Concurrency Control in a System for Distributed Databases (SDD-1)

PHILIP A. BERNSTEIN, DAVID W. SHIPMAN, and JAMES B. ROTHNIE, JR.
Computer Corporation of America

This paper presents the concurrency control strategy of SDD-1. SDD-1, a System for Distributed Databases, is a prototype distributed database system being developed by Computer Corporation of America. In SDD-1, portions of data distributed throughout a network may be replicated at multiple sites. The SDD-1 concurrency control guarantees database consistency in the face of such distribution and replication.

This paper is one of a series of companion papers on SDD-1 [4, 10, 12, 21].

Key Words and Phrases: distributed database system, concurrency control, serializability, timestamps, synchronization, conflict graph
CR Categories: 4.32, 4.33

1. INTRODUCTION

SDD-1 is a prototype distributed database system being designed and implemented at Computer Corporation of America. The system is designed to support databases that can be physically distributed with arbitrary redundancy over a network of hundreds of sites, while keeping data distribution and data redundancy invisible to the user. A principal problem of implementing systems of this type is maintaining the consistency of the database while concurrent user transactions attempt to update it. The concurrency control mechanism that SDD-1 uses to overcome this problem is the subject of this paper.

2. LITERATURE REVIEW

The concurrency control problem in database systems has been a major research focus for some time. In centralized database management systems (DBMSs), the conventional method to control concurrent update activity is *two-phase locking* [9]. Two-phase locking requires that every transaction

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract N00039-77-C-0074, ARPA Order No. 3175-6. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Authors' present addresses: P. A. Bernstein, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138; D. W. Shipman and J. B. Rothnie, Jr., Computer Corporation of America, 575 Technology Square, Cambridge, MA 02139.

(1) lock the data it reads and writes before it actually accesses them, and
(2) not obtain any new locks after it has released a lock.

Once a data item is locked, no other transaction may lock that data item until the owner of that lock releases it. Research into locking-based concurrency controls has analyzed deadlock problems, logical locks described by predicates (instead of by data item names), granularity of locks, and efficient locking algorithms [7, 9, 11, 13, 18].

Locking methods have also been proposed for distributed DBMSs. One technique, called *primary-site*, uses a central lock controller to manage the locks [1]. Alternatively, locks can be distributed with the data. In the *primary copy* method, a primary copy of each redundantly stored file is designated, and only the primary copy is locked [23]. A centralized deadlock detector resolves distributed deadlock. Locks are also distributed in the method of [19, 22], but distributed deadlock detection is avoided by using timestamps to resolve locking conflicts. A method that avoids locks entirely is the *majority consensus algorithm*, in which sites "vote" on update requests to resolve conflicts [24]; however, the amount of concurrency attained here is the same as locking [6]. Another method, which uses timestamped versions, is described in [17]. A survey of distributed concurrency control methods appears in [3].

These distributed locking approaches are quite similar to centralized concurrency controls. However, these mechanisms do differ from centralized schemes in one respect—the possibility of asynchronous failures of sites and communication links while an update is in the midst of being processed. Many of the proposed distributed concurrency controls have concentrated on this problem of failure (e.g., [1, 15, 23, 24]).

The concurrency control mechanism of SDD-1 differs from all of the above mechanisms in at least one way. In SDD-1, information about how transactions can conflict is preanalyzed before the transactions are submitted. This preanalysis step determines the amount of run-time synchronization required; in many cases, preanalysis shows that very little run-time synchronization is needed. Preanalysis is the heart of the SDD-1 concurrency control and is the main topic of this paper. Also, the run-time synchronization mechanisms of SDD-1, which differ considerably from locking, are discussed. An early restricted version of the SDD-1 concurrency control appears in [5].

This paper is organized in 15 sections. We begin, in Section 3, with a review of those aspects of SDD-1 architecture that impact concurrency control. Section 4 defines correctness for a concurrency control mechanism. Then, in Sections 5 and 6, we discuss two important techniques on which the SDD-1 concurrency control is based: timestamps and transaction classes. Sections 7 through 10 develop the preanalysis technique. An overview of the mathematics used in preanalysis has been isolated in Section 9 and can be skipped without loss of continuity. Sections 11 and 13 describe implementation aspects of the mechanism, and Section 12 describes a special protocol for transactions that would otherwise induce tremendous synchronization overhead. In Section 14 we discuss the reliability aspects of the implementation. We conclude in Section 15 with a summary of the advantages of our method.

The concepts and mechanisms of SDD-1 concurrency control are complex, and

therefore their correctness is not obvious. In a companion paper [4], we produce a formal model of the concurrency control mechanism and prove that it does indeed work correctly.

3. REVIEW OF SDD-1 ARCHITECTURE

The architecture of SDD-1 is described in [20, 21]. We review here those aspects of the architecture that are needed for understanding the concurrency control mechanism.

A user of SDD-1 sees a conventional DBMS. The logical database is expressed in a relational data model which from the viewpoint of the user's transaction is nonredundant and nondistributed. Issues that are consequences of physical data distribution and redundancy are entirely handled by the system and are visible to the user transaction only insofar as they affect performance. Transactions are expressed as a program written in a semiprocedural data manipulation language called Datalanguage [8].

Internally, SDD-1 consists of two types of modules, called transaction modules (TMs) and data modules (DMs). Each site can contain either one or both types of modules. DMs store physical data and behave much like conventional nondistributed DBMSs. TMs are responsible for supervising the execution of user transactions, translating from the user's nondistributed view of the data to the realities of its distribution and redundancy.

For purposes of concurrency control the important messages processed by DMs are READ and WRITE messages. A READ message is a request by a TM to read some of the data items stored at a DM and to store them in a local workspace at that DM on behalf of some transaction. A WRITE message is sent by a TM to a DM to report updates produced by a transaction which the TM supervised. Each DM performs READs and WRITEs as atomic operations. This means, for example, that none of the data read by a READ message can be updated by any WRITE during the time the READ is being processed.

The basic unit of user computation in SDD-1 is the *transaction.* A transaction essentially corresponds to a program in a high-level host language with several data manipulation language statements sprinkled within it. The execution of each transaction is supervised by a TM and proceeds in three phases called *read*, *execute*, and *write.*

In the read phase, SDD-1 analyzes the transaction to determine which portions of the (logical) database it reads, called its *read-set.* Since the transaction is coded in terms of the logical database, and since the physical database in general has redundant copies of many logical data items, the TM must choose which copies of the read-set will be read. It reads this copy of the read-set into a private distributed workspace by sending READ messages to those DMs at which the selected copies are stored. When all READ messages have been processed (i.e., when the TM has received positive acknowledgments from all the DMs), the read phase is complete.

During the read phase a TM sends at most one READ message to each DM on behalf of a single transaction. If, for example, a transaction reads data from two data items that reside at the same DM, then only one READ message is issued to read both data items.

During the execute phase the TM supervises the execution of the transaction.

This function of the TM is performed by the access planner and is described in [10, 25]. Since the concurrency control mechanism in the read phase guarantees that the physical read-set obtained by READ messages is internally consistent, the transaction will produce correct output. The output of this phase is a list of data items to be written into the database or displayed to the user. This output list is produced in a workspace, not the permanent database. When the output list is constructed and the transaction terminates, the execute phase is complete.

In the write phase, the output list produced by the transaction is broadcast to the "relevant" DMs as WRITE messages. A DM is *relevant* if it contains a physical copy of some logical data item that is referenced in (i.e., updated by) the output list. So each update to a logical data item, say x, is sent to *all* DMs that have a stored copy of x. Aspects of resiliency to failure are handled in this phase.

Since each transaction produces (at most) one output list, and since that output list is sent to each relevant DM as a single WRITE message, all of a transaction's updates are performed atomically at each individual DM. Since each TM sends at most one READ message to each DM on behalf of a single transaction, this means that each READ message only reads data produced by complete transactions. It is the job of the concurrency control mechanism to guarantee (among other things) that READ messages which are sent on behalf of a single transaction and which are processed at different DMs all read data produced by *the same set* of complete transactions.

4. CONCURRENT CORRECTNESS

The system usually has many transactions in progress at any one time, both because there are multiple TMs operating concurrently within the system and because individual TMs are processing transactions concurrently. If the READs and WRITEs that implement these transactions were arbitrarily interleaved, then serious problems of database consistency could result. The usual method of avoiding these consistency problems is by guaranteeing that the execution of transactions is serializable [9, 16, 19].

We say that an interleaved execution of a set of transactions is *serializable* if it is "equivalent" to a history of operation in which each of the transactions runs alone to completion before the next one begins. Two executions are *equivalent* if in both executions each transaction produces the same output, thereby leading to the same final state of the database. That is, an interleaved execution is serializable if it can be reproduced by a noninterleaved (i.e., serial) execution of the same set of transactions. Note that serializability requires only that there exist *some* serial order equivalent to the actual interleaved execution. There may in fact be *several* such equivalent serial orderings.

The adoption of serializability as the criterion for concurrent correctness is based on the assumption that each user transaction will preserve database consistency if it runs atomically. That is, if only one transaction is allowed to execute at a time, and if the database state is initially consistent, then after executing a transaction the database state must still be consistent. So, a serial ordering of transaction executions will, by induction, result in a consistent database state. Since a serializable execution is equivalent to some serial one, a serializable execution results in a consistent database state as well.

The issue of serializability arises because a system's atomic actions are at a

finer granularity than its users' atomic actions. In SDD-1, the users' atomic actions are transactions, while the system's atomic actions are the execution of READ and WRITE messages at the DMs. When a system allows the execution of several transactions at the same time, then the system's operations corresponding to different transactions are interleaved. If the interleaving is not controlled, there is no guarantee that the behavior of such a system conforms to the user's expectation that each transaction is processed as an indivisible computation.

For example, assume there is a single copy of data item x, which initially has the value $x = 0$. There are two transactions in the system; transaction i sets $x := x + 1$, and transaction j sets $x := x + 2$. The following sequence of events occurs:

transaction i reads $x = 0$;
transaction j reads $x = 0$;
transaction j sets $x := 2$;
transaction i sets $x := 1$.

Any serial execution of the two transactions, one after the other, would have resulted in setting x to 3. However, the result of this interleaved execution is to set x to 1, contrary to the user's intention. This execution history is not serializable, since no serial processing of these transactions can produce the observed effects.

To guarantee serializability in SDD-1, we apparently need to avoid undesirable interleavings of READ and WRITE messages—those that lead to nonserializable executions. We accomplish this goal using two mechanisms. First, we examine each transaction to determine if it is *conceivable* that it could participate in a nonserializable execution. As we will see, many transactions will never produce READs and WRITEs that interleave badly with other transactions, and hence they can be run unsynchronized. Second, for those transactions that are determined to be dangerous because they can participate in nonserializable executions, we *synchronize* their READ and WRITE messages using protocols that avoid undesirable interleavings. These protocols are based on a timestamping mechanism and are quite different from the locking protocols used in conventional centralized DBMSs.

As we will see, most of the effort in distinguishing transactions that require no synchronization from the dangerous ones is done statically when the database is designed. When a transaction is actually submitted, a simple local table look-up is sufficient to determine how much, if any, synchronization is required. The runtime mechanism is the collection of protocols that must be invoked for those transactions that do require synchronization.

Note that these two components of the concurrency control mechanism are independent. Our technique for analyzing transactions to determine sources of nonserializability could be used in conjunction with conventional locking protocols. Or we could run *all* transactions using our timestamp-based protocols and ignore the preanalysis step entirely, as in present systems that use locking without preanalysis. Together the two mechanisms provide a powerful technique for synchronizing concurrent transactions at low cost.

