage internal to the DM. After all pre-commits are processed, the TM issues dm-writes for all copies of all logical data items updated by T. A DM responds to dm-write($x_i$) by copying the value of $x_i$ from secure storage into the stored database. After all dm-writes are installed, T's execution is finished.

<u>Two-Phase Commit in a DDBMS</u>

The problem of atomic commitment is aggravated in a DDBMS by the possibility of one site failing while the remainder of the system continues to operate. Suppose T is updating x,y,z,... stored at $DM_x$, $DM_y$, $DM_z$,... (resp.) and suppose T's TM fails after issuing the dm-write(x), but before issuing the dm-writes for y,z,... At this point, the database contains incorrect information as illustrated in fig. 2.2. In a centralized DBMS, this phenomenon is not harmful because no transaction can access the database until the TM recovers from the failure. However, in a DDBMS, other TMs remain operational, and the incorrect database can be accessed from these TMs.

To avoid this problem, each DM that receives a pre-commit must be able to determine which other DMs are involved in the commitment activity. (This information could be a parameter to the pre-commit operation, stored in a private workspace, etc.) If T's TM fails before issuing all dm-writes, the DMs whose dm-writes were not issued can recognize the situation and consult the other DMs involved in the commitment. If <u>any</u> DM received a dm-write, the remaining ones act as if they had also received the command. Thus, if any DM applies an update to the database, they all do (see also, [HS2]).

## 3. Decomposition of Concurrency Control Problem

In this section we review concurrency control theory with two objectives: to define "correct executions" in precise terms, and to decompose the concurrency control problem into more tractable sub-problems.

### 3.1 Serializability

Let E denote an execution of transactions $T_1$, ..., $T_n$. E is a <u>serial execution</u> if no transactions ever execute concurrently in E; i.e., each transaction is executed to completion before the next one begins. Every serial execution is defined to be <u>correct</u>, because the properties of transactions (see Section 2.1) imply that a serial execution terminates properly and preserves database consistency. An execution is <u>serializable</u> if it is computationally equivalent to a serial execution, that is, if it produces the same output and has the same effect on the database as some serial execution. Since serial executions are correct and every serializable execution is equivalent to a serial one, every serializable execution is also correct. The goal of database concurrency control is to ensure that all executions are serializable.

The only operations that access the stored database are dm-read and dm-write. Hence, insofar as serializability is concerned, it is sufficient to model an execution of transactions by the execution of dm-reads and dm-writes at the various DMs of the DDBMS. In this spirit, we formally model an execution of transactions by a set of logs, one log per DM. Each log indicates the order in which dm-reads and dm-writes are processed at one DM (see fig. 3.1).

An execution modelled by a set of logs is <u>serial</u> if (1) for each log, and for each pair of transactions $T_i$ and $T_j$ whose operations appear in the log, either all of $T_i$'s operations precede all of $T_j$'s operations, or vice versa; and (2) for each pair of transactions, $T_i$ and $T_j$, if $T_i$'s operations precede $T_j$'s operations in one log, then $T_i$'s operations precede $T_j$'s operations in every log in which oper-

Figure 3.1 Modelling Executions as Logs

| Transactions | Database | |
|---|---|---|
| $T_1$: BEGIN; READ(X); WRITE(Y); END | A | $x_1$ $y_1$ |
| $T_2$: BEGIN; READ(Y); WRITE(Z): END | B | $y_2$ $z_2$ |
| $T_3$: BEGIN; READ(Z); WRITE(X); END | C | $z_3$ |

One possible execution of $T_1$, $T_2$, and $T_3$ is represented by the following logs. (Note: $r_i[x]$ denotes the operation dm-read(x) issued by $T_i$; $w_i[x]$ has the analogous meaning)

Log for DM A: $r_1[x_1]$ $w_1[y_1]$ $r_2[y_1]$ $w_3[x_1]$

Log for DM B: $w_1[y_2]$ $w_2[z_2]$

Log for DM C: $w_2[z_3]$ $r_3[z_3]$

ations from both $T_i$ and $T_j$ appear (see fig. 3.2). Intuitively, (1) says that at each DM no two transactions are interleaved, and (2) says that transactions execute in the same order at all DMs.

Two operations conflict if they operate on the same data item and one of the operations is a dm-write. The order in which operations execute is computationally significant iff the operations conflict. To illustrate the notion of conflict, consider a data item x and transactions $T_i$ and $T_j$. If $T_i$ issues dm-read(x) and $T_j$ issues dm-write(x), the value read by $T_i$ will (in general) differ depending on whether the dm-read precedes or follows the dm-write. Similarly, if both transactions issue dm-write(x) operations, the final value of x depends on which dm-write happens last. Those conflict situations are called read-write conflicts and write-write conflicts respectively.

The notion of conflict helps characterize the equivalence of executions. Let $E_1$ and $E_2$ be two executions, modelled by logs $\{L_{1,1},\ldots, L_{1,n}\}$ and $\{L_{2,1},\ldots L_{2,n}\}$, where $L_{i,j}$ models the execution at $DM_j$ for $E_i$. $E_1$ and $E_2$ are computationally equivalent if [PBR, Papadimitriou]: for each j, $1<j<n$, $L_{1,j}$ and $L_{2j}$ contain the same set of dm-reads and dm-writes and each pair of conflicting operations

Figure 3.2 Serial and Non-Serial Logs

The execution modelled in figure 3.1 is serial. Condition (1) holds since each log is itself serial -- i.e., there is no interleaving of operations from different transactions. Condition (2) holds since at DM A, $T_1$ precedes $T_2$ precedes $T_3$; at DM B, $T_1$ precedes $T_2$; and at DM C, $T_2$ precedes $T_3$.

The following execution is not serial; it satisfies (1) but not (2).

DM A: $r_1[x_1]$ $w_1[y_1]$ $r_2[y_2]$ $w_3[x_1]$

DM B: $w_2[z_2]$ $w_1[y_2]$

DM C: $w_2[z_3]$ $r_3[z_3]$

The following execution is also not serial; it doesn't satisfy (1) or (2).

DM A: $r_1[x_1]$ $r_2[y_2]$ $w_3[x_1]$ $w_1[y_1]$

DM B: $w_2[z_2]$ $w_1[y_2]$

DM C: $w_2[z_3]$ $r_3[z_3]$

appears in the same relative order in both logs. Intuitively, computational equivalence must hold in this case because (1) each dm-read operation reads data item values that were produced by the same dm-writes in both executions; and (2) the final dm-write on each data item is the same in both executions. Condition(1) ensures that each transaction reads the same input in both executions (and therefore performs the same computation). Combined with (2), it ensures that both executions leave the database in the same final state.

We can now characterize serializable executions precisely.

