The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The General Case)

1

[redacted]

Technical Report
CCA-77-09
December 15, 1977

Philip A. Bernstein
David W. Shipman
James B. Rothnie
Nathan Goodman

A03/75
1PT

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

Computer Corporation of America

The Concurrency Control Mechanism of SDD-1:
A System for Distributed Databases
(The General Case)

Philip A. Bernstein
David Shipman
James B. Rothnie
and
Nathan Goodman

Technical Report
CCA-77-09

December 15, 1977

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

This research was supported by the Defense Advanced Research Project Agency of the Department of Defense under Contract No. 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 Defense Advanced Research Projects Agency or the U.S. Government.

Abstract

SDD-1, a System for Distributed Databases, is a distributed database system being developed by CCA. SDD-1 permits data to be stored redundantly at several database sites in order to enhance the reliability and responsiveness of the system and to facilitate upwards scaling of system capacity. This paper describes the algorithm used by SDD-1 for updating data that is stored redundantly.

Table of Contents

1. Introduction

SDD-1 is a prototype distributed database system currently being designed at Computer Corporation of America [ROTHNIE and GOODMAN]. The system will use the data storage facilities of Datacomputers [MARILL and STERN] that are scattered around an Arpanet environment [METCALF]. This report describes the basic approach to the problem of redundant update in SDD-1. Descriptions of other aspects of SDD-1, such as retrieval and reliability, are reported elsewhere [ROTHNIE and GOODMAN], [WONG], [HAMMER and SHIPMAN].

Several solutions have recently been suggested to the concurrent update problem in a distributed database system (see discussion in [ROTHNIE and GOODMAN]). The techniques include performing all updates at a primary site [ALSBERG and DAY], or using a voting discipline to perform an update on a data item after the sites that hold a copy of that data item have agreed to the update [THOMAS]. However, these methods suffer from the problem either of a potential bottleneck on updates or of heavy communication traffic.

The approach to be discussed in this paper attempts to overcome both problems by preanalyzing those transactions that will be run frequently, so as to select those transaction types that can be run using little or even no synchronization.

The preanalysis technique determines, for each type of transaction, the level of synchronization required for that transaction type. The analysis is based on knowledge of which portions of the database each transaction will read or write. This analysis is based on invariant properties of each transaction type that are in no sense stochastic. The major assumption is that the types of transactions that account for most of the database activity are predictable in the sense that they only operate on certain restricted portions of the database.

The SDD-1 system will permit data to be stored redundantly around the network without restricting any one copy of a logical data item to be the primary copy for updates. The retrieval algorithm will be truly distributed, aggregating data at a single site for synchronization purposes only when necessary [WONG]. The system will also be able to run in spite of multiple site failures and will be able to recover when down sites return to operation [HAMMER and SHIPMAN].

In this paper we describe the formal methods used to analyze the degree of synchronization required by transactions in SDD-1. While we believe our method to be quite general, the discussion will be limited to its application in the SDD-1 environment.

A simplified version of the SDD-1 concurrent update methodology was presented in [ROTHNIE et al] and [BERNSTEIN et al]. We expand this technique more completely in Sections 2 and 3. The proof of correctness of our synchronization rules is presented in Section 4. In Section 5, a further mechanism is described which extends the earlier results.

2. The SDD-1 Architecture

2.1 Overview

An SDD-1 database system consists of a set of sites, each site residing at a single node of the network. A site provides some or all of the following subsystems:

1. data module - maintains a stored copy of portions of the logical database and supervises read and write operations on its copy;

2. transaction module - processes transactions, one at a time, by communicating with data modules;

3. terminal module - provides a user interface that routes each user transaction to the appropriate transaction module for processing.

From the user's viewpoint, a transaction is entered at a terminal and received by the terminal module that controls that terminal. The terminal module examines the transaction and decides which transaction module should execute it; the transaction module may or may not reside at the same site where the terminal module is located (i.e., the transaction module may be at a foreign site). The terminal module may pass certain synchronization information to the transaction module, in addition to the text of the transaction, to synchronize this transaction with other transactions that ran at the same terminal.

A transaction module receives transactions from many different (possibly foreign) terminal modules. For each transaction it receives, a transaction module interacts with various (possibly foreign) data modules to obtain the portion of the database necessary for processing the read and write operations requested by the transaction. Results of the transaction (e.g. printed output) are passed back to the terminal module that sent the transaction.

A data module is a database management facility that processes read and write operations from (possibly foreign) transaction modules. Certain synchronization facilities are supported by the data module so that transactions are able to obtain a consistent view of the database. The synchronization facilities supplied by a data module are entirely local to that data module and do not require that the data module ever explicitly cooperate (via message passing, say) with other data modules.

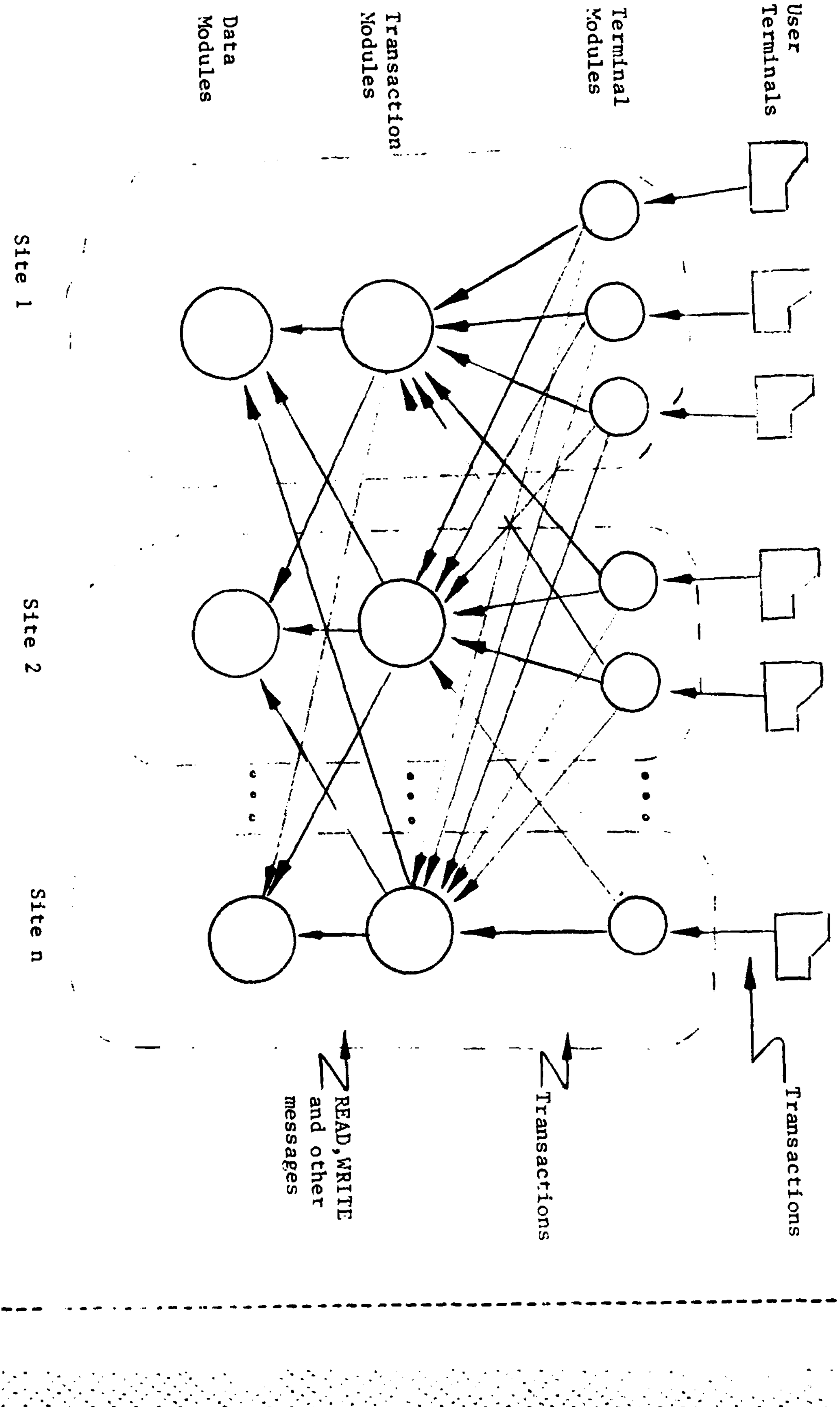


Figure 2.1 Overview of Logical Architecture

The three kinds of modules supported by SDD-1 constitute three levels of virtual machines (see figure 2.1). At the lowest level are the data modules. They provide a facility for processing read and write commands atomically. At the second level are transaction modules. Transaction modules provide a facility for processing transactions and guarantee that the union of all transactions processed by an SDD-1 system is "serially reproducible" (this concept, discussed in [ROTHNIE and GOODMAN], will be developed in great detail in the sequel). At the third level are terminal modules. Terminal modules provide a user interface and guarantee certain consistency conditions among transactio s run at that terminal (in addition to serial reproducibility). While we will not discuss the particular software/hardware structure that will be used to implement the virtual machines, one can think of the three types of modules being implemented as software processes, with each data module incorporating a Datacomputer [MARILL and STERN].

2.2 Distributed Data Organization

A logical database in SDD-1 consists of a set of relations [CODD]. Each relation has one domain named "tuple identifier" (TID) which is a key of the relation; that is, no two tuples of a relation can have identical TID values.

Each relation is partitioned into a set of logical fragments. Logical fragments are defined by first partitioning the set of all possible tuples of the relations into a set of mutually exclusive partitions. For example, the EMPLOYEE relation could be partitioned by DEPARTMENT, so that each partition contains all of the employee tuples in a single department. A logical fragment consists of a projection of a partition on the TID domain and one other domain. The inclusion of the TID domain guarantees that the logical fragment has exactly one tuple for each tuple of the partition from which it was selected.

A stored copy of a logical fragment is called a stored fragment. Stored fragments are the units of data distribution; a stored fragment is either entirely present or entirely absent at a data module. Note that several stored fragments from a single partition of a relation might conveniently be stored as a single file at a data module so that the TID domain need not be repeated for every fragment.

We do not require that two stored copies of a logical fragment at two different data modules be identical at all times. The redundant update mechanism will be responsible for only allowing consistent copies to be read.

Each logical fragment is partitioned into logical data items, a stored copy of which is called a stored data item. A data item is the smallest updatable unit in the database.

Each logical data item may have several associated stored data items. Hence, when referencing a logical data item, it is necessary to choose a particular stored data item to reference. The concept of materialization is convenient here. Formally, a materialization is a total function from the set of logical fragments into the set of stored fragments. That is, a materialization is an assignment of a stored fragment for each logical fragment.

Each transaction is said to run in a particular materialization of the database. The materialization of a transaction specifies which copies of logical fragments are to be read. In order to maintain the internal consistency of all stored copies of a particular logical fragment, a transaction must perform its updates on all stored copies of each logical data item (not just the copy specified by the materialization). As a result, materializations are not useful when considering write operations. The process of updating fragments will be described later in some detail.

There are no logical restrictions on how to configure a materialization, other than that each logical fragment must map into a stored copy of that same fragment. A materialization need not, for example, obtain any of its stored fragments from the site at which it executes. Also, two materializations may use different stored copies of a single logical fragment. Two transactions concurrently running in these materializations may therefore read different stored copies of a single logical fragment concurrently. The system as a whole does not support a single primary copy of a logical fragment for all materializations. How the system avoids race conditions in such an apparently chaotic environment is the main subject of this report.

2.3 Transactions

The basic unit of a user computation in SDD-1 is the transaction. Transactions are structured to execute in three sequential steps:

1. The transaction reads a subset of the database, called its read-set, into a workspace.

2. It does some computation on the workspace.

3. The transaction writes some of the values in its workspace back into a subset of the database, called its write-set.

The read-set and write-set of a transaction are defined on the logical database. That is, the transaction references only logical data items; it has no knowledge of its materialization or of the distribution and redundancy of stored copies.

The workspace into which data is read is, in general, distributed. That is, various parts of the workspace may reside at different data modules. In SDD-1, the execution of a transaction is also, in general, distributed;

processes running at various data modules operate on the portion of the workspace located at that data module. These processes run concurrently and/or sequentially with respect to one another and transfer data between data modules as needed. The processes running at the data modules are initiated and coordinated by the original transaction module to which the transaction was submitted. This function is performed by the access planner sub-module within the transaction module. The access planner converts the original transaction as submitted by the user into a number of local data management processes running at the data modules where the workspace is stored. The algorithms used by the access planner are described in [WONG]. Again, this distribution of processing is entirely internal to SDD-1 and is not reflected in the user's transaction in any way.

To process a transaction, a transaction module must obtain the read-set data for the transaction's input and later write its output into copies of its write-set. These functions are performed by sending READ and WRITE messages, respectively, to data modules.

A READ message for a transaction is sent to a data module and is a request to read some of the stored data items at that data module. Each stored item that is requested must be the particular stored copy of a logical data item in the read-set of the transaction that is specified by the materialization in which the transaction runs. So, if a transaction wants to read logical data item x, and the transaction's materialization associates x with its particular stored copy at data module alpha, then to read x the transaction must send a READ message to alpha.

A WRITE message is sent from a transaction module to a data module to report updates that have taken place to certain data items as a result of executing a transaction by that transaction module. If a transaction updates a particular logical data item x, WRITE messages are sent to all data modules that have a stored copy of x (not just to the one stored copy associated with the transaction's materialization).

A transaction module sends at most one READ message and at most one WRITE message to any particular data module on behalf of a single transaction. If a transaction reads data from two stored fragments which reside at the same data module, for example, then only one READ message will be issued to read from both fragments. This is an important point, as each data module must perform READ's and WRITE's as atomic operations; for example, none of the data read by a READ message can be updated by some WRITE while the READ is being processed.

2.4 System Consistency Guarantees

One of the important advantages of SDD-1 is its ability to maintain multiple copies of the same logical piece of data at several different data modules. It is this capability of SDD-1 that presents the most difficult technical problems. The system must maintain the consistency of all copies of data and ensure that the READ requests for a transaction retrieve a correct state of the database. In addition, transactions reading or writing data in several data modules must be synchronized to ensure that a transaction does not read partial results of another transaction. If transactions are allowed to run in an arbitrary interleaved manner without coordination, various anomalies in system operation may occur. The system design guarantees two properties which prevent these anomalies from occurring.

System Property 1: Convergence - If updates were to be quiesced, then after some finite period of time all transactions which read the same logical data item will retrieve the same value for it. Essentially this means that all physical copies of a logical data item will eventually converge to the same value.

System Property 2: Serial Reproducibility (or Serializability) - The operation of the system when running transactions in an interleaved manner is equivalent to a history of operation in which each of the transactions runs alone to completion before the next one begins. That is, the interleaved operation is reproducible by an equivalent one in which the transactions run serially. By "equivalent", we mean that each transaction produces the same output values and that the final state of the database is the same. The concept of serial reproducibility is crucial to an understanding of the system and will be taken up in detail later.

These two system properties are provided at the transaction module level. That is, the set of all transactions submitted to transaction modules must satisfy these properties. The terminal modules provide a level of system guarantee beyond that of the transaction module. These guarantees however are not the main subject of this paper.

2.5 Terminal Modules

A transaction is entered at a terminal and is received by the terminal module connected to that terminal. The terminal module must determine the read-set and write-set of the transaction. This information will be used to decide which transaction module should execute the transaction, as each transaction module handles only certain classes of transactions. For example, in an airline reservation system, each transaction module may execute transactions corresponding to flights originating at a certain city. By examining the read-set and write-set of a reservation transaction, a terminal module can determine the originating city and thereby is able to choose an appropriate transaction module to execute the transaction.