Before describing the heart of the system—the method for determining the amount of synchronization required by each transaction and the protocols that

effect that synchronization—we must first describe two basic concepts that underlie much of the concurrency control mechanism. These concepts, timestamps and transaction classes, are described in the next two sections.

5. TIMESTAMPS

Each transaction i executed by SDD-1 is assigned a globally unique timestamp, denoted TS_i, by its TM before READ messages are broadcast on its behalf. Transaction timestamps serve a number of purposes for synchronizing READs and WRITEs. To generate globally unique timestamps, a TM reads its local clock and appends its unique TM number as the low-order bits of the timestamps. By requiring that once a clock is read it cannot be read again until it has been incremented, we ensure that every timestamp is globally unique within the system [24].

The clocks are actually maintained as part of the Reliable Network, the reliable communications facility of SDD-1. By using the clock synchronization method described in [14], the system behaves as if there were a single virtual clock available to all sites.

One use of timestamps is in processing WRITE messages that arrive at a DM out of order. The problem is that the WRITE messages sent by two transactions that update the same logical data item may be processed in different orders at different DMs, thereby producing mutually inconsistent copies of the data item. One way to solve this problem is to attach the transaction's timestamp to all of its WRITE messages and then require that WRITE messages be processed in timestamp order at all DMs. A better method that gives more flexibility to DMs in the processing of WRITE messages uses timestamped data items and is adopted in SDD-1 (this method was originally suggested in [24]).

A transaction's timestamp is carried on all of its WRITE messages. In addition, every physical data item at every DM has an associated timestamp. Note that timestamps are attached to *physical* data items; there may be many physical copies of a logical data item, and each one has its own attached timestamp. The timestamp of a data item is the timestamp of the last WRITE message that updated it. Each DM processes WRITE messages according to the following *WRITE message rule*: A data item is updated by a WRITE message if and only if the data item's timestamp is less than the WRITE message's timestamp. (Recall that a WRITE message contains the final values of data items, not computations to be performed on them.) So to process a data item in a WRITE message, the DM compares the timestamp of the WRITE message with the timestamp of its stored copy of the data item. If the timestamp of the WRITE message exceeds the timestamp of the stored data item, then the new value of the data item in the WRITE message is written into the stored data item along with the new timestamp. Otherwise, the update is not performed *on that stored data item.* This is a data item by data item check; some data items in the WRITE message may result in update operations, while others may not.

Whenever a WRITE message for a recent transaction that updates some data item is processed at a DM before a WRITE message for an earlier (i.e., older) transaction that updates the same data item, the latter WRITE message will contain a data item update that is not performed. Such a situation is not an error. It is simply the way that the system reorders updates to occur in the same order

that their generating transactions executed. That is, the net effect of a set of WRITE messages processed at a DM in arbitrary order is the same as the effect of processing them in timestamp order without the WRITE message rule.

The principal advantage of using the WRITE message rule is that WRITE messages can be processed as soon as they are received, thereby avoiding artificial queuing delays at the DMs. However, since later WRITEs may be processed before earlier ones, a database copy may be temporarily inconsistent. As we will see, the concurrency control never permits a transaction to read such an inconsistent state if this could lead to incorrect results.

Note that the WRITE message rule reorders updates into timestamp order even if clocks in different TMs are not synchronized. All other timestamp-related mechanisms in SDD-1 also operate correctly with unsynchronized clocks. For reasons of efficiency, however, it is necessary to assume that clock values in different TMs are reasonably close to each other.

A principal objection to timestamped data items is their cost. However, not all timestamps actually need to be stored. If the timestamp of a data item is earlier than the timestamp of any transaction whose WRITE messages have not yet been processed, then the data item's timestamp is effectively zero. Any WRITE message that tries to update that data item will succeed, because the WRITE message will have a later timestamp than the data item. So we need only maintain the timestamps of recently updated data items. If a data item is not updated for a while (say a few minutes), then its timestamp can be assumed to be zero and therefore dropped. A caching mechanism for timestamps using differential files has been designed for this purpose. Using this mechanism, we judge that the overhead in maintaining timestamps will be small, since only a small portion of the data items will require their timestamps to be stored in the cache at any given time.

6. TRANSACTION CLASSES

A crucial aspect of the SDD-1 concurrency control mechanism is its ability to distinguish between transactions that require synchronization and those that do not. By examining the read-set and write-set of transactions, the system can determine which transactions conflict with each other. Intuitively, two transactions *conflict* if the read-set or write-set of one intersects the write-set of the other. Such conflicts can lead to nonserializability under certain interleavings of READs and WRITEs. Such nonserializable interleavings are avoided in conventional DBMSs by locking data items so that two conflicting transactions never run concurrently. However, synchronizing *all* conflicting READs and WRITEs is more than what is required to guarantee serializability. By analyzing a graph-theoretic representation of the transactions, called a *conflict graph*, the system can isolate the dangerous conflicts that can potentially lead to nonserializability. This analysis technique is described in detail later in the paper.

Unfortunately, analyzing the conflict graph at run-time for all executing transactions is too time consuming. Also, since the transactions are distributed at run-time, assembling a conflict graph would require too much communication. So we transform this run-time analysis into a static analysis done only once at database design time by capitalizing on the predictability of transaction types in the following way.

Relation Schema: INVENTORY (ITEM#, DESCRIPTION, PRICE, QUANTITY)

Class 1
TM: TM1
read-set: INVENTORY [ITEM#, PRICE]
write-set: INVENTORY [PRICE]
comments: transactions that update prices

Class 2
TM: TM2
read-set: INVENTORY [ITEM#, QUANTITY]
WHERE (PRICE > $100)
write-set: INVENTORY [QUANTITY]
comments: transactions that update quantities of high-priced items

Class 3
TM: TM2
read-set: INVENTORY [ITEM#, DESCRIPTION, PRICE]
WHERE (QUANTITY > 0)
write-set: user's terminal
comments: transactions that display item information about items currently in stock

Fig. 1. Class definitions using simple predicates.

When designing the database, the database administrator establishes a static set of *transaction classes.* Formally, each transaction class is defined by a logical read-set and write-set and is assigned to run at a particular TM. A transaction *fits* in a class if the read-set and write-set of the transaction are contained (respectively) in the read-set and write-set of the class. Clearly, a transaction can fit in many classes. Read-set and write-set definitions are expressed using simple restrictions,[1] so that class membership can be checked quickly (see Figure 1).

Note that two classes at different TMs can have identical read-sets and write-sets. However, it is important that they be distinguished as separate classes, and that they be analyzed as separate classes during conflict graph analysis.

The conflict graph analysis is now done on the statically defined transaction classes instead of on the transactions themselves. This analysis yields the type of synchronization, if any, required for each class. At run-time, when a transaction is submitted to a TM, the TM selects a class in which the transaction fits and applies the type of synchronization specified by the analysis for that class.

The utility of classes lies in the property that two transactions that run in different classes conflict only if their classes conflict.[2] Hence, conflicts between transactions can be determined by conflicts between classes. So an analysis of the classes at database design time is sufficient to determine potentially dangerous conflicts between transactions at run time. We believe that for many kinds of applications, the most frequent determination will be that the class participates in no dangerous conflicts and can therefore run with only local synchronization.

[1] A simple restriction is a Boolean expression whose clauses are of the form ⟨attribute⟩⟨rel-op⟩⟨constant⟩, where ⟨rel-op⟩ is =, ≠, <, >, etc.

[2] Of course, transactions within a class may also conflict. Section 7 shows how to synchronize transactions within a class.

```
Do Forever;
  Wait for a transaction, T, to arrive;
  Find a class, C, in which T fits;
  If C cannot be processed locally
    then forward T to a site that can process C
    else begin
      look up the synchronization rules for class C,
      send out appropriate READ messages on
            behalf of T, synchronizing where necessary;
      supervise the distributed execution of T;
      send out WRITE messages on behalf of T
      end
end
```

Fig. 2. How a TM processes a transaction.

For a set of class definitions to be feasible, it must cover *all* transactions that might ever be submitted. It is *not* necessary that *every* TM have enough classes to accept all possible transactions, since a TM can forward a transaction to some other TM for execution. However, it *is* necessary that every possible transaction fit in a class supported by *some* TM. A sketch of how a transaction is routed and executed by TMs appears in Figure 2.

7. SYNCHRONIZING TRANSACTIONS WITHIN A CLASS

To ensure the serializability of transactions which execute in the same class, we require that conflicting transactions within a class be executed in timestamp order. To formalize this requirement, some notation is helpful. Let the processing of a READ message on behalf of transaction i at DM_α be denoted R_α^i.[3] Similarly, let the processing of a WRITE message on behalf of transaction i at DM_α be denoted W_α^i. Then we can express the requirement that transactions within a class run serially as follows:

Class Pipelining Rule. For each DM_α, for each class **i**, and for each pair of transactions i_1 and i_2 in **i**, if i_1 reads some data item x at DM_α and i_2 writes into x at DM_α, then $R_\alpha^{i_1}$ is processed before $W_\alpha^{i_2}$ if and only if $TS_{i_1} < TS_{i_2}$ (i.e., i_1 has an earlier timestamp than i_2).

The class pipelining rule is sufficient to guarantee that the transactions within a single class are serializable. This follows from the fact that a serial execution of the transactions in a class in timestamp order is equivalent to an execution that obeys the class pipelining rule.

The class pipelining rule, although stated in terms of DMs, is actually enforced by mechanisms at both TMs and DMs. For each class that a TM processes, the messages from that class are sent to each DM in an order that is consistent with the pipelining rule. The communications network (ARPANET, in our case) guarantees that messages are received in the order that they were sent, for any point-to-point communications channel. The DMs process messages within a class in the order in which they are received, thereby enforcing the pipelining rule.

---

[3] We use lowercase Greek letters to denote DMs. We use lowercase Roman letters i, j, k, ... to denote transactions. We denote the class in which transaction i executes by **i**.

8. INTERCLASS INTERFERENCE

8.1 An Example of Safe Interference

We say that a set of transactions *interferes* if the system allows the transactions to be interleaved in a nonserializable manner. Given the class pipelining rule, we need not be concerned with interference among transactions in the same class. The problem now is to avoid interference among transactions in different classes. A critical aspect of our solution to this problem is isolating those cases where transactions in different classes never interfere with each other. This requires some subtlety, for even when transactions read and write the same data items, they may not interfere, as illustrated by the following simple example.

Suppose we run two transactions, say i and j, in two different classes, **i** and **j**. Transaction i first finds the EMPLOYEE record whose NAME domain has the value "JAMES BOND," and then writes a new value into the PHONE# domain of that record. Transaction j finds the EMPLOYEE record whose SOC-SEC# domain is 007 (which is JAMES BOND's SOC-SEC#) and writes a new value into the PHONE# domain of that record, different from the PHONE# written by i. Naturally, the final value of JAMES BOND's PHONE#, after both transactions execute, is dependent on the order in which their write operations were processed. However, no matter how their read and write operations are interleaved, the execution will be serializable. The transactions will always appear to have executed serially with the order of their writes determining the order of the transactions in the serialization; the first transaction that writes JAMES BOND's PHONE# appears first in the serialization. Therefore, even though the transactions have overlapping write-sets—a situation that conventionally requires locking—no synchronization is necessary.

To exploit situations where no synchronization is required, we must determine if unsafe patterns of interleaved reads and writes are possible. This determination is accomplished by analyzing conflicts between transaction classes. For example, an analysis of classes **i** and **j** above would show that all patterns of interleaved reads and writes are serializable. This analysis is performed on a graph-theoretic representation of transaction conflicts and is the subject of the next section.