Theorem 1 [PBR, Papadimitriou, SLR] Let $\underline{T}=\{T_1,\ldots, T_n\}$ be a set of transactions and let E be an execution of these transactions modelled by logs $\{L_1,\ldots, L_n\}$. E is serializable if there exists a total ordering of $\underline{T}$ such that for each pair of conflicting operations $O_i$ and $O_j$ from distinct transactions $T_i$ ad $T_j$ (resp.), $O_i$ precedes $O_j$ in a log iff $T_i$ precedes $T_j$ in the total ordering.

The total order hypothesized in Theorem 1 is called a serialization order. A serialization order indicates a serial execution of the transactions $\underline{T}$ that is computationally equivalent to the original execution E. Thus, if the transactions had executed serially in the hypothesized order, the computation

performed by the transactions would have been identical to the computation represented by E.

To attain serializability, the DDBMS must guarantee that all executions satisfy the condition of Theorem 1. Those conditions require that conflicting dm-reads and dm-writes be processed in certain relative orders. Concurrency control is the activity of controlling the relative order of conflicting operations; an algorithm to perform such control is called a synchronization technique. So, to be correct, a DBMS must incorporate synchronization techniques that guarantee the conditions of Theorem 1.

### 3.2 A Paradigm for Concurrency Control

In Theorem 1, read-write and write-write conflicts are treated together under the general notion of conflict. However, we can decompose the concept of serializability by distinguishing these two types of conflict. Let E be an execution modelled by a set of logs. We define three binary relations on transactions in E, denoted ->rw, ->wr, and ->ww. For each pair of transactions, $T_i$ and $T_j$

1. $T_i$ ->rw $T_j$ iff in some log of E, $T_i$ reads some data item into which $T_j$ subsequently writes;
2. $T_i$ ->wr $T_j$ iff in some log of E, $T_i$ writes into some data item that $T_j$ subsequently reads;
3. $T_i$ ->ww $T_j$ iff in some log of E, $T_i$ writes into some data item into which $T_j$ subsequently writes.

Notationally, we use ->rwr = (->rw U ->wr) and -> = (->rwr U ->ww).

Intuitively, -> (with any subscript) means "in any serialization must precede". For example, $T_i$ ->rw $T_j$ means "$T_i$ in any serialization must precede $T_j$". This interpretation follows from Theorem 1: If $T_i$ reads x before $T_j$ writes into x, then the hypothetical serialization in Theorem 1 must have $T_i$ preceding $T_j$.

Every conflict between operations in E is represented by an -> relationship. Therefore, we can restate Theorem 1 in terms of ->. According to Theorem 1, E is serializable if there is a total order of transactions that is consistent with the order of all conflicts. In terms of ->, this means that E is serializable if there is a total order of transactions that is consistent with ->. This latter condition holds iff -> is acyclic ( A relation, ->, is acyclic if there is no sequence $i_1$ -> $i_2$, $i_2$ -> $i_3$,..., $i_{n-1}$ -> $i_n$ such that $i_1 = i_n$.) In addition, we can decompose -> into its components, ->rwr and ->ww, and restate the theorem in terms of these components.

Theorem 2 Let ->rwr and ->ww be associated with execution E. Then E is serializable if (a) ->rwr and ->ww are acyclic, and (b) there is a total ordering of the transactions consistent both with all ->rwr and all ->ww relationships.

Theorem 2 emphasizes a point overlooked in Theorem 1: read-write and write-write conflicts interact only insofar as there must be a total ordering of the transactions consistent with both types of conflicts. This suggests that read-write and write-write conflicts can, to some extent, be synchronized independently. We can use one technique to guarantee an acyclic ->rwr relation (which amounts to read-write synchronization) and a different technique to guarantee an acyclic ->ww relation (write-write synchronization). However, Theorem 2 says that having both ->rwr and ->ww acyclic is not enough. There must also be one serial order consistent with all -> relations. This serial order is the cement that binds together the read-write and write-write synchronization techniques.

Decomposing the serializability problem into the problems of read-write and write-write synchronization is the cornerstone of our paradigm for concurrency control. In Section 4 we describe algorithms that accomplish read-write (rw) and/or write-write (ww) synchronization, and in Section 5 we show how to combine rw and ww synchronization algorithms into correct concurrency control algorithms. It will be important hereafter to distinguish algorithms that attain rw and/or ww synchronization from algorithms that solve the entire distributed concurrency control problem. We shall use synchronization technique for the former type of algorithm, and concurrency control method for the latter.

## 4. Timestamp Ordering (T/O) Techniques

### 4.1 Specification

Timestamp ordering (T/O) is a technique whereby a serialization order is selected a priori and transaction execution is forced to obey this order. When a transaction begins, its TM creates a unique timestamp for it by reading the local clock time and appending a unique TM identifier to the low order bit. The TM also agrees not to assign another timestamp until the next clock tick. Thus timestamps assigned by different TMs differ in their low order bits while timestamps assigned by the same TM differ in their high order bits, and so all timestamps are unique system-wide. (Notice that this algorithm does not require that clocks at different sites be synchronized.)

The TM attaches the timestamp to all dm-read and dm-write operations issued on behalf of the transaction. DMs are required to process conflicting operations in timestamp order. The definition of conflicting operations depends on the type of syn-

chronization being performed. For rw synchronization, two operations conflict iff both operate on the same data item and one is a dm-read and the other is a dm-write. For ww synchronization, two operations conflict iff both operate on the same data item and both are dm-writes.

It is easy to prove that T/O attains an acyclic ->rwr (resp. ->ww) relation when used for rw (resp. ww) synchronization. Since each DM processes conflicting operations in timestamp order, each edge of the ->rwr (resp. ->ww) relation is in timestamp order. Hence, all paths in the relation are in timestamp order and, since all transactions have unique timestamps, no cycles are possible. In addition, the timestamp order is a valid serialization order.

### 4.2 Basic Implementation

An implementation of T/O amounts to building a T/O scheduler, a software module that receives dm-read and dm-write operations and outputs these operations according to the T/O specification. In practice, pre-commits must also be processed through the T/O scheduler for two-phase commit to operate properly. In Sections 4.1-4.8 we describe T/O implementations without considering the impact of two-phase commit. Section 4.9 considers two-phase commitment issues.

The basic T/O implementation distributes the schedulers along with the database. Consider the T/O scheduler at some particular DM. For each data item x stored at the DM, the scheduler keeps track of the largest timestamp of any dm-read (resp. dm-write) that has operated on x. This timestamp is denoted R-timestamp(x)(resp. W-timestamp(x)).

