Concurrency Control in Distributed Database Systems

PHILIP A. BERNSTEIN AND NATHAN GOODMAN

Computer Corporation of America, Cambridge, Massachusetts 02139

In this paper we survey, consolidate, and present the state of the art in distributed database concurrency control. The heart of our analysis is a decomposition of the concurrency control problem into two major subproblems: read-write and write-write synchronization. We describe a series of synchronization techniques for solving each subproblem and show how to combine these techniques into algorithms for solving the entire concurrency control problem. Such algorithms are called "concurrency control methods." We describe 48 principal methods, including all practical algorithms that have appeared in the literature plus several new ones. We concentrate on the structure and correctness of concurrency control algorithms. Issues of performance are given only secondary treatment.

Keywords and Phrases: concurrency control, deadlock, distributed database management systems, locking, serializability, synchronization, timestamp ordering, timestamps, two-phase commit, two-phase locking

CR Categories: 4.33, 4.35

INTRODUCTION

The Concurrency Control Problem

Concurrency control is the activity of coordinating concurrent accesses to a database in a multiuser database management system (DBMS). Concurrency control permits users to access a database in a multiprogrammed fashion while preserving the illusion that each user is executing alone on a dedicated system. The main technical difficulty in attaining this goal is to prevent database updates performed by one user from interfering with database retrievals and updates performed by another. The concurrency control problem is exacerbated in a distributed DBMS (DDBMS) because (1) users may access data stored in many different computers in a distributed system, and (2) a concurrency control mechanism at one computer cannot instantaneously know about interactions at other computers.

Concurrency control has been actively investigated for the past several years, and the problem for nondistributed DBMSs is well understood. A broad mathematical theory has been developed to analyze the problem, and one approach, called *two-phase locking*, has been accepted as a standard solution. Current research on nondistributed concurrency control is focused on evolutionary improvements to two-phase locking, detailed performance analysis and optimization, and extensions to the mathematical theory.

Distributed concurrency control, by contrast, is in a state of extreme turbulence. More than 20 concurrency control algorithms have been proposed for DDBMSs, and several have been, or are being, implemented. These algorithms are usually complex, hard to understand, and difficult to prove correct (indeed, many are incorrect). Because they are described in different terminologies and make different assumptions

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1981 ACM 0010-4892/81/0600-0185 $00.75

المنارة للاستشارات

CONTENTS

about the underlying DDBMS environment, it is difficult to compare the many proposed algorithms, even in qualitative terms. Naturally each author proclaims his or her approach as best, but there is little compelling evidence to support the claims.

To survey the state of the art, we introduce a standard terminology for describing DDBMS concurrency control algorithms and a standard model for the DDBMS environment. For analysis purposes we decompose the concurrency control problem into two major subproblems, called read-write and write-write synchronization. Every concurrency control algorithm must include a subalgorithm to solve each subproblem. The first step toward understanding a concurrency control algorithm is to isolate the subalgorithm employed for each subproblem.

After studying the large number of proposed algorithms, we find that they are compositions of only a few subalgorithms. In fact, the subalgorithms used by all practical DDBMS concurrency control algorithms are variations of just two basic techniques: two-phase locking and timestamp ordering; thus the state of the art is far more coherent than a review of the literature would seem to indicate.

Examples of Concurrency Control Anomalies

The goal of concurrency control is to prevent interference among users who are simultaneously accessing a database. Let us illustrate the problem by presenting two "canonical" examples of interuser interference. Both are examples of an on-line electronic funds transfer system accessed via remote *automated teller machines* (*ATMs*). In response to customer requests, ATMs retrieve data from a database, perform computations, and store results back into the database.

Anomaly 1: Lost Updates. Suppose two customers simultaneously try to deposit money into the same account. In the absence of concurrency control, these two activities could interfere (see Figure 1). The two ATMs handling the two customers could read the account balance at approximately the same time, compute new balances in parallel, and then store the new balances back into the database. The net effect is incorrect: although two customers deposited money, the database only reflects one activity; the other deposit is lost by the system.

Anomaly 2: Inconsistent Retrievals. Suppose two customers simultaneously execute the following transactions.

Customer 1:	Move $1,000,000 from Acme Corporation's savings account to its checking account.
Customer 2:	Print Acme Corporation's total balance in savings and checking.

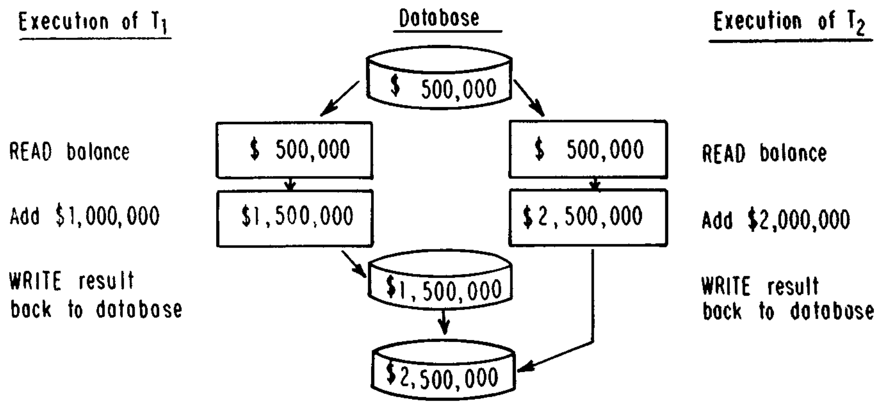


Figure 1. Lost update anomaly.

In the absence of concurrency control these two transactions could interfere (see Figure 2). The first transaction might read the savings account balance, subtract $1,000,000, and store the result back in the database. Then the second transaction might read the savings and checking account balances and print the total. Then the first transaction might finish the funds transfer by reading the checking account balance, adding $1,000,000, and finally storing the result in the database. Unlike Anomaly 1, the final values placed into the database by this execution are correct. Still, the execution is incorrect because the balance printed by Customer 2 is $1,000,000 short.

These two examples do not exhaust all possible ways in which concurrent users can interfere. However, these examples are typical of the concurrency control problems that arise in DBMSs.

Comparison to Mutual Exclusion Problems

The problem of database concurrency control is similar in some respects to that of mutual exclusion in operating systems. The latter problem is concerned with coordinating access by concurrent processes to system resources such as memory, I/O devices, and CPU. Many solution techniques have been developed, including locks, semaphores, monitors, and serializers [BRIN73, DIJK71, HEWI74, HOAR74].

The concurrency control and mutual exclusion problems are similar in that both are concerned with controlling concurrent access to shared resources. However, control schemes that work for one do not necessarily work for the other, as illustrated by the following example. Suppose processes P_1 and P_2 require access to resources R_1 and R_2 at different points in their execution. In an operating system, the following interleaved execution of these processes is perfectly acceptable: P_1 uses R_1, P_2 uses R_1, P_2 uses R_2, P_1 uses R_2. In a database, however, this execution is not always acceptable. Assume, for example, that P_2 transfers funds by debiting one account (R_1), then crediting another (R_2). If P_2 checks both balances, it will see R_1 after it has been debited, but see R_2 *before* it has been credited. Other differences between concurrency control and mutual exclusion are discussed in CHAM74.

1. 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 user interactions. Later we explain how concurrency control algorithms operate in the context of this model.

1.1 Preliminary Definitions and DDBMS Architecture

A distributed database management system (DDBMS) is a collection of sites interconnected by a network [DEPP76,

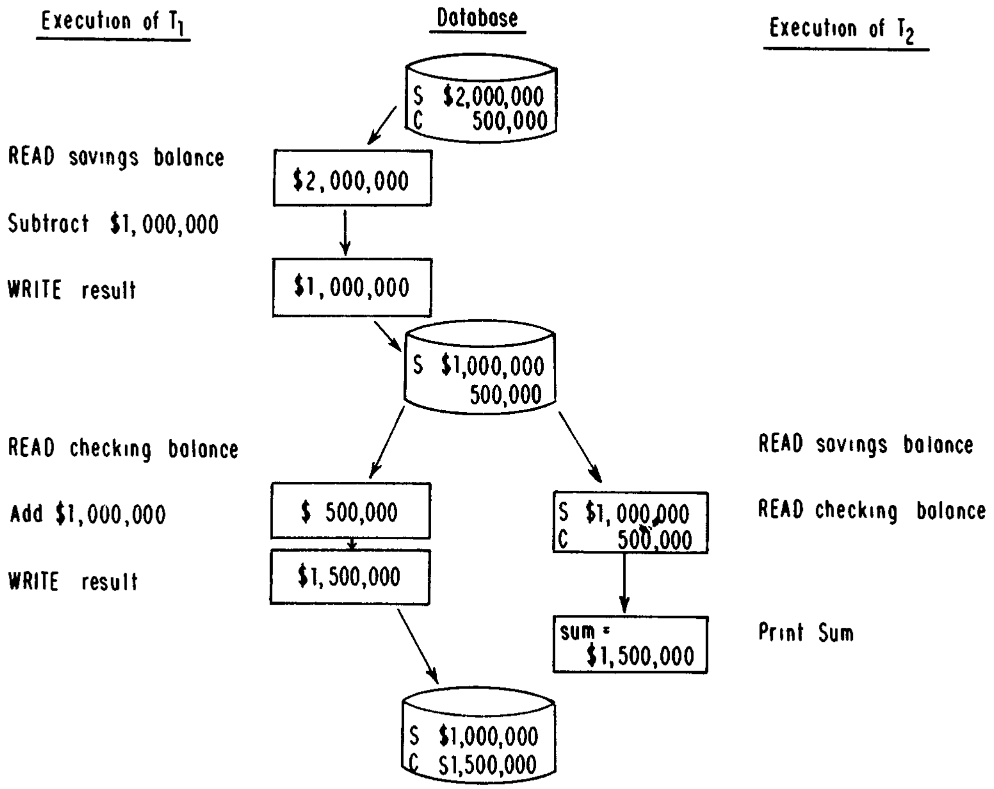


Figure 2. Inconsistent retrieval anomaly.

ROTH77]. Each *site* is a computer running one or both of the following software modules: a transaction manager (TM) or a data manager (DM). *TMs* supervise interactions between users and 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, they may be files, records, etc. A *logical database state* is an assignment of values to the logical data items composing 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.* (When no confusion is possible, we use the term *data item* for stored data item.) The stored copies of logical data item X are denoted $x_1, \ldots, x_m$. We typically use x to denote an arbitrary stored data item. A *stored database state* is an assignment of values to the stored data items in a database.

Users interact with the DDBMS by executing *transactions*. Transactions may be on-line queries expressed in a self-contained query language, or application programs written in a general-purpose programming language. The concurrency control algorithms we study pay no attention to the computations performed by transactions. Instead, these algorithms make all of their decisions on the basis of the data items a transaction reads and writes, and so details of the form of transactions are unimportant in our analysis. However we do assume that transactions represent complete and correct computations; each transaction, if ex-

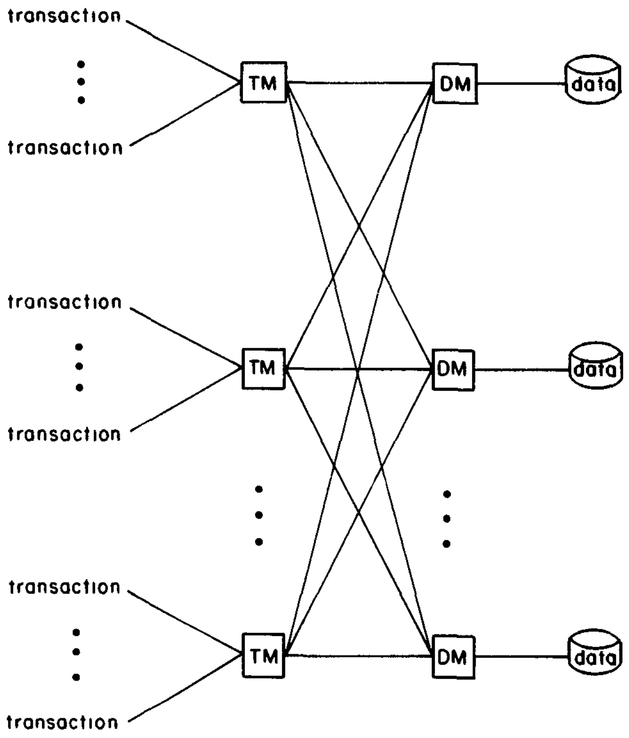


Figure 3. DDBMS system architecture.

ecuted alone on an initially consistent database, would terminate, produce correct results, and leave the database consistent. The *logical readset* (correspondingly, *writeset*) of a transaction is the set of logical data items the transaction reads (or writes). Similarly, *stored readsets* and *stored writesets* are the stored data items that a transaction reads and writes.

The correctness of a concurrency control algorithm is defined relative to users' expectations regarding transaction execution. There are two correctness criteria: (1) users expect that each transaction submitted to the system will eventually be executed; (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. Realizing this expectation is the principal issue in concurrency control.

A DDBMS contains four components (see Figure 3): 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.)

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.

Four operations are defined at the transaction-TM interface. READ(X) returns the value of X (a logical data item) 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 as backend 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 1.2 and 1.3. Section 1.2 describes the TM–DM interaction in a centralized database environment, and Section 1.3 extends the discussion to a distributed database setting.

1.2 Centralized Transaction-Processing Model

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

BEGIN: The TM initializes for T a *private workspace* that functions as a temporary buffer for values read from and written into the database.

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 *dm-read*(x) to the DM to retrieve a copy of X from the database, gives the retrieved value to T, and puts it into T's private workspace.

WRITE(X, new-value): The TM again checks the private workspace for a copy of X. If it finds one, the value is updated to new-value; otherwise a copy of X with the new value is created in the workspace. The new value of X is *not* stored in the database at this time.

END: The TM issues *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 *restart* T any time before a dm-write has been processed. The effect of restarting T is to obliterate its private workspace and to reexecute 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; 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 avoid such partial results by having the property of *atomic commitment*, which requires that either all of a transaction's dm-writes are processed or none are. The "standard" implementation of atomic commitment is a procedure called *two-phase commit* [LAMP76, GRAY78].[1] Suppose T is updating data items X and Y. When T issues its END, the *first phase* of two-phase commit begins, during which the DM issues *prewrite* commands for X and Y. These commands instruct the DM to copy the values of X and Y from T's private workspace onto secure storage. If the DBMS fails during the first phase, no harm is done, since none of T's updates have yet been applied to the stored database. During the *second phase*, the TM issues *dm-write* commands for X and Y which instruct the DM to copy the values of X and Y into the stored database. If the DBMS fails during the second phase, the database may contain incorrect information, but since the values of X and Y are stored on secure storage, this inconsistency can be rectified when the system recovers: the recovery procedure reads the values of X and Y from secure storage and resumes the commitment activity.

We emphasize that this is a *mathematical model* of transaction processing, an approximation to the way DBMSs actually function. While the implementation details of atomic commitment are important in designing a DBMS, they are not central to an understanding of concurrency control. To explain concurrency control algorithms we need a model of transaction execution in which atomic commitment is visible, but not dominant.