8.2 Conflict Graphs

As we observed in Section 6, two transactions from different classes conflict only if their classes conflict. To formalize this, we say that WRITE message W^i_α *conflicts with* a READ message R^j_α iff transaction i's write-set intersects transaction j's read-set. A WRITE message W^i_α *conflicts with* another WRITE message W^j_α iff transaction i's write-set intersects transaction j's write-set. It follows that if R^i_α conflicts with W^j_α, then the read-set of class **i** intersects the write-set of class **j**. By examining class conflicts, we can predict potential transaction conflicts, which are a primary component of the serializability problem. It will turn out that this examination of class conflict will lead us to our goal—a method for determining the amount of synchronization required by each transaction.

The method begins with the construction of a *conflict graph* (see Figure 3). In the graph, each class, say **i**, is modeled by two nodes labeled r^i and w^i. For each class **i** an edge $\langle r^i, w^i \rangle$, called a *vertical edge*, is drawn [Figure 3(a)]. When the

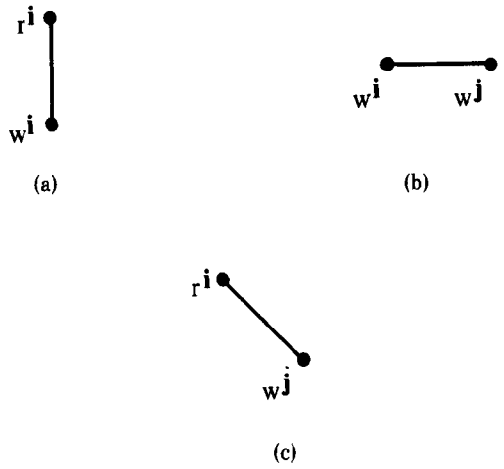


Fig. 3. Conflict graph edges. (a) A vertical edge is drawn between every $\langle r^i, w^i \rangle$ pair. (b) A horizontal edge is drawn between a $\langle w^i, w^j \rangle$ pair iff the write-sets of **i** and **j** intersect. (c) A diagonal edge is drawn between an $\langle r^i, w^j \rangle$ pair iff the read-set of **i** intersects the write-set of **j**.

write-sets of two classes, say **i** and **j**, intersect, then an edge $\langle w^i, w^j \rangle$, called a *horizontal edge*, is drawn [Figure 3(b)]. Similarly, if the read-set of one class (say **i**), intersects the write-set of another class (say **j**), then an edge $\langle r^i, w^j \rangle$, called a *diagonal edge*, is drawn [Figure 3(c)].

For a given set of classes, C, we denote the conflict graph for C by CG_C. A sample conflict graph appears in Figure 4.

We will use the conflict graph to help us predict the amount of synchronization required by each transaction class. The connection between synchronization protocols and conflict graphs is developed in Section 9. Since this development is lengthy and may not be of interest to all readers, we summarize the principal results of Section 9 in Section 10. Hence, if desired, Section 9 can be skipped without loss of continuity.

9. CONFLICT GRAPH ANALYSIS

9.1 Serializing Logs

Depending on the order in which READ and WRITE messages are processed by the system, an interleaved execution of transactions may or may not be serializable. To understand which message orderings *are* serializable, we need a notation that models these orderings. In our notation we represent the ordered processing of READ and WRITE messages at a DM by a log. A *log* is simply a string of R's and W's that have the same DM subscript. For example, $R_\alpha^1 W_\alpha^2 W_\alpha^1 R_\alpha^5 W_\alpha^5 R_\alpha^4$ is a log describing the order in which READ and WRITE messages were processed at DM_α. When we say, for example, that R_α^i *precedes* W_α^j (in DM_α's log), we mean that R_α^i *was processed before* W_α^j at DM_α.

A log is a complete representation of the computations performed on the

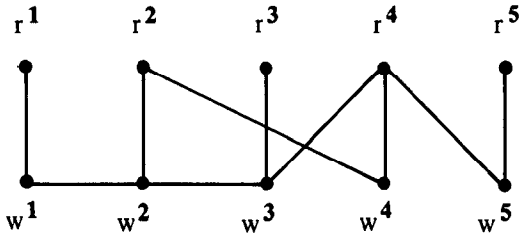


Fig. 4. A sample conflict graph.

database at a DM. If we were to be given the list of data items read by each READ message and written by each WRITE message, as well as the timestamps of transactions (so that we could correctly apply the WRITE message rule), then we would be able to reproduce the computation that was actually performed at the DM. So an "interleaved execution of transactions" in SDD-1 is modeled by a "collection of DM logs, one per DM." We therefore use these two terms interchangeably.

Suppose we are given an interleaved execution of N transactions, represented by a set of DM logs. Which of the $N!$ possible serializations of the transactions is an equivalent serialization of the given logs? A serialization is *equivalent* to the given logs if that serial execution of the transactions on a nondistributed, nonredundant database (represented by the serialization) produces the same computation as the interleaved execution on the distributed, redundant database (represented by the DM logs). It is a theorem that *if each transaction reads from a database that has had exactly the same write operations applied to it in the serialization as were applied to it in the given interleaved execution, then each transaction will perform the same computation in the serialization as it did in the given interleaved execution* [16]. We can guarantee this condition by requiring that the serialization satisfy the following three rules. For each i, j, and DM_α:

(1) If W^i_α precedes and conflicts with R^j_α, then i must precede j in the serialization.
(2) If R^j_α precedes and conflicts with W^i_α, then j must precede i in the serialization.
(3) If W^i_α conflicts with W^j_α, then i and j must appear in the serialization in their timestamp order.

If the serialization obeys rules (1) and (2), then write operations in the serialization precede exactly the same read operations as they did in the given interleaved execution. However, this is not the same as saying that each transaction reads from a database that has had exactly the same write operations applied to it in the serialization as were *applied* to it in the given execution. The reason is that owing to the WRITE message rule, the order in which WRITE messages are *processed* is not the same as the effective order in which the write operations are *applied* to the database; indeed, some write operations are not applied at all. To understand this distinction is to understand the need for rule (3).

In the logs, the WRITE message rule prevents certain write operations from being applied; this occurs when a WRITE message with an early timestamp arrives after a WRITE message with a later timestamp and both WRITE

messages write into a common data item. The WRITE message rule is an artifact of the distributed execution of SDD-1 and would not have been applied if the transaction were executed serially on a nondistributed, nonredundant database. In essence, this means that the serialization must produce the same computation *without* the WRITE message rule that the given logs produced *with* the WRITE message rule. Rules (1) and (2) alone are not strong enough to make this guarantee.

For example, suppose the log for DM_α contains the subsequence $W_\alpha^i W_\alpha^j R_\alpha^k$ where j has an *earlier* timestamp than i and the three messages either write or read only data item x. The WRITE message rule prevents W_α^j from overwriting x, so R_α^k reads x from W_α^i. We want the same relative ordering of R_α^k and W_α^i to appear in the serialization. So transaction j must either precede transaction i or follow transaction k in the serialization. However, the serialization [i, j, k] would be permitted by the rules (1) and (2) alone; this is incorrect because transaction k would read x from j (not i) in this serialization.

Rule (3) guarantees that write operations in the serialization are applied in the same relative order as they are applied in the given logs. It "factors out" the WRITE message rule from the serialization by requiring the write operations to appear in the order in which they were *effectively* applied, rather than the order in which they were processed.

By developing rules (1)-(3), we have related the order of conflicting READ and WRITE messages in DM logs to the order of transactions in serializations. As we know, not all interleaved executions are serializable. So, as we would expect, there are DM logs that have no serialization obeying rules (1)-(3). In principle, we could schedule READ and WRITE messages by continually checking rules (1)-(3) at run time so that the order in which READ and WRITE messages are processed can always be serialized. However this would be very costly in computation time and communication traffic. Instead, we use the conflict graph model of transaction conflicts to guide us in synchronizing READ and WRITE messages so that a serialization obeying rules (1)-(3) is always possible.

The conflict graph is used to determine potentially nonserializable executions of conflicting transactions. Diagonal and horizontal edges can be used to determine if READ and WRITE messages may conflict, leading to the following extension of rules (1)-(3). For each i (in **i**), j (in **j**), and DM_α:

(1′) If $\langle w^i, r^j \rangle$ is a diagonal edge of CG and W_α^i precedes R_α^j in DM_α's log, then i must precede j in any serialization.

(2′) If $\langle r^i, w^j \rangle$ is a diagonal edge of CG and R_α^i precedes W_α^j in DM_α's log, then i must precede j in any serialization.

(3′) If $\langle w^i, w^j \rangle$ is a horizontal edge of CG, then i and j must appear in the serialization in their timestamp order.

Since two transactions conflict only if their classes conflict, any serialization that satisfies rules (1′)-(3′) will satisfy rules (1)-(3) as well. The advantage to using rules (1′)-(3′) in place of rules (1)-(3) is that the former are stated entirely in terms of class conflicts, which are known in advance.

In SDD-1 there is always a serialization of the executed transactions that satisfies rules (1′)-(3′). The mechanisms that are used to guarantee that such a serialization always exists are called *protocols*.

9.2 Protocol P1 and the Acyclicity Theorem

To understand why we need protocols, let us consider a system consisting of two classes, say **i** and **j**, such that only one transaction is processed in each class, say transactions i and j. Under what conditions will these two transactions be serializable? If there are no horizontal or diagonal edges connecting **i** and **j** in the conflict graph, then rules (1′)-(3′) are trivially satisfied. In this case i and j are serializable; in fact, either serialization will do. What if **i** and **j** are connected by some edge?

If $\langle w^i, w^j \rangle$ appears in CG, and if W^i_α and W^j_α are processed (for some DM_α), then according to rule (3′) i and j must be serialized in timestamp order. If this is the only edge connecting **i** and **j**, then the transactions are surely serializable. For example, suppose $TS_i < TS_j$; then no matter how many DMs process WRITE messages from both transactions, each DM will apply the WRITE message rule, thereby making it look as if i was processed before j. Therefore, applying rule (3′) at all DMs will yield the same requirement that i and j be serialized in the same timestamp order. The only way we could get into trouble is if one DM believes i should precede j in the serialization while another believes j should precede i—a clear impossibility using rule (3′). So if $\langle w^i, w^j \rangle$ is the only edge connecting **i** and **j**, we are safe.

If $\langle r^i, w^j \rangle$ appears in CG, then we have a potential problem. Suppose W^j_α precedes and conflicts with R^i_α and R^i_β precedes and conflicts with W^j_β. Rule (1′) applied at DM_α says that j should precede i, while rule (2′) applied at DM_β says that i should precede j. Since both cannot be simultaneously satisfied, we have a nonserializable interleaving. Apparently, we must introduce some synchronization mechanism to avoid this problem produced by the diagonal edge.

Protocol P1 is the mechanism used to synchronize diagonal edge conflicts. The effect of running transaction i under protocol P1 against transaction j is that the relative ordering of READ messages from i and WRITE messages from j are the same at all DMs where both appear and conflict. If for every diagonal edge $\langle r^i, w^j \rangle$, transaction i in **i** obeyed P1 against j in **j**, and if each class ran only a single transaction (and then became permanently inactive), then the nonserializable situation due to the opposite serializations consistent with rules (1′) and (2′) could not occur.