The terminal module makes sequencing guarantees above and beyond those of the transaction modules. The terminal module incorporates certain synchronization information with the transaction before sending it to a transaction module. This information allows the transaction module to avoid certain sequencing anomalies with respect to other transactions entered at the same terminal.

The main body of this paper however concerns the design and interaction of the transaction modules and data modules. For convenience, transaction modules and data modules will be referred to as TM's and DM's, respectively, in the sequel.

2.6 Timestamps

System property 1, convergence, is provided in SDD-1 through the use of a timestamping mechanism. Each TM has a clock used for generating globally unique timestamps. After a clock has been read, it cannot be read again until it has been incremented. By appending the TM number as the low order bits of each timestamp, we ensure that every timestamp is globally unique within the system. This method of generating unique timestamps was suggested in [THOMAS].

None of the mechanisms described in this report require that clocks running in different TM's be at all synchronized. For reasons of efficiency however it is necessary to assume that clock values in different TM's be reasonably close to each other. In [Lamport] a method of synchronizing clocks in a network is described that involves pushing ahead a local clock if a message with a future timestamp is received. This simple method will keep clocks sufficiently well synchronized for the purposes of SDD-1.

Each transaction, before being run, is assigned a unique timestamp. The transaction's timestamp will be carried on all its WRITE messages.

In addition, timestamps are maintained for every updatable physical data item in the database. Note that a timestamp is associated with each physical data item, rather than with the logical data item; there may be many physical copies of a logical data item and each copy of the logical data item has its own timestamp. This timestamp is the timestamp of the last WRITE message which updated that physical data item.

In order to implement property 1, convergence, each data module obeys the following rule: A data item is updated by a WRITE message if and only if the data item's timestamp is less than the timestamp of the WRITE message. So, to process a single WRITE message at a data module the following procedure is used. For each data item in the WRITE message, the timestamp in the WRITE message is compared with the timestamp of the stored data item at that data module. If the timestamp in the WRITE message is greater than 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 with the new timestamp. If the timestamp of the WRITE message is less than the timestamp of the stored data item, then the update is not performed on that 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. Also, if a data item in the WRITE message is part of a fragment that is not stored at the data module, then the update is not performed.

It will be quite common for WRITE messages to contain many data item updates that are not performed. This will happen when 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. Such situations are not errors. They are simply the way that the system reorders updates to occur in the same order that they actually executed.

2.7 Interleaved Transactions

The system usually has many transactions in progress at any one time, both because there are multiple TM's operating concurrently within the system and because individual TM's are processing transactions concurrently. The resulting arbitrary interleavings of READs and WRITEs can introduce serious problems of database consistency. System Property 2, serial reproducibility, deals with this problem.

The issue of serial reproducibility arises because a system's atomic actions are at a finer granularity than its users' atomic actions. In our case, the users' atomic operations are user transactions, while the system's atomic actions can be taken to be the execution of READ and WRITE messages at the DM's. Each DM behaves as if READ's and WRITE's are processed as indivisible units. That is, it is not possible for a READ operation to observe the effects of only a part of a WRITE operation at a DM.

When a system allows the execution of several user transactions at the same time, then the system atomic operations corresponding to different user transactions are interleaved. There is no guarantee that the behavior of such a system conforms to the user's expectation that each transaction is treated as an indivisible unit (a user's transaction should not examine the database during the execution of another user's transaction, when the database is possibly in an inconsistent state).

Serial reproducibility requires that a system operating in an interleaved manner is equivalent to a system in which each transaction is processed in its entirety before another one is begun. In other words, for any given interleaved execution, there exists an ordering of atomic transactions, called a serial ordering, which is equivalent to the interleaved operation which in fact occurs. By "equivalent" we mean that each transaction in the interleaved ordering reads the same data as it would have read if the transactions had been run one at a time in the serial order (and hence, will produce the same output). Note that serial reproducibility requires only that there exists _some_ serial order equivalent to the actual interleaved operation. There may in fact be _several_ such equivalent serial orderings.

The modelling of correct concurrent operation by the concept of serial reproducibility is based on the assumption that each user transaction will preserve database consistency if it runs atomically. That is, if only one transaction were allowed to execute at a time, and if the database state is consistent, then after executing a transaction the database state will be consistent. So, a serial ordering of transaction executions will, by induction, result in a consistent database state. Since a serially reproducible history of operation is equivalent to some serial ordering, then the serially reproducible history results in a consistent database state as well.

If a system does not guarantee serial reproducibility then anomalies can result from operation of the system. Consider, for example, the following scenario in SDD-1. We assume 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 execution of the two transactions one after the other would have resulted in setting x to 3. The result of the interleaved execution was to set x to 1, contrary to the user's intention. To guarantee serial reproducibility, we need a mechanism that prevents these kinds of undesirable interleavings.

2.8 Transaction Classes

The problem of interleaved transactions is not unique to distributed systems. Numerous solutions have been devised for non-distributed systems, most notably locking mechanisms. These techniques do not, however, generalize well to distributed systems. A number of proposals have been suggested for extending locking mechanisms to distributed systems that contain redundant data. These techniques are reviewed in [ROTHNIE and GOODMAN]. We feel, however, that such techniques require unacceptably large amounts of network transmission and delay whenever there is considerable data redundancy.

Yet at first glance the network transmission seems to be necessary. How can one TM safely proceed to run a transaction without first consulting other TM's to determine that it does not interact badly with transactions currently executing elsewhere?

Our solution to this problem is to have the DBA establish a static set of transaction classes. Each transaction class is defined in terms of its logical read-set and write-set and is assigned to run at a particular TM. A transaction can run in a class if the read-set and write-set of the transaction is contained (respectively) in the read-set and write-set of the class. Classes need not be disjoint, so a transaction may fit into more than one class. In this case, the decision as to which class should be chosen is made by the terminal module that accepts the transaction. The terminal module will normally choose a class that requires the least amount of synchronization, and is therefore the least expensive class (synchronization-wise) to use.

The predefined classes reflect the typical transactions that are intended to run at each site in the network. Since each TM is aware of the complete set of transaction classes assigned to foreign transaction modules, it can know exactly what potential conflicts its own transactions have with those that might be running at other TM's.

From the information contained in the class definitions, a TM can determine the degree and nature of coordination necessary to ensure a serially reproducible ordering of transactions. We believe that, for many kinds of applications, the most frequent determination will be that no coordination whatsoever is actually required to run a transaction. In such a case, the transaction is just immediately executed, since it does not interact badly with transactions submitted elsewhere. In other cases, an analysis of the class definitions might indicate that the pending transaction could be involved in a potential conflict and some coordination is necessary with respect to particular foreign classes. Our purpose here is to develop a method of determining exactly what conflicts occur and to provide coordination mechanisms that eliminate the conflict.

If the problem of determining exactly what conflicts might occur required run-time calculations when each transaction was introduced at a class, then the concurrency control mechanism would potentially be quite expensive. Actually, since the class definitions are static, the computations checking for potential conflicts can be done once, when the class definitions are selected. Selecting the appropriate coordination mechanism at run-time amounts to a table look-up. So, the only significant run-time overhead is the coordination mechanism itself. If no coordination is found to be necessary, then the run-time overhead is negligible. This is in contrast to locking mechanisms which always set locks, whether or not the synchronization is really required.

2.9 Class Pipelining Rule

The first question to address is the issue of the serializability of transactions which execute in the same class. To ensure this, we require that within a class all of the transactions are actually executed serially, one after another. This is expressed as follows-

> Class Pipelining Rule: For any particular data module and transaction class, READ and WRITE messages from that class arrive and are processed in timestamp order.

The class pipelining rule forces transactions that run in a single class to be processed serially at all DM's in the same order. So, two transactions from a single class are never interleaved at a single DM nor are they processed by two DM's in two different orders. This is sufficient to guarantee noninterference of any two transactions that run in a single TM.

2.10 Class Conflict Graphs

Given the set of class definitions, we need to detect potentially harmful interactions between classes. The approach used to resolve these questions involves the construction and analysis of a *class conflict graph*.

A class definition specifies a logical read-set and write-set and a materialization. This is the only information required to determine class conflicts. From the read-set and the materialization, the READ messages needed by the class can be predicted. From the write-set, the WRITE messages needed by the class can be predicted, since a WRITE message must be sent to *all* copies of the logical write-set. Since all READ and WRITE messages are predictable, we will be able to predict all possible harmful interactions between classes.

A class is represented in the class conflict graph as three types of nodes connected by edges. The three types of nodes are e, r and w nodes.

An e node represents the execution of a transaction which runs in the class. A class superscript (e.g. e^I)

designates the class identifier for the transaction class. (Throughout this report, transactions will be indicated by lower case letters, and transaction classes by lower case letters with an overscore.) The graph includes exactly one e node per class.

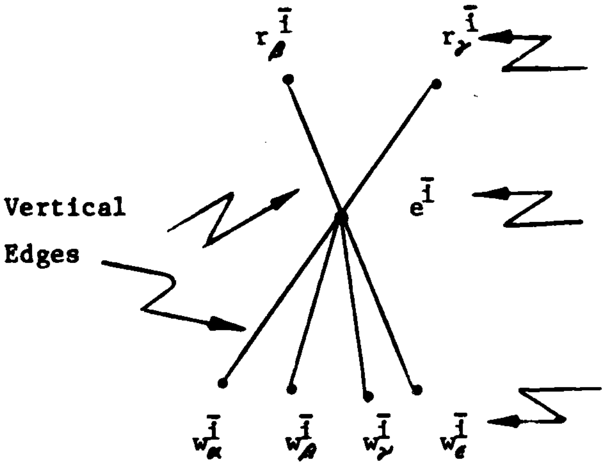
An r node represents the processing of a READ message to retrieve data for transactions in the class. A superscript represents the class identifier and a subscript indicates to which DM the READ message would be sent (e.g. $r^{\bar{j}}_{alpha}$ represents a READ message from a transaction in class $\bar{j}$ to DM_{alpha}). (Lower case Greek letters denote DMs.) For any class, there is one r node for each DM which stores part of the class's (physical) read-set.

A w node represents the processing of a WRITE message issued by a transaction running in the class. Again, a superscript indicates the class identifier and a subscript indicates the DM to which the WRITE message would be sent (e.g. $w^{\bar{i}}_{gamma}$). For any class, there is one w node for each DM on which a copy of (some of) the write-set items lie.

Edges connect the e node for a particular class with the r and w nodes for that class. These edges are called vertical edges, because of the convention that, for each class, r nodes are drawn above the e node and w nodes are drawn below the e node.

Figure 2.2 illustrates the representation of a class whose read-set lies in two datamodules and whose write-set lies on four datamodules.

After all the predefined transaction classes have been placed in the graph, additional edges are added to indicate interactions between the classes.



There are two READ messages, one to DM_{β} and the other to DM_{γ}

This is transaction class #14

Data must be written to four DM's: $\alpha, \beta, \gamma, \epsilon$

Figure 2.2 Representing transaction classes in the graph

Where two classes have a read/write intersection, a *diagonal* edge is drawn. The edge is drawn between an r node which represents the reading of some particular physical data item and a w node which represents the writing of that same item (see Figure 2.3). Note that such a diagonal edge only connects r and w nodes with the same DM subscript, since a physical data item resides at only one DM. If the intersection of one class's read-set and another's write-set spans more than one DM, then several diagonal edges connect the two classes (see Figure 2.4).

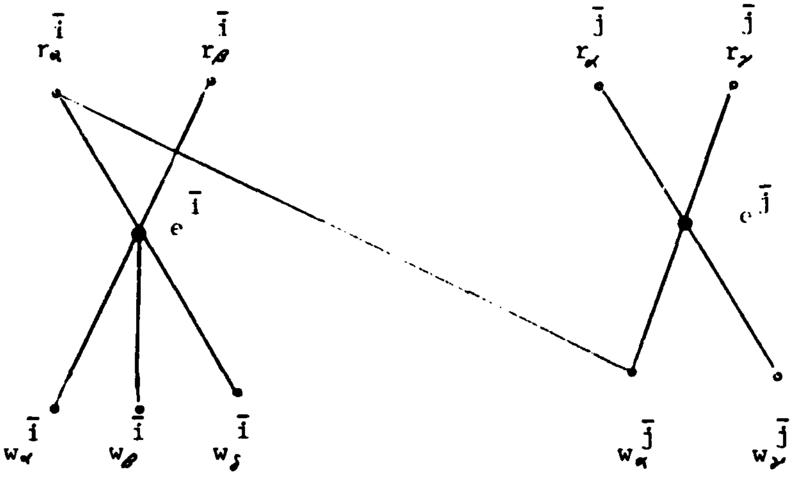


Figure 2.3 - The diagonal edge indicates that class $\bar{i}$ reads some data item from DM_α which can be written by class $\bar{j}$.

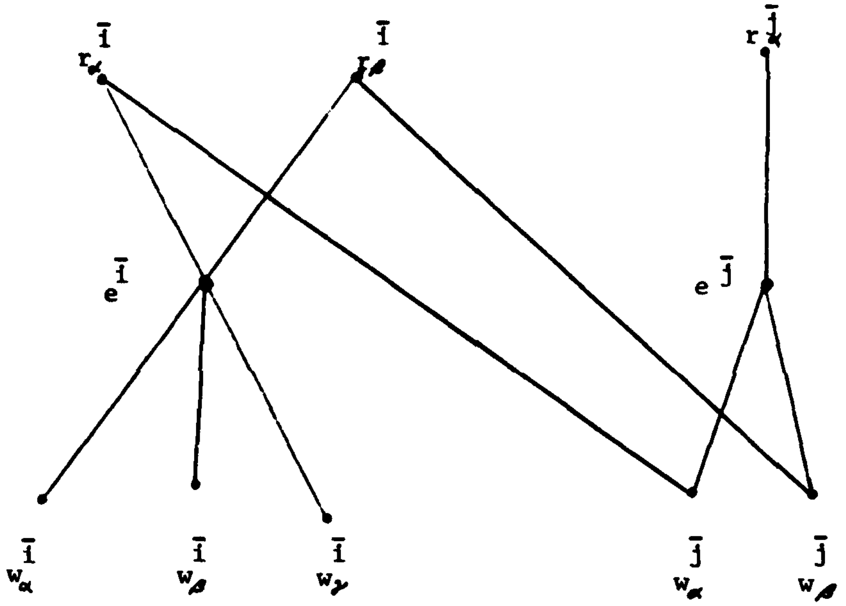


Figure 2.4 - Here two diagonal edges connect the two classes since the read/write intersection exists at both DM_α and DM_β

Horizontal edges are drawn between e nodes of two classes that have a logical write/write intersection (see Figure 2.5).

The graph must contain all classes and all possible vertical, diagonal and horizontal edges.