1.3 Distributed Transaction-Processing Model

Our model of transaction processing in a distributed environment differs from that in a centralized one in two areas: handling private workspaces and implementing two-phase commit.

[1] 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.

In a centralized DBMS we assumed that (1) private workspaces were part of the TM, and (2) 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 *query optimization procedures* which regulate (and, it is hoped, 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 [BERN81]. 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.

The problem of atomic commitment is aggravated in a DDBMS by the possibility of one site failing while the rest of the system continues to operate. Suppose T is updating x, y, z stored at DM_x, DM_y, DM_z, and suppose T's TM fails after issuing dm-write(x), but before issuing the dm-writes for y and z. At this point the database is incorrect. 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 can access the incorrect database.

To avoid this problem, prewrite commands must be modified slightly. In addition to specifying data items to be copied onto secure storage, prewrites also specify which other DMs are involved in the commitment activity. Then if the TM fails during the second phase of two-phase commit, the DMs whose dm-writes were not issued can recognize the situation and consult the other DMs involved in the commitment. If any DM received a dm-write, the remaining ones act as if they had also received the command. The details of this procedure are complex and appear in HAMM80.

As in a centralized DBMS, a transaction T accesses the system by issuing BEGIN, READ, WRITE, and END operations. In a DDBMS these are processed as follows.

BEGIN: The TM creates a private workspace for T. We leave the location and organization of this workspace unspecified.

READ(X): The TM checks T's private workspace to see if a copy of X is present. If so, that copy's value 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 it in the private workspace. The TM returns this value to T.

WRITE(X, new-value): 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.

END: Two-phase commit begins. For each X updated by T, and for each stored copy x_i of X, the TM issues a prewrite (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 prewrites 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.

2. DECOMPOSITION OF THE 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 subproblems.

2.1 Serializability

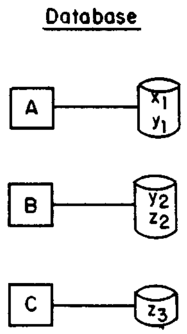
Let E denote an execution of transactions $T_1, \ldots, T_n$. E is a *serial execution* if no transactions execute concurrently in E; that is, each transaction is executed to completion before the next one begins. Every serial execution is defined to be *correct*, because the properties of transactions (see Section 1.1) imply that a serial execution terminates properly and preserves database consistency. An execution is *serializable* 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

Transactions

```
T1. BEGIN;
    READ(X); WRITE(Y); END

T2  BEGIN;
    READ(Y), WRITE(Z); END

T3. BEGIN,
    READ(Z), WRITE(X), END
```



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$_i$; $w_i[x]$ denotes a dm-write(x) issued by T$_i$.)

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. $w_2[z_3]r_3[z_3]$

Figure 4. Modeling executions as logs.

- The execution modeled in Figure 4 is serial. Each log is itself serial; that is, there is no interleaving of operations from different transactions. 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$_3$. Therefore, T$_1$, T$_2$, T$_3$ is a total order satisfying the definition of serial.
- The following execution is not serial. The logs themselves are not serial.

  DM A: $r_1[x_1]r_2[y_1]w_3[x_1]w_1[y_1]$
  DM B: $w_2[z_2]w_1[y_2]$
  DM C: $w_2[z_3]r_3[z_3]$
- The following execution is also not serial Although each log is serial, there is no total order consistent with all logs.

  DM A: $r_1[x_1]w_1[y_1]r_2[y_1]w_3[x_1]$
  DM B: $w_2[z_2]w_1[y_2]$
  DM C: $w_2[z_3]r_3[z_3]$

Figure 5. Serial and nonserial loops.

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 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*, each of which indicates the order in which dm-reads and dm-writes are processed at one DM (see Figure 4). An execution is *serial* if there is a total order of transactions such that if T$_i$ precedes T$_j$ in the total order, then all of T$_i$'s operations precede all of T$_j$'s operations in every log where both appear (see Figure 5). Intuitively, this says that transactions execute serially and 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 if and only if 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* (*rw*) *conflicts* and *write-write* (*ww*) *conflicts*, respectively.

The notion of conflict helps characterize the equivalence of executions. Two executions are *computationally equivalent* if (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 [PAPA77, PAPA79]. 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.

From this we can characterize serializable executions precisely.

Theorem 1 [PAPA77, PAPA79, STEA76]

Let $T = \{T_1, \ldots, T_m\}$ be a set of transactions and let E be an execution of these transactions modeled by logs $\{L_1, \ldots, L_m\}$. E is serializable if there exists a total ordering of T such that for each pair of conflicting operations O_i and O_j from distinct transactions T_i and T_j (respectively), O_i precedes O_j in any log $L_1, \ldots, L_m$ if and only if T_i precedes T_j in the total ordering.

The total order hypothesized in Theorem 1 is called a *serialization order*. If the transactions had executed serially in the serialization 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, namely, 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*. To be correct, a DDBMS must incorporate synchronization techniques that guarantee the conditions of Theorem 1.

2.2 A Paradigm for Concurrency Control

In Theorem 1, rw and ww 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 modeled by a set of logs. We define several binary relations on transactions in E, denoted by $\rightarrow$ with various subscripts. For each pair of transactions, T_i and T_j

(1) $T_i \rightarrow_{rw} T_j$ if in some log of E, T_i reads some data item into which T_j subsequently writes;

(2) $T_i \rightarrow_{wr} T_j$ if in some log of E, T_i writes into some data item that T_j subsequently reads;

(3) $T_i \rightarrow_{ww} T_j$ if in some log of E, T_i writes into some data item into which T_j subsequently writes;

(4) $T_i \rightarrow_{rwr} T_j$ if $T_i \rightarrow_{rw} T_j$ or $T_i \rightarrow_{wr} T_j$;

(5) $T_i \rightarrow T_j$ if $T_j \rightarrow_{rwr} T_j$ or $T_i \rightarrow_{ww} T_j$.

Intuitively, $\rightarrow$ (with any subscript) means "in any serialization must precede." For example, $T_i \rightarrow_{rw} T_j$ 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 $\rightarrow$ relationship. Therefore, we can restate Theorem 1 in terms of $\rightarrow$. According to Theorem 1, E is serializable if there is a total order of transactions that is consistent with $\rightarrow$. This latter condition holds if and only if $\rightarrow$ is acyclic. (A relation, $\rightarrow$, is *acyclic* if there is no sequence $T_1 \rightarrow T_2, T_2 \rightarrow T_3, \ldots, T_n\text{-}1 \rightarrow T_n$ such that $T_1 = T_n$.) Let us decompose $\rightarrow$ into its components, $\rightarrow_{rwr}$ and $\rightarrow_{ww}$, and restate the theorem using them.

Theorem 2 [BERN80a]

Let $\rightarrow_{rwr}$ and $\rightarrow_{ww}$ be associated with execution E. E is serializable if (a) $\rightarrow_{rwr}$ and $\rightarrow_{ww}$ are acyclic, and (b) there is a total ordering of the transactions consistent with all $\rightarrow_{rwr}$ and all $\rightarrow_{ww}$ relationships.

Theorem 2 is an immediate consequence of Theorem 1. (Indeed, part (b) of Theorem 2 is essentially a restatement of the earlier theorem.) However, this way of characterizing serializability suggests a way of decomposing the problem into simpler parts. Theorem 2 implies that rw and ww conflicts can be synchronized independently except insofar as there must be a total ordering of the transactions consistent with both types of conflicts. This suggests that we can use one technique to guarantee an acyclic $\rightarrow_{rwr}$ relation (which amounts to *read-write synchronization*) and a different technique to guarantee an acyclic $\rightarrow_{ww}$ relation (*write-write synchronization*). However, in addition to both $\rightarrow_{rwr}$ and $\rightarrow_{ww}$ being acyclic, there must also be *one* serial order consistent with *all* $\rightarrow$ relations. This serial order is the cement that binds together the rw and ww synchronization techniques.

Decomposing serializability into rw and ww synchronization is the cornerstone of our paradigm for concurrency control. It will be important hereafter to distinguish algorithms that attain either rw or ww synchronization from algorithms that solve the entire distributed concurrency control problem. We use the term *synchronization technique* for the former type of algorithm, and *concurrency control method* for the latter.

3. SYNCHRONIZATION TECHNIQUES BASED ON TWO-PHASE LOCKING

Two-phase locking (2PL) synchronizes reads and writes by explicitly detecting and preventing conflicts between concurrent operations. Before reading data item x, a transaction must "own" a *readlock on* x. Before writing into x, it must "own" a *writelock on* x. The ownership of locks is governed by two rules: (1) different transactions cannot simultaneously own *conflicting locks*; and (2) once a transaction surrenders ownership of a lock, it may never obtain additional locks.

The definition of *conflicting lock* depends on the type of synchronization being performed: for rw synchronization two locks *conflict* if (a) both are locks on the same data item, and (b) one is a readlock and the other is a writelock; for ww synchronization two locks *conflict* if (a) both are locks on the same data item, and (b) both are writelocks.

The second lock ownership rule causes every transaction to obtain locks in a *two-phase* manner. During the *growing phase* the transaction obtains locks without releasing any locks. By releasing a lock the transaction enters the *shrinking phase*. During this phase the transaction releases locks, and, by rule 2, is prohibited from obtaining additional locks. When the transaction terminates (or aborts), all remaining locks are automatically released.

A common variation is to require that transactions obtain all locks before beginning their main execution. This variation is called *predeclaration*. Some systems also require that transactions hold all locks until termination

Two-phase locking is a correct synchronization technique, meaning that 2PL attains an acyclic $\rightarrow_{rwr}$ ($\rightarrow_{ww}$) relation when used for rw (ww) synchronization [BERN79b, ESWA76, PAPA79]. The serialization order attained by 2PL is determined by the order in which transactions obtain locks. The point at the end of the growing phase, when a transaction owns all the locks it ever will own, is called the *locked point* of the transaction [BERN79b]. Let E be an execution in which 2PL is used for rw (ww) synchronization. The $\rightarrow_{rwr}$ ($\rightarrow_{ww}$) relation induced by E is identical to the relation induced by a *serial* execution E′ in which every transaction executes at its locked point. Thus the locked points of E determine a serialization order for E.

3.1 Basic 2PL Implementation

An implementation of 2PL amounts to building a *2PL scheduler*, a software module that receives lock requests and lock releases and processes them according to the 2PL specification.

The basic way to implement 2PL in a distributed database is to distribute the schedulers along with the database, placing the scheduler for data item x at the DM were x is stored. In this implementation readlocks may be implicitly requested by dm-reads and writelocks may be implicitly requested by prewrites. If the requested lock cannot be granted, the operation is placed on a waiting queue for the desired data item. (This can produce a *deadlock*, as discussed in Section 3.5.) Writelocks are implicitly released by dm-writes. However, to release readlocks, special lock-release operations are required. These lock releases may be transmitted in parallel with the dm-writes, since the dm-writes signal the start of the shrinking phase. When a lock is released, the operations on the waiting queue of that data item are processed first-in/first-out (FIFO) order.

Notice that this implementation "automatically" handles redundant data correctly. Suppose logical data item X has copies $x_1, \ldots, x_m$. If basic 2PL is used for rw synchronization, a transaction may read any copy and need only obtain a readlock on the copy of X it actually reads. However, if a transaction updates X, then it must update all copies of X, and so must obtain writelocks on all copies of X (whether basic 2PL is used for rw or ww synchronization).

3.2 Primary Copy 2PL

Primary copy 2PL is a 2PL technique that pays attention to data redundancy [STON79]. One copy of each logical data item is designated the *primary copy*; before accessing any copy of the logical data item, the appropriate lock must be obtained on the primary copy.

For readlocks this technique requires more communication than basic 2PL. Suppose x_1 is the primary copy of logical data item X, and suppose transaction T wishes to read some other copy, x_i, of X. To read x_i, T must communicate with two DMs, the DM where x_1 is stored (so T can lock x_1) and the DM where x_i is stored. By contrast, under basic 2PL, T would only communicate with x_i's DM. For writelocks, however, primary copy 2PL does not incur extra communication. Suppose T wishes to update X. Under basic 2PL, T would issue prewrites to all copies of X (thereby requesting writelocks on these data items) and then issue dm-writes to all copies. Under primary copy 2PL the same operations would be required, but only the prewrite (x_1) would request a writelock. That is, prewrites would be sent for $x_1, \ldots, x_m$, but the prewrites for $x_2, \ldots, x_m$ would not implicitly request writelocks.

3.3 Voting 2PL

Voting 2PL (or *majority consensus 2PL*) is another 2PL implementation that exploits data redundancy. Voting 2PL is derived from the majority consensus technique of Thomas [THOM79] and is only suitable for ww synchronization.

To understand voting, we must examine it in the context of two-phase commit. Suppose transaction T wants to write into X. Its TM sends prewrites to each DM holding a copy of X. For the voting protocol, the DM always responds immediately. It acknowledges receipt of the prewrite and says "lock set" or "lock blocked." (In the basic implementation it would not acknowledge at all until the lock is set.) After the TM receives acknowledgments from the DMs, it counts the number of "lockset" responses: if the number constitutes a majority, then the TM behaves as if all locks were set. Otherwise, it waits for "lockset" operations from DMs that originally said "lock blocked." Deadlocks aside (see Section 3.5), it will eventually receive enough "lockset" operations to proceed.

Since only one transaction can hold a majority of locks on X at a time, only one transaction writing into X can be in its second commit phase at any time. All copies of X thereby have the same sequence of writes applied to them. A transaction's locked point occurs when it has obtained a majority of its writelocks on each data item in its writeset. When updating many data items, a transaction must obtain a majority of locks on every data item before it issues any dm-writes.

In principle, voting 2PL could be adapted for rw synchronization. Before reading any copy of X a transaction requests readlocks on all copies of X; when a majority of locks are set, the transaction may read any copy. This technique works but is overly strong: Correctness only requires that a single copy of X be locked—namely, the copy that is read—yet this technique requests locks on all copies. For this reason we deem voting 2PL to be inappropriate for rw synchronization.

3.4 Centralized 2PL

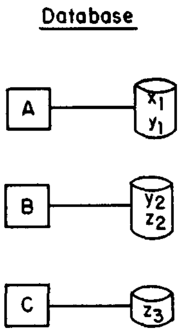
Instead of distributing the 2PL schedulers, one can centralize the scheduler at a single site [ALSB76a, GARC79a]. Before accessing data at any site, appropriate locks must be obtained from the central 2PL scheduler. So, for example, to perform dm-read(x) where x is not stored at the central site, the TM must first request a readlock on x from the central site, wait for the central site to acknowledge that the lock has been set, then send dm-read(x) to the DM that holds x. (To save some communication, one can have the TM send both the lock request and dm-read (x) to the central site and let the central site directly forward dm-read(x) to x's DM; the DM then responds to the TM when dm-read (x) has been processed.) Like primary copy 2PL, this approach tends to require more communication than basic

Transactions

```
T1: BEGIN;
    READ(X); WRITE(Y); END

T2  BEGIN;
    READ(Y); WRITE(Z); END

T3. BEGIN,
    READ(Z), WRITE(X), END
```



- Suppose transactions execute concurrently, with each transaction issuing its READ before any transaction issues its END.
- This partial execution could be represented by the following logs

DM A: $r_1[x_1]$
DM B: $r_2[y_2]$
DM C: $r_3[z_3]$

- At this point, T_1 has readlock on x_1
  T_2 has readlock on y_2
  T_3 has readlock on z_3
- Before proceeding, all transactions must obtain writelocks.
  T_1 requires writelocks on y_1 and y_2
  T_2 requires writelocks on z_2 and z_3
  T_3 requires writelock on x_1
- But

  T_1 cannot get writelock on y_2, until T_2 releases readlock
  T_2 cannot get writelock on z_3, until T_3 releases readlock
  T_3 cannot get writelock on x_1, until T_1 releases readlock

  This is a deadlock

Figure 6. Deadlock.