However, this effect of P1 is insufficient to synchronize a diagonal edge when *multiple* transactions run in each class. To illustrate the potential problem, suppose i and i′ run in class **i**, j and j′ run in class **j**, and $\langle r^i, w^j \rangle$ is in the conflict graph. Consider an execution history in which

(a) W^j_α precedes R^i_α at DM_α;
(b) $R^{i'}_\beta$ precedes $W^{j'}_\beta$ at DM_β;
(c) $TS_i < TS_{i'}$, and W^i_γ and $W^{i'}_\gamma$ execute at DM_γ;
(d) $TS_{j'} < TS_j$, and W^j_δ and $W^{j'}_\delta$ execute at DM_δ.

These four properties of the execution tell us that in any serialization j must precede i (by property (a) and rule (1′)), i must precede i′ (by property (c) and rule (3′)), i′ must precede j′ (by property (b) and rule (2′)), and j′ must precede j (by property (d) and rule (3′)). However, this implies that j both precedes and follows j′ in the serialization, which is an impossibility.

To avoid nonserializable executions due to a diagonal edge when many transactions run in each class, we use the following definition of P1. *Transactions in* **i** *obey protocol P1 with respect to transactions in* **j** if for any i, i′ in **i**, and j, j′ in **j**, whenever W^j_α precedes and conflicts with R^i_α at DM_α, and $R^{i'}_\beta$ precedes and conflicts with $W^{j'}_\beta$ at DM_β, and either $TS_i < TS_{i'}$, or $i = i'$, then $TS_j < TS_{j'}$. We require that if $\langle r^i, w^j \rangle$ is an edge in CG, then transactions in **i** must obey P1 with respect to transactions in **j**. This is sufficient to prevent rules (1′) and (2′) from ever leading to opposite serializations of transactions in **i** and **j**, even when multiple transactions execute in each class [4].

The above observations regarding single edge conflicts between two classes generalize directly to paths of conflicts. Suppose there is a single edge conflict between **i** and **k**, and another one between **k** and **j**. Assume again that one transaction runs in each class, say i, j, and k. Rules (1′)-(3′) only restrict the order of serialization between pairs of conflicting transactions. They will either require that i and j have a defined relative ordering (i.e., either i precedes k and k precedes j, or i follows k and k follows j) or that they have no special required order (i.e., either i precedes k and j precedes k, or i follows k and j follows k). In either case the three transactions are serializable.

The only way the transactions might not be serializable is if there were *two different paths* from **i** to **j**. Then one path could lead to i preceding j according to rules (1′)-(3′), while the other path could lead to i following j. If this occurred, then the execution would be nonserializable. But note that it can only occur if there are *two* distinct paths. Two distinct paths that link **i** to **j** constitute a cycle. So as long as there are no cycles in the conflict graph and each class runs one transaction, P1 is sufficient to guarantee serializability.

The class pipelining rule guarantees that transactions within a single class are serializable. So the above statement about acyclic conflict graphs generalizes to the case of multiple transactions per class. (A proof of this fact is nontrivial and appears in [4].)

Our observations in this section can now be stated more formally as follows:

Acyclicity Theorem. For a given set of transaction classes C, if

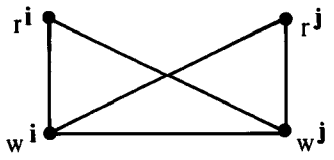
(1) CG_C has no cycles,
(2) all classes in C obey the class pipelining rule, and
(3) for each diagonal edge $\langle r^i, w^j \rangle$ in CG_C, transactions in **i** obey P1 with respect to transactions in **j**,

then all possible interleavings of transactions in classes in C are serializable.

To make the acyclicity theorem effective, we need to demonstrate an implementation for P1. This we will do in Section 11. First, however, we will show how to synchronize nonserializable situations caused by cycles.

9.3 Cycles, P3, and the Serializability Theorem

We have shown that if no cycles exist in the conflict graph and if P1 is properly applied, then all possible interleaved executions of transactions are serializable. We have also observed that cycles in the conflict graph can cause a nonserializable



(a)

log for DM_α: $R^i_\alpha R^j_\alpha W^i_\alpha W^j_\alpha$

(b)

Fig. 5. A nonserializable execution caused by a cycle. Classes **i** and **j** have data item x in their read-sets and write-sets. (a) The conflict graph. (b) A nonserializable log of transactions from classes **i** and **j**.

execution. If two distinct paths exist between two classes **i** and **j**, then the paths may lead to opposite serializations of transactions i in **i** and j in **j** according to rules (1′)-(3′)—a nonserializable situation. To eliminate this possibility, we introduce a protocol that forces any two paths between **i** and **j** to always lead to the same relative ordering of i and j in all serializations. To illustrate the problem and the protocol that solves it, let us consider another example.

This time suppose that the database has one data item x stored at DM_α. Classes **i** and **j** both read from and write into x; for example, they both run transactions that increment x. The conflict graph of these classes contains two distinct edges, $\langle r^i, w^j \rangle$ and $\langle w^i, r^j \rangle$, connecting **i** and **j**. These two edges together with $\langle r^i, w^i \rangle$ and $\langle r^j, w^j \rangle$ constitute a cycle (see Figure 5). The problem is that the diagonal edges may force opposite serializations of transactions in **i** and **j**.

Consider, for instance, transactions i in **i** and j in **j** which execute their READ and WRITE messages in the following order: $R^i_\alpha R^j_\alpha W^i_\alpha W^j_\alpha$ (cf. example in Section 4). Notice that P1 is trivially obeyed. Since R^i_α precedes and conflicts with W^j_α, rule (2′) implies that i must be serialized before j. Since R^j_α precedes and conflicts with W^i_α, the same rule implies that j must be serialized before i. Since both cannot be simultaneously satisfied, the execution is nonserializable. This occurs because the edges between **i** and **j** lead to opposite serializations.

Protocol P3 prevents executions such as this one by making the following guarantee: *If two transactions belong to two classes connected by a diagonal edge in a cycle, then the timestamp order of the two transactions is the same as the relative ordering dictated by rules* (1′) *or* (2′) *applied to the messages that correspond to the edge.* Before examining how P3 accomplishes this task, let us first see how P3 corrects the above example.

Since $\langle r^i, w^j \rangle$, $\langle w^j, r^j \rangle$, $\langle r^j, w^i \rangle$, $\langle w^i, r^i \rangle$ comprise a cycle, P3 applies to transactions i and j. Suppose the timestamp of i is smaller than the timestamp of j. We have observed that rule (2′) requires that i be serialized before j because R^i_α precedes W^j_α, and that j be serialized before i because R^j_α precedes W^i_α. But the latter requirement violates P3. Since $\langle r^j, w^i \rangle$ is in a cycle, protocol P3 implies

that rule (2′) applied to R_α^j and W_α^i must lead to the serialization of i and j in timestamp order. However, the opposite occurs. What P3 must do, therefore, is make sure that W_α^i precedes R_α^j. Then both edges will lead to the serialization of i and j in timestamp order, and the nonserializability problem goes away.

Formally we define protocol P3 as follows. Transactions in **i** obey protocol P3 with respect to transactions in **j** if for each i in **i**, j in **j**, and each DM_α at which R_α^i and W_α^j both appear and conflict, R_α^i and W_α^j are processed in timestamp order. We require that for each diagonal edge $\langle r^i, w^j \rangle$ in a cycle, transactions in **i** must obey P3 with respect to transactions in **j**.

Protocol P3 synchronizes multiclass cycles as well as the simple two-class cycle just illustrated. In a cycle consisting of several diagonal and horizontal edges, P3 requires that each conflict due to a diagonal edge lead to the pair of transactions being serialized in timestamp order. Rule (3′) makes the very same requirement for horizontal edges. So insofar as this cycle is concerned, if rules (1′)-(3′) say anything about the relative ordering of two transactions whose classes are on the cycle, then the requirement must be that the transactions be serialized in timestamp order. Since there is only one timestamp ordering of transactions, conflicting serialization orderings are impossible. Generalizing this observation for the case of multiple transactions per class as we did for the acyclicity theorem leads to the correctness theorem for the SDD-1 concurrency control [4].

Serializability Theorem. For a given set of transaction classes C, if

(1) all classes in C obey the class pipelining rule, and
(2) for each diagonal edge $\langle r^i, w^j \rangle$ in CG_C, transactions in **i** obey P1 with respect to transactions in **j**, and
(3) for each diagonal edge $\langle r^i, w^j \rangle$ in a cycle in CG_C, transactions in **i** obey P3 with respect to transactions in **j**,

then all possible interleavings of transactions in classes in C are serializable.

9.4 P2: A Faster Protocol for Read-Only Transactions

While P3 is sufficient for synchronizing all diagonal edges in a cycle, we can do somewhat better with those transactions that intersect the cycle only with their r-nodes. These read-only transactions contribute to nonserializability only because they may observe certain WRITE messages being processed in reverse timestamp order.[4] Protocol P2 is a weaker version of P3 that prevents this situation and thereby provides a less expensive alternative for synchronizing such transactions.

Let us begin by taking a slightly different view of P3. Suppose transactions i and j execute in classes **i** and **j** (respectively) and that **i** and **j** lie on a cycle. P3 attains serializability by guaranteeing that i and j can be serialized in timestamp order. It accomplishes this by requiring that for each diagonal edge in the cycle, the READ and WRITE messages corresponding to the edge's endpoints are processed in timestamp order. In essence, each diagonal edge in the cycle is individually synchronized. However, we can relax the synchronization requirements somewhat, synchronizing certain pairs of edges as a unit.

[4] Strictly speaking, these transactions need not be read-only. It is just that their write operations, if they have any, do not participate in a conflict graph cycle.

To illustrate, suppose the edges $\langle w^j, r^i \rangle$ and $\langle r^i, w^k \rangle$ lie on a cycle. Since **j** and **k** are also connected by some other path, we must ensure that executions of reads and writes corresponding to this two-edge path are consistent with serializing transactions in **j** and **k** in timestamp order. This two-edge path will prevent a timestamp ordered serialization *only if* transaction i observes WRITE messages from j and k in reverse timestamp order. For example, suppose $TS_j < TS_k$. If R_α^i precedes and conflicts with W_α^j and R_β^i follows and conflicts with W_β^k, then from i's viewpoint and according to rules (1′) and (2′), k must be serialized before i, which must be serialized before j. If either R_α^i had followed W_α^j or R_β^i had preceded W_β^k, j and k could have been serialized in timestamp order. Protocol P2 is designed to make precisely this guarantee, without requiring that the edge be synchronized as strongly as by P3.

Transactions in **i** *obey protocol P2 with respect to transactions in* **j** *and* **k** if for any i in **i**, j in **j**, k in **k**, and for any α

(1) if R_α^i precedes and conflicts with W_α^j and $TS_k > TS_j$, then R_β^i precedes W_β^k at every DM_β where they both appear and conflict, and
(2) if R_α^i follows and conflicts with W_α^j and $TS_j > TS_k$, then R_β^i follows W_β^k at every DM_β where they both appear and conflict.

That is, if $TS_j < TS_k$, then transaction i observes a WRITE message from transaction k only if it has observed all WRITE messages from transaction j, and conversely if $TS_k < TS_j$. Protocol P2 prevents i from observing a WRITE message from the later transaction unless it has observed all WRITE messages from the earlier one.

Protocol P2 is strictly weaker than P3 in that if **i** obeys P3 with respect to **j** and **k**, then it obeys P2 with respect to **j** and **k** (but not conversely). Yet we can use it correctly for synchronizing classes which only intersect cycles with their r-nodes. Stated precisely, if $[\langle w^j, r^i \rangle, \langle r^i, w^k \rangle]$ is a subpath of a cycle, and if we require that transactions in **i** obey P2 with respect to transactions in **j** and **k**, then we need not synchronize these two diagonal edges using P3. That is, if we use P2 to synchronize edge combinations $[\langle w^j, r^i \rangle, \langle r^i, w^k \rangle]$ in cycles and use P3 to synchronize all other diagonal edges in cycles, then the serializability theorem in the last section still holds.