The conflict graph is used to determine unsafe interactions among a set of classes. By "unsafe" we mean

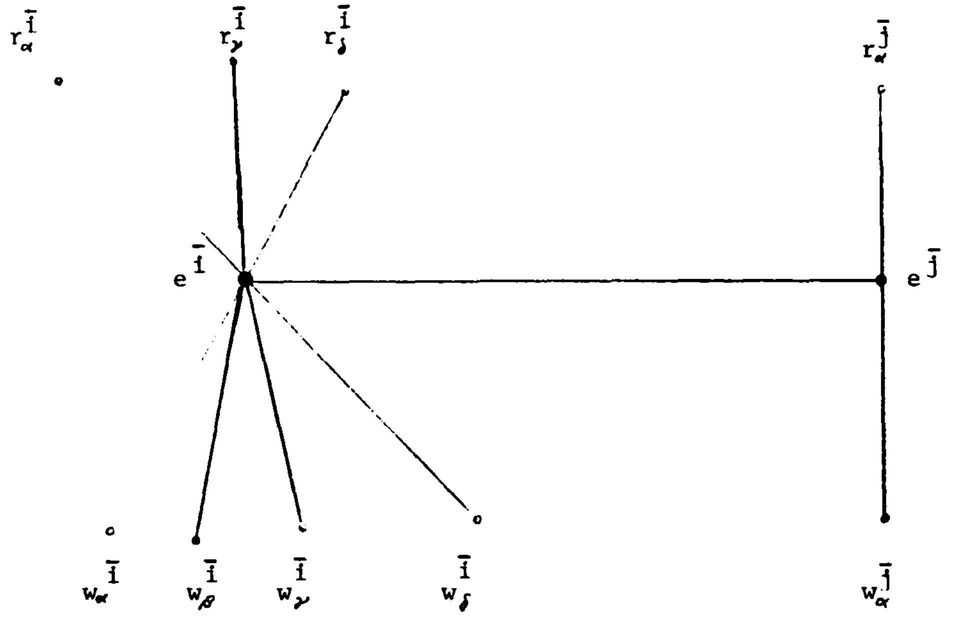


Figure 2.5 - A horizontal edge is added to the graph when two classes write the same data item.

that the classes can interact in such a way that there is no serial ordering of transactions that is equivalent to the interleaved execution that actually occurred. The interpretation of the diagonal and horizontal edges applied to a given interleaved execution is the key to determining transaction serializability.

2.11 Graph Cycles and Nonserializability

Suppose the system executes in a manner that permits the interleaving of READ and WRITE messages from different transactions. We call such an interleaved execution a log. If the execution is not interleaved, that is, if transactions execute serially one after the other, then we call the execution a serial log. Our goal is to only permit the system to produce logs that are serially reproducible. This means that for each log resulting from the execution of the system, there must exist a serial log that produces the same effect on the database. We say that two logs are equivalent if they produce the same effect on the database.

Of course if the transactions in a log are arbitrarily reordered into a serial log, the resulting serial log will not necessarily be equivalent to the given log. The conflict graph helps us to characterize precisely those serial logs that produce the same effect as a given log.

Consider diagonal graph edges. A diagonal edge represents a read/write intersection between two classes. If one transaction from each of the two classes appears in the given log, then in any equivalent serial log the transactions should appear in the same relative order as their intersecting READ and WRITE messages were processed in the given log. For if the READ message of one transaction preceded the WRITE message of the other in the given log, but the transactions appear in the reverse order in the serial log, then in the serial log the READ message may read different values for some of its inputs in the serial log than reads in the given log. So, the transaction corresponding to the READ may produce a different output in the serial log than in the given log. That is, the two logs are not necessarily equivalent. This is just to say that only some serial reorderings of the given log are possible, given the existence of this diagonal edge. (Actually, the above claim about permissible serial reorderings is somewhat too strong, as shown in [PAPADIMITRIOU et al]. However, the reasons are quite technical in nature and are not needed to gain an understanding of the interpretation of conflict graphs.)

Consider classes $\bar{I}$ and $\bar{J}$ in figure 2.3. We denote READ and WRITE messages using a notation similar to that of node labels. The processing of the READ message for transaction i at DM_{alpha} is denoted R^{i}_{alpha}; the processing of the WRITE message for transaction i at DM_{alpha} is denoted W^{i}_{alpha}.

Assume two transactions, say i and j, are running concurrently in classes $\bar{i}$ and $\bar{j}$ respectively. If the READ message R^{i}_{alpha} is processed at DM_{alpha} before the WRITE message W^{j}_{alpha} is processed, then any equivalent serial ordering must have transaction i precede transaction j. This must be so, for otherwise transaction i would have read the results of the update made by transaction j. On the other hand, if the WRITE message W^{j}_{alpha} is processed before the READ message R^{i}_{alpha}, then transaction j must precede transaction i.

To reiterate, a diagonal edge implies a particular relative ordering in any serial log that is equivalent to the given interleaved execution. The particular ordering that is chosen depends on the particular order in which READ and WRITE messages were processed; however the relative serial ordering of transactions from classes with a diagonal edge connecting them is not arbitrary.

Horizontal edges also affect possible reorderings of transactions. A horizontal edge indicates an intersection of write-sets. Whenever two transactions write the same data, the update from the transaction with the greater (i.e. later) timestamp takes precedence over the update from the transaction with the smaller (i.e. earlier) timestamp. If two transactions in different classes appear in an interleaved execution and have a write/write intersection, then they must appear in timestamp order in any equivalent serial log. Otherwise, the effect of the intersecting write messages would be reversed, thereby producing a different database state. Notice that it is the timestamp order of the transactions and not the order in which the WRITE messages were processed that is significant here. This is because the rule by which WRITE messages are processed uses the timestamps, not the order of arrival of the WRITE messages, to determine which write operations are actually applied.

So, a horizontal edge also implies a particular relative ordering of certain transactions in any serial log that is equivalent to the given interleaved execution. This ordering is always the timestamp ordering of the transactions that have the write/write intersection.

In the same way that diagonal and horizontal edges restrict the ways in which transactions can be reordered without upsetting the resulting database state, paths of edges can restrict reorderings of transactions as well. For example, a particular diagonal edge may imply that transaction i must precede transaction j and an adjacent horizontal edge may indicate that transaction j must precede transaction k (see figure 2.6). So, the net effect of this path is that transaction i must precede transaction k, even though no single edge may connect their respective classes in the conflict graph.

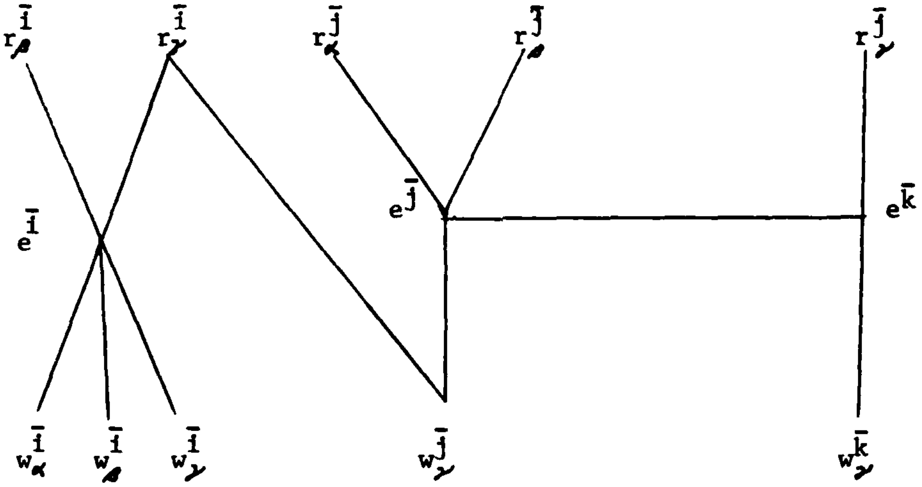


Figure 2.6 - A path between two classes in the graph indicates that transactions in those classes must be serialized in some particular order.

Now, suppose again that we have a conflict graph and a log of interleaved transactions. Suppose that for each pair of transactions, say i and j, the log and graph edges never imply both that i must precede j and that j must precede i in the serial reordering. That is, either i and j can appear in an arbitrary order, or there is only one order that will do. Then it is easy to see that there must be a serial log equivalent to the given log. Any serialization that preserves the relative orderings that are demanded by the graph serves the purpose.

However, suppose instead that there are two transactions such that one path in the graph requires that they appear in one order and another path in the graph requires that they appear in the other order. Then there is no equivalent serial log that includes these two transactions, for whatever order that they appear in the serial log, the graph indicates that they must also appear in the other order. In this case, there are two different paths connecting the two transactions' classes in the graph. These two paths constitute a cycle in the graph. So, apparently a cycle in the graph corresponds to a non-serializable execution of transactions. If there are no cycles, then there is at most one path connecting any pair of classes. Hence, the graph can only require that two transactions be serialized one way or the other, but never both ways. So, a cycle-free graph implies that every log is serializable, and _no synchronization whatsoever is required_. The preceding informal argument demonstrating this fact will be proved quite rigorously in Section 4.

Consider the cycle in Figure 2.7 consisting of two diagonal edges and four vertical edges. If we examine a

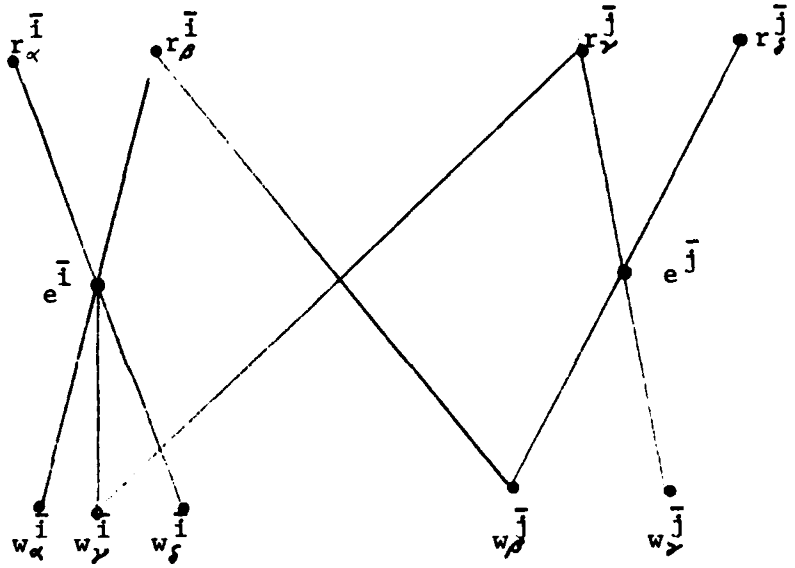


Figure 2.7 - Cycles represent situations in which non-serializability is possible.

case of concurrent transactions in each of the two classes and the particular sequence of events in which the READ message R^{i}_{beta} is processed before the WRITE message W^{j}_{beta}, and the READ message R^{j}_{gamma} is processed before the WRITE message W^{i}_{gamma}, then there is no serial ordering of the two transactions which is equivalent to their interleaved ordering. This follows because the $r^{\bar{I}}_{beta}$-$w^{\bar{J}}_{beta}$ edge requires that the transaction in class $\bar{I}$ occurs before the transaction in class $\bar{J}$, yet the $r^{\bar{J}}_{gamma}$-$w^{\bar{I}}_{gamma}$ edge implies the opposite relative ordering. Therefore, it must be the case that no equivalent serial ordering exists.

We have shown that potentially dangerous interleavings can be identified by a cycle in the class conflict graph. So, as long as no cycles exist, the class pipelining rule is sufficient to guarantee serializability. Where cycles do exist, some synchronization among classes is required. In SDD-1, this synchronization is accomplished by protocols.

2.12 Protocol P3

When a cycle exists in the conflict graph, then an interleaved execution might be such that a pair of transactions, i and j, must be serialized with i preceding j and j preceding i, clearly an impossibility. Protocol P3 prevents this situation 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 the diagonal edge. For example, suppose the edge $(r^{\bar{i}}_{alpha}, w^{\bar{j}}_{alpha})$ lies on a cycle and transaction i executes in class $\bar{I}$ and j executes in class $\bar{J}$. Then, assuming protocol P3 is observed, R^{i}_{alpha} is processed before W^{j}_{alpha} if and only if the timestamp of i is smaller that the timestamp of j. Before describing how P3 accomplishes this task, let us first examine how P3 prevents nonserializable executions.

Consider again transaction i and j above. Since they apparently must be serialized in both orders, there must be two independent paths connecting them in the graph, such that one path requires that i precede j and the other requires that j precede i. Suppose the timestamp of i is smaller than that of j. So, the path that requires j to precede i in the serial reordering is trying to serialize them in reverse timestamp order. But suppose every transaction pair connected by a diagonal edge in this path observes P3. Then each such pair must be serialized in timestamp order, as P3 requires. Consider a pair of transactions connected on the path by a horizontal edge. Following the discussion about horizontal edges in the last section, they too must be serialized in timestamp order. Thus, every pair of transactions in the interleaved execution that corresponds to a graph edge along this path must be serialized in timestamp order. The net effect (by induction on the length of the path) is that the entire path requires that i and j be serialized in timestamp order. But this is a contradiction, since the chosen path was one that required the transactions to be serialized in reverse timestamp order. The conclusion is that all paths in the graph between $\bar{i}$ and $\bar{j}$ require that i and j be serialized in timestamp order. Protocol P3 prevents the case that there are two independent paths between $\bar{i}$ and $\bar{j}$ that require opposite relative orderings.

To implement protocol P3, we need to synchronize the READ and WRITE messages of transactions that correspond to the endpoints of a diagonal edge in a cycle. To explain the operation of P3, suppose that the edge $(r^{\bar{I}}_{alpha}, w^{\bar{J}}_{alpha})$ is a diagonal edge in a cycle; so, for each transaction i in class $\bar{I}$, R^{i}_{alpha} has to run P3 against class $\bar{J}$ at DM_{alpha}. This is accomplished by appending a read condition to each read message R^{i}_{alpha}. The read condition includes the timestamp of transaction i, say TS_i, and the name of the class against which P3 is being run, in this case $\bar{J}$. A data module, upon encountering a READ message with the attached read condition $< TS_i, \bar{J} >$, must not process the READ until it is certain that all WRITE messages from $\bar{J}$ with timestamps prior to TS_i have been received and processed, and that it has not processed any WRITE messages from $\bar{J}$ with a timestamp greater than TS_i. This ensures that the READ messages R^{i}_{alpha} is processed before a WRITE message from $\bar{J}$ if and only if TS_i is smaller than the timestamp of the transaction corresponding to the WRITE message. That is, it guarantees that the diagonal edge forces transactions from the two classes to be serialized in timestamp order. We refer to this mechanism as protocol P3, and would say, for example, that transactions in class $\bar{I}$ run protocol P3 against transactions in class $\bar{J}$ at DM_{alpha}.

Several problems arise about the operation of protocol P3. Suppose the DM has already processed a WRITE message from the specified class $\bar{J}$ with a timestamp greater than TS_i.

In this case, the READ message must be *rejected* by DM_{alpha}. The initiating TM then assigns a new timestamp to the transaction and resubmits its READ requests. Notice that *all* READ messages must be resubmitted if any READ message is rejected.

A more serious problem is how to guarantee that a DM has received all WRITE messages through some particular time. The solution lies in the class pipelining rule. Recall that READ and WRITE messages from a class to a DM must be processed in timestamp order. If DM_{alpha} wants to process all WRITE messages from $\bar{j}$ up to but not past time TS_i, it simply processes all WRITE messages from $\bar{j}$ until it receives one with a timestamp greater than TS_i. It holds this WRITE message until R^{i}_{alpha} is processed, thereby satisfying the read condition attached to R^{i}_{alpha}.

Unfortunately, if class $\bar{j}$ is idle because it has no transactions to process, DM_{alpha} may need to wait for a long time until a message timestamped later than TS_i arrives from $\bar{j}$. To handle this problem we have TM's send out NULLWRITE messages to appropriate DM's. A NULLWRITE message specifies a class and a timestamp. It is semantically equivalent to a WRITE message that does not update any data. 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.