2PL, since dm-reads and prewrites usually cannot implicitly request locks.

3.5 Deadlock Detection and Prevention

The preceding implementations of 2PL force transactions to wait for unavailable locks. If this waiting is uncontrolled, deadlocks can arise (see Figure 6).

Deadlock situations can be characterized by *waits-for graphs* [HOLT72, KING74], directed graphs that indicate which transactions are waiting for which other transactions. Nodes of the graph represent transactions, and edges represent the "waiting-for" relationship: an edge is drawn from transaction T_i to transaction T_j if T_i is waiting for a lock currently owned by T_j. There is a deadlock in the system if and only if the waits-for graph contains a *cycle* (see Figure 7).

Two general techniques are available for deadlock resolution: *deadlock prevention* and *deadlock detection.*

3.5.1 Deadlock Prevention

Deadlock prevention is a "cautious" scheme in which a transaction is restarted when the system is "afraid" that deadlock might occur. To implement deadlock prevention, 2PL schedulers are modified as follows. When a lock request is denied, the scheduler tests the requesting transaction (say T_i) and the transaction that currently owns the lock (say T_j). If T_i and T_j pass the test, T_i is permitted to wait for T_j as usual. Otherwise, one of the two is aborted. If T_i is restarted, the deadlock prevention algorithm is called *nonpreemptive*; if T_j is restarted, the algorithm is called *preemptive.*

The test applied by the scheduler must

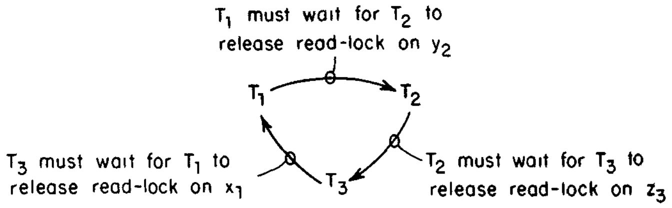

T_1 must wait for T_2 to release read-lock on y_2
T_1
T_2
T_3 must wait for T_1 to release read-lock on x_1
T_3
T_2 must wait for T_3 to release read-lock on z_3

Figure 7. Waits-for graph for Figure 6.

guarantee that if T_i waits for T_j, then deadlock cannot result. One simple approach is never to let T_i wait for T_j. This trivially prevents deadlock but forces many restarts.

A better approach is to assign *priorities* to transactions and to test priorities to decide whether T_i can wait for T_j. For example, we could let T_i wait for T_j if T_i has lower priority than T_j (if T_i and T_j have *equal* priorities, T_i cannot wait for T_j, or vice versa). This test prevents deadlock because, for every edge $\langle T_i, T_j \rangle$ in the waits-for graph, T_i has lower priority than T_j. Since a cycle is a path from a node to itself and since T_i cannot have lower priority than itself, no cycle can exist.

One problem with the preceding approach is that *cyclic restart* is possible—some unfortunate transaction could be continually restarted without ever finishing. To avoid this problem, Rosenkrantz et al. [ROSE78] propose using "timestamps" as priorities. Intuitively, a transaction's timestamp is the time at which it begins executing, so old transactions have higher priority than young ones.

The technique of ROSE78 requires that each transaction be assigned a *unique* timestamp by its TM. When a transaction begins, the TM reads the local clock time and appends a unique TM identifier to the low-order bits [THOM79]. The resulting number is the desired timestamp. 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 (since different TMs have different identifiers), while timestamps assigned by the same TM differ in their high-order bits (since the TM does not use the same clock time twice). Hence timestamps are unique throughout the system. Note that this algorithm does not require clocks at different sites to be precisely synchronized.

Two timestamp-based deadlock prevention schemes are proposed in ROSE78. *Wait-Die* is the nonpreemptive technique. Suppose transaction T_i tries to wait for T_j. If T_i has lower priority than T_j (i.e., T_i is younger than T_j), then T_i is permitted to wait. Otherwise, it is aborted ("dies") and forced to restart. It is important that T_i not be assigned a new timestamp when it restarts. *Wound-Wait* is the preemptive counterpart to *Wait-Die.* If T_i has higher priority than T_j, then T_i waits; otherwise T_j is aborted.

Both Wait-Die and Wound-Wait avoid cyclic restart. However, in Wound-Wait an old transaction may be restarted many times, while in Wait-Die old transactions never restart. It is suggested in ROSE78 that Wound-Wait induces fewer restarts in total.

Care must be exercised in using preemptive deadlock prevention with two-phase commit: a transaction must not be aborted once the second phase of two-phase commit has begun. If a preemptive technique wishes to abort T_j, it checks with T_j's TM and cancels the abort if T_j has entered the second phase. No deadlock can result because if T_j is in the second phase, it cannot be waiting for any transactions.

Preordering of resources is a deadlock avoidance technique that avoids restarts altogether. This technique requires predeclaration of locks (each transaction obtains all its locks before execution). Data items are numbered and each transaction requests locks one at a time in numeric order. The priority of a transaction is the number of the highest numbered lock it owns. Since a transaction can only wait for transactions with higher priority, no deadlocks can occur. In addition to requiring predeclaration, a principal disadvantage of this technique is that it forces locks to be obtained sequentially, which tends to increase response time.

- Consider the execution illustrated in Figures 6 and 7.
- Locks are requested at DMs in the following order:

DM A	DM B	DM C
readlock x_1 for T_1	readlock y_2 for T_2	readlock z_3 for T_3
writelock y_1 for T_1	writelock z_2 for T_2	
*writelock x_1 for T_3	*writelock y_2 for T_1	*writelock z_3 for T_2

- None of the "starred" locks can be granted and the system is in deadlock. However, the waits-for graphs at each DM are acyclic.

$$\begin{tikzcd}
\text{DM A} & & \text{DM B} & & \text{DM C} & \\
T_3 \arrow[r] & T_1 & T_1 \arrow[r] & T_2 & T_2 \arrow[r] & T_3
\end{tikzcd}$$

Figure 8. Multisite deadlock.

3.5.2 Deadlock Detection

In *deadlock detection*, transactions wait for each other in an uncontrolled manner and are only aborted if a deadlock actually occurs. Deadlocks are detected by explicitly constructing the waits-for graph and searching it for cycles. (Cycles in a graph can be found efficiently using, for example, Algorithm 5.2 in AHO75.) If a cycle is found, one transaction on the cycle, called the *victim*, is aborted, thereby breaking the deadlock. To minimize the cost of restarting the victim, victim selection is usually based on the amount of resources used by each transaction on the cycle.

The principal difficulty in implementing deadlock detection in a distributed database is constructing the waits-for graph efficiently. Each 2PL scheduler can easily construct the waits-for graph based on the waits-for relationships local to that scheduler. However, these local waits-for graphs are not sufficient to characterize all deadlocks in the distributed system (see Figure 8). Instead, local waits-for graphs must be combined into a more "global" waits-for graph. (Centralized 2PL does not have this problem, since there is only one scheduler.) We describe two techniques for constructing global waits-for graphs: centralized and hierarchical deadlock detection.

In the *centralized* approach, one site is designated the deadlock detector for the distributed system [GRAY78, STON79]. Periodically (e.g., every few minutes) each scheduler sends its local waits-for graph to the deadlock detector. The deadlock detector combines the local graphs into a system-wide waits-for graph by constructing the union of the local graphs.

In the *hierarchical* approach, the database sites are organized into a hierarchy (or tree), with a deadlock detector at each node of the hierarchy [MENA79]. For example, one might group sites by *region*, then by *country*, then by *continent*. Deadlocks that are local to a single site are detected at that site; deadlocks involving two or more sites of the same region are detected by the regional deadlock detector; and so on.

Although centralized and hierarchical deadlock detection differ in detail, both involve periodic transmission of local waits-for information to one or more deadlock detector sites. The periodic nature of the process introduces two problems. First, a deadlock may exist for several minutes without being detected, causing response-time degradation. The solution, executing the deadlock detector more frequently, increases the cost of deadlock detection. Second, a transaction T may be restarted for reasons other than concurrency control (e.g., its site crashed). Until T's restart propagates to the deadlock detector, the deadlock detector can find a cycle in the waits-for graph that includes T. Such a cycle is called a *phantom deadlock*. When the deadlock detector discovers a phantom deadlock, it may unnecessarily restart a transaction other than T. Special precautions are also needed to avoid unnecessary restarts for deadlocks in voting 2PL.[2]

[2] Suppose logical data item X has copies x_1, x_2, and x_3, and suppose using voting 2PL T_i owns write-locks on x_1 and x_2 but T_i's lock request for x_3 is blocked by T_j.

A major cost of deadlock detection is the restarting of partially executed transactions. Predeclaration can be used to reduce this cost. By obtaining a transaction's locks before it executes, the system will only restart transactions that have not yet executed. Thus little work is wasted by the restart.

4. SYNCHRONIZATION TECHNIQUES BASED ON TIMESTAMP ORDERING

Timestamp ordering (T/O) is a technique whereby a serialization order is selected a priori and transaction execution is forced to obey this order. Each transaction is assigned a unique timestamp by its TM. The TM attaches the timestamp to all dm-reads and dm-writes issued on behalf of the transaction, and DMs are required to process *conflicting operations* in timestamp order. The timestamp of operation O is denoted ts(O).

The definition of conflicting operations depends on the type of synchronization being performed and is analogous to conflicting locks. For rw synchronization, two operations *conflict* if (a) both operate on the same data item, and (b) one is a dm-read and the other is a dm-write. For ww synchronization, two operations *conflict* if (a) both operate on the same data item, and (b) both are dm-writes.

It is easy to prove that T/O attains an acyclic $\rightarrow_{\text{rwr}}$ ($\rightarrow_{\text{ww}}$) relation when used for rw (ww) synchronization. Since each DM processes conflicting operations in timestamp order, each edge of the $\rightarrow_{\text{rwr}}$ ($\rightarrow_{\text{ww}}$) relation is in timestamp order. Consequently, 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.

4.1 Basic T/O Implementation

An implementation of T/O amounts to building a *T/O scheduler*, a software module that receives dm-reads and dm-writes and outputs these operations according to the T/O specification [SHAP77a, SHAP77b]. In practice, prewrites must also be processed through the T/O scheduler for two-phase commit to operate properly. As was the case with 2PL, the basic T/O implementation distributes the schedulers along with the database [BERN80a].

If we ignore two-phase commit, the basic T/O scheduler is quite simple. At each DM, and for each data item x stored at the DM, the scheduler records the largest timestamp of any dm-read(x) or dm-write(x) that has been processed. These are denoted R-ts(x) and W-ts(x), respectively. For rw synchronization, scheduler S operates as follows. Consider a dm-read(x) with timestamp TS. If TS < W-ts(x), S *rejects* the dm-read and aborts the issuing transaction. Otherwise S outputs the dm-read and sets R-ts(x) to max(R-ts(x),TS). For a dm-write(x) with timestamp TS, S rejects the dm-write if TS < R-ts(x); otherwise it outputs the dm-write and sets W-ts(x) to max(W-ts(x),TS). For ww synchronization, S rejects a dm-write(x) with timestamp TS if TS < W-ts(x); otherwise it outputs the dm-write and sets W-ts(x) to TS.

When a transaction is aborted, it is assigned a new and larger timestamp by its TM and is restarted. Restart issues are discussed further below.

Two-phase commit is incorporated by timestamping prewrites and accepting or rejecting prewrites instead of dm-writes. Once a scheduler accepts a prewrite, it must guarantee to accept the corresponding dm-write no matter when the dm-write arrives. For rw (or ww) synchronization, once S accepts a prewrite(x) with timestamp TS it must not output any dm-read(x) (or dm-write(x)) with timestamp greater than TS until the dm-write(x) is output. The effect is similar to setting a writelock on x for the duration of two-phase commit..

To implement the above rules, S *buffers* dm-reads, dm-writes, and prewrites. Let min-R-ts(x) be the minimum timestamp of any buffered dm-read(x), and define min-W-ts(x) and min-P-ts(x) analogously. Rw synchronization is accomplished as follows:

1. Let R be a dm-read(x). If ts(R) < W-ts(x), R is rejected. Else if ts(R) > min-P-ts(x), R is buffered. Else R is output.

Insofar as x_3's scheduler is concerned, T_i is waiting for T_j. However, since T_i has a *majority* of the copies locked, T_i can proceed without waiting for T_j. This fact should be incorporated into the deadlock resolution scheme to avoid unnecessary restarts.

Let R = dm-read (x).
Let W = dm-write (x).
R is *ready* if it precedes the earliest prewrite request:
if ts(R) < min-P-ts(x).
W is *ready* if it precedes the earliest dm-read request:
if ts (W) < min-R-ts(x).
When a dm-write(x) arrives, do the following:

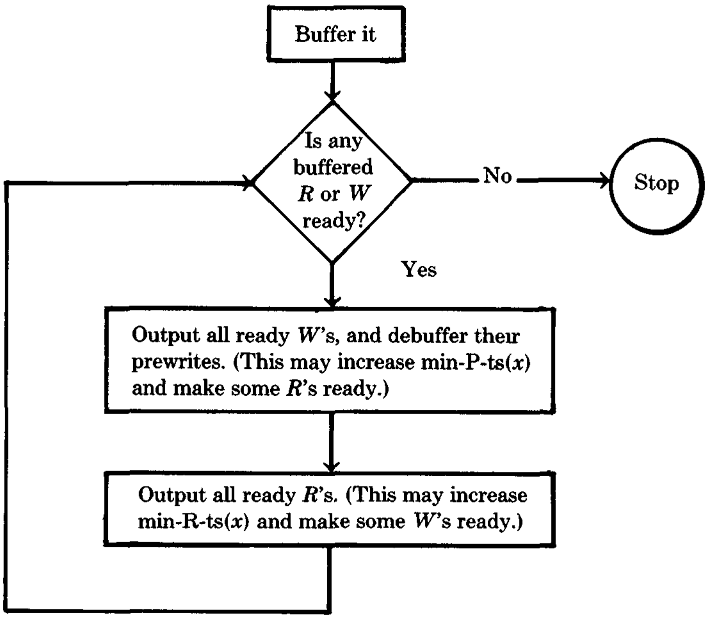

Buffer it
Is any buffered R or W ready?
No
Stop
Yes
Output all ready W's, and debuffer their prewrites. (This may increase min-P-ts(x) and make some R's ready.)
Output all ready R's. (This may increase min-R-ts(x) and make some W's ready.)

Figure 9. Buffer emptying for basic T/O rw synchronization.