10. A SUMMARY OF THE PROTOCOL SELECTION RULES

In Section 9 we have described the three basic protocols for synchronizing transactions and the conflict graph topologies that require the use of the protocols. While the analysis that leads to the protocols is somewhat complex, the rules for selecting the protocols are not. It is these *protocol selection rules* that completely govern the concurrency control mechanism of SDD-1. We present these rules here in order to summarize and encapsulate the results of Section 9 and to incorporate a few more details to make the statement of the rules precise.

First, let us restate each of the three protocols.

Protocol P1. Transactions in **i** obey protocol P1 with respect to transactions in **j** if for any i, i′ in **i** and j, j′ in **j**, whenever W_α^j precedes R_α^i at DM_α, and $R_\beta^{i'}$ precedes $W_\beta^{j'}$ at DM_β, and either $TS_i < TS_{i'}$, or $i = i'$, then $TS_j < TS_{j'}$.

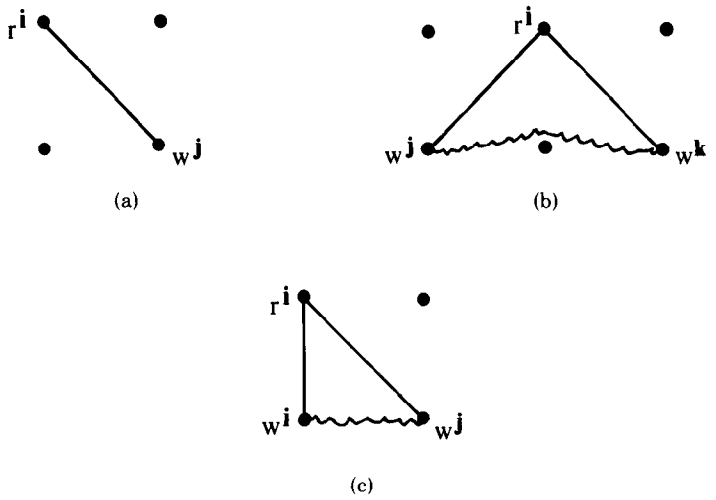


Fig. 6. Protocol selection rules. (a) Transactions in **i** must obey P1 with respect to transactions in **j**. (b) Transactions in **i** must obey P2 with respect to transactions in **j** and **k**. (c) Transactions in **i** must obey P3 with respect to transactions in **j**.

Protocol P2. Transactions in **i** obey protocol P2 with respect to transactions in **j** and **k** if for any i in **i**, j in **j**, k in **k**, and any DMα

(1) if R_α^i is processed before and conflicts with W_α^j and $TS_k > TS_j$, then R_β^i is processed before W_β^k at every DM_β where they both appear and conflict, and
(2) if R_α^i is processed after and conflicts with W_α^j and $TS_k < TS_j$, then R_β^i is processed after W_β^k at every DM_β where they both appear and conflict.

Protocol P3. Transactions in **i** obey protocol P3 with respect to transactions in **j** if for each i in **i**, j in **j**, and each DM_α at which R_α^i and W_α^j both appear and conflict, R_α^i and W_α^j are processed in timestamp order.

Briefly, these protocols serve the following purposes:

P1 Prevents READ messages from one transaction that conflict with WRITE messages from another transaction from being processed in different relative orders at different DMs.
P2 Prevents a READ message from seeing WRITE messages from two other transactions in reverse timestamp order.
P3 Prevents two transactions that read each other's output from both reading before either writes, i.e., prevents a classical race condition.

The protocol selection rules state which protocols should be invoked by which transactions. They are as follows.

I. For all classes in **i** and **j** such that $\langle r^i, w^j \rangle$ is in the conflict graph, transactions in **i** must obey protocol P1 with respect to transactions in **j** [see Figure 6(a)].
II. For each cycle in the conflict graph the following hold:
   (a) for all distinct classes **i**, **j**, **k**, if edges $\langle r^i, w^j \rangle$ and $\langle r^i, w^k \rangle$ lie on the cycle, then transactions in **i** must obey P2 with respect to transactions in **j** and **k** [see Figure 6(b)]; and

(b) for all distinct classes **i** and **j** such that $\langle r^i, w^i \rangle$ and $\langle r^i, w^j \rangle$ lie on the cycle, then transactions in **i** must obey P3 with respect to transactions in **j** [see Figure 6(c)].

The protocol selection rules are easily transformed into an algorithm that analyzes the conflict graph and produces the protocols that each class must obey. However, the definitions of the protocols are not algorithmic. To make the protocols effective, we now show how TMs and DMs can enforce the relative orderings of READ and WRITE messages required by the protocols.

11. IMPLEMENTING THE PROTOCOLS

11.1 Implementing Protocol P1

Each protocol demands that certain relative orderings of READ and WRITE messages be obeyed. These orderings are enforced by synchronization information that is carried entirely by the READ messages from **i** in the form of *read conditions.*

A read condition is attached to a READ message and specifies which WRITE messages from certain other classes must be processed before the READ message can be correctly processed. The read condition includes a timestamp, say TS, and one or more classes, say $\{\mathbf{j}_1, \ldots, \mathbf{j}_m\}$. The DM can only process the READ message when the attached read condition is satisfied. Read condition $\langle$TS, $\{\mathbf{j}_1, \ldots, \mathbf{j}_m\}\rangle$ is *satisfied* when *all* WRITE messages from classes $\{\mathbf{j}_1, \ldots, \mathbf{j}_m\}$ with timestamps earlier than TS have been processed and *no* WRITE messages from classes $\{\mathbf{j}_1, \ldots, \mathbf{j}_m\}$ with timestamps later than TS have been processed. Then the READ message can be processed. If a READ message contains multiple read conditions, then all of them must be simultaneously satisfied when the DM processes the READ.

Before discussing the implementation of read conditions at DMs, let us first show how read conditions implement protocol P1. Suppose transactions in **i** must obey P1 with respect to transactions in **j**. To process a transaction i in **i**, we select a timestamp TS_i' (not necessarily equal to the transaction's timestamp) and require that the read condition $\langle TS_i', \{\mathbf{j}\}\rangle$ be attached to each READ message sent on behalf of i to each DM at which conflicting WRITE messages from **j** are processed. In addition, we require that for any i_1, i_2 in **i**, if i_1 has an earlier timestamp than i_2, TS_{i_1}' for i_1 must be earlier than TS_{i_2}' for i_2.

To see why this implementation is correct, recall the definition of P1: Transactions in **i** obey P1 with respect to transactions in **j** if for any i_1, i_2 in **i**, and j_1, j_2 in **j**, whenever (1) $W_\alpha^{j_1}$ precedes and conflicts with $R_\alpha^{i_1}$ at DM_α, and (2) $R_\beta^{i_2}$ precedes and conflicts with $W_\beta^{j_2}$ at DM_β, and (3) either $TS_{i_1} < TS_{i_2}$ or $i_1 = i_2$, then $TS_{j_1} < TS_{j_2}$. So, suppose we have i_1, i_2, j_1, and j_2 satisfying (1), (2), and (3). Let TS_{i_1}' and TS_{i_2}' be the timestamps used on read conditions for i_1 and i_2. By (1) and read condition $\langle TS_{i_1}', \{\mathbf{j}\}\rangle$ attached to $R_\alpha^{i_1}$, we have $TS_{j_1} < TS_{i_1}'$. By (3) and the implementation of P1, we have $TS_{i_1}' \leq TS_{i_2}'$. By (2) and read condition $\langle TS_{i_2}', \{\mathbf{j}\}\rangle$, we have $TS_{i_2}' < TS_{j_2}$. So by transitivity, $TS_{j_1} < TS_{j_2}$ as desired.

To effectively implement read conditions, we need a mechanism that allows a DM to determine when it has received all WRITE messages with timestamps earlier than some TS from a specified set of classes and none with later timestamps from these classes. The mechanism we use is called *WRITE pipelining.*

WRITE pipelining requires that WRITE messages from each class must be processed in timestamp order at all DMs. That is, for each class **i**, for each DM_α, and for any pair of transactions i_1 and i_2 in **i**, $W_\alpha^{i_1}$ is processed before $W_\alpha^{i_2}$ at DM_α only if $TS_{i_1} < TS_{i_2}$. WRITE pipelining can be implemented in the same way as the class pipelining rule (cf. Section 7); each class sends its WRITE messages to all DMs in timestamp order, the network ensures messages are received in the order sent, and DMs process messages from each class in the order in which they were received.

Given that WRITE pipelining is used, a DM can determine when a read condition $\langle TS, \{j\} \rangle$ is satisfied. Since WRITE messages from any given class are processed in timestamp order at every DM, as soon as the DM receives a WRITE message timestamped later than TS, it knows it must hold it and process the READ message first. Of course, if a WRITE message from **j** with timestamp later than TS has been processed before the read condition is received, then the read condition cannot be satisfied without backing out the WRITE message. In SDD-1, no WRITE message is backed out for concurrency control reasons. So in this case the READ message would have to be *rejected*, and the originating class must resubmit it with a later timestamp. Notice that *all* READ messages on behalf of transaction i have to be resubmitted with a new transaction timestamp, since their read conditions are now obsolete. WRITE messages from the resubmitted transaction carry the new timestamp.

There is some danger that a transaction may be continually rejected. This can be avoided by choosing a timestamp for the resubmitted transaction such that the difference between the transaction timestamp and that of the previously submitted transaction grows with each resubmission. Eventually, the READ messages for the transaction will be received at all DMs before any earlier timestamped WRITE messages from conflicting classes. Since a DM never processes a WRITE message which would force the rejection of an already received READ message, the transaction will not be rejected in this case. In practice, however, choosing transaction timestamps equal to the local clock times will keep the number of rejections very small, as long as clocks at different sites are reasonably well synchronized.

An important optimization is used when transactions in **i** only conflict with transactions in **j** at one DM, say DM_α. In this case we avoid read conditions entirely by using *READ pipelining*: READ messages from **i** to DM_α are processed in order of transaction timestamps. That is, if i_1 has an earlier timestamp than i_2, then $R_\alpha^{i_1}$ is processed before $R_\alpha^{i_2}$. The implementation of READ pipelining is exactly analogous to WRITE pipelining. If transactions in **i** use READ pipelining at DM_α and transactions in **j** use WRITE pipelining at DM_α, then P1 of **i** with respect to **j** is obeyed. To see this, suppose $W_\alpha^{j_1}$ precedes and conflicts with $R_\alpha^{i_1}$, $R_\alpha^{i_2}$ precedes and conflicts with $W_\alpha^{j_2}$, and $TS_{i_1} < TS_{i_2}$. By READ pipelining, $R_\alpha^{i_1}$ is processed before $R_\alpha^{i_2}$ and, hence, $W_\alpha^{j_1}$ is processed before $W_\alpha^{j_2}$. Now, by WRITE pipelining, we have $TS_{j_1} < TS_{j_2}$, thereby satisfying P1. We expect that most diagonal edges will correspond to single DM conflicts, so this optimization will usually apply.

When read conditions *are* used, a problem arises when class **j** is idle because it has no transactions to process. In this case the DM will wait for a long time until a WRITE message timestamped later than TS arrives. One way to solve this

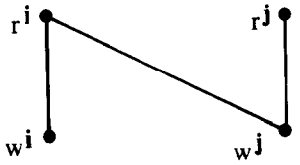


Fig. 7. A conflict graph illustrating P1.

problem is to have idle classes periodically send NULLWRITE messages.[5] A NULLWRITE message specifies the originating class and a timestamp and is interpreted as an empty WRITE message from that class with that timestamp. When a DM receives such a NULLWRITE message, it can be sure that it has received all WRITE messages from the indicated class through the given timestamp. If a DM chooses not to wait passively for a WRITE or NULLWRITE message from **j**, it can request a NULLWRITE by sending a SENDNULL message to **j**.