TM's will send out NULLWRITEs on a periodic basis. In addition, a TM may be specifically requested to send a NULLWRITE for a particular class and timestamp. This specific request is in the form of a SENDNULL message and may be sent by either another TM or a DM. A discussion and analysis of various strategies for sending NULLWRITE and SENDNULL messages will appear in a later report.

To illustrate the use of protocol P3 for eliminating bad interleaved executions, let us reconsider the anomalous scenario discussed in section 2.7, this time adding a bit more structure to the problem.

We assume a single copy of data item x, residing at DM_{alpha}, with initial value x=0. Class $\bar{I}$ has been defined to run at TM_{alpha} with read-set = {x} and write-set = {x}. Class $\bar{J}$ has been defined to run at TM_{beta} with read-set = {x} and write-set = {x}. The class graph in this situation is shown in figure 2.8. Notice that a cycle is present and that transactions in class $\bar{I}$ must run P3 against class $\bar{J}$ and that transactions in class $\bar{J}$ must run P3 against class $\bar{I}$.

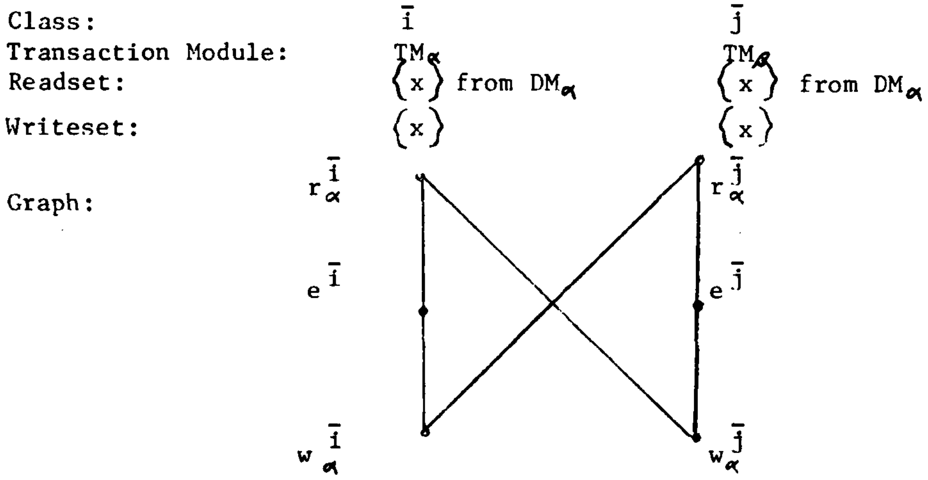


Figure 2.8 - Class Conflict Graph for Example in Section 2.12.

A transaction, i, arrives at TM_{alpha} of the form x:=x+1. TM_{alpha} assigns the transaction to class $\bar{i}$ and gives it timestamp TS_i. A transaction, j, arrives at TM_{beta} of the form x:=x+2. TM_{beta} assigns the transaction to class $\bar{j}$ and gives it timestamp TS_j. TS_i and TS_j cannot be equal because all timestamps in the system are unique. Let us assume that $TS_j < TS_i$. Now the following sequence of events occurs:

1. TM_{alpha} sends a READ message, R^{i}_{alpha}, to DM_{alpha} to retrieve the value of data item x for transaction i. This READ includes a P3 read condition against class $\bar{j}$. The READ message cannot be immediately processed because WRITE messages through time TS_{i} from class $\bar{j}$ have not yet been received at DM_{alpha}.

2. TM_{beta} sends a READ message, R^{j}_{alpha}, to DM_{alpha} to retrieve the value of data item x for transaction j. The READ message can be immediately processed (the presence of a class $\bar{I}$ READ message at DM_{alpha} with timestamp $TS_{i} > TS_{j}$ insures that all WRITE messages from class $\bar{I}$ have been received through time TS_{j}). The result of the READ is x=0.

3. TM_{beta} sends a WRITE message for transaction j to DM_{alpha} setting x:=2.

4. TM_{beta} sends a NULLWRITE message to DM_{alpha} with timestamp $TS_{j'} > TS_{i}$. (This message may be a response to a SENDNULL request from TM_{alpha}. The class pipelining rule requires that this message could not be sent before the WRITE message with time $TS_{j} < TS_{j'}$). The READ message for transaction i can now be processed. (The presence of the NULLWRITE message at DM_{alpha} with timestamp $TS_{j'} > TS_{i}$ satisfies the P3 read condition.) The result of the READ is x=2.

5. TM_{alpha} sends a WRITE message for transaction i to DM_{alpha} setting x:=3. Notice that this WRITE message overwrites the earlier value of x=2 because the earlier value was associated with timestamp TS_j and the current WRITE message has timestamp $TS_i > TS_j$.

The final value of data item x is 3, as expected. The anomalous interleaving that was described in the example of section 2.7 has been prevented by the use of protocol P3.

We have seen that by locating graph cycles, by finding every class that lies at the r-end of a diagonal edge embedded in a cycle, and by having transactions in that class run protocol P3, we can guarantee that all interleaved executions will be serializable. However, there are situations in which weaker protocols (i.e., protocols that allow more concurrency) than P3 may be used. This leads us to a discussion of protocols P2 and P2f.

2.13 Protocol P2

The main opportunity for weakening the P3 protocol arises in connection with the transactions that participate in a conflict graph only with their read-nodes. These read-only transactions contribute to non-serializability only because they may observe certain WRITE messages being processed in reverse timestamp order. For example, suppose we have classes $\bar{i}$, $\bar{j}$, and $\bar{k}$ connected by the edges $(w^{\bar{i}}_{alpha}, r^{\bar{j}}_{alpha})$ and $(r^{\bar{j}}_{alpha}, w^{\bar{k}}_{alpha})$ as shown in figure 2.9. Class $\bar{j}$ is a read-only transaction whose read-set intersects the write-sets of classes $\bar{i}$ and $\bar{k}$. Suppose transactions i, j, and k execute in classes $\bar{i}$, $\bar{j}$, and $\bar{k}$ (respectively) such that k is timestamped before i which is timestamped before j. At DM_{alpha}, the following sequence of events might occur: first W^{i}_{alpha} is processed, then R^{j}_{alpha} is processed, then W^{k}_{alpha} is processed. In this case, even though k is timestamped earlier that i, from j's point of view transaction i precedes transaction k, since it sees i's update but has not yet seen k's update. That is, this interleaved execution requires that transaction i be serialized in front of transaction k, which is the reverse timestamp order. If another path in the conflict graph connected $\bar{i}$ to $\bar{k}$ such that the interleaved execution required the timestamp ordering, then the impossible requirement that i both precede and follow k in the serial reordering means that the execution is not serializable. In the previous section we showed that if R^{j}_{alpha} ran P3 against $\bar{i}$ and $\bar{k}$ (due to the two diagonal graph edges), then this non-serializable situation could not arise. However, there is a weaker protocol that R^{j}_{alpha} can run in this

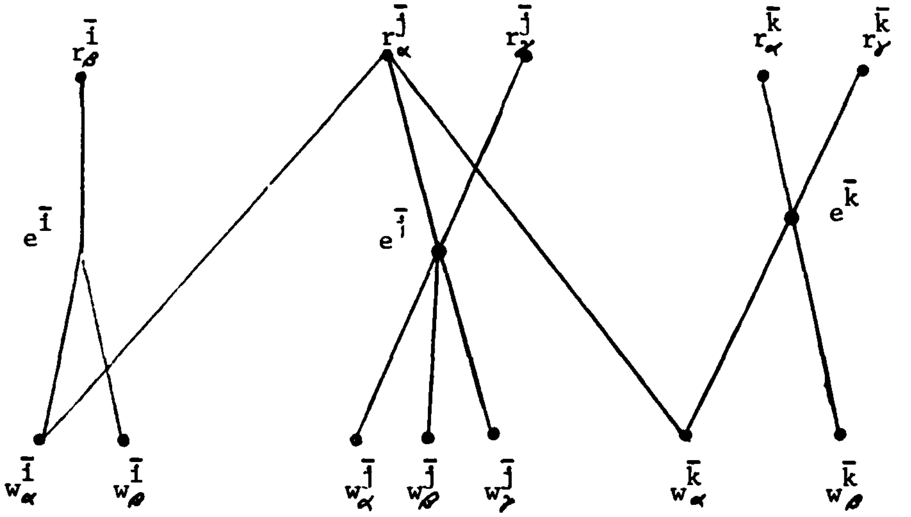


Figure 2.9 - A transaction reading from two other transaction classes may force a relative ordering of these classes' transactions in equivalent serial orderings.

situation that has the same effect.

The effect we want to produce is that if W^{i}_{alpha} is timestamped after W^{k}_{alpha} and W^{i}_{alpha} is processed before R^{j}_{alpha}, then W^{k}_{alpha} is processed before R^{j}_{alpha} as well. If this condition is made to be true (by some protocol) then R^{j}_{alpha} cannot observe W^{i}_{alpha} and W^{k}_{alpha} to execute in reverse timestamp order. The protocol that has this effect is called P2.

Protocol P2 applies to a read message R^{j}_{alpha} if and only if there are classes $\bar{i}$ and $\bar{k}$ such that $(w^{\bar{i}}_{alpha}, r^{\bar{j}}_{alpha}, w^{\bar{k}}_{alpha})$ is a subpath in a cycle in the conflict graph (where j runs in class $\bar{j}$). In this case, we say that R^{j}_{alpha} must run protocol P2 against classes $\bar{i}$ and $\bar{k}$ at DM_{alpha}. If protocol P2 is used, then R^{j}_{alpha} need not run protocol P3 against $\bar{i}$ and $\bar{k}$, as would normally be indicated by the diagonal edges. Since P2 prevents R^{j}_{alpha} from observing transactions in $\bar{i}$ and $\bar{k}$ in reverse timestamp order, R^{j}_{alpha} will not interfere with serializing transactions in $\bar{i}$ and $\bar{k}$ in timestamp order, as desired.

To run R^{j}_{alpha} under P2 against $\bar{i}$ and $\bar{k}$, DM_{alpha} must ensure that, at the time R^{j}_{alpha} is processed, there is a timestamp TS_{o}, such that all WRITE messages from classes $\bar{i}$ and $\bar{k}$ whose timestamps are less than TS_{o} have been processed at DM_{alpha}, and no WRITE messages from classes $\bar{I}$ and $\bar{k}$ whose timestamps are greater that TS_o have been processed. The specific timestamp, TS_o, is not given by the READ message R^j_{alpha} but rather is selected by DM_{alpha}. As long as there exists some TS_o through which WRITEs from the classes $\bar{I}$ and $\bar{k}$ have been processed but beyond which they have not been processed, then R^j_{alpha} will only be able to observe transactions in classes $\bar{I}$ and $\bar{k}$ to have been run in their relative timestamp order.

The implementation of protocol P2 requires an extension to the read condition mechanism. Since the DM is expected to choose a convenient TS_o (cf. P3 where the timestamp is prespecified in the READ message), the timestamping in the read condition cannot be determined until the READ message is processed. So, a named <u>timestamp marker</u> may be supplied in place of a particular timestamp in the read condition. Whenever a DM encounters a timestamp marker in a read condition, it may choose an appropriate time itself, with the proviso that when two or more read conditions are given for a single READ message, all timestamp markers with the same name must be assigned the same timestamp value.

For R^j_{alpha} to run P2 against classes $\bar{I}$ and $\bar{k}$, R^j_{alpha}'s READ message must include two read conditions, <TSM, $\bar{I}$>

and <TSM, $\bar{J}$>, where TSM is a timestamp marker. By satisfying the read conditions, DM_{alpha} fulfills the protocol P2 condition against classes $\bar{I}$ and $\bar{k}$, as desired.

It is interesting to note that protocol P2 is strictly weaker than P3 in the following sense. If R^{j}_{alpha} runs P3 against classes $\bar{I}$ and $\bar{k}$ at DM_{alpha}, then R^{j}_{alpha} satisfies the P2 constraint against $\bar{I}$ and $\bar{k}$ as well. The converse is not true. Since P2 always permits more concurrency than P3, it is always advantageous to run P2 in place of P3 where ever possible.

An example will illustrate the use of protocol P2. Suppose there are two data items of interest, x and y, which reside at both DM_{alpha} and DM_{beta}; initially x=0 and y=0. We assume there is an integrity constraint requiring that $y \leq x^2$. Three classes have been defined. Class $\bar{I}$ runs at TM_{alpha}, reads x from DM_{alpha} and writes x. Class $\bar{J}$ runs at TM_{alpha}, reads x from DM_{alpha} and writes y. Class $\bar{k}$ runs at TM_{beta}, and reads x and y from DM_{beta}. A class conflict graph for this configuration is shown in figure 2.10. Notice that a cycle is present and that transactions in class $\bar{J}$ must run P3 against transactions in class $\bar{I}$ at DM_{alpha} and that transactions in class $\bar{k}$ must run protocol P2 against classes $\bar{I}$ and $\bar{J}$ at DM_{beta}.

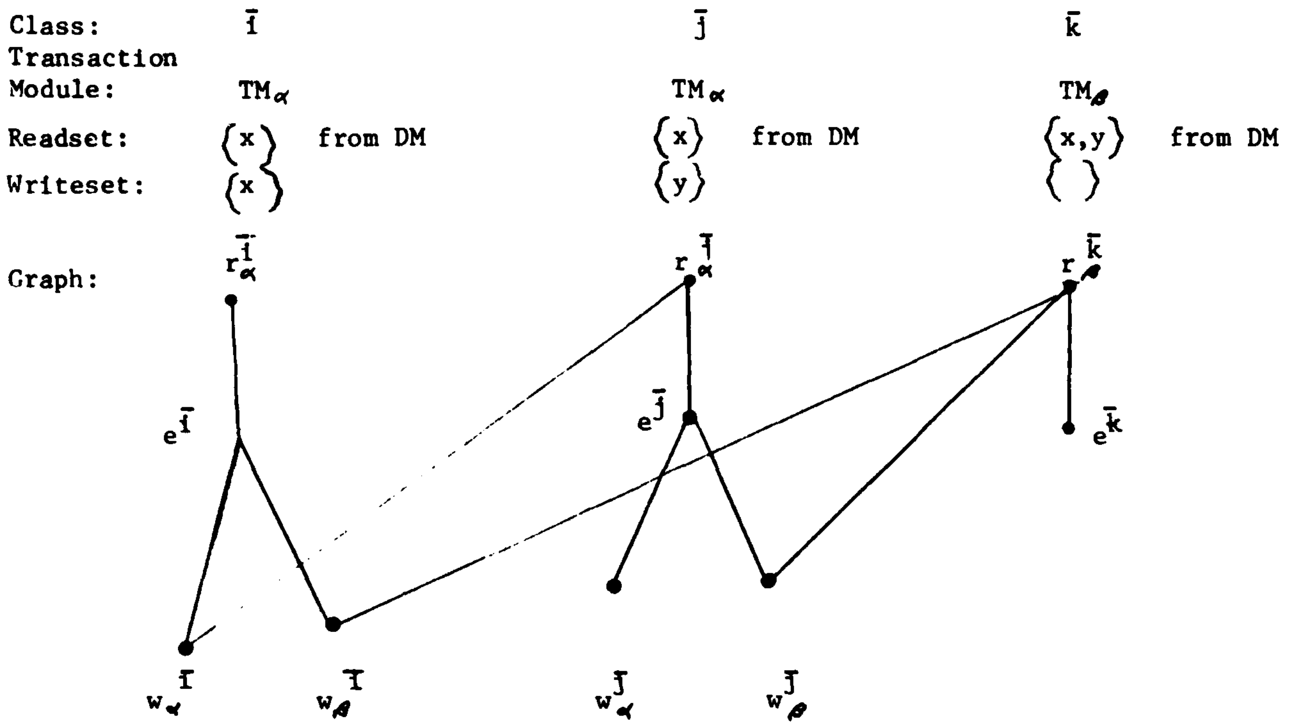


Figure 2.10 - Class Conflict Graph for Example in Section 2.13

Transaction i is received at TM_{alpha}, requests to perform the computation x := x+1, is assigned to class $\bar{i}$, and is given timestamp TS_i. Transaction j is received at TM_{alpha}, requests to perform y := x^2, is assigned to class $\bar{j}$, and is given timestamp $TS_j > TS_i$. Transaction k is received at TM_{beta}, requests to print the values of x and y on the user's terminal, is assigned to class $\bar{k}$, and is given timestamp $TS_k > TS_j$. Notice that each of these transactions preserve the constraint that $y \leq x^2$. No serial ordering of the transactions could invalidate this condition.