2. Let P be a prewrite(x). If ts(P) < R-ts(x), P is rejected. Else P is buffered.
3. Let W be a dm-write(x). W is never rejected. If ts(W) > min-R-ts(x), W is buffered. (If W were output it would cause a buffered dm-read(x) to be rejected.) Else W is output.
4. When W is output, the corresponding prewrite is debuffered. If this causes min-P-ts(x) to increase, the buffered dm-reads are retested to see if any of them can be output. If this causes min-R-ts(x) to be increased, the buffered dm-writes are also retested, and so forth. This process is diagramed in Figure 9.

Ww synchronization is accomplished as follows:

1. Let P be a prewrite(x). If ts(P) < W-ts(x), P is rejected; else P is buffered.
2. Let W be a dm-write(x). W is never rejected. If ts(W) > min-P-ts(x), W is buffered; else W is output.
3. When W is output, the corresponding prewrite is debuffered. If this causes min-P-ts(x) to be increased, the buffered dm-writes are retested to see if any can now be output. See Figure 10.

As with 2PL, a common variation is to require that transactions *predeclare* their readsets and writesets, issuing all dm-reads and prewrites before beginning their main execution.[3] If all operations are accepted,

[3] These prewrites are nonstandard relative to the definition in Section 1.4. Since new values for the data items in the writeset are not yet known, these prewrites do *not* instruct DMs to store values on secure storage; instead, prewrite (x) merely "warns" the DM to expect a dm-write (x) in the near future. However, these prewrites are processed by synchronization algorithms exactly as "standard" ones are.

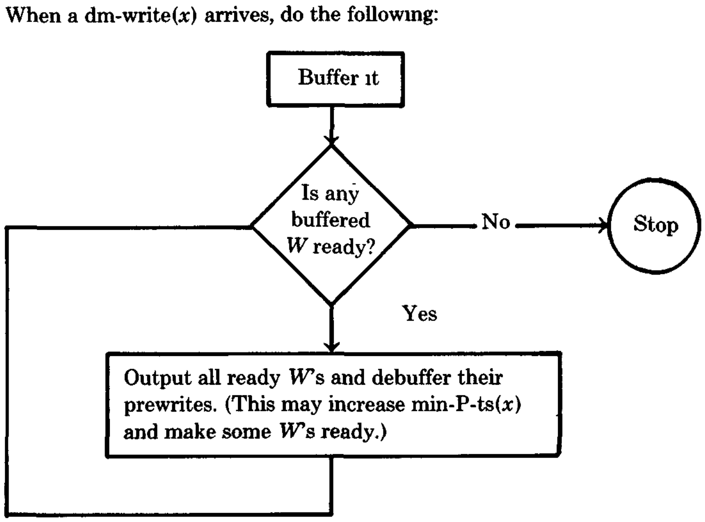

When a dm-write(x) arrives, do the following:
Buffer it
Is any buffered W ready?
No
Stop
Yes
Output all ready W's and debuffer their prewrites. (This may increase min-P-ts(x) and make some W's ready.)

Figure 10. Buffer emptying for basic T/O ww synchronization.

the transaction is guaranteed to execute without danger of restart. Another variation is to *delay* the processing of operations to wait for operations with smaller timestamps. The extreme version of this heuristic is *conservative T/O*, described in Section 4.4.

4.2 The Thomas Write Rule

For ww synchronization the basic T/O scheduler can be optimized using an observation of THOM79. Let W be a dm-write(x), and suppose ts(W) < W-ts(x). Instead of rejecting *W we can simply ignore it*. We call this the *Thomas Write Rule (TWR)*. Intuitively, TWR applies to a dm-write that tries to place 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.

If TWR is used, there is no need to incorporate two-phase commit into the ww synchronization algorithm; the ww scheduler always accepts prewrites and never buffers dm-writes.

4.3 Multiversion T/O

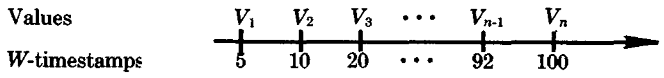
For rw synchronization the basic T/O scheduler can be improved using *multiversion data items* [REED78]. For each data item x there is a *set* of R-ts's and a set of ⟨W-ts, value⟩ pairs, called *versions*. The R-ts's of x record the timestamps of all executed dm-read(x) operations, and the versions record the timestamps and values of all executed dm-write(x) operations. (In practice one cannot store R-ts's and versions forever; techniques for deleting old versions and timestamps are described in Sections 4.5 and 5.2.2.)

Multiversion T/O accomplishes rw synchronization as follows (ignoring two-phase commit). Let R be a dm-read(x). R is processed by reading the version of x with largest timestamp less than ts(R) and adding ts(R) to x's set of R-ts's; see Figure 11a. R is never rejected. Let W be a dm-write(x), and let *interval*(W) be the interval from ts(W) to the smallest W-ts(x) > ts(W);[4] see Figure 11b. If any R-ts(x) lies in interval(W), W is rejected; otherwise W is output and creates a new version of x with timestamp ts(W).

To prove the correctness of multiversion T/O, we must show that every execution is equivalent to a serial execution in timestamp order [BERN80b]. Let R be a dm-read(x) that is processed "out of order"; that is, suppose R is executed after a dm-write(x) whose timestamp exceeds ts(R). Since R ignores all versions with time-

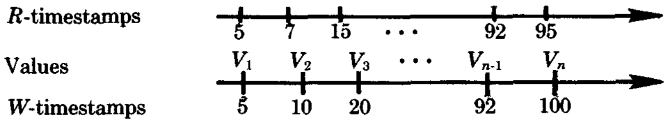
[4] Interval(W) = (ts(W),∞) if no W-ts(x) > ts (W) exists.

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



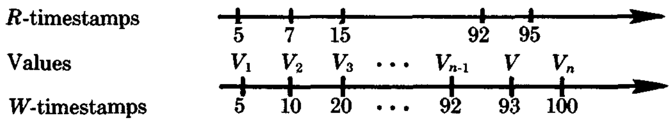
To process a dm-read(x) with timestamp 95, find the biggest W-timestamp less than 95; in this case 92. That is the version you read. So in this case, the value read by the dm-read is V_{n-1}.

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



Let W be dm-write(x) with timestamp 93. Interval(W) = (93,100).

To process W we create a new version of x with that timestamp.



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.

Figure 11. Multiversion reading and writing.

stamps greater than ts(R), the value read by R is identical to the value it would have read had it been processed in timestamp order. Now let W be a dm-write(x) that is processed "out of order"; that is, suppose it is executed after a dm-read(x) whose timestamp exceeds ts(W). Since W was not rejected, there exists a version of x with timestamp TS such that ts(W) $<$ TS $<$ ts(dm-read). Again the effect is identical to a timestamp-ordered execution.

For ww synchronization, multiversion T/O is essentially an embellished version of TWR. A dm-write(x) always creates a new version of x with timestamp ts(dm-write) and is never rejected.

Integrating two-phase commit requires that dm-reads and prewrites (but not dm-writes) be buffered as in basic T/O. Let P be a buffered prewrite(x): *interval*(P) is the interval from ts(P) to the smallest W-ts(x) $>$ ts(P). Rw synchronization is performed as follows:

1. Let R be a dm-read(x). R is never rejected. If ts(R) lies in interval(prewrite(x)) for some buffered prewrite(x), then R is buffered. Else R is output.
2. Let P be a prewrite(x). If some R-ts(x) lies in interval(P), P is rejected. Else P is buffered.
3. Let W be a dm-write(x). W is always output immediately.
4. When W is output, its prewrite is debuffered, and the buffered dm-reads are retested to see if they can now be output. See Figure 12.

Two-phase commit is not an issue for ww synchronization, since dm-writes are never rejected for ww synchronization.

Let R = dm-read(x). R is *ready* if ts(R) $\notin$ interval (P), where P is any buffered prewrite(x).

When a dm-write arrives do the following:

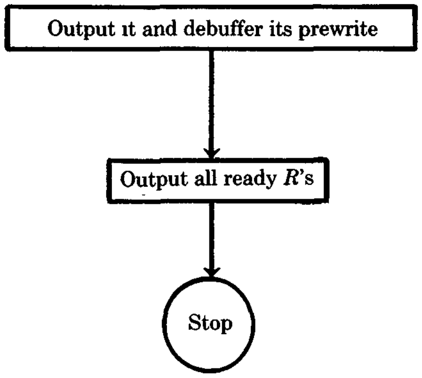


Figure 12. Buffer emptying for multiversion T/O.

4.4 Conservative T/O

Conservative timestamp ordering is a technique for eliminating restarts during T/O scheduling [BERN80a]. When a scheduler receives an operation O that might cause a future restart, the scheduler *delays* O until it is sure that no future restarts are possible.

Conservative T/O requires that each scheduler receive dm-reads (or dm-writes) from each TM in timestamp order. For example, if scheduler S_j receives dm-read(x) followed by dm-read(y) from TM_i, then ts(dm-read(x)) $\leq$ ts(dm-read(y)). Since the network is assumed to be a FIFO channel, this timestamp ordering is accomplished by requiring that TM_i *send* dm-reads (or dm-writes) to S_j in timestamp order.[5]

Conservative T/O buffers dm-reads and dm-writes as part of its normal operation. When a scheduler buffers an operation, it remembers the TM that sent it. Let min-R-ts(TM_i) be the minimum timestamp of any buffered dm-read from TM_i, with min-R-ts(TM_i) = $-\infty$ if no such dm-read is buffered. Define min-W-ts(TM_i) analogously.

Conservative T/O performs rw synchronization as follows:

1. Let R be a dm-read(x). If ts(R) > min-W-ts(TM) for any TM in the system, R is buffered. Else R is output.
2. Let W be a dm-write(x). If ts(W) > min-R-ts(TM) for any TM, W is buffered. Else W is output.
3. When R or W is output or buffered, this may increase min-R-ts(TM) or min-W-ts(TM); buffered operations are retested to see if they can now be output.

The effect is that R is output if and only if (a) the scheduler has a buffered dm-write from every TM, and (b) ts(R) < minimum timestamp of any buffered dm-write. Similarly, W is output if and only if (a) there is a buffered dm-read from every TM, and (b) ts(W) < minimum timestamp of any buffered dm-read. Thus R (or W) is output if and only if the scheduler has received every dm-write (or dm-read) with smaller timestamp that it will ever receive.

Ww synchronization is accomplished as follows:

1. Let W be a dm-write(x). If ts(W) > min-W-ts(TM) for any TM in the system, W is buffered; else it is output.
2. When W is buffered or output, this may increase min-W-ts(TM); buffered dm-writes are retested accordingly.

The effect is that the scheduler waits until it has a buffered dm-write from every TM and then outputs the dm-write with smallest timestamp.

Two-phase commit need not be tightly integrated into conservative T/O because dm-writes are never rejected. Although prewrites must be issued for all data items updated, these operations are not processed by the conservative T/O schedulers.

The above implementation of conservative T/O suffers three major problems: (1) If some TM never sends an operation to some scheduler, the scheduler will "get stuck" and stop outputting. (2) To avoid the first problem, every TM must communicate regularly with every scheduler; this is infeasible in large networks. (3) The im-

[5] This can be implemented by requiring that TMs process transactions serially Alternatively, we can require that transactions issue all dm-reads before beginning their main execution, and all dm-writes after terminating their main execution. Then transactions can execute concurrently, although they must terminate in timestamp order.

- A *class* is defined by a readset and a writeset. For example,

  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. For example,

  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

Figure 13. Transaction classes.

plementation is overly conservative; the ww algorithm, for instance, processes *all* dm-writes in timestamp order, not merely conflicting ones. These problems are addressed below.

Null Operations. To solve the first problem, TMs are required to periodically send timestamped *null operations* to each scheduler in the absence of "real" traffic. A null operation is a dm-read or dm-write whose sole purpose is to convey timestamp information and thereby unblock "real" dm-reads and prewrites. An impatient scheduler can prompt a TM for a null operation by sending a "request message." For example, for rw synchronization suppose scheduler S wants to process a dm-read with timestamp TS, but does not have a buffered dm-write from TM_i. S can send a message to TM_i requesting a null-dm-write with timestamp greater than TS.

A variation is to use null operations with *very large* (perhaps infinite) timestamps. For example, if TM_i rarely needs to issue dm-reads to S, TM_i can send S a null-dm-read with infinite timestamp signifying that TM_i does not intend to communicate with S until further notice.

Transaction Classes. Transaction classes [BERN78a, BERN80d] is a technique for reducing communication in conservative T/O and for supporting a less conservative scheduling policy. As in predeclaration, assume that every transaction's readset and writeset are known in advance. A *class* is defined by a readset and a writeset (see Figure 13). Transaction T is a *member* of class C if readset(T) is a subset of readset(C) and writeset(T) is a subset of writeset(C). (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 that are available to 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 that has no class containing it, the transaction is forwarded to another TM that does. We assume that every class is associated with exactly one TM, and vice versa. The class associated with TM_i is denoted C_i. To execute transactions that are members of class C at two TMs, we define another class C′ with the same definition as C and associate C with one TM and C′ with the other. To execute transactions that are members of two classes at one site, we multiprogram two TMs at that site.

Classes are exploited by conservative T/O schedulers as follows. Consider rw synchronization and suppose scheduler S wants to output a dm-read(x). Instead of waiting for dm-writes with smaller timestamps from all TMs, S need only wait for dm-writes *from those TMs whose class writeset contains x*. Similarly, to process a dm-write (x), S need only wait for dm-reads with smaller timestamp from those TMs whose class readset contains x. Thus communication requirements are decreased, and the level of concurrency in the system is increased. Ww synchronization proceeds similarly.

Conflict Graph Analysis. Conflict graph analysis is a technique for further improving the performance of conservative T/O with classes. A *conflict graph* 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 represent the readset and writeset of C_i. The edges of the graph are defined as follows (see Figure 14): (1) For each class

Define C_1, C_2, C_3 as in Figure 13.

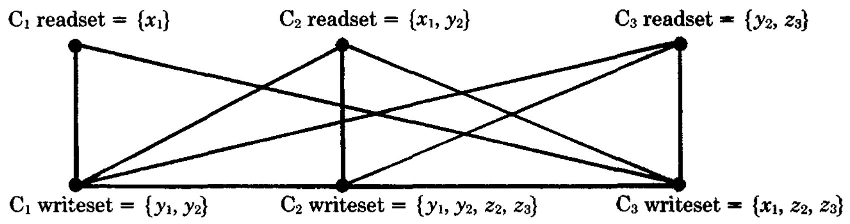

C_1 readset = $\{x_1\}$
C_2 readset = $\{x_1, y_2\}$
C_3 readset = $\{y_2, z_3\}$
C_1 writeset = $\{y_1, y_2\}$
C_2 writeset = $\{y_1, y_2, z_2, z_3\}$
C_3 writeset = $\{x_1, z_2, z_3\}$

Figure 14. Conflict graph.

C_i there is a *vertical edge* between r_i and w_i; (2) 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 if and only if writeset(C_i) intersects writeset(C_j); (3) 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 if and only if readset(C_i) intersects writeset(C_j).