For rw synchronization the basic T/O scheduler operates as follows. To process a dm-read(x), the scheduler compares the timestamp of the dm-read to W-timestamp(x). If the former timestamp is larger, the scheduler outputs the dm-read and updates R-timestamp(x) to the maximum of (a) the old R-timestamp(x), or (b) the timestamp of the dm-read. If the timestamp of the dm-read is smaller than W-timestamp(x), the dm-read is rejected and the issuing transaction is aborted. Similarly, to process a dm-write(x), the scheduler compares the timestamp of the dm-write to R-timestamp(x). If the former timestamp is larger, the dm-write is output and W-timestamp(x) is updated to the maximum of (a) the old W-timestamp(x), or (b) the timestamp of the dm-write. Otherwise, the dm-write is rejected and the transaction is aborted.

For ww synchronization, the T/O scheduler operates as follows. To process a dm-write(x), scheduler compares the timestamp of the dm-write to the W-timestamp(x). If the dm-write has a larger timestamp, the dm-write is output and W-timestamp(x) is set equal to the timestamp of the dm-write. Otherwise, the dm-write is rejected and the transaction is aborted.

When a transaction is aborted, it is assigned a larger timestamp by its TM and is restarted. This restart policy can lead to a cyclic restart situation, meaning that some transaction can be continually restarted without ever finishing. Cyclic restart can be avoided by assigning an especially large timestamp to the transaction, thereby reducing the probability of a subsequent restart. Other restart policies are discussed in later sections.

This implementation of T/O requires a substantial amount of storage for maintaining timestamps. Techniques for reducing this storage requirement are discussed in Section 4.8.

### 4.3 The Thomas Write Rule

For ww synchronization the basic T/O scheduler can be optimized using an observation of [Thomas 1,2]. Suppose the timestamp of a dm-write(x) is smaller than W-timestamp(x). Instead of rejecting the dm-write (and restarting the issuing transaction) we can simply ignore the dm-write. We call this the Thomas Write Rule (TWR).

Intuitively, TWR only applies to a dm-write that tries to put obsolete information into the database. The rule guarantees that the effect of applying a set of dm-writes to x is identical to what would have happened had the dm-writes been applied in timestamp order.

### 4.4 Multi-Version T/O

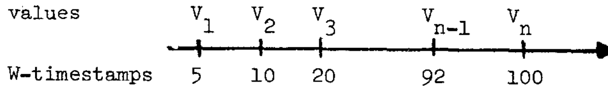
For rw synchronization the basic T/O scheduler can be improved by using the multi-version data item concept of [Reed]. For each data item x we maintain a set of R-timestamps, and a set of <W-timestamp, value> pairs (called versions). The R-timestamps of x record the timestamps of all dm-reads that have ever read x; the versions record the timestamps of all dm-writes that have ever written into x, along with the values written.

Using multi-versions, one can achieve rw synchronization without ever rejecting dm-reads. Consider a dm-read(x) with timestamp TS. To process this operation, we simply read the version(x) with largest timestamp less than TS; see fig. 4.1a. However, dm-writes can still be rejected. Consider a dm-write(x) with timestamp $TS_1$, and let $TS_2$* be the smallest W-timestamp(x) greater than $TS_1$; see fig. 4.1b. If any R-timestamp(x) lies between $TS_1$ and $TS_2$ then the dm-write is rejected. If no R-timestamp lies in that range, then the scheduler outputs the dm-write; this causes a new version of x to be created with timestamp $TS_1$.

To prove the correctness of this technique, consider a dm-read(x) with timestamp $TS_1$ that is processed "out of order"; i.e., suppose the dm-read(x) has timestamp $TS_1$ yet it is processed after some dm-write(x) with a larger timestamp $TS_2$. The dm-read ignores all versions(x) with timestamps

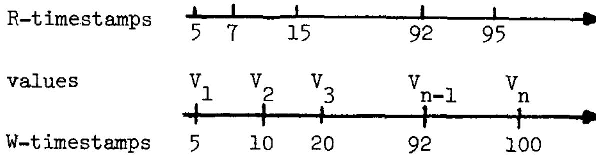
Figure 4.1 Multi-version Reading and Writing

a) Let us represent the versions of a data item x on a "time line"

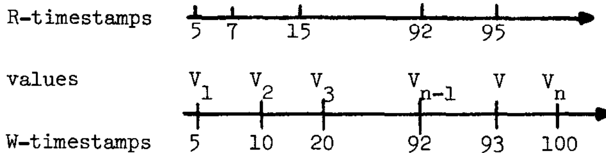



To process a dm-read(x) with timestamp 95, find the biggest W-timestamp less than 95; in this case 92. That is the version you read. So in this case, the value read by the dm-read is $V_{n-1}$.

b) Let us represent the R-timestamps of x similarly




To process a dm-write(x) with timestamp 93, we create a new version of x with that timestamps.




However, this new version "invalidates" the dm-read of part (a), because if the dm-read had arrived after the dm-write, it would have read value V instead of $V_{n-1}$. Therefore, we must reject the dm-write.

larger than $TS_1$; thus, the value read by the dm-read equals the value it would have read had it been processed "in order". Now consider a dm-write(x) that is processed "out of order". I.e., suppose the dm-write is processed after some dm-read with a larger timestamp $TS_2$. Since the dm-write was not rejected, there must exist a version(x) with timestamp $TS_1$ such that $TS_1 < TS_1 < TS_2$. Again the effect is identical to that of a timestamp ordered execution. Q.E.D.

Notice that the multi-version concept achieves ww synchronization "automatically"; insofar as ww synchronization is concerned, multi-versions are an embellished implementation of TWR.

It is usually not possible to keep all versions forever, so a technique for <u>forgetting</u> (i.e., deleting) versions is needed (see Section 4.8).

### 4.5 Conservative T/O

<u>Conservative timestamp ordering</u> is a technique for eliminating restarts during T/O scheduling [BP, BSR, HV, KNTH, SM1, SM2]. When a scheduler receives an operation O that might cause a future restart, the scheduler <u>delays</u> O until it is certain that no future restarts are possible.

Imagine that each T/O scheduler has a set of input queues, one <u>R-queue</u> and one <u>W-queue</u> per TM. Each R-queue (resp. W-queue) is a FIFO channel for transmitting dm-reads (resp. dm-writes) from one TM to one scheduler. In addition, each TM is required to place operations into any given queue in timestamp order.

This structure can be used for rw synchronization as follows. Suppose scheduler $S_j$ receives a dm-read(x) with timestamp TS. If $S_j$ outputs this dm-read "too early", subsequent dm-writes may have to be rejected. $S_j$ can avoid this possibility by scanning its W-queues and only outputting the dm-read if (a) every W-queue is non-empty, and (b) the first dm-write on each W-queue has timestamp greater than TS. This guarantees that $S_j$ will not output the dm-read until it has processed every dm-write with timestamp less than TS that $S_j$ will <u>ever</u> receive. To avoid the rejection of dm-reads, $S_j$ can use multi-version T/O, or it can delay the processing of dm-writes until it is has processed all dm-reads with smaller timestamps using an algorithm similar to the above.