The choice of timestamps for read conditions and the rate at which NULLWRITEs are sent are important tuning parameters to avoid the frequent use of SENDNULLs. In addition, the choice of timestamp for read conditions will affect how long a READ message has to wait for conflicting WRITE messages to be processed. Essentially, the timestamp should be as small as possible without actually forcing the read condition to be rejected.

To illustrate the operation of protocol P1, let us consider a database that consists of two data items, x and y, where x is stored at DM_α and y is stored at DM_β. Class **j** writes both x and y, and class **i** reads both x and y. For definiteness, suppose class **i** runs at TM_i and **j** runs at TM_j. The conflict graph for this situation is shown in Figure 7. The edge $\langle r^i, w^j \rangle$ implies transactions in **i** must obey P1 with respect to transactions in **j**.

For TM_i to process a transaction, say i, it must send READ messages R_α^i to DM_α and R_β^i to DM_β. By P1, both messages must have a read condition $\langle TS_i', \{j\} \rangle$ attached. DM_α will not process R_α^i to read x until it has received (but not processed) a WRITE message or a NULLWRITE for TM_j on behalf of **j** with timestamp later than TS. DM_β will behave the same way. So R_α^i will wait for (i.e., will be processed after) WRITE messages from the same set of transactions in **j** as R_β^i will wait for. Hence, for each j in **j**, rules (1′) and (2′) will require the same serialization order for i and j at both DM_α and DM_β, and the result will be serializable. The nonserializable situation of R_α^i preceding W_α^j but R_β^i following W_β^j cannot occur.

11.2 Implementing Protocol P3

The same read condition mechanism that we described for implementing P1 is sufficient for implementing P3 as well. For transaction i to obey P3 with respect to transactions **j** at DM_α, R_α^i must be processed after all W_α^j with earlier timestamps and before all W_α^j with later timestamps. Attaching the read condition $\langle TS_i, \{j\} \rangle$ to R_α^i will force DM_α to process R_α^i according to P3; DM_α will wait for

[5] The use of periodic NULLWRITE messages can be avoided by use of special protocols that are tailored for low-frequency classes. However, their description is beyond the scope of this paper.

exactly those W_a^j with $TS_j < TS_i$. Rejected READ messages are handled exactly as per P1.

From this implementation we see immediately that protocol P3 is strictly stronger than protocol P1. If transactions in **i** obey P3 with respect to transactions in **j** at DM_a, then they obey P1 with respect to transactions in **j** at DM_a. The difference between P1 and P3 is that P1 allows any timestamp to appear in the read condition while P3 requires that timestamp to be TS_i. Also note that the class pipelining rule is essentially an implementation of P3; class pipelining has the effect of **i**'s obeying P3 with respect to **i**.

Our earlier remarks about NULLWRITEs and SENDNULLs apply here as well. We noted under P1 that choosing a timestamp for the read condition was important to avoid lengthy delays. Since the read condition timestamp is the transaction's timestamp in P3, we must be careful to run the P3 transaction as early as possible—early enough so that READ messages need not wait for many WRITE messages, but not so early as to require its being rejected.

11.3 Implementing Protocol P2

As with the other protocols, P2 is implemented using read conditions. If transaction i must obey P2 with respect to transactions in j and **k**, then it must attach a read condition ⟨TS, {**j**, **k**}⟩ to each of its READ messages that are sent to a DM that processes conflicting WRITE messages from **j** or **k**. As in P1, any timestamp for the read condition will do. Since some DMs will only process conflicting WRITE messages for either **j** or **k** (but not both), these DMs will only use one of the two classes in the second read condition parameter.

If i conflicts with WRITE messages from **j** and **k** at *only one* DM, an interesting optimization is possible. Rather than specifying the timestamp TS in the read condition, the DM can select the timestamp itself. As long as there is *some* time TS such that all earlier WRITE messages and no later WRITE messages from **j** and **k** have been processed, P2 will be obeyed. However, if two or more DMs are involved, the timestamp must be fixed in advance because all DMs must use the *same* timestamp; they cannot choose timestamps independently.

12. P4: A CYCLE-BREAKING PROTOCOL

Although P1, P2, and P3 are sufficient to guarantee serializability, from an efficiency standpoint these protocols have a very serious problem. The problem is that a single class can cause many cycles and thereby force many classes to use P2 and P3, *even though very few transactions are ever run in that class.*

While we expect that the vast majority of transactions that we wish to execute are predictable and belong to predefined classes, we still want to be able to execute an unexpected transaction that does not fit into any of our class definitions. One way to accomplish this is to define a "very large" class, call it $\mathbf{i}_{total}$, that has a read-set and write-set that include the entire logical database. Every conceivable transaction can fit into $\mathbf{i}_{total}$, so this apparently solves the problem. But the cost is enormous, for $\mathbf{i}_{total}$ induces a two-class cycle with *every other class* in the system. So every class has to run P3 against $\mathbf{i}_{total}$, and $\mathbf{i}_{total}$ has to run P3 against every other class. Since P3 is the most expensive protocol (measured by the delay of the transaction obeying it), this is an unfortunate state of affairs. It is especially unfortunate because transactions will rarely need to execute in $\mathbf{i}_{total}$,

since most transactions fit into other less expensive classes. So $\mathbf{i}_{total}$ introduces considerable synchronization overhead for synchronizing against a class that will rarely run a transaction.

In general, any class in which transactions are only infrequently run, but which creates many cycles in the conflict graph, exhibits this phenomenon. Although the problem of proliferation of cycles is especially acute in $\mathbf{i}_{total}$, other classes with smaller read-sets and write-sets may manifest the same problem.

To alleviate these problems, we introduce a new protocol called P4, the purpose of which is to "break" cycles in the conflict graph. That is, if a class runs P4, then other classes that are in a cycle with the P4 class can behave as if the cycle did not exist.

One way to implement P4 is to shut off the system when a P4 transaction is introduced. No new transactions are processed, and the system works until all outstanding WRITE messages from transactions already in progress have been processed. When the system has finally quiesced, we can safely run the P4 transaction serially. After all of the P4 transaction's WRITE messages are processed, we can safely permit the system to process transactions again. Since the execution before and after the P4 transaction ran was serializable (by the serializability theorem) and since the P4 transaction ran serially, the entire execution is serializable.

We can state the desired effect of P4 more formally as follows: *Transactions in* $\mathbf{i}$ *obey P4 with respect to transactions in* $\{\mathbf{j}_1, \ldots, \mathbf{j}_p\}$ if for each transaction i in $\mathbf{i}$, j_1 in $\mathbf{j}_u$, and j_2 in $\mathbf{j}_v$ $(1 \leq u, v \leq p)$,

(a) if j_1 must precede j_2 in any serialization satisfying rules (1′)-(3′) and $TS_i \leq TS_{j_1}$,[6] then $TS_i < TS_{j_2}$; and

(b) if j_1 must precede j_2 in any serialization satisfying rules (1′)-(3′) and $TS_i \geq TS_{j_2}$, then $TS_i > TS_{j_1}$.

To "break a cycle" in which $\mathbf{i}$ lies, transactions in $\mathbf{i}$ must obey P4 with respect to all classes on the cycle (including i). In this case the remaining transactions in the cycle need not obey P2 and P3 as would normally be required. Note that one need not break all cycles on which $\mathbf{i}$ lies; one can use P4 to break some of i's cycles and use {P1, P2, P3} as usual to synchronize the others.

The key property of protocol P4 is summarized as follows. If transaction i obeys P4 with respect to transactions in $\{\mathbf{j}_1, \ldots, \mathbf{j}_p\}$, then there is a serialization such that for each transaction j processed in one of $\mathbf{j}_1, \ldots, \mathbf{j}_p$, j precedes i in the serialization if and only if j's timestamp is smaller than i's timestamp. Thus, processing i under P4 has the same effect as shutting off the system when i is submitted, executing i, and then returning to normal operation. To see why P4 guarantees the above property, suppose $TS_j < TS_i$ but j must follow i in all serializations according to rules (1′)-(3′). Substituting i for j_1 and j for j_2 in part (a) of the definition of P4, we see that $TS_j > TS_i$, which is a contradiction. The inverse situation is handled by part (b) of P4.

Implementing P4 by shutting off the system—even temporarily—is likely to be unacceptable because of a severe performance degradation. We can improve this implementation considerably by exploiting two observations. First, a P4 trans-

[6] It is possible that $TS_i = T_{j_1}$. This occurs when $\mathbf{i} = \mathbf{j}_u$ and $i = j_1$.

action need only synchronize against classes that lie on a cycle that includes the P4 class, since only classes on cycles can cause nonserializability. Second, even these classes need not quiesce completely before running a P4 transaction. Only *conflicting* WRITE messages must be completed before the P4 transaction executes and subsequently allows the other classes to resume processing. WRITE messages that do not conflict with READs in the same cycle cannot affect the ordering of transactions in the serialization according to rules (1')-(3'), and therefore they do not require synchronization under the definition of P4.

The implementation of P4 differs structurally from the other protocols in two ways. First, P4 requires some direct communication between TMs. By this communication the TM supervising the P4 class requests that certain other TMs perform synchronization to avoid interfering with the P4 transaction. Second, P4 requires an augmented form of read condition. Recall that a standard read condition is a pair of the form ⟨timestamp, {classes}⟩. For P4, the timestamp may be interpreted as a "minimum time," i.e., ⟨mintime = timestamp, {classes}⟩. This condition is satisfied if all WRITE messages from {classes} timestamped less than "timestamp" have been processed. It does *not* require that no messages from {classes} timestamped greater than "timestamp" be processed (as in standard read conditions). The utility of mintime read conditions is explained shortly.

To implement P4, we use three additional types of messages that are sent from TMs to TMs (*not* from TMs to DMs). A P4-ALERT message is sent from a TM supervising a P4 class to a TM supervising some other class. A P4-ALERT message includes the name of the P4 class and the timestamp of the P4 transaction as its parameters. A class responds to a P4-ALERT with either a P4-ACCEPT or a P4-REJECT.

To run a transaction i in the P4 class **i** with respect to some cycle CYC, one performs the following P4 algorithm:

(1) Choose a timestamp for i, say TS_i.
(2) For every class that lies on CYC, send a message P4-ALERT(**i**, TS_i) to the TM supervising that class.
(3) Wait for the P4-ACCEPTs to be received from all classes to which a P4-ALERT was sent. If a P4-REJECT is received, then restart the protocol from step (1).
(4) Construct the READ messages for i. For each DM_α and class **j** such that $\langle r^i, w^j \rangle$ lies on CYC and **j** sends WRITE messages to DM_α that can conflict with R^i_α, attach the read condition $\langle TS_i, \{j\} \rangle$ to R^i_α.

When a TM receives a P4-ALERT(**i**, TS_i) for a particular class, **j**, it performs the following P4-ALERT algorithm:

(1) If the TM has run or begun running a transaction in **j** with a timestamp greater than TS_i, then respond to the TM supervising **i** by sending P4-REJECT. Otherwise, send P4-ACCEPT and do not run another transaction in **j** timestamped earlier than TS_i.
(2) For each DM_α to which **j** sends a READ message and for each class **k** which sends WRITE messages to DM_α and for which $\langle r^j, w^k \rangle$ lies on CYC, the first transaction in **j** with timestamp greater than TS_i which issues a READ

message to DM_α must attach the read condition (mintime = TS_i, {**k**}) to R^j_α. These conditions are in addition to those normally carried by R^j_α. (*Note*: Only do this step for the *first* transaction in **j** with timestamp later than TS_i which sends a READ message to DM_α.)