Intuitively, a horizontal edge indicates that a scheduler S 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 (w_i, w_j), indicating that their class writesets intersect. If S receives a dm-write from C_i, it must delay the dm-write until it receives all dm-writes with smaller timestamps from C_j. Similarly, a diagonal edge indicates that S may need to delay operations for rw synchronization.

Conflict graph analysis improves the situation by identifying interclass conflicts that cannot cause nonserializable 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 horizontal edge between w_i and 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 [BERN80c]. That is, if conditions (1) and (2) do not hold, scheduler S 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 diagonal edge between r_i and 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.

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

4.5 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," for example, to reject a dm-read(x) with timestamp TS_1 that arrives after a dm-write(x) with timestamp TS_2, where $TS_1 < TS_2$. In principle, TS_1 and TS_2 can differ by an arbitrary amount. However, in practice it is unlikely that these timestamps will differ by more than a few minutes. Consequently, timestamps can be stored in small tables which are periodically purged.

R-ts's are stored in the *R-table* with entries of the form $\langle x$, R-ts$\rangle$; for any data item x, there is at most one entry. In addition, a variable, *R-min*, tells the maximum value of any timestamp that has been purged from the table. To find R-ts(x), a scheduler searches the R-table for an $\langle x$, TS$\rangle$ entry. If such an entry is found, R-ts(x) = TS; otherwise, R-ts(x) $\leq$ R-min. To err on the side of safety, the scheduler assumes R-ts(x) = R-min. To update R-ts(x), the scheduler modifies the $\langle x$, TS$\rangle$ 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-ts's are managed similarly, and analogous techniques can be devised for multiversion databases.

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

5. INTEGRATED CONCURRENCY CONTROL METHODS

An integrated concurrency control method consists of two components—an rw and a ww synchronization technique—plus an interface between the components that attains condition (b) of Theorem 2: a total ordering of the transactions consistent with all $\rightarrow_{rwr}$ and $\rightarrow_{ww}$ relationships. In this section we list 48 concurrency control methods that can be constructed using the techniques of Sections 3 and 4.

Approximately 20 concurrency control methods have been described in the literature. Virtually all of them use a *single* synchronization technique (either 2PL or T/O) for both rw and ww synchronization. Indeed, most methods use the same *variation* of a single technique for both kinds of synchronization. However, such homogeneity is neither necessary nor especially desirable.

For example, the analysis of Section 3.2 suggests that using basic 2PL for rw synchronization and primary copy 2PL for ww synchronization might be superior to using basic 2PL (or primary copy 2PL) for both. More outlandish combinations may be even better. For example, one can combine basic 2PL with TWR. In this method ww conflicts never cause transactions to be delayed or restarted; multiple transactions can write into the same data items concurrently (see Section 5.3).

In Sections 5.1 and 5.2 we describe methods that use 2PL and T/O techniques for both rw and ww synchronization. The concurrency control methods in these sections are easy to describe given the material of Sections 3 and 4; the description of each method is little more than a description of each component technique. In Section 5.3 we list 24 concurrency control methods that combine 2PL and T/O techniques. As we show in Section 5.3, methods of this type have useful properties that cannot be attained by pure 2PL or T/O methods.

5.1 Pure 2PL Methods

The 2PL synchronization techniques of Section 3 can be integrated to form 12 *principal 2PL methods*:

Method	rw technique	ww technique
1	Basic 2PL	Basic 2PL
2	Basic 2PL	Primary copy 2PL
3	Basic 2PL	Voting 2PL
4	Basic 2PL	Centralized 2PL
5	Primary copy 2PL	Basic 2PL
6	Primary copy 2PL	Primary copy 2PL
7	Primary copy 2PL	Voting 2PL
8	Primary copy 2PL	Centralized 2PL
9	Centralized 2PL	Basic 2PL
10	Centralized 2PL	Primary copy 2PL
11	Centralized 2PL	Voting 2PL
12	Centralized 2PL	Centralized 2PL

Each method can be further refined by the choice of deadlock resolution technique (see Section 3.5).

The interface between each 2PL rw technique and each 2PL ww technique is straightforward. It need only guarantee that "two-phasedness" is preserved, meaning that *all* locks needed for both the rw and ww technique must be obtained before any lock is released by *either* technique.

5.1.1 *Methods Using Basic 2PL for rw Synchronization*

Methods 1–4 use basic 2PL for rw synchronization. Consider a logical data item X with copies $x_1, \ldots, x_m$. To read X, a transaction sends a dm-read to *any* DM that stores a copy of X. This dm-read implicitly requests a readlock on the copy of X at that DM. To write X, a transaction sends prewrites to *every* DM that stores a copy of X. These prewrites implicitly request writelocks on the corresponding copies of X. For all four methods, these writelocks conflict with *readlocks* on the same copy, and may also conflict with other *writelocks* on the same copy, depending on the specific ww synchronization technique used by the method.

Since locking conflict rules for writelocks will vary from copy to copy, we distinguish three types. An *rw writelock* only conflicts with readlocks on the same data item. A *ww writelock* only conflicts with ww writelocks on the same data item. And an *rww writelock* conflicts with readlocks, ww writelocks, and rww writelocks. Thus, using basic 2PL for rw synchronization, every prewrite sets rw writelocks, and may set stronger locks depending on the ww technique.

Method 1: *Basic 2PL for ww synchronization.* All writelocks are rww writelocks; that is, for $i = 1, \ldots, m$, a writelock on x_i conflicts with either a readlock or a writelock on x_i. This is the "standard" distributed implementation of 2PL.

Method 2: *Primary copy 2PL for ww synchronization.* Writelocks only conflict on the primary copy. An rww writelock is used on the primary copy, while rw writelocks are used on the others.

Method 3: *Voting 2PL for ww synchronization.* A DM responds to a prewrite(x_i) by *attempting* to set an rww writelock on x_i. However, if another transaction already owns an rww writelock on x_i, the DM only sets an rw writelock and leaves a request for an rww writelock pending. A transaction can write into any copy of X after it obtains rww writelocks on a majority of copies. This is similar to the method proposed in GIFF79.

Method 4: *Centralized 2PL for ww synchronization.* To write into X, a transaction must first explicitly request a ww writelock on X from a centralized 2PL scheduler. The rw writelocks set by prewrites *never* conflict with each other.

In all four methods, readlocks are explicitly released by lock releases while writelocks are implicitly released by dm-writes. Lock releases may be transmitted in parallel with dm-writes. In Method 4, *after all* dm-writes have been executed, additional lock releases must be sent to the centralized scheduler to release writelocks held there.

5.1.2 Methods Using Primary Copy 2PL for rw Synchronization

Methods 5–8 use primary copy 2PL for rw synchronization. Consider a logical data item X with copies $x_1, \ldots, x_m$, and assume x_1 is the primary copy. To read X, a transaction must obtain a readlock on x_1. It may obtain this lock by issuing a dm-read(x_1). Alternatively, the transaction can send an explicit lock request to x_1's DM; when the lock is granted the transaction can read *any* copy of X.

To write into X, a transaction sends prewrites to *every* DM that stores a copy of X. A prewrite(x_1) implicitly requests an rw writelock. Prewrites on other copies of X may also request writelocks depending on the ww technique.

Method 5: *Basic 2PL for ww synchronization.* For $i = 2, \ldots, m$, prewrite(x_i) requests a ww writelock. Since the writelock on x_1 must also conflict with readlocks on x_1, prewrite(x_1) requests an rww writelock.

Method 6: *Primary copy 2PL for ww synchronization.* Prewrite(x_1) requests an rww writelock on x_1. Prewrites on other copies do not request any locks. This method was originally proposed by STON79 and is used in Distributed INGRES [STON77].

Method 7: *Voting 2PL for ww synchronization.* When a scheduler receives a prewrite(x_i) for $i \neq 1$, it tries to set a ww writelock on x_i. When it receives a prewrite(x_1), it tries to set an rww writelock on x_1; if it cannot, then it sets an rw writelock on x_1 (if possible) before waiting for the rww writelock. A transaction can write into every copy of X after it obtains a ww (or rww) writelock on a majority of copies of X.

Method 8: *Centralized 2PL for ww synchronization.* Transactions obtain ww writelocks from a centralized 2PL scheduler. Thus a prewrite(x_1) requests an rw writelock on x_1; for $i = 2, \ldots, m$, prewrite(x_i) does not request any lock.

Lock releases for Methods 5–8 are handled as in Section 5.1.1.

5.1.3 Methods Using Centralized 2PL for rw Synchronization

The remaining 2PL methods use centralized 2PL for rw synchronization. Before reading (or writing) any copy of logical data item X, a transaction must obtain a readlock (or rw writelock) on X from a centralized 2PL scheduler. Before writing X, the transaction must also send prewrites to every DM that stores a copy of X. Some of these prewrites implicitly request ww writelocks on copies of X, depending on the specific method.

Method 9: *Basic 2PL for ww synchronization.* Every prewrite requests a ww writelock.

Method 10: *Primary copy 2PL for ww synchronization.* If x_1 is the primary copy of X, a prewrite(x_1) requests a ww writelock. Prewrites on other copies do not request any writelocks.

Method 11: *Voting 2PL for ww synchronization.* Every prewrite *attempts* to set a ww writelock. A transaction can write into every copy of X after it obtains ww writelocks on a majority of copies of X.

Method 12: *Centralized 2PL for ww synchronization.* All locks are obtained at the centralized 2PL scheduler. Before writing into any copy of X, an rww writelock on X is obtained from the centralized scheduler. Prewrites set no locks at all. Method 12 is the "standard" implementation of centralized 2PL (called *primary site* in ALSB76a).

Lock releases for Methods 9–12 are handled as in Section 5.1.1.

5.2 Pure T/O Methods

The T/O synchronization techniques of Section 5 can also be integrated to form 12 *principal T/O methods*:

Method	rw technique	ww technique
1	Basic T/O	Basic T/O
2	Basic T/O	Thomas Write Rule (TWR)
3	Basic T/O	Multiversion T/O
4	Basic T/O	Conservative T/O
5	Multiversion T/O	Basic T/O
6	Multiversion T/O	TWR
7	Multiversion T/O	Multiversion T/O
8	Multiversion T/O	Conservative T/O
9	Conservative T/O	Basic T/O
10	Conservative T/O	TWR
11	Conservative T/O	Multiversion T/O
12	Conservative T/O	Conservative T/O

(That there are also 12 2PL methods is coincidental.)

Each T/O method that incorporates a conservative component can be refined by including classes and conflict graph analysis (see Sections 4.4.2 and 4.4.3).

The interface between rw and ww synchronization techniques is even simpler for T/O methods than for 2PL. The only requirement is that both techniques use the *same* timestamp for any given transaction.

5.2.1 Methods Using Basic T/O for rw Synchronization

Methods 1–4 use basic T/O for rw synchronization. All four methods require R-ts's for each data item. Methods 1, 2, and 4 require W-ts's, while in Method 3 each data item has a set of timestamped versions; for Method 3, let W-ts(x) denote x's largest timestamp. Each method buffers dm-reads and prewrites for two-phase commitment purposes; let min-R-ts(x) and min-P-ts(x) be the minimum timestamps of any buffered dm-read(x) and prewrite(x), respectively.

These methods can be described by the following steps. Let R be a dm-read(x), P a prewrite(x), and W a dm-write(x).

1. If ts(R) $<$ W-ts(x), R is rejected. Else if ts(R) $>$ min-P-ts(x), R is buffered. Else R is output and R-ts(x) is set to max(R-ts(x), ts(R)).
2. If ts(P) $<$ R-ts(x) or *condition (A)*[6] holds, P is rejected. Else P is buffered.
3. If ts(W) $>$ min-R-ts(x) or *condition (B)*[6] holds, W is buffered. Else W is output and W-ts(x) is set to max(W-ts(x), ts(W)). For Method 3, a new version of x is created with timestamp ts(W).
4. When W is output, its prewrite is debuffered and the buffered dm-reads and dm-writes are retested to see if any can now be output.

Method 1: *Basic T/O for ww synchronization.* Condition (A) is ts(P) $<$ W-ts(x) and condition (B) is ts(W) $>$ min-P-ts(x). Note that min-R-ts(x) $>$ min-P-ts(x), since R is buffered only if ts(R) $>$ min-P-ts(x). Also, when W is output, ts(W) $>$ W-ts(x), since condition (B) forces dm-writes on a given x to be output in timestamp order. Thus step 3 simplifies to

3. If ts(W) $>$ min-P-ts(x), W is buffered. Else W is output and W-ts(x) is set to ts(W).

Method 2: *TWR for ww synchronization.* Conditions (A) and (B) are null. However,

[6] Conditions (A) and (B) are determined by the new technique. See the following.

if ts(*W*) < W-ts(*x*), *W* has no effect on the database.

Method 3: *Multiversion T/O for ww synchronization.* Like Method 2 except that *W* always creates a new version of *x*.

Method 4: *Conservative T/O for ww synchronization.* Condition (A) is null. For each TM, let min-W-ts(TM) be the minimum timestamp of any buffered dm-write from that TM. Condition (B) is ts(*W*) > min-W-ts(TM) for some TM. As in Method 1, this causes dm-writes on a given *x* to be output in timestamp order, and step 3 simplifies to

3. If ts(*W*) > min-R-ts(*x*) or ts(*W*) > min-W-ts(TM) for some TM, *W* is buffered. Else *W* is output and W-ts(*x*) is set to ts(*W*).

5.2.2 Methods Using Multiversion T/O for rw Synchronization

Methods 5–8 use multiversion T/O for rw synchronization and require a set of R-ts's and a set of versions for each data item. These methods can be described by the following steps. Define *R*, *P*, *W*, min-R-ts, min-W-ts, and min-P-ts as above; let interval(P) be the interval from ts(*P*) to the smallest W-ts(*x*) > ts(*P*).

1. *R* is never rejected. If ts(*R*) lies in interval(prewrite(*x*)) for some buffered prewrite(*x*), then *R* is buffered. Else *R* is output and ts(*R*) is added to *x*'s set of R-ts's.
2. If some R-ts(*x*) lies in interval(*P*) or *condition* (*A*) holds, then *P* is rejected. Else *P* is buffered.
3. If *condition* (*B*) holds, *W* is buffered. Else *W* is output and creates a new version of *x* with timestamp ts(*W*).
4. When *W* is output, its prewrite is debuffered, and buffered dm-reads and dm-writes are retested.

Method 5: *Basic T/O for ww synchronization.* Condition (A) is ts(*P*) < max-W-ts(*x*) and condition (B) is ts(*W*) > min-P-ts(*x*). Condition (A) implies that interval(*P*) = (ts(*P*), ∞); some R-ts(*x*) lies in that interval if and only if ts(*P*) < maximum R-ts(*x*). Thus step 2 simplifies to

2. If ts(*P*) < max W-ts(*x*) or ts(*P*) < max R-ts(*x*), then *P* is rejected. Else it is buffered.

Because of this simplification, the method only requires that the maximum R-ts(*x*) be stored.