For ww synchronization, the scheduler need only wait until every W-queue is nonempty and then output the dm-write with smallest timestamp. If conservative T/O is used for <u>both</u> rw and ww synchronization, the scheduler waits until every queue is nonempty and then outputs the operation with smallest timestamp.

The above implementation of conservative T/O suffers three major problems. First, the implementation does not guarantee termination -- if some TM never sends an operation to some scheduler, the scheduler will "get stuck" due to the empty queue and will never output any operations. Second, the implementation requires that all TMs communicate regularly with all schedulers -- this is infeasible in large networks. Third, the implementation is overly conservative -- e.g., the combined rw and ww algorithm <u>processes all operations in timestamp order</u>, not merely conflicting operation. The first two problems are addressed below. The third is considered in Section 4.6.

<u>Guaranteeing Termination -- Null operations</u>

To guarantee termination, we require that TMs periodically send timestamped <u>null-operations</u> to each scheduler, in the absence of any "real" traffic. A null-operation is a dm-read or dm-write that does not reference a data item. When $TM_i$ sends a null-dm-read (resp. null-dm-write) with timestamp TS to

scheduler $S_j$, this signifies that $TM_i$ will not send $S_j$ any more dm-reads (resp. dm-writes) with timestamps smaller than TS. Thus, any scheduling decision requiring that $S_j$ receive all dm-reads (resp. dm-writes) from $TM_i$ timestamped less than TS can be made after that null-dm-read (resp. null-dm-write) is received. An impatient scheduler can prompt a TM for a null-operation by sending a <u>request-null</u> operation to it.

<u>Avoiding Unnecessary Communication</u>

To avoid unnecessary communication between TMs and schedulers, null-operations with <u>very large</u> timestamps can be used. In extreme cases, $TM_i$ can send $S_j$ a null-operation with <u>infinite</u> timestamp, signifying that $TM_i$ does not intend to communicate with $S_j$ until further notice. Of course, when $TM_i$ needs to send a "real" operation to $S_j$, some mechanism is required to retract the infinite timestamp and replace it by a finite one.

### 4.6 Conservative T/O with Transaction Classes

Another technique for reducing communication is transaction classes [BRGP]. Here, we assume that the readset and writeset of every transaction is known in advance. This information is used to group transactions into predefined <u>classes</u>. Class definitions help support a less conservative scheduling policy.

A <u>transaction class</u> is defined by a readset and writeset (see fig. 4.2). Transaction T is a <u>member</u>

Figure 4.2 Transaction Classes

•A class is defined by a readset and a writeset. E.g.,

$C_1$: readset = $\{x_1\}$ , writeset = $\{y_1,y_2\}$

$C_2$: readset = $\{x_1,y_2\}$ , writeset = $\{y_1,y_2,z_2,z_3\}$

$C_3$: readset = $\{y_2, z_3\}$ , writeset = $\{x_1,z_2,z_3\}$

•A transaction is a member of a class if its readset is a subset of the class readset and its writeset is a subset of the class writeset. E.g.,

$T_1$: readset = $\{x_1\}$ , writeset = $\{y_1, y_2\}$

$T_2$: readset = $\{y_2\}$ , writeset = $\{z_2, z_3\}$

$T_3$: readset = $\{z_3\}$ , writeset = $\{x_1\}$

•$T_1$ is a member of $C_1$ and $C_2$

•$T_2$ is a member of $C_2$ and $C_3$

•$T_3$ is a member of $C_3$

of class C iff T's readset is a subset of C's readset, and T's writeset is a subset of C's writeset. (Classes need not be disjoint.) Class definitions are not expected to change frequently during normal operation of the system. Changing a class definition is akin to changing the database schema and requires mechanisms beyond the scope of this paper. We assume that class definitions are stored in static tables which are available at any site requiring them.

Classes are associated with TMs. Every transaction that executes at a TM must be a member of a class associated with the TM. If a transaction is submitted to a TM at which this property does not hold, the transaction is forwarded to another TM that has an appropriate class. We assume that every class is associated with exactly one TM, and conversely, every TM is associated with exactly one class. We use $C_i$ to denote the class associated with $TM_i$. This notation simplifies our discussion, but does not constrain system operation in any way. For example, to execute transactions that are members of class $C_1$ at two TMs, we define another class $C_2$ with the same readset and writeset as $C_1$ and associate $C_1$ with one TM and $C_2$ with the other. On the other hand, to execute transactions that are members of two classes at one site, we multiprogram two TMs at the same site.

Transaction classes are exploited by conservative T/O schedulers as follows. Consider rw synchronization and suppose scheduler $S_j$ wants to output a dm-read(x) with timestamp TS. Instead of waiting for dm-writes with smaller timestamp from all TMs, $S_j$ need only wait for dm-writes <u>from those TMs whose class writeset contains x</u>. Similarly, to process a dm-write(x) with timestamp TS, $S_j$ need only wait for dm-reads with smaller timestamp from those TMs whose class readset contains x. Thus, the level of concurrency in the system is increased. ww synchronization proceeds analogously.

This technique also reduces communication requirements, since a TM need only communicate with a scheduler if its class readset or writeset contains data items protected by the scheduler.

### 4.7 Conservative T/O with Conflict Graph Analysis

<u>Conflict graph analysis</u> is a technique for further improving the performance of conservative T/O with classes. A <u>conflict graph</u> is an undirected graph that summarizes potential conflicts between transactions in different classes. For each class $C_i$ the graph contains two nodes, denoted $r_i$ and $w_i$, which intuitively represent the readset and writeset of $C_i$. The edges of the graph are defined as follows (see fig. 4.3). (i) For each class $C_i$, there is a <u>vertical edge</u> between $r_i$ and $w_i$; (ii) For each pair of classes $C_i$ and $C_j$ (with $i \neq j$) there

is a horizontal edge between $w_i$ and $w_j$ iff the writeset of $C_i$ intersects the writeset of $C_j$. (iii) For each pair of classes $C_i$ and $C_j$ (with $i \neq j$) there is a diagonal edge between $r_i$ and $w_j$ iff the readset of $C_i$ intersects the writeset of $C_j$.

Intuitively, a horizontal edge indicates that a scheduler $S_k$ may be forced to delay dm-writes for purposes of ww synchronization. Suppose classes $C_i$ and $C_j$ are connected by a horizontal edge (i.e., there is an edge between $w_i$ and $w_j$). Then the class writesets intersect and so, if $S_k$ receives a dm-write from $C_i$, $S_k$ must delay the dm-write until $S_k$ receives all dm-writes with smaller timestamps from $C_j$. Similarly, a diagonal edge indicates that $S_k$ may need to delay operations for rw synchronization.