The combination of P4-ALERT and the read conditions in step (4) of the P4 algorithm are enough to guarantee that P4 is obeyed. Step (4) of the P4 algorithm guarantees that WRITE messages from conflicting transactions in CYC with timestamps earlier than i are processed before i's READ messages. Step (2) of the P4-ALERT algorithm guarantees that WRITE messages from transactions conflicting with classes on CYC other than **i** and with timestamps earlier than TS_i are processed before READ messages from conflicting transactions with timestamps later than TS_i. This ensures that the transactions sending the WRITE messages can be serialized before i. P4-REJECT messages are needed in case the first transaction in **j** with timestamp later than TS_i is already in progress, for then it is too late to attach the read condition required by step (2) of the P4-ALERT algorithm. The mintime read condition is sufficient because only conflicting transactions with earlier timestamps need to be processed before the first transaction in **j**; later ones can be safely processed. Together, the rules guarantee that transactions with earlier timestamps can be serialized before i and those with later timestamps can be serialized after i.

13. THE CONCURRENCY MONITOR

The implementation of the run-time concurrency control mechanism primarily lies in a software module at the DMs called the *concurrency monitor*. The concurrency monitor at a DM accepts READ, WRITE, and NULLWRITE messages from TMs and schedules their execution at the DM. In essence, it is responsible for determining the ordering of events for local DM logs. In this section we describe the operation of the concurrency monitor. As we will see, the mechanism is quite simple.

The concurrency monitor accepts and schedules messages of three types:

WRITE(TS, CLASS, UPDATES)

TS is the timestamp of the transaction issuing the WRITE, and CLASS is its transaction class. UPDATES is a list of data item identifiers and values. When a WRITE is processed, the indicated data items are updated to the specified values according to the WRITE message rule (see Section 5).

NULLWRITE(TS, CLASS)

This message indicates that all future messages in CLASS will have timestamp greater than TS. Processing the NULLWRITE simply involves taking note of this fact in the internal tables of the concurrency monitor.

READ(TS, CLASS, READSET, CONDITIONS)

TS and CLASS are the timestamp and transaction class of the transaction issuing the READ message. CONDITIONS is a list of read conditions associated with the READ message. Processing a READ involves reading the current values for data specified by READSET into a local transaction workspace.

The read conditions have the following format:

⟨TYPE, CLASSES, TS⟩

CLASSES is a list of transaction classes. TS is either a timestamp or is blank, depending on TYPE. If TYPE is "normal," then the read condition is satisfied when all WRITE messages from the listed

classes with timestamps less than TS have been processed, but no WRITE messages from those classes with greater timestamps have been processed. "Normal" read conditions are used in all four protocols. If TYPE is "DMchoice," then the TS specification is blank; the read condition is satisfied when the condition for "normal" read conditions can be satisfied for some selected value for TS. "DMchoice" read conditions are used in the single-DM optimized version of protocol P2. If TYPE is "mintime," then the read condition is satisfied when all WRITE messages from the listed classes with timestamps less than TS have been processed. "Mintime" read conditions are used in the P4 protocol. The TS specification in a read condition must always be less than the transaction TS specified in the READ message itself (to prevent a deadlock within the concurrency monitor).

The DM returns an ACCEPT-READ message when all the read conditions on a READ message have been satisfied and the READ has been processed. If the read conditions cannot be satisfied, even by waiting for new WRITE messages to be processed, then a REJECT-READ message is returned to the originator of the READ.

The function of the concurrency monitor is to schedule the processing of READ and WRITE messages under the constraints imposed by read conditions. A READ message can be processed as soon as its read conditions are satisfied. While WRITE messages should be processed without unnecessary delay, a WRITE message will be delayed if its immediate processing causes the rejection of a pending READ message. When a READ message is received, it is checked to see if it is immediately rejectable. If it is not, then the READ will eventually be satisfied because the concurrency monitor will not process any WRITE messages that will cause it to be rejected.

The concurrency table, shown in Table I, contains the information needed by the concurrency monitor to resolve the status of read conditions. For each class it holds a timestamp associated with the most recently processed WRITE or NULLWRITE message and a pointer to a queue of pending messages from that class to be processed. Messages are ordered on the queue by their arrival order. To avoid violating any of the pipelining rules, the concurrency monitor schedules the messages on each queue in the order in which they appear (with one exception, noted below). The message at the head of the queue is said to be *immediately pending.*

The concurrency monitor chooses the next message to be processed from among those immediately pending. Process any pending NULLWRITE. If there are none, process any immediately pending WRITE, as long as this does not cause any pending READ to be rejected. If there are no such WRITEs, process any immediately pending READs whose read conditions are satisfied.

It is important that the concurrency monitor not postpone indefinitely the processing of any immediately pending message either because of timing anomalies or deadlock. One way to guarantee this would be to schedule immediately pending messages according to the following priority rule. The priority of an immediately pending NULLWRITE or WRITE message is the TS parameter in the message; for a READ message it is the lowest timestamp in an unsatisfied read condition in the READ. The concurrency monitor schedules lowest priority first.

Unfortunately, this scheduling mechanism still has a potential deadlock. As an example, suppose the immediately pending message for i is a READ with a P3 read condition $\langle TS_i, \{j\} \rangle$ and the immediately pending message for j is a READ

Table I. Concurrency Table

Class	Timestamp of most recently processed WRITE	Timestamp of most recently processed NULLWRITE	Pointer to pending message queue
i	425179	425221	→
.	.	.	.
.	.	.	.
.	.	.	.

with read condition $\langle TS_j, \{i\} \rangle$ where $TS_i < TS_j$. Furthermore, suppose no WRITE or NULLWRITE message from **i** timestamped later than TS_j has been received and that none from **j** timestamped later than TS_i has been received—that is, neither read condition is satisfied. If **j** has no more WRITE messages to send with timestamp earlier than TS_j, then the concurrency monitor is deadlocked, waiting for nonexistent WRITE messages from each of the classes.

To avoid this type of deadlock, we need one more scheduling rule: If the immediately pending message with smallest priority is a READ whose lowest priority unsatisfied read condition is $\langle TS, \{j\} \rangle$, and if **j**'s immediately pending message is a READ, then a SENDNULL message must be sent to **j**'s TM, requesting a NULLWRITE with timestamp greater than TS. If j's TM does not respond with a NULLWRITE, then a WRITE with the appropriate timestamp must be on the way (eventually). When it arrives, it must be processed ahead of **j**'s pending READ to break the deadlock. Note that this cannot violate class pipelining because the WRITE message's timestamp is earlier than that of **j**'s pending READ.

Given this deadlock prevention rule, we can now show the lowest-priority-first scheduler to be deadlock free. Let M be the message with lowest priority. If M is a NULLWRITE, it can be processed immediately. If M is a WRITE, then it will be held up only if there is an immediately pending READ with a read condition that has a timestamp smaller than M's. But then the READ would have a lower priority than M, contradicting the choice of M. So the WRITE can be immediately processed. Suppose that M is a READ. If all its read conditions are satisfied, it can be immediately processed and we are done. So assume not and that $\langle TS_R, \{j, \ldots\} \rangle$ is its unsatisfied read condition with smallest timestamp. Let M′ be the immediately pending message on **j**'s queue. If the queue is empty, then a WRITE or NULLWRITE message with timestamp greater than TS_R will eventually appear on **j**'s queue, since there are only a finite number of timestamps smaller than TS_R. If M′ is a WRITE, then M′ must have a timestamp greater than TS_R (by choice of M); and since **j** obeys WRITE pipelining, the READ condition is already satisfied, which is a contradiction. Similarly, M′ cannot be a NULLWRITE. If M′ is a READ, then the deadlock prevention rule is invoked, and M will eventually be processed. Finally, since there are only a finite number of timestamps less than any priority, this also argues for proper termination, since every message will eventually be the one with the lowest priority.

It may not be wise to follow this priority rule strictly, since a lowest priority READ may wait a considerable period of time for all the necessary WRITEs to arrive. This would unnecessarily create a large backlog of other unprocessed

messages. However, the above argument demonstrates feasibility; of course, any more efficient variation which never indefinitely postpones is also acceptable.

14. RELIABILITY CONSIDERATIONS[7]

14.1 Overview

The reliability mechanisms of SDD-1 provide two kinds of protection. First, the system must continue to operate correctly in the face of site and communication failures. That is, the serializability guarantee must be maintained. Second, the procedures by which this is done must not force protocols to wait for failed sites to recover before they can safely proceed. Otherwise, transactions at nonfailed sites could experience arbitrarily long delays before being allowed to continue.

Details of the reliability mechanism are described in a separate companion paper [12]. That paper centers around the description of an extended communications facility called the reliability network or *RelNet.* In that paper we show how the concurrency control mechanisms described here are made robust by use of the RelNet's capabilities.

For our present purposes, the RelNet can be modeled as a virtual machine with the following properties:

(1) The RelNet never fails.[8]
(2) Between any sender-receiver pair, messages are received in the order that they are sent.
(3) Messages may be buffered within the RelNet. This implies that delivery is guaranteed as soon as the message is accepted by the RelNet; the message need not arrive at its final destination for delivery to be assured. In particular, messages may be sent to a failed site and will be delivered upon its recovery.
(4) WRITE messages are not processed by a DM until a COMMIT message is received. Alternatively, uncommitted WRITEs may be canceled by an ABORT message. For each transaction a single COMMIT command is issued to the RelNet, which then guarantees that all of the DMs participating in the transaction will receive COMMIT messages from the RelNet. When a TM crashes, the RelNet sends an ABORT message to those DMs participating in its uncommitted transactions.
(5) The RelNet maintains the clock used in assigning timestamps for concurrency control. The system behaves as if there were a single global clock accessible to all sites.
(6) The RelNet monitors site status. In addition to reporting status as up/down, the RelNet indicates a global clock time for which that status is valid.
(7) The RelNet may be queried as to whether all messages sent from a given site prior to a given time have been received. This may be used, for example, to ensure that all messages sent from a failed site prior to its failure have been received.

---

[7] The mechanisms reported in this section were developed by M. M. Hammer and D. W. Shipman.
[8] Of course, since 100 percent reliability is impossible to achieve, the actual RelNet may in fact fail. We consider this to be a "catastrophe" for which manual procedures may be required to repair any damage done.

A more detailed specification of the RelNet interface, as well as a description of its internal design, is given in [12]. Since the most difficult design issues have been relegated to the RelNet, what is required here is to describe the ways these facilities are used in providing reliable and timely protocol implementations.

We are principally concerned with failures affecting the read phase[9] of a transaction. During the execute phase, the status of participating DMs is monitored by the TM, which will abort the transaction on a DM failure. Failures of the controlling TM during the execute phase result in transaction abortion by the RelNet. During the write phase, if the controlling TM fails before all WRITE messages have been sent and the transaction committed, then the RelNet aborts the transaction. If the controlling TM fails after the transaction is committed, then all WRITEs are guaranteed safe delivery to their destinations and committed by the RelNet.