First, we consider an anomalous scenario in which transaction k does not run protocol P2 as is required:

1. TM_{alpha} sends a READ message to DM_{alpha} for transaction i and retrieves x=0.

2. TM_{alpha} sends WRITE messages to DM_{alpha} and DM_{beta} for transaction i. Each WRITE message contains timestamp TS_i and the assignment x := 1.

3. DM_{alpha} processes the WRITE for transaction i (but DM_{beta} has not yet done so).

4. TM_{alpha} sends a NULLWRITE message for class $\bar{I}$ with timestamp $TS_{i'} > TS_j$ to DM_{alpha}.

5. TM_{alpha} sends a READ message to DM_{alpha} for transaction j and retrieves x=1. (The P3 read condition on this READ message is immediately satisfied because of the previously received NULLWRITE message.)

6. TM_{alpha} sends WRITE messages to DM_{alpha} and DM_{beta} for transaction j. Each WRITE message contains timestamp TS_j and the assignment y := 1.

7. DM_{alpha} processes j's WRITE message.

8. DM_{beta} processes j's WRITE message.

9. TM_{beta} sends a READ message to DM_{beta} for transaction k, retrieving x=0, y=1.

10. Transaction k prints x=0, y=1 on the user's terminal.

11. DM_{beta} processes the WRITE message from i, thereby setting x=1.

The user has seen an impossible state of the database (i.e., x=0, y=1) printed by transaction k, with $y > x^2$. The problem is that k is reading both the input and output of another transaction, j. However, k is reading the new value of the output but an old value of the input on which that output is based.

If k had run protocol P2 as required, then this situation could not have occurred. By replacing steps (9)-(11) with the following, we obtain a correct scenario in which k satisfies P2.

9. TM_{beta} sends a READ message to DM_{beta} for transaction k. The P2 read condition requires that WRITE's from classes $\bar{I}$ and $\bar{J}$ be processed through some common time. Now $\bar{J}$ has been processed through

time TS_j but class $\bar{I}$ has not been processed through that time yet.

10. DM_{beta} processes the WRITE message for i.

11. A NULLWRITE message arrives at DM_{beta} for class $\bar{I}$ with timestamp $TS_{i'} > TS_j$.

12. DM_{beta} can now process the READ message from k, since WRITE's from both $\bar{I}$ and $\bar{J}$ have been processed through time TS_j. It retrieves x=1, y=1.

13. Transaction k prints x=1, y=1 at the user's terminal.

Notice that it was not necessary for transaction k to use protocol P3 to obtain a correct result. It only had to wait until WRITE's from classes $\bar{I}$ and $\bar{J}$ had been processed through time TS_j, not through time TS_k (its own timestamp).

2.14 Protocol P2f

Protocol P2f is quite similar to protocol P2. It is used in cycles that contain a w-r-e-r-w subpath such as the subpath ($w^{\bar{I}}_{alpha}$, $r^{\bar{J}}_{alpha}$, $e^{\bar{J}}$, $r^{\bar{J}}_{beta}$, $w^{\bar{k}}_{beta}$) shown in Figure 2.11. The "f" in P2f refers to the fact that reading is being done from a <u>foreign</u> DM. As in a P2 subpath, a transaction in class $\bar{j}$ is able to observe an ordering of transactions in classes $\bar{I}$ and $\bar{k}$; protocol P2f is designed to ensure that the observed ordering is always the timestamp ordering of the transactions. If the above subpath is part of a cycle, then each transaction, j, in class $\bar{j}$ must run P2f against $\bar{I}$ at DM_{alpha} and $\bar{k}$ at DM_{beta}. This means that there must be a timestamp, say TS_o, such that all WRITE messages from $\bar{I}$ timestamped before TS_o and none timestamped after TS_o are processed before R^j_{alpha} at DM_{alpha}, and all WRITE messages from $\bar{k}$ timestamped before TS_o and none timestamped after TS_o are processed before R^j_{beta} at DM_{beta}. Protocol P2f essentially runs half of P2 (against $\bar{I}$) at one DM and half of P2 (against $\bar{k}$) at another DM.

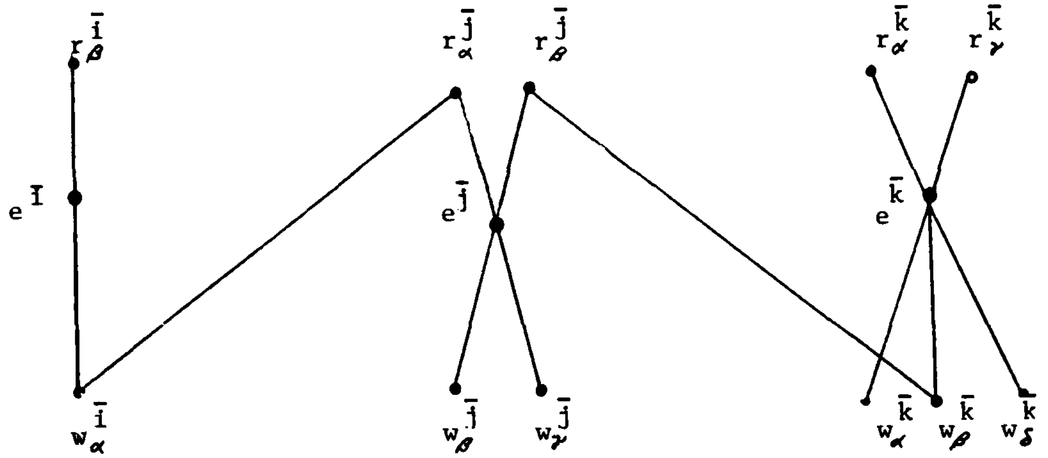


Figure 2.11 - A w-r-e-r-w subpath calls for the use of protocol P2-F when it forms part of a cycle

Since reading is being done from two separate DM's, it is not possible to use the timestamp marker mechanism. (If timestamp markers were used, it would be necessary for the two DM's involved to carry on a conversation to determine a mutually satisfactory timestamp to substitute for the marker. This kind of synchronization overhead is exactly what we are trying to avoid.) Instead, the TM issuing the READ messages <u>chooses</u> a timestamp (i.e., TS_o above) and includes a read condition on each READ with this timestamp. That is, if $\bar{j}$ must run P2f against $\bar{i}$ at DM_{alpha} and $\bar{k}$ at DM_{beta}, then a transaction j in class $\bar{j}$ includes the read condition $<TS_o, \bar{i}>$ in R^{j}_{alpha} and $<TS_o,$

$\bar{k}$> in R^{j}_{beta} for some chosen value of TS_{o}. Unfortunately, choosing a TS_{o} for P2f is not quite as nice as using timestamp markers in P2, because the P2f READ messages have a greater likelihood of being rejected or having to wait. The primary difference between read conditions issued as part of protocol P3 and those issued as part of protocol P2f is that the read condition timestamp for protocol P3 must be the same as the timestamp of the issuing transaction while the read condition timestamp for protocol P2f may have any value.

2.15 Protocol P1

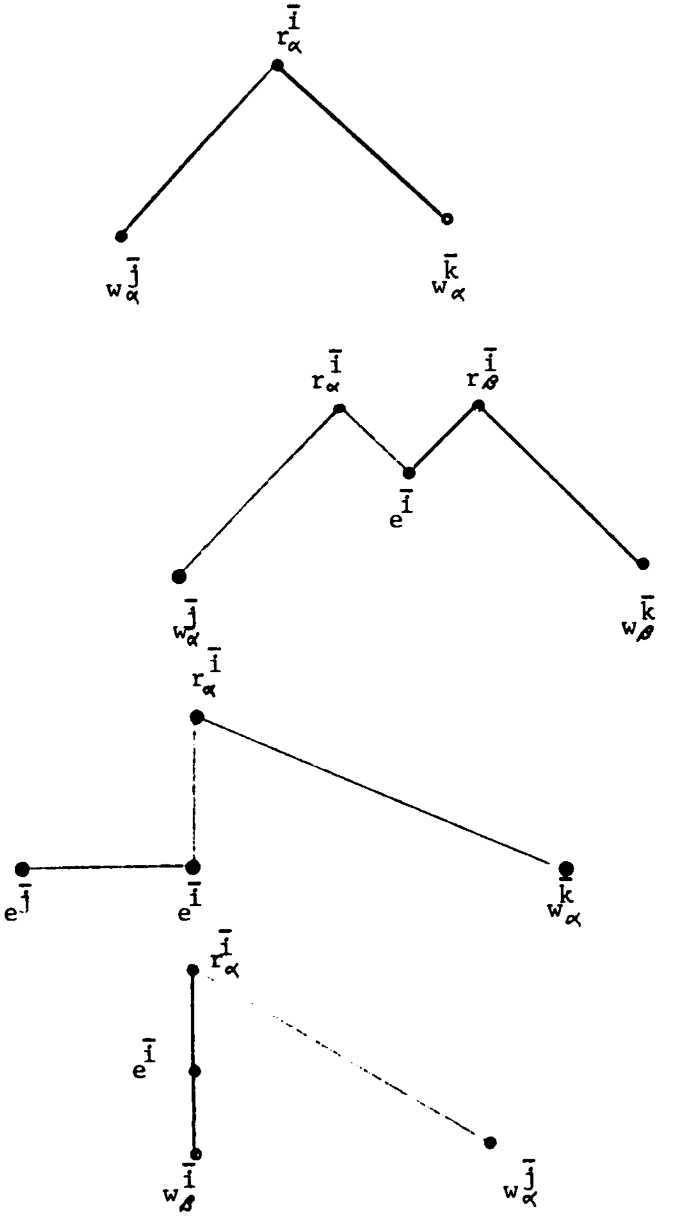
If a transaction class appears in the graph but does not run one of protocols P2, P2f, or P3, then we say it runs protocol P1. That is to say, protocol P1 is the protocol that involves no synchronization other than the data item timestamping rule and the class pipelining rule.

P1, P2, P2f, and P3 provide a graduated set of mechanisms in terms of concurrency and synchronization expense. A goal in designing a particular application is to distribute the data and define the classes to use the lower numbered protocols most frequently.

Graph Topology

(the subpath shown is part of a cycle)

Protocol Requirement



Transactions in class $\bar{i}$ must run protocol P2 with respect to classes $\bar{j}$ and $\bar{k}$.

Transactions in class $\bar{i}$ must run protocol P2-F with respect to classes $\bar{j}$ and $\bar{k}$.

Transactions in class $\bar{i}$ must run protocol P3 with respect to class $\bar{k}$.

Transactions in class $\bar{i}$ must run protocol P3 with respect to class $\bar{j}$.

Figure 2.12 - Protocol requirements are suggested by the graph topology

2.16 Pre-Analysis of the Class Conflict Graph

Figure 2.12 summarizes the results so far, illustrating how particular graph topologies indicate that particular protocols must be run.

If it were necessary to compute graph edges and cycles before executing each transaction, the cost of doing so would clearly be prohibitive. Fortunately, this is not necessary. The class definitions are specified by a DBA at application design time and at that time the class conflict graph can be computed and analyzed. The result of such an analysis will be a list of read conditions for each class. Note that a class may have more than one or two read conditions which it must use. This is because the class may be a part of several cycles.

When a transaction is entered at a TM, the TM first determines its read-set and write-set. It then determines to which class that transaction belongs (if the transaction can run in more than one class, the class with the fewest synchronization requirements is chosen). Having identified the transaction's class, only a table lookup is required to determine what read conditions the transaction must use.

2.17 Safe Cycles

It happens that there are graph cycles which never cause a non-serializable interleaving of transactions. In particular, any cycle which does not contain a vertical edge is always safe. Thus, a cycle composed entirely of diagonal edges or entirely of horizontal edges will never lead to a serializability problem and classes lying on such cycles can safely run P1 (at least insofar as the safe cycles are concerned). The cycle shown in Figure 2.13 is an example of a safe cycle.

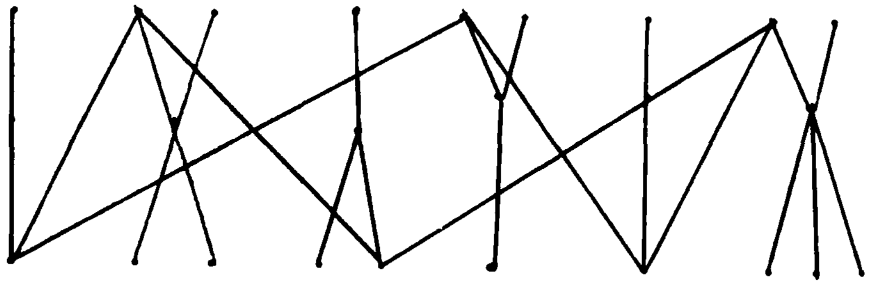


Figure 2.13 - A cycle is safe if it contains no vertical edges

This result is not immediately apparent through intuitive understanding and is illustrative of the fact that a more formal and precise treatment of serializability criteria is needed.

(Some intuitive understanding can be gained, however, through the following arguments. First, if the cycle consists entirely of horizontal edges then a serializability problem cannot arise because horizontal edges always imply a timestamp ordering of the transactions. Second, if the cycle consists entirely of diagonal edges then all the nodes on the cycle have the same DM subscript. Also such a cycle consists of a series of W-R-W subpaths. Remember from the discussion of protocol P2 that on such a subpath the reading transaction may observe a particular ordering of the writing transactions and that the observed ordering depends on the actual order in which the WRITEs were processed by the DM. Since all of the WRITEs on the cycle are being processed by the same DM, it must be the case that the reading transactions all observe the same relative ordering among the writing transactions and hence all transactions on the cycle will be serializable.)

2.18 Summary and Conclusions

In reviewing the concepts presented in section 2, it is helpful to distinguish between three kinds of properties of an SDD-1 system:

1. properties that are intrinsic to the way the SDD-1 software operates;

2. properties that arise from database design decisions.

3. properties that arise from the analysis of the database design.

In category (1) are the way data modules process READ messages and WRITE messages, the way clocks operate, the pipelining rules, and the way each protocol works. In category (2) are the choice of the location of SDD-1 sites on the network, the choice of logical fragments, the location of physical fragments, the configuration of materializations, the choice of read-sets and write-sets for each class, and the assignment of materializations to each class. Finally, in category (3) is the assignment of protocols to each class.