Conflict graph analysis improves the situation by identifying inter-class conflicts that never cause non-serializable behavior. This corresponds to identifying horizontal and diagonal edges that do not require synchronization. In particular, schedulers need only synchronize dm-writes from $C_i$ and $C_j$ if either (1) the edge $(w_i, w_j)$ is embedded in a cycle of the conflict graph; or (2) portions of the intersection of $C_i$'s writeset and $C_j$'s writeset are stored at two or more DMs[BS]. That is, if conditions (1) and (2) do not hold, a scheduler $S_k$ need not process dm-writes from $C_i$ and $C_j$ in timestamp order. Similarly, dm-reads from $C_i$ and dm-writes from $C_j$ need only be processed in timestamp order if either (1) the edge $(r_i, w_j)$ is embedded in a cycle of the conflict graph; or (2) portions of the intersection of $C_i$'s readset and $C_j$'s writeset are stored at two or more DMs[BS].

Since classes are defined statically, conflict graph analysis is also performed statically. The output of this analysis is a table indicating which horizontal and vertical edges require synchronization and which do not. This information, like class definitions, is distributed in advance to all schedulers that require it.

Conservative T/O with conflict graph analysis has been implemented in the SDD-1 DDBMS [BSR]. In principle, conflict graph analysis can be applied to other synchronization techniques to improve their performance as well. Theoretical aspects of this integration are examined in [BSW], but many details remain to be worked out.

### 4.8 Timestamp Management

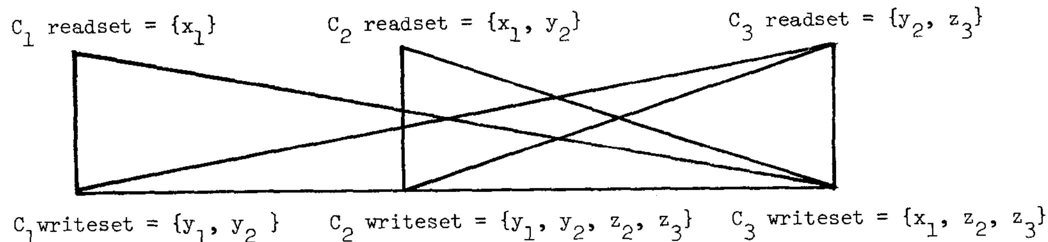
A common criticism of T/O schedulers is that too much memory is needed to store timestamps. This problem can be overcome by "forgetting" old timestamps.

Timestamps are used in basic T/O to reject operations that "arrive late", e.g., to reject a dm-read(x) with timestamp $TS_1$ that arrives after a dm-write(x) with timestamp $TS_2 > TS_1$. In principle, $TS_1$ and $TS_2$ can differ by an arbitrary amount, but in practice these timestamps are unlikely to differ by more than a few minutes. Consequently we may store timestamps in small tables which are periodically purged.

R-timestamps are stored in the R-table with entries of the form <x, R-timestamp>; for any data item x, there is at most one entry. In addition, there is a variable, R-min, which tells the maximum value of any timestamp that has been purged from the table. To find R-timestamp(x), a scheduler searches the R-table for an <x, TS> entry. If such an entry is found, TS = R-timestamp(x); otherwise, R-timestamp(x) < R-min and to err on the side of safety, the scheduler assumes R-timestamp(x) = R-

Figure 4.3 Conflict Graph

Define $C_1$, $C_2$, $C_3$ as in figure 4.2



min. To update R-timestamp(x), the scheduler modifies the <x, TS> entry, if one exists; otherwise, a new entry is created and added to the table. When the R-table is full, the scheduler selects an appropriate value for R-min and deletes all entries from the table with smaller timestamp. W-timestamps are managed similarly; analogous techniques can be devised for multi-version databases.

Maintaining timestamps for conservative T/O is even cheaper, since conservative T/O only requires timestamped operations, not timestamped data. If conservative T/O is used for rw synchronization, the R-timestamps of data items are rendered useless and may be discarded. If conservative T/O is used for both rw and ww synchronization, W-timestamps can be eliminated too.

### 4.9 Integrating Two-Phase Commit into T/O

It is necessary to integrate two-phase commit into the T/O implementations described above to ensure atomic commitment of updates (see Section 2). This is done by timestamping pre-commits and modifying the T/O implementations to accept or reject pre-commits instead of dm-writes. If a scheduler rejects a pre-commit, the issuing transaction is aborted. However, if a scheduler accepts a pre-commit, it must accept the corresponding dm-write no matter when that operation arrives. To make this guarantee, the scheduler may be forced to delay conflicting operations that arrive before the dm-write.

#### Integrating Two-Phase Commit Into Basic T/O

Consider a pre-commit(x) with timestamp TS. Let P denote this operation and let W denote the corresponding dm-write. Assume that basic T/O is used for rw synchronization. P can be accepted by a scheduler iff TS > R-timestamp(x); i.e., P is accepted iff the scheduler can still output W. Once the scheduler accepts P, it must guarantee that TS will remain greater than R-timestamp(x) until W is received. To make this guarantee, the scheduler refuses to output any dm-read(x) with timestamp greater than TS, until W is received. All such dm-reads that arrive before W are placed on a waiting queue.

For ww synchronization, P is accepted by the scheduler iff TS > W-timestamp(x). Once the scheduler accepts P, it agrees not to output any dm-write(x) with timestamp greater than TS until it receives W. All such dm-writes that arrive before W are placed on a waiting queue as above.

#### Integrating Two-Phase Commit Into Thomas Write Rule

TWR applies only to ww synchronization and eliminates the possibility of rejecting dm-writes for purposes of ww synchronization. Hence there is no need to incorporate two-phase commit into the ww synchronization algorithm. Pre-commits must still be sent to all sites being updated, but the pre-commits need not be processed by the ww scheduler.

#### Integrating Two-Phase Commit Into Multi-Version T/O

Like TWR, multi-versions eliminate the need for two-phase commit insofar as ww synchronization is concerned. However, two-phase commit remains as issue for rw synchronization.

Let P be a pre-commit(x) with timestamp $TS_1$ and let W be the corresponding dm-write. When P arrives at a scheduler, the scheduling rule of Section 4.4 is applied: let $TS_2$ be the smallest W-timestamp(x) > $TS_1$; if any R-timestamp(x) lies between $TS_1$ and $TS_2$, P is rejected, otherwise P is accepted. If the scheduler accepts P, it agrees not to output any dm-read(x) with timestamp between $TS_1$ and $TS_2$ until W is received. As before, all such dm-reads that arrive before W are placed on a waiting queue.