We need to consider three issues arising in the read phase:

(1) The possibility that some data item in the read-set is not available.
(2) The possibility that the concurrency monitor, in order to validate a read condition, must wait for additional WRITE or NULLWRITE messages from a failed site. Since the site may take arbitrarily long to recover, the concurrency monitor must be able to proceed in resolving the read condition without waiting for additional messages from that site.
(3) The possibility that an ACCEPT/REJECT response to a P4-ALERT message is required from a failed TM. Here again it is unacceptable to wait for the failed site to recover in order for it to make the ACCEPT/REJECT decision.

The next three subsections deal with these issues.

14.2 Data Item Not Available

If all physical copies of a data item are unavailable because the DMs at which they are stored have failed,[10] then the transaction cannot proceed. It is aborted and the user is informed.

It may happen that the originally chosen physical copy of the data item is unavailable, but that another copy of a data item is available at a different DM. In this case the other copy is used for reading instead. It should be noted that the choice of physical copy to be read does not affect the protocols which must be run. This is because the protocol requirements are expressed solely in terms of *logical* data item conflicts.

14.3 Read Conditions

When the timestamp on a read condition against a class is greater than the timestamp on any WRITE or NULLWRITE message which has been received from that class, it is necessary to wait for the arrival of a WRITE or NULLWRITE

---

[9] Recall that the three phases of a transaction's execution are called read, execute, and write (cf. Section 3).

[10] If all physical copies are unavailable because of communication failures, then a RelNet catastrophe has occurred. Approaches for dealing with such catastrophes are discussed in [12].

message from that class which has a greater timestamp than that of the read condition. When the class in question runs at a TM which has failed, it might appear that the concurrency monitor would have to wait for that TM to recover in order to receive the needed message.

The problem is easily solved assuming the existence of a global clock facility. Upon encountering a read condition which requires waiting for messages from a failed site, the concurrency monitor, after processing all WRITEs which the site had sent prior to its failure, simply accepts the read condition. This is sound for the following reason. Upon recovery, all new transactions at the TM in question will have a timestamp greater than that of the read condition. This follows from the fact that the read condition timestamp is less than or equal to the timestamp of the transaction which issued it, that all transaction timestamps are obtained from the global clock, and that the global clock will have necessarily advanced past the timestamp of the reading transaction by the time the failed site recovers. Therefore, it is not possible for a WRITE message to arrive after the failed site's recovery, such that the WRITE message has a timestamp less than that specified in the read condition; it is thus safe to accept the read condition immediately.

14.4 Protocol P4

Protocol P4 calls for issuing a set of P4-ALERT messages to a number of TMs and awaiting ACCEPT/REJECT responses. If a TM is down, of course, it cannot respond and the P4 transaction might appear to have to wait until the failed TM recovers.

Again, our solution to this problem is based on the global clock facility. An ACCEPT response is assumed from any TM which was down at the time of the P4 transaction. Upon recovery and before starting any new transactions, the recovering TM reads all P4-ALERT messages which were sent to it while it was down (these have been buffered in the RelNet). These P4-ALERTs are accepted. This is because no transactions will have been processed at the recovering TM with timestamp greater than that of the P4 transaction (the TM was down at the time of the P4), and all new transactions after the receipt of the P4-ALERT have a timestamp greater than that of the P4 transaction. These are exactly the conditions necessary for accepting a P4-ALERT.

15. ADVANTAGES OF THE SDD-1 CONCURRENCY CONTROL MECHANISM

The SDD-1 approach to concurrency control is in many ways quite different from other proposed mechanisms. We see many strengths in the approach. Unfortunately, there are few analytic methods for verifying these strengths, say by comparing the relative performance of our mechanism to other database concurrency controls. Furthermore, most of the proposed mechanisms are not yet implemented, so empirical comparisons are not possible either. Hence, the analysis of our mechanism must necessarily be more intuitive than analytical or empirical. And comparisons to other specific mechanisms are avoided for lack of objective evidence. The specific criteria on which we base our performance include the amount of communication required to synchronize transactions, the average delay incurred by a transaction due to concurrency control, the amount of concurrency among transactions allowed by the concurrency control, and the

overhead involved in making the mechanism resilient to communications and node failures.

At the architectural level the SDD-1 concurrency control mechanism has two important properties. First, the architecture makes a strong separation between concurrency control issues and those of query processing and reliability. From a project management standpoint, this separation has allowed us to attack the concurrency control problem independently from and in parallel with query processing and reliability problems. From a software engineering standpoint, this division of labor has led naturally to a division of function in software components. The concurrency control mechanisms are isolated in a small number of modules, making them easily modifiable and tunable.

Second, the architecture fully distributes the concurrency control. While each transaction is controlled from a single site, different sites are concurrently supervising the synchronization of many different transactions. No one site is in charge of *any* system-wide activity. The main advantage of this full distribution is enhanced reliability. A site failure only affects those transactions executing and/or using data at that site.

However, it is in the specific synchronization mechanisms that the most important advantages lie: conflict graph analysis and the timestamp-based protocols. We believe the technique of conflict graph analysis to be our most important contribution. By preanalysis of transaction conflicts, the number of transactions that need to be synchronized is drastically reduced. This has a beneficial effect on all aspects of concurrency control performance. It allows more concurrency among transactions; and for those transactions that require little or no synchronization, it cuts delay, communications overhead, and costs associated with resiliency mechanisms. As shown in [6], the technique is quite general and can be used with a variety of synchronization protocols, including conventional locking. In principle, every proposed concurrency control mechanism could be improved by adding conflict graph analysis as a preprocessing step to eliminate run-time synchronization for some transactions.

The timestamp-based protocols {P1, P2, P3, P4} also offer important advantages over other proposed concurrency controls. First, all of the protocols are deadlock-free. This avoids the communication overhead of distributed deadlock detection, which is required by many locking systems (e.g., [23]). Second, the protocols synchronize transactions only against *named* transaction classes. Even if two transaction classes must be synchronized relative to certain data, other classes can concurrently access those data; in fact, other classes can independently be synchronized against those very same data without affecting the first two classes at all. This is in contrast to locking protocols, which set blanket locks that apply to all transactions that access the shared data. Third, SDD-1 offers a *range* of synchronization protocols. Protocol P2 is a fast synchronization protocol for read-only transactions that can afford to read an old, but consistent copy of the database. While with a locking strategy read-only transactions could choose not to lock the data they read, the unlocked data may be inconsistent. Protocol P4 allows infrequently executed transactions to take a larger share of the synchronization burden. By running such transactions under P4, other frequently executed transactions can run P1 with less delay and more concurrency than they

would obtain if they ran P2 or P3 as otherwise required. The P4 capability is currently unique to the SDD-1 mechanism.

Quantitative comparisons among reliability mechanisms are not yet within the state of the art. However, as indicated in the previous section, SDD-1 has incorporated recovery mechanisms that insulate it from the effects of network and node failure. The mechanisms are an example of a general approach to resiliency discussed in [12].

REFERENCES

1. ALSBERG, P.A., AND DAY, J.D. A principle for resilient sharing of distributed resources. Proc. 2nd Int. Conf. on Software Engineering, IEEE, N.Y., 1976, pp. 562–570.
2. ALSBERG, P.A., ET AL. Synchronization and deadlock. CAC Doc. 185, CCTC-WAD Doc. 6503, Center for Advanced Computation, U. of Illinois, Urbana, Ill., 1975.
3. BERNSTEIN, P.A., AND GOODMAN, N. Approaches to concurrency control in distributed database systems. Proc. AFIPS 1979 NCC, Vol. 48, AFIPS Press, Arlington, Va., pp. 813–820.
4. BERNSTEIN, P.A., AND SHIPMAN, D.W. The correctness of concurrency control mechanisms in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (March 1980), 52–68.
5. BERNSTEIN, P.A., ROTHNIE, J.B., GOODMAN, N., AND PAPADIMITRIOU, C.H. The concurrency control mechanism of SDD-1: A system for distributed databases (the fully redundant case). *IEEE Trans. Software Eng. SE-4*, 3 (May 1978), 154–168.
6. BERNSTEIN, P.A., SHIPMAN, D.W., AND WONG, W.S. Formal aspects of serializability in database concurrency control mechanisms. *IEEE Trans. Software Eng. SE-5*, 3 (May 1979), 203–215.
7. CHAMBERLIN, D.D., BOYCE, R.F., AND TRAIGER, I.L. A deadlock-free scheme for resource locking in a database environment. Information Processing 74, North-Holland Pub. Co., Amsterdam, 1974, pp. 340–343.
8. COMPUTER CORPORATION OF AMERICA. *Datacomputer Version 5 User Manual*, Cambridge, Mass., July 1978.
9. ESWARAN, K.P., GRAY, J.N., LORIE, R.A., AND TRAIGER, I.L. The notions of consistency and predicate locks in a database system. *Comm. ACM 19*, 11 (Nov. 1976), 624–633.
10. GOODMAN, N., BERNSTEIN, P.A., REEVE, C., ROTHNIE, J.B., AND WONG, E. Query processing in SDD-1: A system for distributed databases. Submitted for publication.
11. GRAY, J.N., LORIE, R.A., AND PUTZOLU, G.R. Granularity of locks and degrees of consistency in a shared database. Proc. Int. Conf. on Very Large Databases, ACM, N.Y., Sept. 1975, pp. 428–451.
12. HAMMER, M.M., AND SHIPMAN, D.W. The reliability mechanisms in SDD-1: A system for distributed databases. Submitted for publication.
13. KING, P.F., AND COLLMEYER, A.J. Database sharing—an efficient method for supporting concurrent processes. Proc. AFIPS 1973 NCC, Vol. 42, AFIPS Press, Arlington, Va., pp. 271–275.
14. LAMPORT, L. Time, clocks and ordering of events in a distributed system. *Comm. ACM 21*, 7 (July 1978), 558–565.
15. MENASCE, D.A., POPEK, G.J., AND MUNTZ, R.R. A locking protocol for resource coordination in distributed databases. To appear in *ACM Trans. Database Syst. 5*, 2 (June 1980).
16. PAPADIMITRIOU, C.H., BERNSTEIN, P.A., AND ROTHNIE, J.B. Some computational problems related to database concurrency control. Proc. Conf. on Theoretical Computer Science, U. of Waterloo, Waterloo, Ont., Canada, Aug. 1977, pp. 275–282.
17. REED, D.P. Naming and synchronization in a decentralized computer system. Ph.D. Th., Rep. MIT/LCS/TR-205, Massachusetts Institute of Technology, Cambridge, Mass., Sept. 1978.
18. RIES, D.R., AND STONEBRAKER, M. Effects of locking granularity in a database management system. *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 233–246.
19. ROSENKRANTZ, D.J., STEARNS, R.D., AND LEWIS, P.M. System level concurrency control for distributed database systems. *ACM Trans. Database Syst. 3*, 2 (June 1978), 178–198.
20. ROTHNIE, J.B., AND GOODMAN, N. An overview of the preliminary design of SDD-1: A system for distributed databases. Proc. 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Lab., U. of California, Berkeley Calif., May 1977, pp. 39–57.

21. ROTHNIE, J.B. JR., ET AL. Introduction to a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (March 1980), 1–17.
22. STEARNS, R.E., LEWIS, P.M., II, AND ROSENKRANTZ, D.J. Concurrency controls for database systems. Proc. 17th Ann. Symp. on Foundations of Computer Science, IEEE, N.Y., 1976, pp. 19–32.
23. STONEBRAKER, M. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. Software Eng. SE-5*, 3 (May 1979), 203–215.
24. THOMAS, R.H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4*, 2 (June 1979), 180–209.
25. WONG, E. Retrieving dispersed data from SDD-1: A system for distributed databases. Proc. 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Lab., U. of California, Berkeley Calif., May 1977, pp. 217–235.

Received December 1978; revised August 1979