The description we have presented of the SDD-1 redundant update mechanism has a serious defect. We have shown that certain situations cause serializability problems and have introduced mechanisms to resolve those problems. Yet how can we be sure that we have identified all possible dangerous situations? And how can we be sure that the protocols prevent all possible instances of these dangerous situations?

We believe that in order to fully understand these results, to be confident of their correctness, and to use them intelligently in designing systems, we must prove their correctness in a precise and formal manner. This is the purpose of the Sections 3 and 4.

3. Selection and Analysis of Protocols

3.1 Logs

To develop the criteria for selecting a protocol for each class, we need a formal model for transaction processing. The model we have chosen, called logs, consists of a string of symbols that represents the execution of transactions, READ messages, and WRITE messages. Our claim will be that logs embody all of the information about system execution that is needed to reproduce its input-output behavior. Verifying this claim will permit us to use logs as a formal model for investigating other aspects of the behavior of SDD-1.

There are three kinds of events that are of interest for building logs : READ messages, WRITE messages, and local transaction execution. We represent the processing of a READ message for a transaction, a, at a data module, alpha, by R^{a}_{alpha}. We represent the processing of a WRITE message for a transaction, a, at a data module, alpha, by

W^{a}_{alpha}. Finally, we represent the local execution of a transaction (in its transaction module), a, by E^{a}. In the sequel, we will use lower case Roman letters near the beginning of the alphabet to represent transactions, and lower case Greek letters near the beginning of the alphabet to represent data modules.

The behavior of each data module is modelled as a string of R's and W's, which represents the order in which READ messages and WRITE messages were processed (as opposed to received) by the data module. We call such a string a local data module log. Each local data module log must obey certain syntactic constraints that represent physical properties that data modules must satisfy. In a local data module log, say for data module alpha, the following must hold:

D1. All R's and W's must have the same subscript, alpha, since they are all processed at data module alpha.

D2. For each transaction, a, at most one R^{a}_{alpha} and one W^{a}_{alpha} can appear, since each transaction can send at most one READ and one WRITE message to any given data module.

The behavior of classes is modelled as a string of E's called a global transaction log, which represents the order in which transactions were executed as reflected by their timestamps. The only syntactic constraint on a global transaction log is

E1. For each transaction, a, only one E^a appears, since a transaction receives only one timestamp.

A global transaction log induces certain additional syntactic restrictions on a local data module log, which indicate the proper orderings based on the pipelining rules. In a local data module log, say for data module alpha, the following must hold: if transaction a and transaction a' run in the same class and E^a ,recedes $E^{a'}$ in the global transaction log, then

D3. (R-R pipelining) If R^a_{alpha} and $R^{a'}_{alpha}$ appear, then R^a_{alpha} precedes $R^{a'}_{alpha}$;

D4. (W-W pipelining) If W^a_{alpha} and $W^{a'}_{alpha}$ appear, then W^a_{alpha} precedes $W^{a'}_{alpha}$;

D5. (W-R pipelining) If W^a_{alpha} and $R^{a'}_{alpha}$ appear, then W^a_{alpha} precedes $R^{a'}_{alpha}$.

A log models an execution history of transactions on the database.

To obtain a complete picture of the effect that logs have on the database, we require the following additional information, relating to database design:

for each transaction - the read-set of the transaction, the write-set of the transaction, and the class in which the transaction ran;

for each data module - the set of physical fragments that is stored there; and

for each class - the materialization it uses for reading.

For the sake of economy of the model and to enhance mathematical tractability, we will normally leave the transactions uninterpreted (in the sense of the program schema theory [Manna]). That is, for each logical data item in the write-set of each transaction, we associate a unique uninterpreted function letter that maps all of the read-set into that write-set data item.

Given the above database design information, we must add two more syntactic constraints on local data module logs that guarantee that all of the relevant READ and WRITE messages are actually issued. If E^a appears in the global transaction log, then

1. If some data item in the read-set of transaction a is obtained by the materialization of the class under which transaction a runs from data module alpha, then R^a_{alpha} appears in alpha's local data module log.

2. If some data in the write-set of transaction a is stored at data module alpha, then W^a_{alpha} appears in alpha's local data module log.

In addition to these syntactic constraints, there is the obvious semantic constraint that the logs accurately represent the order in which R's and W's (in the case of a local data module logs) or E's (in a global transaction log) actually were processed.

Suppose we have a global transaction log and a collection of local data module logs that represent the execution of the system during some period. These logs can be merged into a single <u>global system log</u> by satisfying the following conditions:

G1. All symbols in the global transaction log appear in the global system log and appear in the same order (e.g., if E^a precedes E^b in the global transaction log, then E^a and E^b appear in the global system log and E^a precedes E^b).

G2. For each local data module log, all symbols in the local log appear in the global system log and appear in the same order.

G3. For each transaction, a, and for each data module, alpha, if R^a_{alpha} appears in the global system log then E^a also appears in the global system log and R^a_{alpha} precedes E^a.

G4. For each transaction, a, and for each data module, alpha, if W^a_{alpha} appears in the global system log then E^a also appears in the global system log and E^a precedes W^a_{alpha}.

Given a global log and its associated database design information, we would like to show that this model is sufficiently powerful to reproduce the essential aspects of SDD-1 operation.

Claim C The log model of SDD-1 operation is complete in the sense that given an initial value for all data items in the database, a log, and an interpretation of the function symbols for transactions, then there is a mechanical procedure that could analyze the log and reproduce the exact value history of each stored data item at each data module.

The essence of claim of C is that timestamping information for transactions and the parameters of READ and WRITE messages are not needed in order to duplicate the actual operation of the system, given that the log and associated transaction and data distribution information is provided. To prove this claim formally, we would need a formal model for the operation of SDD-1 (at the level, say, of a RAM or Turing machine) and a formal model of logs. Then we would need to show an isomorphism between the value histories of all stored data items of each model. We will not perform this tedious task. Rather we will demonstrate an interpreter that can simulate SDD-1's operation with only the information available in logs and the associated transaction and data distribution information. We argue along intuitive lines only that the interpreter is indeed simulating correctly.

The interpreter maintains a simulated physical copy of each stored data item in each data module. Instead of storing a timestamp with each stored data item, we associate the transaction name of the last transaction that successfully wrote into that data item. Given the total ordering of E's in the global system log, this "transaction label" will be sufficient to reproduce all of the essential timestamping information in the system.

Given the database design information, we can obtain the read-set and write-set associated with each R and W in the log. We also assume that for each uninterpreted function letter in a transaction there is an interpretation (i.e. a program).

Now, to execute a global system log, the interpreter begins by initializing all stored data items to their initial state and their associated transaction labels to NULL. It then selects log symbols, one at a time proceeding from left to right; for each symbol it does the following:

i. If the symbol is a read, say R^{a}_{alpha}, then read that portion of the read-set of transaction a that is stored at data module alpha according to the materialization of the class in which transaction a executes. Store these values in a temporary work space associated with transaction a.

ii. If the symbol is an E, say E^a, execute the interpretation of transaction a on the read-set values stored in its workspace. The resulting write-set values should be stored back into its workspace.

iii. If the symbol is a write, say W^a_{alpha}, then for each data item in the write-set of transaction a that also is stored at data module alpha, take the value of the data item and store it in the stored data item at alpha with transaction label = a if and only if one of the following holds:

1. the transaction label for the data item at alpha is NULL; or

2. the transaction label for the data item at alpha is some b where E^b precedes E^a in the global system log.

First, notice that the parameters (i.e. conditions) of read messages are not needed, in that the global system log already specifies exactly which WRITE messages are processed ahead of each READ message. Second, the conditions for performing WRITE messages are exactly those induced by the timestamping rules. The use of ordered E's in the log to embody timestamping information is a crucial conceptual simplification that makes the proofs in later sections possible. Were we forced to use actual timestamps instead, the notation would be much more difficult to understand and manipulate.

3.2 Correctness Criteria

To determine how to assign protocols to classes to yield correct system operation, we must first develop precise conditions for correct system operation. We define two conditions that characterize the correctness of distributed database systems such as SDD-1. One condition, called convergence, states that all copies of each logical data item must be "converging" toward the same value. The other condition, called serial reproducibility, essentially states that the values toward which the database is converging are mutually consistent. We proceed more formally with a discussion of each of these criteria.

3.2.1 Convergence

A *log is convergent* if, given a database state in which all stored copies of each data item are equivalent, then the log transforms that state into another state with the same property. (In the sequel, we use "log" to mean "global system log".) A *system is convergent* if all of the logs it can generate are convergent. One way to look at system convergence is to imagine that if the processing of E's were to stop at any time and all WRITE messages for completed E's were processed, then the resulting log would be convergent.

Theorem CONV Let L be a log generated by SDD-1. If for each E in L all of E's WRITE messages are in L, then L is convergent.

Proof Consider an arbitrary logical data item, x, and let E^a be the last transaction execution which has x in its write-set. Since all write messages for transaction a are eventually processed (by hypothesis), for each data module, alpha, that has a stored copy of x, W^a_{alpha} will be the last WRITE message in L that successfully updates x. Hence, all copies of x will be equivalent. Q.E.D.

Corollary SDD-1 is convergent.

3.2.2 Serial Reproducibility

We define two logs, L1 and L2, to be equivalent if for all initial database states and for all interpretations of the transactions, L1 and L2 leave the database in the same final state. In a log L, we say that a READ message R^{a}_{alpha} reads from a message W^{b}_{alpha} if

i. There is a stored data item x at alpha that is in the read-set of a and the write-set of b; and

ii. W^{b}_{alpha} precedes R^{a}_{alpha} in L; and

iii. W^{b}_{alpha} successfully updates x when it is processed (i.e., E^{b} appears later in L than x's current transaction label when W^{b}_{alpha} is processed); and

iv. There is no c such that W^{c}_{alpha} follows W^{b}_{alpha}, W^{c}_{alpha} and precedes R^{a}_{alpha} in L, and W successfully writes into x (i.e., W^{b}_{apha} is the last write operation into x before R^{a}_{apha}).

The notion of "reading from" characterizes log equivalence in the following sense.

Theorem E Let L1 and L2 be logs that contain the same set of transactions. If every R reads each of its data items from the same W in both L1 and L2, then L1 is equivalent to L2.

Proof The proof uses Herbrand interpretations to show that each data item displays the same final value in both logs. This is a standard program schema theoretic result and can be found (for example) in [MANNA].

Theorem E can be extended to be both a necessary and sufficient condition for equivalence by incorporating the notion of "deadness" as in [Papadimitriou et al]. However, for later results, we only need the sufficient condition for equivalence.

We define a log to be *serial* if for each transaction a in the log, all R^a symbols *immediately* precede E^a and all W^a symbols *immediately* follow E^a. That is, a serial log is of the form:

$$R^a_{\text{alpha}} \cdots R^a_{\text{omega}} E^a W^a_{\text{alpha}} \cdots W^a_{\text{omega}} R^b_{\text{alpha}} \cdots R^b_{\text{omega}} E^b W^b_{\text{alpha}} \cdots W^b_{\text{omega}} \; R^c_{\text{alpha}} \cdots R^c_{\text{omega}} \; E^c \; W^c_{\text{alpha}} \cdots W^c_{\text{omega}} \cdots$$

A *log is serially reproducible* if it is equivalent to a serial log. A *system is safe* if all of the logs it can generate are serially reproducible. The use of serial reproducibility as a correctness criterion has been used by many researchers [ESWARN et al], [GRAY et al], and [HEWITT] and arises from the following model. Our goal is to show that the database is maintained in a "consistent" state, where "consistency" is characterized, say, by a predicate which is true for all consistent states. We assume that every transaction preserves the consistency of the database: given a copy of its read-set that is consistent then it will produce a copy of its write-set that is also consistent. Clearly, every serial log preserves database consistency if each of its transactions preserves database consistency; in this case, all data items are updated cosynchronously, because all WRITE messages of a transaction are processed before the next READ message is processed. Since a serially reproducible log is equivalent to a serial log, serially reproducible logs preserve consistency as well.

SDD-1 guarantees serial reproducibility by the rules that govern the selection of protocols for classes. That is, if every class executes all of its transactions according to the prespecified protocols, then the log of all transactions executed by all classes is serially reproducible. In the remainder of Section 3 we will develop these protocol selection rules. In Section 4 we will prove that they do in fact make SDD-1 logs serially reproducible.

3.3 Log Transformations

To determine if a log is serially reproducible, we will define an effective procedure to transform a log into an equivalent serial one. The procedure is based on equivalence preserving transformations on logs. These transformations are in the form of "switching rules", i.e., equivalence preserving rules for switching adjacent log symbols. Each of the following switching rules is of the form "... x1 x2 ... $\equiv$ ... x2 x1 ... under condition C", which means that if symbols x1 and x2 are adjacent in a log and they satisfy condition C, then they can be switched and the resulting log is equivalent to the log before the switch.

TR1. ... R^{a}_{alpha} R^{b}_{beta} ... $\equiv$ ... R^{b}_{beta} R^{a}_{alpha} ... where a and b run in different classes

TR2. ... R^{a}_{alpha} R^{b}_{beta} ... $\equiv$ ... R^{b}_{beta} R^{a}_{alpha} ... where alpha $\neq$ beta

TR3. $\ldots E^{a} E^{b} \ldots \equiv \ldots E^{b} E^{a} \ldots$ where a and b run in different classes and have nonintersecting write-sets.

TR4. $\ldots W^{a}_{\mathrm{alpha}} W^{b}_{\mathrm{alpha}} \ldots \equiv \ldots W^{b}_{\mathrm{alpha}} W^{a}_{\mathrm{alpha}}$ ... if a and b run in different classes

TR5. $\ldots W^{a}_{\mathrm{alpha}} W^{b}_{\mathrm{beta}} \ldots \equiv \ldots W^{b}_{\mathrm{beta}} W^{a}_{\mathrm{alpha}} \ldots$ if alpha $\neq$ beta

TR6. $\ldots R^{a}_{\mathrm{alpha}} W^{b}_{\mathrm{beta}} \ldots \equiv \ldots W^{b}_{\mathrm{beta}} R^{a}_{\mathrm{alpha}} \ldots$ if alpha $\neq$ beta

TR7. $\ldots R^{a}_{\mathrm{alpha}} W^{b}_{\mathrm{alpha}} \ldots \equiv \ldots W^{b}_{\mathrm{alpha}} R^{a}_{\mathrm{alpha}}$ ... if a and b run in different classes and there is no stored data item at alpha that is common to transaction a's read-set and transaction b's write-set.

Theorem TR The transformations TR1 - TR7 are sound, i.e., they preserve log equivalence.

Proof Follows directly from theorem E and the definitions of the transformations. Q.E.D.

We note in passing that the transformations TR1 - TR7 are in no sense complete with respect to equivalence. That is, given tvo equivalent logs, L1 and L2, there may be no sequence of applications of TR1-TR7 to L1 that yields L2. There are several reasons for this. First, all of the transformations presc~ve the pipelining rules in addition to equivalence, which thereby weakens them. Second, the transformations preserve certain timing information, which in some cases is not needed to preserve equivalence. Finally, pairwise switching is not sufficient to handle all equivalence situations; logs can be constructed which have entire sublogs that can be switched in an equivalence preserving way, such that no sequence of pairwise switches can reproduce the sublog switch. These observations are parenthetical to the results that follow, since the soundness of TR1 - TR7 is all that is required.

3.4 Conflict Graphs

From TR1 - TR7 we can derive the set of invalid switches, i.e., those switches that are not permitted by TR1 - TR7. These invalid switches correspond to potential conflicts between transactions and, as we will see, can lead to non-serially reproducible logs. The invalid switches, called conflicts, are:

NTR1. ... R^{a}_{alpha} R^{b}_{alpha} ... where a and b run in the same class.