#### Integrating Two-Phase Commit Into Conservative T/O

Two-phase commit need not be tightly integrated into conservative T/O, because dm-writes are never rejected. However, scheduling delay can be reduced by transmitting pre-commits via W-queues. For example, suppose conservative T/O is used for rw synchronization, and suppose scheduler $S_j$ wants to output a dm-read(x) with timestamp TS. $S_j$ need only delay this dm-read until each W-queue contains a pre-commit with timestamp greater than TS; it need not wait for the corresponding dm-writes. (However, the dm-read may have to wait for some dm-writes with smaller timestamp; i.e., if $S_j$ has accepted a pre-commit(x) with timestamp $TS' < TS$, the dm-read cannot be output until the dm-write(x) with timestamp $TS'$ is received.)

### 4.10 Heuristics for Reducing Restarts

This section describes three heuristics for reducing the cost or probability of restarts for non-conservative T/O implementations.

#### Predeclaration of Readsets and Writesets

To reduce the cost of restarts, transactions should issue their dm-reads and pre-commits as early as possible. The extreme version of this heuristic calls for transactions to predeclare their readsets and writesets, so that dm-reads and pre-commits are issued for the entire readset and writeset before a transaction begins its main execution. If no operation is rejected, the transaction is guaranteed to execute with no danger of restart.

#### Delaying of Operations

To reduce the probability of restart, a scheduler can delay the processing of operations to wait for "earlier" operations (i.e., ones with smaller timestamps) to arrive. This heuristic is essentially a compromise between conservative and non-conservative T/O, and trades response time for a

reduction in restart probability. The amount of delay can be tuned to optimize this trade-off.

Reading Old Versions

The performance of multi-version T/O can be improved by permitting queries (i.e., read-only transactions) to read old versions of data items. Recall that in multi-version T/O, dm-read operations are never rejected but may cause subsequent pre-commits to be rejected. (E.g., once dm-read(x) with timestamp TS is processed, a subsequent pre-commit(x) with timestamp TS′, where TS′ < TS, may be rejected.) To reduce the probability of rejecting a pre-commit, we may assign old (i.e. small) timestamps to queries. Of course, this also causes the query to read older data. Thus, this technique entails a compromise between system performance and timeliness of data. Little is known about this tradeoff in general, but a good compromise should often be achievable. For example, if queries are assigned timestamps that are five minutes old, we would expect few queries to interfere with updates. And in many applications, five minute old data is perfectly acceptable.

As a fringe benefit, this technique also improves the response time for queries by reducing the probability that a query′s dm-reads will be blocked by pre-commits.

## 5. Integrated T/O Concurrency Control Methods

The synchronization techniques of Section 4 can be integrated to form twelve principal T/O concurrency control methods:

| # | rw technique | ww technique |
|---|---|---|
| 1 | basic T/O | basic T/O |
| 2 | basic T/O | Thomas Write Rule (TWR) |
| 3 | basic T/O | multi-version T/O |
| 4 | basic T/O | conservative T/O |
| 5 | multi-version T/O | basic T/O |
| 6 | multi-version T/O | TWR |
| 7 | multi-version T/O | multi-version T/O |
| 8 | multi-version T/O | conservative T/O |
| 9 | conservative T/O | basic T/O |
| 10 | conservative T/O | TWR |
| 11 | conservative T/O | multi-version T/O |
| 12 | conservative T/O | conservative T/O |

Each T/O method that incorporates a non-conservative component can be further refined by including (1) techniques for forgetting timestamps (see Section 4.8) and (2) heuristics for reducing restarts (see Section 4.10). Each method that incorporates a conservative component may also incorporate classes (see Section 4.6) and conflict graph analysis (see Section 4.7). Thus, these 12 principal methods produce over 50 distinct methods. In this section we describe the twelve principal methods in detail.

### 5.1 Using Basic T/O for rw Synchronization

Methods 1-4 use basic T/O for rw synchronization. Each stored data item e.g. $x_i$, has an R-timestamp and a W-timestamp. Let T be a transaction with timestamp TS. To read $x_i$, T issues a dm-read($x_i$) with timestamp TS; this dm-read is accepted iff TS > W-timestamp($x_i$). To write $x_i$, T issues a pre-commit($x_i$) with timestamp TS; this pre-commit is accepted iff (a) TS > R-timestamp ($x_i$), and (b) a condition determined by the ww synchronization technique is also satisfied.

Method 1 -- Basic T/O for ww synchronization. The pre-commit is accepted iff TS > R-timestamp ($x_i$) and TS > W-timestamp ($x_i$.)

Method 2 -- TWR for ww synchronization. The pre-commit is accepted iff TS > the largest R-timestamp($x_i$). However, if the pre-commit is accepted and TS < the W-timestamp($x_i$), the corresponding dm-write has no effect on the database. This method represents an optimization of Method 1 that is apparently preferable in most situations.

Method 3 -- Multi-version T/O for ww synchronization. The pre-commit is accepted iff TS > R-timestamp($x_i$); the W-timestamp is irrelevant. If the pre-commit is accepted, the corresponding dm-write creates a new version of $x_i$. While this method appears to be a space-inefficient version of Method 2, it can yield better performance by letting queries read old versions of data items; see Section 4.10.

Method 4 -- Conservative T/O for ww synchronization. Pre-commits are processed by each scheduler in timestamp order. I.e., a scheduler S will not process a pre-commit with timestamp TS until it has processed all pre-commits with smaller timestamp. When S processes a pre-commit($x_i$) with timestamp TS, it accepts the pre-commit iff TS > R-timestamp($x_i$). At first glance this method appears to be a time-inefficient version of Method 2. However, unlike Method 2, this method applies updates to each DM in timestamp order. Consequently, the database at each DM is always consistent between updates, a property which may be useful for reliability reasons.

### 5.2 Using Multi-version T/O for rw Synchronization

Methods 5-8 use multi-version T/O for rw synchronization. Let T be a transaction with timestamp TS. To read $x_i$, T issues a dm-read($x_i$) with timestamp TS; this dm-read is always accepted. To write $x_i$, T issues a pre-commit($x_i$) with timestamp TS; this pre-commit is accepted iff (a) there is no R-timestamp($x_i$) that lies between TS and the smallest

W-timestamp($x_i$) larger than TS, and (b) a condition determined by the ww synchronization technique is also satisfied.

Method 5 -- Basic T/O for ww synchronization. For basic T/O, condition (b) requires that TS be greater-than the largest W-timestamp($x_i$). So, for Method 5, conditions (a) and (b) may be simplified: The pre-commit is accepted iff TS > largest R-timestamp($x_i$) and the largest W-timestamp($x_i$). If the pre-commit is accepted, the corresponding dm-write creates a new version of $x_i$.