Condition (B) forces dm-writes on a given data item to be output in timestamp order. This supports a systematic technique for "forgetting" old versions. Let max-W-ts(*x*) be the maximum W-ts(*x*) and let min-ts be the minimum of max-W-ts(*x*) over all data items in the database. No dm-write with timestamp less than min-ts can be output in the future. Therefore, insofar as update transactions are concerned, we can safely forget all versions timestamped less than min-ts. TMs should be kept informed of the current value of min-ts and queries (read-only transactions) should be assigned timestamps greater than min-ts. Also, after a new min-ts is selected, older versions should not be forgotten immediately, so that active queries with smaller timestamps have an opportunity to finish.

Method 6: *TWR for ww synchronization.* This method is *incorrect.* TWR requires that W be ignored if ts(*W*) < max W-ts(*x*). This may cause later dm-reads to be read incorrect data. See Figure 15. (Method 6 is the only incorrect method we will encounter.)

Method 7: *Multiversion T/O for ww synchronization.* Conditions (A) and (B) are null. Note that this method, unlike all previous ones, never buffers dm-writes.

Method 8: *Conservative T/O for ww synchronization.* Condition (A) is null. Condition (B) is ts(*W*) > min-W-ts(TM) for some TM. Condition (B) forces dm-writes to be output in timestamp order, implying interval(*P*) = (ts(*P*), ∞). As in Method 5, this simplifies step 2:

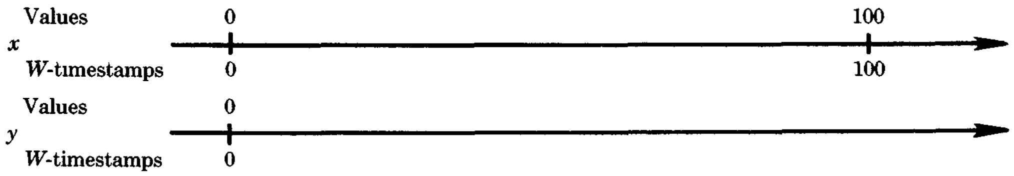
2. If ts(*P*) < max R-ts(*x*), P is rejected; else it is buffered.

Like Method 5, this method only requires that the maximum R-ts(*x*) be stored, and it supports systematic "forgetting" of old versions described above.

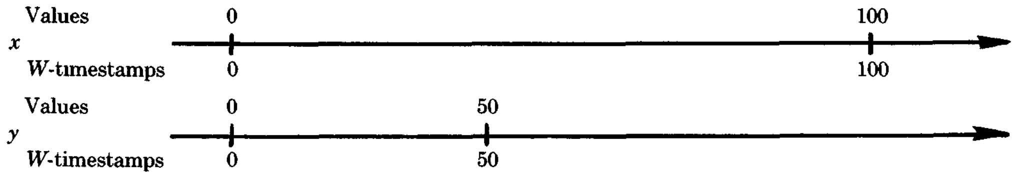
5.2.3 Methods Using Conservative T/O for rw Synchronization

The remaining T/O methods use conservative T/O for rw synchronization. Methods

- 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 $x = 50, y = 50$.

Figure 15. Inconsistent retrievals in Method 6.

9 and 10 require W-ts's for each data item, and Method 11 requires a set of versions for each data item. Method 12 needs no data item timestamps at all. Define R, P, W and min-P-ts as in Section 5.2.1; let min-R-ts(TM) (or min-W-ts(TM)) be the minimum timestamp of any buffered dm-read (or dm-write) from TM.

1. If ts(R) > min-W-ts(TM) for any TM, R is buffered; else it is output.
2. If *condition* (A) holds, P is rejected. Else P is buffered.
3. If ts(W) > min-R-ts(TM) for any TM or *condition* (B) holds, W is buffered. Else W is output.
4. When W is output, its prewrite is debuffered. When R or W is output or buffered, buffered dm-reads and dm-writes are retested to see if any can now be output.

Method 9: *Basic T/O for ww synchronization.* Condition (A) is ts(P) < W-ts(x), and condition (B) is ts(W) > min-P-ts(x).

Method 10: *TWR for ww synchronization.* Conditions (A) and (B) are null. However, if ts(W) < W-ts(x), W has no effect on the database.

This method is essentially the SDD-1 concurrency control [BERN80d], although in SDD-1 the method is refined in several ways. SDD-1 uses classes and conflict graph analysis to reduce communication and increase the level of concurrency. Also, SDD-1 requires predeclaration of read-sets and only enforces the conservative scheduling on dm-reads. By doing so, it forces dm-reads to wait for dm-writes, but does not insist that dm-writes wait for all dm-reads with smaller timestamps. Hence dm-reads can be rejected in SDD-1.

Method 11: *Multiversion T/O for ww synchronization.* Conditions (A) and (B) are null. When W is output, it creates a new version of x with timestamp ts(W). When R is output it reads the version with largest timestamp less than ts(R).

This method can be optimized by noting the multiversion T/O "automatically" prevents dm-reads from being rejected, and makes it unnecessary to buffer dm-writes. Thus step 3 can be simplified to

3. W is output immediately.

Method 12: *Conservative T/O for ww synchronization.* Condition (A) is null; condition (B) is ts(W) > min-W-ts(TM) for some TM. The effect is to output W if the scheduler has received *all* operations with timestamps less than ts(W) that it will ever receive. Method 12 has been proposed in CHEN80, KANE79, and SHAP77a.

5.3 Mixed 2PL and T/O Methods

The major difficulty in constructing methods that combine 2PL and T/O lies in developing the interface between the two techniques. Each technique guarantees an acyclic $\rightarrow_{\mathrm{rwr}}$ (or $\rightarrow_{\mathrm{ww}}$) relation when used for rw (or ww) synchronization. The interface between a 2PL and a T/O technique must guarantee that the combined $\rightarrow$ relation (i.e., $\rightarrow_{\mathrm{rwr}} \cup \rightarrow_{\mathrm{ww}}$) remains acyclic. That is, the interface must ensure that the serialization order induced by the rw technique is consistent with that induced by the ww technique. In Section 5.3.1 we describe an interface that makes this guarantee. Given such an interface, *any* 2PL technique can be integrated with *any* T/O technique. Sections 5.3.2 and 5.3.3 describe such methods.

5.3.1 The Interface

The serialization order induced by any 2PL technique is determined by the locked points of the transactions that have been synchronized (see Section 3). The serialization order induced by any T/O technique is determined by the timestamps of the synchronized transactions. So to interface 2PL and T/O we *use locked points to induce timestamps* [BERN80b].

Associated with each data item is a *lock timestamp*, L-ts(x). When a transaction T sets a lock of x, it simultaneously retrieves L-ts(x). When T reaches its locked point it is assigned a timestamp, ts(T), greater than any L-ts it retrieved. When T releases its lock on x, it updates L-ts(x) to be max(L-ts(x), ts(T)).

Timestamps generated in this way are consistent with the serialization order induced by 2PL. That is, ts(T_j) < ts(T_k) if T_j must precede T_k in any serialization induced by 2PL. To see this, let T_1 and T_n be a pair of transactions such that T_1 must precede T_n in any serialization. Thus there exist transactions $T_1, T_2 \ldots, T_{n-1}, T_n$ such that for $i = 1, \ldots, n\text{-}1$ (a) T_i's locked point precedes T_{i+1}'s locked point, and (b) T_i released a lock on some data item x before T_{i+1} obtained a lock on x. Let L be the L-ts(x) retrieved by T_{i+1}. Then ts(T_i) < L < ts(T_{i+1}), and by induction ts(T_1) < ts(T_n).

5.3.2 Mixed Methods Using 2PL for rw Synchronization

There are 12 principal methods in which 2PL is used for rw synchronization and T/O is used for ww synchronization:

Method	rw technique	ww technique
1	Basic 2PL	Basic T/O
2	Basic 2PL	TWR
3	Basic 2PL	Multiversion T/O
4	Basic 2PL	Conservative T/O
5	Primary copy 2PL	Basic T/O
6	Primary copy 2PL	TWR
7	Primary copy 2PL	Multiversion T/O
8	Primary copy 2PL	Conservative T/O
9	Centralized 2PL	Basic T/O
10	Centralized 2PL	TWR
11	Centralized 2PL	Multiversion T/O
12	Centralized 2PL	Conservative T/O

Method 2 best exemplifies this class of methods, and it is the only one we describe in detail. Method 2 requires that every stored data item have an L-ts and a W-ts. (One timestamp can serve both roles, but we do not consider this optimization here.)

Let X be a logical data item with copies $x_1, \ldots, x_m$. To read X, transaction T issues a dm-read on any copy of X, say x_i. This dm-read implicitly requests a readlock on x_i, and when the readlock is granted, L-ts(x_i) is returned to T. To write into X, T issues prewrites on every copy of X. These prewrites implicitly request rw writelocks on the corresponding copies, and as each writelock is granted, the corresponding L-ts is returned to T. When T has obtained all of its locks, ts(T) is calculated as in Section 5.3.1. T attaches ts(T) to its dm-writes, which are then sent.

Dm-writes are processed using TWR. Let W be dm-write(x_j). If ts(W) > W-ts(x_j), the dm-write is processed as usual (x_j is updated). If, however, ts(W) < W-ts(x_j), W is *ignored.*

The interesting property of this method is that *writelocks never conflict with writelocks.* The writelocks obtained by prewrites are only used for rw synchronization, and only conflict with readlocks. This permits transactions to execute concurrently to completion even if their writesets intersect. Such concurrency is never possible in a pure 2PL method.

5.3.3 *Mixed Methods Using T/O for rw Synchronization*

There are also 12 principal methods that use T/O for rw synchronization and 2PL for ww synchronization:

Method	rw technique	ww technique
13	Basic T/O	Basic 2PL
14	Basic T/O	Primary copy 2PL
15	Basic T/O	Voting 2PL
16	Basic T/O	Centralized 2PL
17	Multiversion T/O	Basic 2PL
18	Multiversion T/O	Primary copy 2PL
19	Multiversion T/O	Voting 2PL
20	Multiversion T/O	Centralized 2PL
21	Conservative T/O	Basic 2PL
22	Conservative T/O	Primary copy 2PL
23	Conservative T/O	Voting 2PL
24	Conservative T/O	Centralized 2PL

These methods all require *predeclaration of writelocks.* Since T/O is used for rw synchronization, transactions must be assigned timestamps before they issue dm-reads. However, the timestamp generation technique of Section 5.3.1 requires that a transaction be at its locked point before it is assigned its timestamp. Hence every transaction must be at its locked point before it issues any dm-reads; in other words, every transaction must obtain all of its writelocks before it begins its main execution.

To illustrate these methods, we describe Method 17. This method requires that each stored data item have a set of R-ts's and a set of ⟨W-ts, value⟩ pairs (i.e., versions). The L-ts of any data item is the maximum of its R-ts's and W-ts's.

Before beginning its main execution, transaction T issues prewrites on every copy of every data item in its writeset.[7] These prewrites play a role in ww synchronization, rw synchronization, and the interface between these techniques.

Let P be a prewrite(x). The ww role of P is to request a ww writelock on x. When the lock is granted, L-ts(x) is returned to T; this is the interface role of P. Also when the lock is granted, P is buffered and the rw synchronization mechanism is informed that a dm-write with timestamp greater than L-ts(x) is pending. This is its rw role.

When T has obtained all of its writelocks, ts(T) is calculated as in Section 5.3.1 and T begins its main execution. T attaches ts(T) to its dm-reads and dm-writes and rw synchronization is performed by multiversion T/O, as follows:

1. Let R be a dm-read(x). If there is a buffered prewrite(x) (other than one issued by T), and if L-ts(x) < ts(T), then R is buffered. Else R is output and reads the version of x with largest timestamp less than ts(T).
2. Let W be a dm-write(x). W is output immediately and creates a new version of x with timestamp ts(T).
3. When W is output, its prewrite is debuffered, and its writelock on x is released. This causes L-ts(x) to be updated to max(L-ts(x), ts(T)) = ts(T).

One interesting property of this method is that restarts are needed only to prevent or break deadlocks caused by ww synchronization; rw conflicts never cause restarts. This property cannot be attained by a pure 2PL method. It can be attained by pure T/O methods, but only if conservative T/O is used for rw synchronization; in many cases conservative T/O introduces excessive delay or is otherwise infeasible.

The behavior of this method for queries is also interesting. Since queries set no writelocks, the timestamp generation rule does not apply to them. Hence *the system is free to assign any timestamp it wishes to a query.* It may assign a small timestamp, in which case the query will read old data but is unlikely to be delayed by buffered prewrites; or it may assign a large timestamp, in which case the query will read current data but is more likely to be delayed. No matter which timestamp is selected, however, *a query can never cause an update to be rejected.* This property cannot be easily attained by any pure 2PL or T/O method.

We also observe that this method creates versions in timestamp order, and so sys-

[7] Since new values for the data items in the writeset are not yet known, these prewrites do not instruct DMs to store values on secure storage, they merely "warn" DMs to "expect" the corresponding dm-writes See footnote 3.

tematic forgetting of old versions is possible (see Section 5.2.2). In addition, the method requires only *maximum* R-ts's; smaller ones may be instantly forgotten.

CONCLUSION

We have presented a framework for the design and analysis of distributed database concurrency control algorithms. The framework has two main components: (1) a system model that provides common terminology and concepts for describing a variety of concurrency control algorithms, and (2) a problem decomposition that decomposes concurrency control algorithms into read-write and write-write synchronization subalgorithms.

We have considered synchronization subalgorithms outside the context of specific concurrency control algorithms. Virtually all known database synchronization algorithms are variations of two basic techniques—two-phase locking (2PL) and timestamp ordering (T/O). We have described the principal variations of each technique, though we do not claim to have exhausted all possible variations. In addition, we have described ancillary problems (e.g., deadlock resolution) that must be solved to make each variation effective.

We have shown how to integrate the described techniques to form complete concurrency control algorithms. We have listed 47 concurrency control algorithms, describing 25 in detail. This list includes almost all concurrency control algorithms described previously in the literature, plus several new ones. This extreme consolidation of the state of the art is possible in large part because of our framework set up earlier.

The focus of this paper has primarily been the *structure* and *correctness* of synchronization techniques and concurrency control algorithms. We have left open a very important issue, namely, performance.

The main performance metrics for concurrency control algorithms are system throughput and transaction response time. Four cost factors influence these metrics: intersite communication, local processing, transaction restarts, and transaction blocking. The impact of each cost factor on system throughput and response time varies from algorithm to algorithm, system to system, and application to application. This impact is not understood in detail, and a comprehensive quantitative analysis of performance is beyond the state of the art. Recent theses by Garcia-Molina [GARC79a] and Reis [REIS79a] have taken first steps toward such an analysis but there clearly remains much to be done.

We hope, and indeed recommend, that future work on distributed concurrency control will concentrate on the performance of algorithms. There are, as we have seen, many known methods; the question now is to determine which are best.

APPENDIX. OTHER CONCURRENCY CONTROL METHODS

In this appendix we describe three concurrency control methods that do not fit the framework of Sections 3-5: the certifier methods of Badal [BADA79], Bayer et al. [BAYE80], and Casanova [CASA79], the majority consensus algorithm of Thomas [THOM79], and the ring algorithm of Ellis [ELLI77]. We argue that these methods are not practical in DDBMSs. The certifier methods look promising for *centralized* DBMSs, but severe technical problems must be overcome before these methods can be extended correctly to distributed systems. The Thomas and Ellis algorithms, by contrast, are among the earliest algorithms proposed for DDBMS concurrency control. These algorithms introduced several important techniques into the field but, as we will see, have been surpassed by recent developments.