NTR2. ... W^{a}_{alpha} W^{b}_{alpha} ... where a and b run in the same class.

NTR3. ... R^{a}_{alpha} W^{b}_{alpha} ... or ... W^{b}_{alpha} R^{a}_{alpha} ... where either a and b run in the same class or there is a stored data item at alpha that is common to transaction a's read-set and transaction b's write-set.

NTR4. ... R^{a}_{alpha} E^{a} ...

NTR5. ... E^{a} W^{a}_{alpha} ...

NTR6. ... E^{a} E^{b} where a and b run in the same class or have intersecting write-sets.

It is easily checked that these are the only pairs that cannot be switched using TR1 - TR7.

The above conflicts can be modelled by a node-labelled undirected graph whose nodes represent generic log symbols and whose edges represent potential conflicts between log symbols. The graph is defined over a finite set of classes, denoted $\{\bar{a},\bar{b},\bar{c},\ldots\bar{z}\}$, and associated with each class is a read-set, a write-set, and a materialization.

We define a conflict graph CG = <V,E> as follows (it denotes set union):

$V = \{e^{\bar{a}}$: all classes $\bar{a}\} + \{r^{\bar{a}}_{alpha}$: all classes $\bar{a}$ and all data modules alpha$\} + \{w^{\bar{a}}_{lpha}$: all classes $\bar{a}$ and all data modules alpha$\}$

$E = E_{vert} + E_{horiz} + E_{diag}$

$E_{vert} = \{(r^{\bar{a}}_{alpha}, e^{\bar{a}})$: all classes $\bar{a}$ and all data modules alpha$\}$ + $\{(e^{\bar{a}}, w^{\bar{a}}_{alpha})$: all classes $\bar{a}$ and all data modules alpha$\}$

$E_{horiz} = \{(e^{\bar{a}}, e^{\bar{b}})$: all classes $\bar{a},\bar{b}$ where the write-sets of $\bar{a}$ and $\bar{b}$ have a nonempty intersection$\}$

$E_{diag} = \{(r^{\bar{a}}_{alpha}, w^{\bar{b}}_{alpha})$: all classes $\bar{a},\bar{b}$ and all data modules alpha such that the portion of $\bar{b}$'s write-set stored at alpha has a nonempty intersection with the portion of $\bar{a}$'s read-set which is stored at alpha under $\bar{a}$'s materialization$\}$

The notions of vertical, horizontal and diagonal edges derive from the following convention for drawing conflict graphs. For each class $\bar{a}$, we draw all of $\bar{a}$'s r nodes in a row, beneath which we uraw $\bar{a}$'s e node, beneath which we draw $\bar{a}$'s w nodes in a row. (See figure 2.2.) The E_{vert} edges connect each e to all of its r's and w's; these edges are (in a manner of speaking) vertica. Groups of nodes for different classes are arranged in a row (see figure 2.3). The E_{horiz} edges connecting e's in different classes are therefore horizontal, and the E_{diag} edges connecting an R and W from different classes are diagonal. We have found these conventions to be very convenient when discussing conflict graphs.

3.5 Protocol Selection Rules

A conflict graph cycle that contains a vertical edge can lead to a nonserializable log, because the edges of the cycle can correspond to conflicting (and hence unswitchable) symbols in the log. The rules for selecting which protocols to use for each READ message in each class are built around cycles in the conflict graph. We conclude Section 3 by enumerating these rules. In Section 4 we prove that if all transactions obey these rules, then all logs are serially reproducible. The rules are:

PSR3. If $r^{\bar{a}}_{\text{alpha}}$ lies on a cycle in the conflict graph and the cycle contains the subpath ($w^{\bar{b}}_{\text{alpha}}$, $r^{\bar{a}}_{\text{alpha}}$, $e^{\bar{a}}$, $w^{\bar{a}}_{\text{beta}}$) or the sub ($w^{\bar{b}}_{\text{alpha}}$, $r^{\bar{a}}_{\text{alpha}}$, $e^{\bar{a}}$, $e^{\bar{c}}$) for some classes $\bar{b}$ and $\bar{c}$ and some data module beta, then for each transaction a in $\bar{a}$, run R^{a}_{alpha} under protocol P3 with respect to $\bar{b}$.

PSR2F. If $r^{\bar{a}}_{\text{alpha}}$ and $r^{\bar{a}}_{\text{beta}}$ lie on a cycle in the conflict graph and the cycle contains the subpath ($w^{\bar{b}}_{\text{beta}}$, $r^{\bar{a}}_{\text{beta}}$, e^{a}, $r^{\bar{a}}_{\text{alpha}}$, $w^{\bar{c}}_{\text{alpha}}$) for some classes $\bar{b}$ and $\bar{c}$, then for each transaction a in $\bar{a}$, run R^{a}_{alpha} and R^{a}_{beta} under protocol schema P2F against $\bar{b}$ at beta and against $\bar{c}$ at alpha.

PSR2. If $r^{\bar{a}}_{alpha}$ lies on a cycle in the conflict graph and the cycle both contains a vertical edge and contains the subpath ($w^{\bar{b}}_{alpha}$, $r^{\bar{a}}_{alpha}$, $w^{\bar{c}}_{alpha}$) for some $\bar{b}$ and $\bar{c}$, then for each transaction a in $\bar{a}$, run R^{a}_{alpha} under protocol P2 against $\bar{b}$ and $\bar{c}$ at alpha.

These protocols must be satisfied for all cycles in the conflict graph. That is, if an r lies on several cycles and thereby satisfies several of the PSRs, then that READ message must include conditions to satisfy all of its PSRs. If an r satisfies none of the above PSRs, either because it lies on no cycles or because none of the cycles on which it lies have the undesirable properties, then that r can run protocol P1. It is expected that under a suitable database design and for many applications, most transactions need only run under protocol schema P1.

Theorem SR If all of the transactions in a log use the correct protocol as outlined by the protocol selection rules, then the log is serially reproducible.

Proof See Section 4.

Corollary SDD-1 is safe.

4. Proof of Serial Reproducibility

4.1 Introduction

This section contains a proof of theorem SR, which demonstrates that the SDD-1 protocol selection rules lead to serially reproducible logs. Since the proof is rather long and its details may not be of interest to all readers, we will first present a brief overview of the proof. To prove the theorem formally, we need to formalize the concepts of the previous sections. This formalism is presented in Section 4.2. The proof itself comes in two parts and is presented in Sections 4.3 and 4.4.

This proof only includes protocols P1, P2, P2f, and P3. A proof that also embodies protocol P4 has been produced and will appear in a later report.

To prove that all logs are serially reproducible, we assume the converse and show a contradiction. That is, we assume that there is some log, say LOG_{given}, which resulted from the correct operation of SDD-1 and that LOG_{given} is not serially reproducible. The general approach we will take is to try to serialize LOG_{given} using the transformations TR1 - TR7. When we get stuck, as we must since LOG_{given} ıs not serially reproducible, we examine the "stuck" log and derive from the log certain properties of the conflict graph that demonstrate that LOG_{given} must have violated the protocol selection rules (PSRs). Thus, the proof proceeds in two stages: first, the attempt to serialize LOG_{given}; second, the construction of the PSR contradiction.

To serialize LOG_{given}, we begin at the left end of the log and try to serialize each R so that it is adjacent to its corresponding E and each W so that it is adjacent to its corresponding E. Suppose, for example, that we are trying to serialize R^a_{alpha} to be adjacent to E^a. By applying switches permitted by TR1 - TR7 of adjacent symbols in the sublog that separate R^a_{alpha} from E^a, we try to move R^a_{alpha} closer to E^a. That is, we try to move each symbol in this sublog either to the left of R^a_{alpha} or to the right of E^a. If we can move all of the symbols in this sublog out of the way, then we will end up with R^a_{alpha} adjacent to E^a. We can apply essentially the same procedure to move each W^a to be adjacent to E^a.

Since LOG_{given} is not serially reproducible, this procedure that tries to serialize LOG_{given} will eventually fail to be able to serialize some R or W. Suppose that R^{a}_{alpha} cannot be serialized with E^{a}. Then we have a sublog of the form $R^{a}_{alpha} \ldots E^{a}$ in which every intermediate symbol is in conflict with some symbol both on its right and on its left, since otherwise the symbol would have been removed by the above applications of TR1 - TR7. Similarly, had we gotten stuck by a W^{a}_{alpha}, we would have obtained a sublog $E^{a} \ldots W^{a}_{alpha}$ with the same property. Finding this blocked sublog completes the first stage of the proof.

Suppose, again, that the blocked sublog is $R^{a}_{alpha} \ldots E^{a}$. The second stage of the proof begins with the observation that since every symbol in the sublog conflicts with some symbol on its left and right in the sublog and since every conflict corresponds to an edge in the conflict graph, then there is a path from R^{a}_{alpha} to E^{a} in the conflict graph. Furthermore, we know that the edge (R^{a}_{alpha}, E^{a}) is in E_{vert}, hence completing a cycle. Since R^{a}_{alpha} lies on a cycle, it is subject to the PSRs. By analyzing the blocked sublog in more detail, enumerating the possible symbols that could be R^{a}_{alpha}'s conflicting right neighbor and those that could be E^{a}'s conflicting left neighbor, we show that in each and every case either R^{a}_{alpha} violated a protocol it was supposed to use according to the PSRs or that LOG_{given} must have violated one of the pipelining rules. The conclusion, then, must be that this blocked sublog could not have arisen in the process of trying to serialize LOG_{given}. The very same kind of argument can be applied if the blocked sublog E^a ... W^a_{alpha} had resulted from stage one. So, the attempt to serialize must inevitably succeed and LOG_{given} is serially reproducible.

There are numerous pitfalls in this line of proof that require a rigorous approach to be taken. We proceed, now, with this rigorous development.

4.2 A Formal Model for SDD-1

4.2.1 A Database Design

A database design for SDD-1 is a ten-tuple

D = <DELTA, KAPPA, LAMBDA, SIGMA, MATZN, logical,
matzn-of-class, stored-data, readset, writeset>

where the components of D are defined as follows (upper case components are sets and lower case components are functions):

1. DELTA = {alpha, beta, gamma, delta,...} is the set of all data modules.

2. KAPPA = {$\bar{a}$,$\bar{b}$,$\bar{c}$,...} is the set of all classes.

3. LAMBDA is the set of all logical fragments.

4. SIGMA is the set of all stored fragments.

5. logical: SIGMA -> LAMBDA. Each stored fragment sigma in SIGMA is a physical incarnation of some logical fragment specified by logical(sigma).

6. MATZN = $\{matzn_1, matzn_2, matzn_3, \ldots\}$ is the set of all materializations. Each materialization is a total function and $matzn_i$: LAMBDA -> SIGMA such that for each lambda in LAMBDA, $logical(matzn_i(lambda)) = lambda$.

7. matzn-of-class: KAPPA -> MATZN. Each class $\bar{a}$ in KAPPA runs in some materialization, specified by matzn-of-class($\bar{a}$).

8. stored-data: SIGMA -> DELTA. Each stored fragment sigma in SIGMA is stored at a data module, specified by stored-data(sigma).

9. readset: KAPPA -> 2^{LAMBDA}. Each class $\bar{a}$ in KAPPA has a readset that is a subset of LAMBDA, specified by readset($\bar{a}$).

10. writeset: KAPPA -> 2^{LAMBDA}. Each class $\bar{a}$ in KAPPA has a writeset that is a subset of LAMBDA, specified by writeset($\bar{a}$).

When designing a database, one has to specify data distribution and class structure by specifying each of the above ten components.

4.2.2 Logs

The execution of the system is completely characterized by a log. Logs are built on transactions. We define a transaction set over a database design D to be a four-tuple TAU(D) = <TN, transclass, transreadset, transwriteset> where the components of TAU are:

1. TN = {a,b,c,d,...} is a set of transaction names.

2. transclass: TN -> KAPPA. Each transaction a in TN runs in a single class specified by transclass(a).

3. transreadset: TN -> 2^{LAMBDA}. Each transaction a in TN has a readset that is a subset of LAMBDA, specified by transreadset(a), such that transreadset(a) is contained in readset(transclass(a)).

4. transwriteset: TN -> 2^{LAMBDA}. Each transaction a in TN has a writeset that is a subset of LAMBDA, specified by transwriteset(a), such that transwriteset(a) is contained in writeset(transclass(a)).

A log is a string defined over a database design D and a transaction set TAU. The symbols of a log, L, are selected from the set $\bar{R}$ +$\bar{E}$ +$\bar{W}$ ('+' is set union) where

$\bar{R}$ = {R^{a}_{alpha} : all a in TN, all alpha in DELTA}
$\bar{E}$ = {E^{a} : all a in TN}
$\bar{W}$ = {W^{a}_{alpha} : all a in TN, all alpha in DELTA}.

A <u>well-formed</u> log, L, satisfies the following restrictions:

1. No element of $\bar{R}$ + $\bar{E}$ + $\bar{W}$ appears more than once in L.

2. For each a in TN, if E^{a} appears in L then for all alpha in DELTA:

   i. if matzn-of-class(transclass(a))(readset(transclass(a)) has a non-empty intersection with stored-data^{-1}(alpha), then R^{a}_{alpha} appears in L and precedes E^{a};* and

ii. if transwriteset(a) has a nonempty intersection with stored-data^{-1}(alpha), then W^{a}_{alpha} appears in L and E^{a} precedes W^{a}_{alpha}. (Note: by "precedes" we mean "appears somewhere in the string to the left of".)

A well-formed log, L, <u>satisfies the pipelining rules</u> if for each a and a' where E^{a} precedes $E^{a'}$ in L and transclass(a) = transclass(a') then

1. (R-R rule) for each alpha in DELTA where R^{a}_{alpha} and $R^{a'}_{alpha}$ are both in L, R^{a}_{alpha} precedes $R^{a'}_{alpha}$;

2. (W-W rule) for each alpha in DELTA where W^{a}_{alpha} and $W^{a'}_{alpha}$ are both in L, W^{a}_{alpha} precedes $W^{a'}_{alpha}$;

3. (W-R rule) for each alpha in DELTA where W^{a}_{alpha} and $R^{a'}_{alpha}$ are both in L, W^{a}_{alpha} precedes $R^{a'}_{alpha}$.

---

* This definition implies a READ message is sent to alpha if the materialization obtains part of the class read-set from alpha, even if the particular transaction does not read any data from alpha. In the implementation of SDD-1, read conditions make it possible to avoid sending the READ messages in the latter case, by adding extra read conditions to the next READ message that goes to alpha from transclass(a).

A system is well-formed and satisfies the pipelining rules if all of the logs it can generate have these properties. Unless explicitly stated otherwise, in the sequel we assume that all logs are well-formed and satisfy the pipelining rules.

4.2.3 Conflict Graphs

We redefine conflict graphs, the protocols, and the protocol selection rules in terms of the above formalism.