Method 6 -- TWR for ww synchronization. This method is incorrect. TWR requires that a dm-write($x_i$) with timestamp TS be ignored if TS < the maximum W-timestamp($x_i$). This may cause subsequent dm-reads to read inconsistent data; see fig. 5.1. (Method 6 is the only incorrect method we will encounter.)

Method 7 -- Multi-version T/O for ww synchronization. This achieves the goals of TWR in conjunction with multi-version rw synchronization. The pre-commit is accepted iff condition (a) holds. If the pre-commit is accepted, the corresponding dm-write creates a new version of $x_i$. This method is similar to the algorithms of [Reed, Montgomery].

Method 8 -- Conservative T/O for ww synchronization. A scheduler S will not process a pre-commit with timestamp TS until it has processed all pre-commits with smaller timestamps, and none with larger timestamps. This permits us to simplify the condition for acceptance of a pre-commit: A pre-commit($x_i$) with timestamp TS is accepted iff TS is greater than the largest R-timestamp($x_i$).

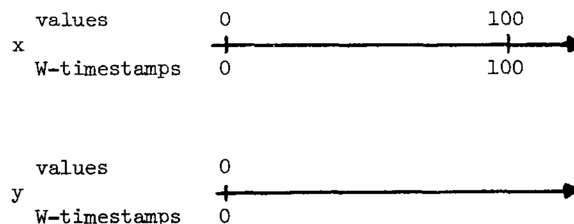
## Systematic Forgetting of Old Version

In Methods 5 and 8, the versions of each data item $x_i$ are created in timestamp order. That is, once a version of $x_i$ has been created with timestamp TS, no subsequent transaction can create a version with a smaller timestamp. When this property holds, it is possible to forget (i.e., delete) old versions such that we never delete a version needed by a later transaction.

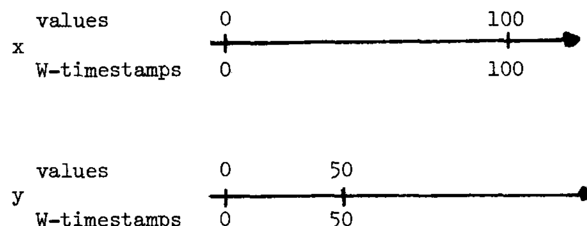
Let W-max($x_i$) be the maximum W-timestamp($x_i$) and W-min be the minimum value of W-max($x_i$) over all data items $x_i$. Observe that no pre-commit with timestamp smaller than W-min can be accepted in the future: since W-min < W-max($x_i$) for all $x_i$, all future update transactions with timestamps less than W-min are guaranteed to be restarted. So, insofar as update transactions are concerned, we can safely forget all versions of every data item timestamped less than W-min. Queries are handled in this framework by interpreting all dm-reads with timestamps less than W-min as if they had timestamps equal to W-min.

Figure 5.1 Inconsistent Retrievals in Method 6

• Consider data items x and y with the following versions




• Now suppose T has timestamp 50 and writes x:=50, y:50. Under Method 6, the update to x is ignored, and the result is




• Finally, suppose T' has timestamp 75 and reads x and y. The values it will read are x=0, y=50 which is incorrect. T' should read x=50, y=50.

Notice also that Methods 5 and 8 only require that the largest R-timestamp of each data item be stored. Smaller R-timestamps may be forgotten at once.

## Systematic Reading of Old Versions

Methods 5 and 8 also support a systematic technique for assigning old timestamps to queries (see Section 4.10) so that (a) no dm-read issued by a query will ever cause a pre-commit to be rejected; and (b) the timestamp assigned to the query is the largest one satisfying (a). This technique is similar to the technique for systematic forgetting of old versions.

Let Q be a query. The technique we describe requires that Q's readset be predeclared. Before Q begins its main execution Q's readset is examined; for each $x_i$ in the readset, W-max($x_i$) is ascertained. In addition, we calculate W-min = min{W-max($x_i$)| $x_i$ is in Q's readset}. The timestamp assigned to Q is W-min - 1. The correctness of this technique is shown in [BG2].

## 5.3 Using Conservative T/O for rw Synchronization

The remaining T/O methods use conservative T/O for

rw synchronization. In these methods, a scheduler S will not process a dm-read($x_i$) with timestamp TS until it has processed all pre-commits with smaller timestamps and none with larger timestamps. Symmetrically, S will not process a pre-commit($x_i$) with timestamp TS until it has processed all dm-reads with smaller timestamps and none with larger timestamps. When S processes a pre-commit($x_i$) with timestamp TS, its action depends on the ww technique.

Method 9 -- Basic T/O for ww synchronization. The pre-commit is accepted iff TS > W-timestamp($x_i$).

Method 10 -- TWR for ww synchronization. The pre-commit is always accepted. However, if TS < W-timestamp($x_i$), the corresponding dm-write has no effect on the database.

Method 10 is essentially the concurrency control of SDD-1 [BSR]. In SDD-1, however, the method is refined in several ways to reduce delay. First, SDD-1 uses classes and conflict graph analysis and requires predeclaration of readsets. In addition, SDD-1 only enforces the conservative scheduling rule on dm-reads, meaning that dm-reads wait for pre-commits, but pre-commits need not wait for all dm-reads with smaller timestamps. Consequently, it is possible for dm-reads to be rejected in SDD-1. The SDD-1 designers accepted this possibility for two reasons: (1) since readsets are predeclared, all dm-reads are issued before the transaction begins its main execution and the cost of rejecting a dm-read is modest. (2) The probability that a dm-read will be rejected can be reduced by assigning large timestamps to transactions. Other techniques for reducing restarts are described by [Lin].

Method 11 -- Multi-version T/O for ww synchronization. The pre-commit is always accepted and the corresponding dm-write always creates a new version of $x_i$. When multi-versions are used, the conservative rw technique can be optimized as follows: a dm-read can never be rejected, and so there is no reason to force pre-commits to wait for dm-reads. (dm-reads must still wait for pre-commits to ensure that pre-commits are never rejected.)

Method 12 -- Conservative T/O for ww synchronization. Scheduler S will not process a pre-commit with timestamp TS until it has processed all pre-commits with smaller timestamps and none with larger timestamps. Combined with conservative rw synchronization, the effect is to process all operations in timestamp order. Method 12 has been recommended by [BP, HV, KNTH, SM1, SM2].

## 6. Conclusion

We have presented a framework for DDBMS concurrency control and have used that framework to describe a number of DDBMS concurrency control methods. The framework has two main parts: (1) a model of distributed transaction execution, in which transactions execute by issuing dm-read, pre-commit, and dm-write operations; and (2) a decomposition of the concurrency control problem into the sub-problems of rw and ww synchronization.