A1. Certifiers

A1.1 The Certification Approach

In the certification approach, dm-reads and prewrites are processed by DMs first-come/first-served, with no synchronization whatsoever. DMs do maintain summary information about rw and ww conflicts, which they update every time an operation is processed. However, dm-reads and prewrites are never blocked or rejected on the basis of the discovery of such a conflict.

Synchronization occurs when a transaction attempts to terminate. When a transaction T issues its END, the DBMS decides whether or not to *certify*, and thereby commit, T.

To understand how this decision is made, we must distinguish between "total" and "committed" executions. A *total execution* of transactions includes the execution of all operations processed by the system up to a particular moment. The *committed execution* is the portion of the total execution that only includes dm-reads and dm-writes processed on behalf of committed transactions. That is, the committed execution is the total execution that would result from aborting all active transactions (and not restarting them).

When T issues its END, the system tests whether the committed execution augmented by T's execution is serializable, that is, whether after committing T the resulting committed execution would still be serializable. If so, T is committed; otherwise T is restarted.

There are two properties of certification that distinguish it from other approaches. First, synchronization is accomplished entirely by restarts, never by blocking. And second, the decision to restart or not is made *after* the transaction has finished executing. No concurrency control method discussed in Sections 3–5 satisifies both these properties.

The rationale for certification is based on an optimistic assumption regarding run-time conflicts: if very few run-time conflicts are expected, assume that most executions are serializable. By processing dm-reads and prewrites without synchronization, the concurrency control method never delays a transaction while it is being processed. Only a (fast, it is hoped) certification test when the transaction terminates is required. Given optimistic transaction behavior, the test will usually result in committing the transaction, so there are very few restarts. Therefore certification simultaneously avoids blocking and restarts in optimistic situations.

A certification concurrency control method must include a *summarization algorithm* for storing information about dm-reads and prewrites when they are processed and a *certification algorithm* for using that information to certify transactions when they terminate. The main problem in the summarization algorithm is avoiding the need to store information about already-certified transactions. The main problem in the certification algorithm is obtaining a *consistent* copy of the summary information. To do so the certification algorithm often must perform some synchronization of its own, the cost of which must be included in the cost of the entire method.

A1.2 Certification Using the $\rightarrow$ Relation

One certification method is to construct the $\rightarrow$ relation as dm-reads and prewrites are processed. To certify a transaction, the system checks that $\rightarrow$ is acyclic [BADA79, BAYE80, CASA79].[8]

To construct $\rightarrow$, each site remembers the most recent transaction that read or wrote each data item. Suppose transactions T_i and T_j were the last transactions to (respectively) read and write data item x. If transaction T_k now issues a dm-read(x), $T_j \rightarrow T_k$ is added to the summary information for the site and T_k replaces T_i as the last transaction to have read x. Thus pieces of $\rightarrow$ are distributed among the sites, reflecting run-time conflicts at each site.

To certify a transaction, the system must check that the transaction does not lie on a cycle in $\rightarrow$ (see Theorem 2, Section 2). Guaranteeing acyclicity is sufficient to guarantee serializability.

There are two problems with this approach. First, it is in general not correct to delete a certified transaction from $\rightarrow$, even if all of its updates have been committed. For example, if $T_i \rightarrow T_j$ and T_i is active but T_j is committed, it is still possible for $T_j \rightarrow T_i$ to develop; deleting T_j would then cause the cycle $T_i \rightarrow T_j \rightarrow T_i$ to go unnoticed when T_i is certified. However, it is obviously not feasible to allow $\rightarrow$ to grow indefinitely. This problem is solved by Casanova [CASA79] by a method of encoding information about committed transactions in space proportional to the number of active transactions.

A second problem is that *all* sites must be checked to certify *any* transaction. Even

[8] In BAYE80 certification is only used for rw synchronization whereas 2PL is used for ww synchronization.

sites at which the transaction never accessed data must participate in the cycle checking of $\rightarrow$. For example, suppose we want to certify transaction T. T might be involved in a cycle $T \rightarrow T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_{n-1} \rightarrow T_n \rightarrow T$, where each conflict $T_k \rightarrow T_{k+1}$ occurred at a different site. Possibly T only accessed data at one site; yet the $\rightarrow$ relation must be examined at n sites to certify T. This problem is currently unsolved, as far as we know. That is, any *correct* certifier based on this approach of checking cycles in $\rightarrow$ must access the $\rightarrow$ relation at *all* sites to certify each and every transaction. Until this problem is solved, we judge the certification approach to be impractical in a distributed environment.

A2. Thomas' Majority Consensus Algorithm

A2.1 The Algorithm

One of the first published algorithms for distributed concurrency control is a certification method described in THOM79. Thomas introduced several important synchronization techniques in that algorithm, including the Thomas Write Rule (Section 3.2.3), majority voting (Section 3.1.1), and certification (Appendix A1). Although these techniques are valuable when considered in isolation, we argue that the overall Thomas algorithm is not suitable for distributed databases. We first describe the algorithm and then comment on its application to distributed databases.

Thomas' algorithm assumes a fully redundant database, with every logical data item stored at every site. Each copy carries the timestamp of the last transaction that wrote into it.

Transactions execute in two phases. In the first phase each transaction executes locally at one site called the transaction's *home site*. Since the database is fully redundant, any site can serve as the home site for any transaction. The transaction is assigned a unique timestamp when it begins executing. During execution it keeps a record of the timestamp of each data item it reads and, when its executes a write on a data item, processes the write by recording the new value in an *update list*. Note that each transaction must read a copy of a data item before it writes into it. When the transaction terminates, the system augments the update list with the list of data items read and their timestamps at the time they were read. In addition, the timestamp of the transaction itself is added to the update list. This completes the first phase of execution.

In the second phase the update list is sent to every site. Each site (including the site that produced the update list) *votes* on the update list. Intuitively speaking, a site votes yes on an update list if it can certify the transaction that produced it. After a site votes yes, the update list is said to be *pending* at that site. To cast the vote, the site sends a message to the transaction's home site, which, when it receives a majority of yes or no votes, informs all sites of the outcome. If a majority voted yes, then all sites are required to commit the update, which is then installed using TWR. If a majority voted no, all sites are told to discard the update, and the transaction is restarted.

The rule that determines when a site may vote "yes" on a transaction is pivotal to the correctness of the algorithm. To vote on an update list *U*, a site compares the timestamp of each data item in the readset of *U* with the timestamp of that same data item in the site's local database. If any data item has a timestamp in the database different from that in *U*, the site votes *no*. Otherwise, the site compares the readset and writeset of *U* with the readset and writeset of each pending update list at that site, and if there is no rw conflict between *U* and any of the pending update lists, it votes *yes*. If there is an rw conflict between *U* and one of those pending requests, the site votes *pass* (abstain) if *U*'s timestamp is larger than that of all pending update lists with which it conflicts. If there is an rw conflict but *U*'s timestamp is smaller than that of the conflicting pending update list, then it sets *U* aside on a *wait queue* and tries again when the conflicting request has either been committed or aborted at that site.

The voting rule is essentially a certification procedure. By making the timestamp comparison, a site is checking that the readset was not written into since the transaction read it. If the comparisons are satisfied, the situation is as if the transaction had locked its readset at that site and held the locks until it voted. The voting rule is thereby guaranteeing rw synchronization with a certification rule approximating rw 2PL. (This fact is proved precisely in BERN79b.)

The second part of the voting rule, in which U is checked for rw conflicts against pending update lists, guarantees that conflicting requests are not certified concurrently. An example illustrates the problem. Suppose T_1 reads X and Y, and writes Y, while T_2 reads X and Y, and writes X. Suppose T_1 and T_2 execute at sites A and B, respectively, and X and Y have timestamps of 0 at both sites. Assume that T_1 and T_2 execute concurrently and produce update lists ready for voting at about the same time. Either T_1 or T_2 must be restarted, since neither read the other's output; if they were both committed, the result would be nonserializable. However both T_1's and T_2's update lists will (concurrently) satisfy the timestamp comparison at both A and B. What stops them from both obtaining unanimous yes votes is the second part of the voting rule. After a site votes on one of the transactions, it is prevented from voting on the other transaction until the first is no longer pending. Thus it is not possible to certify conflicting transactions concurrently. (We note that this problem of concurrent certification exists in the algorithms of Section A1.2, too. This is yet another technical difficulty with the certification approach in a distributed environment.)

With the second part of the voting rule, the algorithm behaves as if the certification step were atomically executed at a primary site. If certification were centralized at a primary site, the certification step at the primary site would serve the same role as the majority decision in the voting case.

A2.2 Correctness

No simple proof of the serializability of Thomas' algorithm has ever been demonstrated, although Thomas provided a detailed "plausibility" argument in THOM79. The first part of the voting rule can correctly be used in a centralized concurrency control method since it implies 2PL [BERN79b], and a centralized method based on this approach was proposed in KUNG81. The second part of the voting rule guarantees that for every pair of conflicting transactions that received a majority of yes votes, all sites that voted yes on both transactions voted on the two transactions in the same order. This makes the certification step behave just as it would if it were centralized, thereby avoiding the problem exemplified in the previous paragraph.

A2.3 Partially Redundant Databases

For the majority consensus algorithm to be useful in a distributed database environment, it must be generalized to operate correctly when the database is only partially redundant. There is reason to doubt that such a generalization can be accomplished without either serious degradation of performance or a complete change in the set of techniques that are used.

First, the majority consensus decision rule apparently must be dropped, since the voting algorithm depends on the fact that all sites perform exactly the same certification test. In a partially redundant database, each site would only be comparing the timestamps of the data items stored at that site, and the significance of the majority vote would vanish.

If majority voting cannot be used to synchronize concurrent certification tests, apparently some kind of mutual exclusion mechanism must be used instead. Its purpose would be to prevent the concurrent, and therefore potentially incorrect, certification of two conflicting transactions, and would amount to locking. The use of locks for synchronizing the certification step is not in the spirit of Thomas' algorithm, since a main goal of the algorithm was to avoid locking. However, it is worth examining such a locking mechanism to see how certification can be correctly accomplished in a partially redundant database.

To process a transaction T, a site produces an update list as usual. However, since the database is partially redundant, it may be necessary to read portions of T's readset from other sites. After T terminates, its update list is sent to every site that contains part of T's readset or writeset. To certify an update list, a site first sets local locks on the readset and writeset, and then (as in the fully redundant case) it compares the update list's timestamps with the database's timestamps. If they are identical, it votes yes; otherwise it votes no. A unanimous vote of yes is needed to commit the updates. Local locks cannot be released until the voting decision is completed.

While this version of Thomas' algorithm for partially redundant data works correctly, its performance is inferior to standard 2PL. This algorithm requires that the same locks be set as in 2PL, and the same deadlocks can arise. Yet the probability of restart is higher than in 2PL, because even after all locks are obtained the certification step can still vote no (which cannot happen in 2PL).

One can improve this algorithm by designating a primary copy of each data item and only performing the timestamp comparison against the primary copy, making it analogous to primary copy 2PL. However, for the same reasons as above, we would expect primary copy 2PL to outperform this version of Thomas' algorithm too.

We therefore must leave open the problem of producing an efficient version of Thomas' algorithm for a partially redundant database.

A2.4 Performance

Even in the fully redundant case, the performance of the majority consensus algorithm is not very good. First, repeating the certification and conflict detection at each site is more than is needed to obtain serializability: a centralized certifier would work just as well and would only require that certification be performed at one site. Second, the algorithm is quite prone to restarts when there are run-time conflicts, since restarts are the only tactic available for synchronizing transactions, and so will only perform well under the most optimistic circumstances. Finally, even in optimistic situations, the analysis in GARC79a indicates that centralized 2PL outperforms the majority consensus algorithm.

A2.5 Reliability

Despite the performance problems of the majority consensus algorithm, one can try to justify the algorithms on reliability grounds. As long as a majority of sites are correctly running, the algorithm runs smoothly. Thus, handling a site failure is free, insofar as the voting procedure is concerned.

However, from current knowledge, this justification is not compelling for several reasons. First, although there is no cost when a site fails, substantial effort may be required when a site recovers. A centralized algorithm using backup sites, as in ALSB76a, lacks the symmetry of Thomas' algorithm, but may well be more efficient due to the simplicity of site recovery. In addition, the majority consensus algorithm does not consider the problem of atomic commitment and it is unclear how one would integrate two-phase commit into the algorithm.

Overall, the reliability threats that are handled by the majority consensus algorithm have not been explicitly listed, and alternative solutions have not been analyzed. While voting is certainly a possible technique for obtaining a measure of reliability, the circumstances under which it is cost-effective are unknown.

A3. Ellis' Ring Algorithm

Another early solution to the problem of distributed database concurrency control is the ring algorithm [ELLI77]. Ellis was principally interested in a proof technique, called *L systems*, for proving the correctness of concurrent algorithms. He developed his concurrency control method primarily as an example to illustrate L-system proofs, and never made claims about its performance. Because the algorithm was only intended to illustrate mathematical techniques, Ellis imposed a number of restrictions on the algorithm for mathematical convenience, which make it infeasible in practice. Nonetheless, the algorithm has received considerable attention in the literature, and in the interest of completeness, we briefly discuss it.

Ellis' algorithm solves the distributed concurrency control problem with the following restrictions:

(1) The database must be fully redundant.
(2) The communication medium must be a ring, so each site can only communicate with its successor on the ring.

(3) Each site-to-site communication link is pipelined.
(4) Each site can supervise no more than one active update transaction at a time.
(5) To update any copy of the database, a transaction must first obtain a lock on the entire database at all sites.

The effect of restriction 5 is to force all transactions to execute *serially*; no concurrent processing is ever possible. For this reason alone, the algorithm is fundamentally impractical.

To execute, an update transaction migrates around the ring, (essentially) obtaining a lock *on the entire database at each site*. However, the lock conflict rules are nonstandard. A lock request from a transaction that originated at site A conflicts at site C with a lock held by a transaction that originated from site B if B = C and either A = B or A's priority < B's priority. The daisy-chain communication induced by the ring combined with this locking rule produces a deadlock-free algorithm that does not require deadlock detection and never induces restarts. A detailed description of the algorithm appears in GARC79a.

There are several problems with this algorithm in a distributed database environment. First, as mentioned above, it forces transactions to execute serially. Second, it only applies to a fully redundant database. And third, the daily-chain communication requires that each transaction obtain its lock at one site at a time, which causes communication delay to be (at least) linearly proportional to the number of sites in the system.

A modified version of Ellis' algorithm that mitigates the first problem is proposed in GARC79a. Even with this improvement, performance analysis indicates that the ring algorithm is inferior to centralized 2PL. And, of course, the modified algorithm still suffers from the last two problems.

ACKNOWLEDGMENT

This work was supported by Rome Air Development Center under contract F30602-79-C-0191.

REFERENCES

AHO75 AHO, A. V., HOPCROFT, E., AND ULLMAN, J. D. *The design and analysis of computer algorithms,* Addison-Wesley, Reading, Mass., 1975.

ALSB76a ALSBERG, P. A , AND DAY, J. D. "A principle for resilient sharing of distributed resources," in *Proc. 2nd Int. Conf. Software Eng.*, Oct. 1976, pp. 562–570.

ALSB76b ALSBERG, P. A., BELFORD, G.C., DAY, J. D., AND GRAPLA, E. "Multi-copy resiliency techniques," Center for Advanced Computation, AC Document No. 202, Univ. Illinois at Urbana-Champaign, May 1976.

BADA78 BADAL, D. Z., AND POPEK, G. J. "A proposal for distributed concurrency control for partially redundant distributed data base system," in *Proc. 3rd Berkeley Workshop Distributed Data Management and Computer Networks*, 1978, pp. 273–288

BADA79 BADAL, D. Z. "Correctness of concurrency control and implications in distributed databases," in *Proc COMPSAC 79 Conf.*, Chicago, Ill., Nov. 1979.

BADA80 BADAL, D. Z. "On the degree of concurrency provided by concurrency control mechanisms for distributed databases," in *Proc. Int Symp. Distributed Databases,* Versailles, France, March 1980.

BAYE80 BAYER, R., HELLER, H., AND REISER, A. "Parallelism and recovery in database systems," *ACM Trans. Database Syst.* **5,** 2 (June 1980), 139–156.

BELF76 BELFORD, G. C., SCHWARTZ, P. M., AND SLUIZER, S. "The effect of back-up strategy on database availability," CAC Document No. 181, CCTCWAD Document No. 5515, Center for Advanced Computation, Univ. Illinois at Urbana-Champaign, Urbana, Feb. 1976.

BERN78a BERNSTEIN, P. A., GOODMAN, N., ROTHNIE, J B., AND PAPADIMITRIOU, C. A. "The concurrency control mechanism of SDD-1: A system for distributed databases (the fully redundant case)," *IEEE Trans. Softw. Eng.* **SE-4,** 3 (May 1978), 154–168.

BERN79a BERNSTEIN, P. A., AND GOODMAN, N. "Approaches to concurrency control in distributed databases," in *Proc. 1979 Natl. Computer Conf.*, AFIPS Press, Arlington, Va., June 1979.

BERN79b BERNSTEIN, P. A., SHIPMAN, D. W., AND WONG, W. S. "Formal Aspects of Serializability in Database Concurrency Control," *IEEE Trans. Softw Eng.* **SE-5,** 3 (May 1979), 203-215.

BERN80a BERNSTEIN, P. A., AND GOODMAN, N. "Timestamp based algorithms for concurrency control in distributed database systems," *Proc 6th Int. Conf. Very Large Data Bases*, Oct. 1980.

BERN80b BERNSTEIN, P. A., GOODMAN, N., AND LAI, M. Y. "Two Part Proof Schema for Database Concurrency Control," in *Proc 5th Berkeley Workshop Distributed Data Management and Computer Networks,* Feb. 1980.

BERN80c BERNSTEIN, P. A , AND SHIPMAN, D. W "The correctness of concurrency control mechanisms in a system for distributed databases (SDD-1)," in *ACM Trans. Database Syst.* **5,** 1 (March 1980), 52–68.

BERN80d BERNSTEIN, P., SHIPMAN, D. W., AND ROTHNIE, J. B. "Concurrency control in a system for distributed databases (SDD-1)," in *ACM Trans. Database Syst* **5,** 1 (March 1980), 18–51.

BERN81 BERNSTEIN, P. A., GOODMAN, N., WONG, E., REEVE, C. L., AND ROTHNIE, J. B. "Query processing in SDD-1," *ACM Trans. Database Syst.* **6,** 2, to appear.

BREI79 BREITWIESER, H., AND KERSTEN, U. "Transaction and catalog management of the distributed file management system DISCO," in *Proc. Very Large Data Bases*, Rio de Janerio, 1979

BRIN73 BRINCH-HANSEN, P. *Operating system principles*, Prentice-Hall, Englewood Cliffs, N. J., 1973.

CASA79 CASANOVA, M. A. "The concurrency control problem for database systems," Ph.D. dissertation, Harvard Univ., Tech. Rep. TR-17-79, Center for Research in Computing Technology, 1979.

CHAM74 CHAMBERLIN, D. D., BOYCE, R. F., AND TRAIGER, I. L. "A deadlock-free scheme for resource allocation in a database environment," *Info. Proc.* **74,** North-Holland, Amsterdam, 1974.

CHEN80 CHENG, W. K., AND BELFORD, G. C. "Update Synchronization in Distributed Databases," in *Proc. 6th Int. Conf. Very Large Data Bases*, Oct. 1980.

DEPP76 DEPPE, M. E., AND FRY, J. P. "Distributed databases· A summary of research," in *Computer networks*, vol. 1, no. 2, North-Holland, Amsterdam, Sept. 1976.

DIJK71 DIJKSTRA, E. W. "Hierarchical ordering of sequential processes," *Acta Inf.* **1,** 2 (1971), 115–138.

ELLI77 ELLIS, C. A. "A robust algorithm for updating duplicate databases," in *Proc 2nd Berkeley Workshop Distributed Databases and Computer Networks*, May 1977.

ESWA76 ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. "The notions of consistency and predicate locks in a database system." *Commun. ACM* **19,** 11 (Nov. 1976), 624–633.

GARC78 GARCIA-MOLINA, H "Performance comparisons of two update algorithms for distributed databases," in *Proc. 3rd Berkeley Workshop Distributed Databases and Computer Networks*, Aug. 1978.

GARC79a GARCIA-MOLINA, H. "Performance of update algorithms for replicated data in a distributed database," Ph.D. dissertation, Computer Science Dept., Stanford Univ., Stanford, Calif., June 1979.

GARC79b GARCIA-MOLINA, H. "A concurrency control mechanism for distributed data bases which use centralized locking controllers," in *Proc. 4th Berkeley Workshop Distributed Databases and Computer Networks*, Aug. 1979.

GARC79c GARCIA-MOLINA, H. "Centralized control update algorithms for fully redundant distributed databases," in *Proc. 1st Int. Conf. Distributed Computing Systems* (IEEE), New York, Oct. 1979, pp. 699–705.

GARD77 GARDARIN, G., AND LEBAUX, P. "Scheduling algorithms for avoiding inconsistency in large databases," in *Proc. 1977 Int. Conf. Very Large Data Bases* (IEEE), New York, pp. 501–516.

GELE78 GELEMBE, E., AND SEVCIK, K. "Analysis of update synchronization for multiple copy databases," in *Proc. 3rd Berkeley Workshop Distributed Databases and Computer Networks*, Aug. 1978.

GIFF79 GIFFORD, D. K. "Weighted voting for replicated data," in *Proc. 7th Symp. Operating Systems Principles*, Dec. 1979.

GRAY75 GRAY, J. N., LORIE, R. A., PUTZULO, G. R., AND TRAIGER, I. L. "Granularity of locks and degrees of consistency in a shared database," IBM Res. Rep. RJ1654, Sept. 1975.

GRAY78 GRAY, J. N. "Notes on database operating systems," in *Operating Systems: An Advanced Course*, vol. 60, *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1978, pp. 393–481.

HAMM80 HAMMER, M. M., AND SHIPMAN, D. W. "Reliability mechanisms for SDD-1: A system for distributed databases," *ACM Trans. Database Syst.* **5,** 4 (Dec. 1980), 431–466.

HEWI74 HEWITT, C. E. "Protection and synchronization in actor systems," Working Paper No. 83, M.I.T. Artificial Intelligence Lab., Cambridge, Mass., Nov. 1974.

HOAR74 HOARE, C. A. R. "Monitors. An operating system structuring concept," *Commun. ACM* **17,** 10 (Oct. 1974), 549–557.

HOLT72 HOLT, R. C. "Some deadlock properties of computer systems," *Comput. Surv.* **4,** 3 (Dec. 1972) 179–195.

KANE79 KANEKO, A., NISHIHARA, Y., TSURUOKA, K., AND HATTORI, M. "Logical clock synchronization method for duplicated database control," in *Proc. 1st Int. Conf. Distributed Computing Systems* (IEEE), New York, Oct. 1979, pp. 601–611.

KAWA79 KAWAZU, S, MINAMI, ITOH, S., AND TERANAKA, K. "Two-phase deadlock detection algorithm in distributed databases," in *Proc. 1979 Int. Conf. Very Large Data Bases* (IEEE), New York.

KING74 KING, P. F., AND COLLMEYER, A J. "Database sharing—an efficient method for supporting concurrent processes," in *Proc. 1974 Nat. Computer Conf.*, vol. 42, AFIPS Press, Arlington, Va., 1974.

KUNG79 KUNG, H. T., AND PAPADIMITRIOU, C. H. "An optimality theory of concurrency control for databases," in *Proc. 1979*

ACM-SIGMOD Int. Conf Management of Data, June 1979.

KUNG81 KUNG, H. T., AND ROBINSON, J. T. "On optimistic methods for concurrency control," *ACM Trans. Database Syst.* **6,** 2, (June 81), 213-226.

LAMP76 LAMPSON, B., AND STURGIS, H. "Crash recovery in a distributed data storage system," Tech. Rep., Computer Science Lab., Xerox Palo Alto Research Center, Palo Alto, Calif., 1976.

LAMP78 LAMPORT, L. "Time, clocks and ordering of events in a distributed system," *Commun. ACM* **21,** 7 (July 1978), 558-565.

LELA78 LELANN, G. "Algorithms for distributed data-sharing sytems which use tickets," in *Proc. 3rd Berkeley Workshop Distributed Databases and Computer Networks*, Aug. 1978.

LIN79 LIN, W. K. "Concurrency control in multiple copy distributed data base system," in *Proc 4th Berkeley Workshop Distributed Data Management and Computer Networks*, Aug. 1979.

MENA79 MENASCE, D. A., AND MUNTZ, R. R. "Locking and deadlock detection in distributed databases," *IEEE Trans. Softw. Eng.* **SE-5,** 3 (May 1979), 195-202.

MENA80 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), 103-138.

MINO78 MINOURA, T. "Maximally concurrent transaction processing," in *Proc. 3rd Berkeley Workshop Distributed Databases and Computer Networks*, Aug. 1978.

MINO79 MINOURA, T. "A new concurrency control algorithm for distributed data base systems," in *Proc. 4th Berkeley Workshop Distributed Data Management and Computer Networks*, Aug. 1979.

MONT78 MONTGOMERY, W. A. "Robust concurrency control for a distributed information system," Ph.D. dissertation, Lab. for Computer Science, M.I.T., Cambridge, Mass , Dec. 1978.

PAPA77 PAPADIMITRIOU, C. H., BERNSTEIN, P A , AND ROTHNIE, J. B. "Some computational problems related to database concurrency control," in *Proc. Conf. Theoretical Computer Science*, Waterloo, Ont., Canada, Aug. 1977.

PAPA79 PAPADIMITRIOU, C. H. "Serializability of concurrent updates," *J. ACM* **26,** 4 (Oct. 1979), 631-653.

RAHI79 RAHIMI, S K., AND FRANTS, W. R. "A posted update approach to concurrency control in distributed database systems," in *Proc. 1st Int. Conf. Distributed Computing Systems* (IEEE), New York, Oct. 1979, pp. 632-641.

RAMI79 RAMIREZ, R. J , AND SANTORO, N. "Distributed control of updates in multiple-copy data bases: A time optimal algorithm," in *Proc. 4th Berkeley Workshop Distributed Data Management and Computer Networks*, Aug. 1979.

REED78 REED, D. P. "Naming and synchronization in a decentralized computer system, Ph.D. dissertation, Dept. of Electrical Engineering, M.I.T., Cambridge, Mass., Sept., 1978.

REIS79a REIS, D. "The effect of concurrency control on database management system performance," Ph.D. dissertation, Computer Science Dept., Univ. California, Berkeley, April 1979.

REIS79b REIS, D. "The effects of concurrency control on the performance of a distributed database management system," in *Proc. 4th Berkeley Workshop Distributed Data Management and Computer Networks*, Aug. 1979.

ROSE79 ROSEN, E. C. "The updating protocol of the ARPANET's new routing algorithm: A case study in maintaining identical copies of a changing distributed data base," in *Proc. 4th Berkeley Workshop Distributed Data Management and Computer Networks*, Aug. 1979.

ROSE78 ROSENKRANTZ, D. J., STEARNS, R E., AND LEWIS, P. M. "System level concurrency control for distributed database systems," *ACM Trans. Database Syst.* **3,** 2 (June 1978), 178-198.

ROTH77 ROTHNIE, J. B., AND GOODMAN, N. "A survey of research and development in distributed databases systems," in *Proc 3rd Int. Conf. Very Large Data Bases* (IEEE), Tokyo, Japan, Oct. 1977.

SCHL78 SCHLAGETER, G. "Process synchronization in database systems." *ACM Trans. Database Syst.* **3,** 3 (Sept. 1978), 248-271.

SEQU79 SEQUIN, J., SARGEANT, G., AND WILNES, P. "A majority consensus algorithm for the consistency of duplicated and distributed information," in *Proc. 1st Int. Conf. Distributed Computing Systems* (IEEE), New York, Oct. 1979, pp. 617-624.

SHAP77a SHAPIRO, R. M., AND MILLSTEIN, R. E. "Reliability and fault recovery in distributed processing," in *Oceans '77 Conf Record*, vol II, Los Angeles, 1977.

SHAP77b SHAPIRO, R. M., AND MILLSTEIN, R. E. "NSW reliability plan," Massachusetts Tech. Rep. 7701-1411, Computer Associates, Wakefield, Mass., June 1977.

SILB80 SILBERSCHATZ, A., AND KEDEM, Z. "Consistency in hierarchical database systems," *J. ACM* **27,** 1 (Jan. 1980), 72-80.

STEA76 STEARNS, R. E., LEWIS, P. M. II, AND ROSENKRANTZ, D. J. "Concurrency controls for database systems," in *Proc. 17th Symp. Foundations Computer Science* (IEEE), 1976, pp. 19-32.

STEA81 STEARNS, R. E., AND ROSENKRANTZ, J. "Distributed database concurrency controls using fore-values," in *Proc 1981 SIGMOD Conf.* (ACM).

STON77 STONEBRAKER, M., AND NEUHOLD, E. "A distributed database version of INGRES," in *Proc. 2nd Berkeley Workshop Distributed Data Management and Computer Networks*, May 1977.

STON79 STONEBRAKER, M. "Concurrency control and consistency of multiple copies of data in distributed INGRES, *IEEE Trans. Softw. Eng.* **SE-5,** 3 (May 1979), 188–194.

THOM79 THOMAS, R. H. "A solution to the concurrency control problem for multiple copy databases," in *Proc. 1978 COMPCON Conf.* (IEEE), New York.

VERH78 VERHOFSTAD, J. S. M. "Recovery and crash resistance in a filing system," in *Proc. SIGMOD Int Conf. Management of Data* (ACM), New York, 1977, pp 158–167.

A Partial Index of References

Received April 1980; final revision accepted February 1981

المنارة للاستشارات