We presented several timestamp-based synch ronization techniques for solving each sub-problem. Four of these techniques were deemed to be "principal": basic T/O, the Thomas Write Rule, multi-version T/O, and conservative T/O. These techniques vary substantially in their behavior but are united by a common underlying objective: each technique seeks to execute conflicting operations in timestamp order, or in some equivalent order. Basic T/O achieves this objective by rejecting operations that are received out of timestamp order. The Thomas Write Rule ignores operations that are received out of timestamp order. (This technique is only suitable for ww synchronization.) Multi-version T/O retains multiple "versions" of data items to permit many operations that are received out of order to be executed as if they had been received in order. And conservative T/O delays operations that are received out of order to permit all operations with smaller timestamps to be processed first.

Finally we showed how to integrate any principal rw technique with any principal ww technique to yield a principal concurrency control method. Twelve principal methods can be constructed in this way. Each principal method can be refined by several non-principal techniques so that more than 50 distinct concurrency control algorithms can be built using the framework and material of this paper.

Most of the principal methods we describe are new algorithms. These are Methods 1-4 (which use basic T/O for rw synchronization); Methods 5 and 8 (multi-version T/O for rw, with basic T/O or conservative T/O for ww); and Methods 9 and 11 (conservative T/O for rw, with basic T/O or multi-version T/O for ww). Of the remaining methods, Method 6 (multi-version T/O with TWR) is an incorrect method; Method 7 (multi-version T/O for rw and ww) is similar but not identical to the algorithms of [Montgomery, Reed]; Method 10 (conservative T/O with TWR) is essentially the SDD-1 concurrency control algorithm [BSR]; and Method 12 (conservative T/O for rw and ww) is essentially the algorithm recommended by [BP, HV, KNTH, SM1,2].

A major issue we have not addressed concerns the performance of these algorithms. This issue is addresssed qualitatively in [BG2]. However, little quantitative performance analysis has been reported in the literature and this remains a topic for future research.

---

* The term "two-phase commit" is commonly used to denote the distributed version of this procedure. However, since the centralized and distributed versions are identical in structure, we use "two-phase commit" to describe both.

* $TS_2$ equals infinity if $TS_1$ is the largest W-timestamp(x).

## References


[BG1]
Bernstein, P.A., and Goodman, N., "Approaches to Concurrency Control in Distributed Databases", Proc. 1979 National Computer Conf., June 1979.

[BG2]
Bernstein, P.A., and Goodman, N., "Fundamental Algorithms for Concurrency Control in Distributed Database Systems", Tech. Rep., Computer Corp. of Am., Feb. 1980.

[BP]
Badal, D.Z.; and Popek, G.J. "A Proposal for Distributed Concurrency Control for Partially Redundant Distributed Data Base System," Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks, 1978, pp. 273-288.

[BS]
Bernstein P.A. and Shipman D.W., "The Correctness of Concurrency Control Mechanisms in a System for Distributed Databases (SDD-1)", ACM Trans. on Database Sys., Vol. 5, No. 1, March 1980.

[BSR]
Bernstein P., Shipman D., and Rothnie J., "Concurrency Control in a System for Distributed Databases (SDD-1)", ACM Trans. on Database Syst., Vol. 5, No. 1, March 1980.

[BSW]
Bernstein, P. A., Shipman D. W. and Wong, W. S., "Formal Aspects of Serializability in Database Concurrency Control", IEEE Trans. on Software Engineering, Vol. SE-5, No. 3, May 1979.

[GBWRR]
Goodman, N., P.A. Bernstein, E. Wong, C.L. Reeve, and J.B. Rothnie, "Query Processing in SDD-1", Tech. Rep. 79-06, Computer Corp. of Am., Oct. 1979

[Gray]
Gray, J. N. Notes on Database Operating Systems, unpublished lecture notes. IBM San Jose Research Laboratory, San Jose, Calif., 1977.

[HS1]
Hammer, M. M. and Shipman, D. W., "An Overview of Reliability Mechanisms for a Distributed Data Base System", Proc. 1977 COMPCON, IEEE, N.Y.

[HS2]
Hammer, M.M., and Shipman, D.W., "Reliability Mechanisms for SDD-1", Tech. Rep. 79-05, Computer Corp. of Am., July 1979.

[HV]
Herman, D. and J.P. Verjus, "An Algorithm for Maintaining the Consistencyof Multiple Copies", Proc. First International Conf. on Distributed Computing Systems, IEEE, N.Y., pp. 625-631.

[KNTH]
Kaneko, A., Y.Nishihara, K. Tsuruoka, and M. Hattori, "Logical Clock Synchronization Method for Duplicated Database Control", Proc. First International Conf. on Distributed Computing Systems, IEEE, N.Y., Oct. 1979, pp. 601-611.

[LS]
Lampson, B. and Sturgis, H., "Crash Recovery in a Distributed Data Storage System", Tech. Rep., Computer Science Lab., Xerox Palo Alto Research Center, 1976.

[Lin]
Lin, W. K., "Concurrency Control in a Multiple Copy Distributed Data Base System", Proc. 4th Berkeley Work. on Distributed Data Management & Computer Networks, August 1979.

[Montgomery]
Montgomery, W.A., "Robust Concurrency Control for a Distributed Information System", Ph.D. dissertation, Laboratory for Computer Science, MIT, Dec. 1978.

[PBR]
Papadimitriou, C. H., Bernstein, P. A. and Rothnie, J. B., Jr., "Some Computational Problems Related to Database Concurrency Control," Proc. Conf. on Theoretical Computer Science, Waterloo, Ontario, August 1977.

[Papadimitriou]
Papadimitriou, C. H., "Serializability of Concurrent Updates", J. of the ACM, Vol. 26, No. 4, Oct. 1979, pp. 631-653.

[Reed]
Reed, D.P., Naming and Synchronization in a Decentralized Computer System, Ph.D. Thesis, M.I.T. Department of Electrical Engineering, Sept. 1978.

[SM1]
Shapiro, R.M. and Millstein, R.E., "Reliability and Fault Recovery in Distributed Processing", Oceans '77 Conference Record, Vol. II, Los Angeles, 1977.

[SM2]
Shapiro, R.M. and Millstein, R.E., NSW Reliability Plan, Mass. Computer Associates, Tech. Rep. 7701-1411, June 1977.

[SLR]
Stearns, R.E., Lewis, P.M. II and Rosenkrantz, D.J., "Concurrency Controls for Database Systems", Proc. 17th Symp. on Found. of Computer Science, IEEE, 1976, pp. 19-32.

[Thomas1]
Thomas, R.H., "A Solution to the Concurrency Control Problem for Multiple Copy Databases", Proc. 1978 COMPCON Conference., IEEE, N.Y.

[Thomas2]
Thomas, R.H., "A Majority Consensous Approach to Concurrency Control for Multiple Copy Databases", ACM Trans. on Database Syst., Vol. 4, No. 2, June 1979, pp. 180-209.