A conflict graph

$$CG(D) = \langle \text{VERTICES, EDGES} \rangle$$

is a vertex-labelled undirected graph defined over a database design D as follows:

$$\begin{aligned}\text{VERTICES} = {} & \{r^{\bar{a}}_{\text{alpha}}\text{: all } \bar{a} \text{ in KAPPA, all alpha in DELTA}\} + \\ & \{e^{\bar{a}}\text{: all } \bar{a} \text{ in KAPPA}\} + \\ & \{w^{\bar{a}}_{\text{alpha}}\text{: all } \bar{a} \text{ in KAPPA, all alpha in DELTA}\}\end{aligned}$$

$$\text{EDGES} = \text{EDGES}_{\text{vert}} + \text{EDGES}_{\text{horiz}} + \text{EDGES}_{\text{diag}}$$

$$\begin{aligned}\text{EDGES}_{\text{vert}} = {} & \\ & \{(r^{\bar{a}}_{\text{alpha}}, e^{\bar{a}})\text{: all } \bar{a} \text{ in KAPPA, all alpha in DELTA}\} \\ & + \{e^{\bar{a}}, w^{a}_{\text{alpha}})\text{: all } \bar{a} \text{ in KAPPA, all alpha in DELTA}\}\end{aligned}$$

$\text{EDGES}_{\text{horiz}} = \{(e^{\bar{a}}, e^{\bar{b}}) : \bar{a} \neq \bar{b}$ and the intersection of writeset($\bar{a}$) and writeset($\bar{b}$) is nonempty$\}$

$\text{EDGES}_{\text{diag}} = \{(r^{\bar{a}}_{\text{alpha}}, w^{\bar{b}}_{\text{alpha}}) : \bar{a} \neq \bar{b}$ and the three-way intersection of matzn-of-class($\bar{a}$)(readset($\bar{a}$)) and logical^{-1}(writeset($\bar{b}$)) and stored-data^{-1}(alpha) is nonempty$\}$

In a conflict graph, CG(D), a path is a sequence $(a_1, a_2, \ldots, a_n)$ where for each i, $1 \leq i < n$, (a_i, a_{i+1}) is an edge of CG(D). If $a_1 = a_n$ and no edge appears twice in the path, then the path is called a cycle.

An edge (a_i, a_j) in CG(D) is called heterogeneous if the two nodes have different superscripts (i.e., are in different classes). A path (or cycle) is nonredundant if each class is a superscript in at most two heterogeneous edges in the path (or cycle).

4.2.4 Protocols and Protocol Selection Rules

The protocols are now defined purely in terms of logs. The timestamping mechanisms described in Section 2 can be thought of as a method of implementing the protocols.

A read operation R^a_{alpha} satisfies protocol P2 in log L with respect to classes $\{\bar{a}_1,\ldots,\bar{a}_n\}$ in KAPPA if there exists a transaction b such that R^a_{alpha} satisfies the "partitioned writes property" with respect to E^b and $\{\bar{a}_1,\ldots,\bar{a}_n\}$. A read operation R^a_{alpha} satisfies the <u>partitioned writes property</u> with respect to E^b and $\{\bar{a}_1,\ldots,\bar{a}_n\}$ in log L if for each transaction c with E^c in L and transclass(c) in $\{\bar{a}_1,\ldots,\bar{a}_n\}$:

1. If E^c precedes E^b and W^c_{alpha} appears in L, then W^c_{alpha} precedes R^a_{alpha} in L; and

2. If (b = c or E^b precedes E^c) and W^c_{alpha} appears in L, then R^a_{alpha} precedes W^c_{alpha}.

Two read operations R^a_{alpha} and R^a_{beta}, satisfy protocol P2f with respect to classes $\{\bar{a}_1,\ldots,\bar{a}_m\}$ and $\{\bar{a}_{m+1},\ldots,\bar{a}_n\}$ (respectively) in log L if there exists a transaction, b, such that R^a_{alpha} satisfies the partitioned writes property with respect to E^b and $\{\bar{a}_1,\ldots,\bar{a}_m\}$ and R^a_{beta} satisfies the partitioned writes property with respect to E^b and $\{\bar{a}_{m+1},\ldots,\bar{a}_n\}$.

A read operation R^a_{alpha} satisfies protocol P3 with respect to $\{\bar{a}_1,\ldots,\bar{a}_n\}$ in log L if it satisfies the partitioned writes property with respect to E^a and $\{\bar{a}_1,\ldots,\bar{a}_n\}$.

Two remarks should be made regarding these protocols. First, the protocols are mutually compatible in the following sense: if R^a_{alpha} satisfies protocol P3 with respect to $\{\bar{a}_1,\ldots,\bar{a}_m\}$ and R^a_{beta} satisfies protocol P3 with respect to $\{\bar{a}_{m+1},\ldots,\bar{a}_n\}$, then R^a_{alpha} and R^a_{beta} satisfy protocol P2f with respect to $\{\bar{a}_1,\ldots\bar{a}_m\}$ and $\{\bar{a}_{m+1},\ldots,\bar{a}_n\}$ (respectively) and R^a_{alpha} (for example) satisfies protocol P2 with respect to $\{\bar{a}_1,\ldots\bar{a}_m\}$. Second, protocol P2 allows a single R^a_{alpha} to satisfy P2 with respect to two different sets of classes using two different transaction b's. That is, R^a_{alpha} can satisfy P2 with respect to $\{\bar{a}_1,\ldots,\bar{a}_n\}$ because it satisfies the partitioned writes property with respect to E^b and $\{\bar{a}_1,\ldots,\bar{a}_n\}$ while in the same log it satisfies P2 with respect to $\{\bar{c}_1,\ldots,\bar{c}_m\}$ because it satisfies the partitioned writes property with respect to $E^{b'}$ and $\{\bar{c}_1,\ldots,\bar{c}_m\}$. Yet, there may be no single $E^{b''}$ such that

R^{a}_{alpha} satisfies the partitioned writes property with respect to both sets. This subtlety cannot be hanuled by the read conditions described in Section 2 without some modification.

We complete our formal model by defining the protocol selection rules (abbr. PSRs). Let CG(D) be a conflict graph over the design D and let L be a log defined over D and transaction set TAU. Then L satisfies the protocol selection rules if each of the following hold:

> PSR_1. For all alpha in DELTA and a in TAU, if $r^{\bar{a}}_{alpha}$ (where $\bar{a}$ = transclass(a)) lies on a nonredundant cycle in CG(D) in a subpath of the form $(w^{\bar{b}}_{alpha}, r^{\bar{a}}_{alpha}, e^{\bar{a}}, w^{\bar{a}}_{beta})$ or $(w^{\bar{b}}_{alpha}, r^{\bar{a}}_{alpha}, e^{a}, e^{c})$ for some $\bar{b}$, $\bar{c}$ in KAPPA and beta in DELTA, then R^{a}_{alpha} is in L and R^{a}_{alpha} satisfies protocol P3 with respect to class $\bar{b}$ at alpha in L.
>
> PSR_2. For all alpha, beta in DELTA and a in TAU, if $r^{\bar{a}}_{alpha}$ and $r^{\bar{a}}_{beta}$ ($\bar{a}$ = transclass(a)) lie on a nonredundant cycle in CG(D) in a subpath of the form $(w^{\bar{b}}_{beta}, r^{\bar{a}}_{beta}, e^{\bar{a}}, r^{\bar{a}}_{alpha}, w^{\bar{c}}_{alpha})$, for some $\bar{b}$ and $\bar{c}$ in KAPPA ($\bar{b} \neq \bar{c}$), then R^{a}_{alpha} and R^{a}_{beta} appear in L and R^{a}_{alpha} and R^{a}_{beta} satisfy protocol P2f with respect to $\{\bar{c}\}$ and $\{\bar{b}\}$ respectively in L.

PSR_3. For all alpha in DELTA and a in TAU, if $r^{\bar{a}}_{alpha}$ ($\bar{a}$ = transclass(a)) lies on a nonredundant cycle that contains a vertical edge in CG(D) in a subpath of the form ($w^{\bar{b}}_{alpha}$, $r^{\bar{a}}_{alpha}$, $w^{\bar{c}}_{alpha}$), for some $\bar{b}$ and $\bar{c}$ ın KAPPA ($\bar{b} \neq \bar{c}$), then R^{a}_{alpha} appears in L and R^{a}_{alpha} satisfies protocol P2 with respect to {$\bar{b}$, $\bar{c}$} in L.

A system satisfies the protocol selection rules if for any database design D and transaction set TAU, all logs defined over D and TAU satisfy the protocol selection rules.

4.3 Serialization

Theorem SR If a system is well-formed, satisfies the pipelining rules, and satisfies the protocol selection rules, then all logs that it can generate are serially reproducible.

The first stage of the proof of this theorem is to develop an algorithm, called the serialization procedure, that attempts to serialize a given log. If the procedure gets stuck, then certain conditions are shown to hold by lemma S (the serialization lemma).

4.3.1 Conflicts

We begin by defining a new log symbol, called a composite atom, which is an adjacent group of symbols (R's, W's, and an E) that all have the same superscript (i.e., all in the same transaction) and include an E. The symbolic notation for a composite atom is $A^a[R_{alpha-1},\ldots R_{alpha-m}, W_{alpha-(m+1)},\ldots,W_{alpha-n}]$, which is equivalent to the sublog

$$R^a_{alpha-1} \cdots R^a_{alpha-m}\ E^a\ W^a_{alpha-(m+1)} \cdots W^a_{alpha-n}.$$

Frequently, we will simply write A^a for the composite atom, as an abbreviation. Note that not all R^a's and W^a's that appear in a log must be members of A^a. The only log symbol that must be a member of A^a is E^a. Also, note that since for each transaction, a, no more than one E^a occurs in a log, therefore only one A^a can appear in a log. The introduction of composite atoms is simply a notational convenience so that groups of symbols for a single transaction can be handled as a unit.

We define an atom to be either a composite atom or an isolated R or W that is not adjacent to its corresponding

E (i.e. is not a member of a composite atom). In the sequel, we assume that all logs consist of atoms. That is, all E's are replaced by A's. To do this, the log transformations, TR1 - TR7, and conflicts, NTR1-NTR6, must be extended to handle A's. The extensions are direct consequences of the original transformations and conflicts and the definition of atom. Since we only need conflicts in our proof, we will only extend conflicts and not bother with the transformations. In the following, note that composite atoms are never split up. The conflicts are:

NTR$_1$'. ...R^{a}_{alpha} R^{b}_{alpha}... where transclass(a) = transclass(b).

NTR$_2$'. ...W^{a}_{alpha} W^{b}_{alpha} ... where transclass(a) = transclass(b).

NTR$_3$'. ...R^{a}_{alpha} W^{b}_{alpha} ... or ... W^{b}_{alpha} R^{a}_{alpha} ... , where either transclass(a) = transclass(b) or the three-way intersection of matzn-of-class (transclass(a))(transreadset(a)) and logical^{-1}(transwriteset(b)) and stored-data^{-1}(alpha) is nonempty.

NTR_4'. ... R^a_{alpha} A^a...

NTR_5'. ... $A^a W^a_{alpha}$ ...

NTR_6'. ... $A^a A^b$ ... where at least one of following hold:

i. transclass(a) = transclass(b), or

ii. transwriteset(a) and transwriteset(b) have a non-empty intersection, or

iii. R^a_{alpha} is in A^a and W^b_{alpha} is in A^b and R^a_{alpha} W^b_{alpha} conflict by NTR_3', or

iv. W^a_{alpha} is in A^a and R^b_{alpha} is in A^b and W^a_{alpha} R^b_{alpha} conflict by NTR_3'.

NTR_7'. ... $R^a_{alpha} A^b$ ... or ...$A^b R^a_{alpha}$ ... where either

i. W^b_{alpha} is in A^b and R^a_{alpha} W^b_{alpha} conflict by NTR_3', or

ii. R^a_{alpha} is in A^b and transclass(a) = transclass(b).

NTR_8'. ... W^a_{alpha} A^b ... or ...$A^b W^a_{alpha}$ ... where either

i. R^b_{alpha} is in A^b and R^b_{alpha} W^a_{alpha} conflict by NTR_3' or

ii. W^b_{alpha} is in A^b and transclass(a) = transclass(b).

Lemma C If a pair of adjacent atoms in log L are not in conflict, then the log resulting from switching these atoms is equivalent to L.

Proof Follows directly from Theorem TR in Section 3.3. Q.E.D.

4.3.2 Augmented Conflicts

In the second stage of the proof, we will frequently draw contradictions regarding possible orderings of atoms in a log by appealing to certain protocols. However, after a log has been partially serialized, many of the atoms will no longer be in the same order in which they appeared in the original log before any attempt was made at serialization. Therefore, the fact that a partially serialized log violates the PSRs does not necessarily imply that the given log violates the PSRs. That is, it is *only* the given log which, by hypothesis, must satisfy the PSRs. Hence, we are unable to draw the desired contradiction.

What we require is a proof mechanism to guarantee that certain protocol violations in a partially serialized log imply the same violations in the given log. To do this, we introduce additional conflicts (called *augmented conflicts*), so that while trying to serialize a given log, we do not destroy some of the protocol information. These additional conflicts can be reflected in additional edges in the conflict graph (called *augmented edges*). We proceed by defining these concepts formally.

All augmented conflicts are between pairs of E's. Since we have replaced E's by A's for the purposes of the proof, we will state the augmented conflict rules in terms of A's.

> $ANTR_{P2}$: $\ldots A^a A^b \ldots$ if there is a transaction c in TAU ($c \neq a$, $c \neq b$) and an alpha in DELTA such that R^c_{alpha} must (according to the PSRs) satisfy P2 with respect to transclass(a) and transclass(b) at alpha.

> $ANTR_{P2f}$: $\ldots A^a A^b \ldots$ if there is a transaction c in TAU and alpha and beta in DELTA such that R^c_{alpha} and R^c_{beta} must (according to the PSRs) satisfy protocol P2f with respect to transclass(a) and transclass(b) respectively.

> $ANTR_{P3}$: $\ldots A^a A^b \ldots$ if there is an alpha in DELTA such that either R^a_{alpha} must (according to the PSRs) run P3 with respect to $\bar{b}$ or R^b_{alpha} must (according to the PSRs) run P3 with respect to $\bar{a}$.

Two atoms are in augmented conflict if they conflict by NTR_1' - NTR_8' or by $ANTR_1$ - $ANTR_3$.

Corollary C-AUG If a pair of adjacent atoms are not in augmented conflict in log L, the log resulting from switching these atoms is equivalent to L.

Proof Follows immediately from lemma C. Q.E.D.

Each of the above conflicts must generate an edge in the conflict graph. We define an augmented conflict graph, ACG(D)=<VERTICES, EDGES + $\text{EDGES}_{\text{aug}}$>, as a vertex labelled undirected graph defined over database design D where VERTICES and EDGES are identical to those in CG(D) and $\text{EDGES}_{\text{aug}}$ is:

$\text{EDGES}_{\text{aug}} = \text{EDGES}_{\text{aug-P3}} + \text{EDGES}_{\text{aug-P2f}} + \text{EDGES}_{\text{aug-P2}}$

$\text{EDGES}_{\text{aug-P3}} = \{(e^{\bar{a}}, e^{\bar{b}})$: for all classes $\bar{a}, \bar{b}$ in KAPPA such that there exists an alpha in DELTA such that $r^{\bar{a}}_{\text{alpha}}$ lies on a nonredundant cycle in CG(D) in a subpath $(w^{\bar{b}}_{\text{alpha}}, r^{\bar{a}}_{\text{alpha}}, e^{\bar{a}}, w^{\bar{a}}_{\text{beta}})$ or $(w^{\bar{b}}_{\text{alpha}}, r^{\bar{a}}_{\text{alpha}}, e^{\bar{a}}, e^{\bar{c}})$ for some beta in DELTA and $\bar{c}$ in KAPPA.$\}$

$\text{EDGES}_{\text{aug-P2f}} = \{(e^{\bar{a}}, e^{\bar{b}})$: for all classes $\bar{a}$, $\bar{b}$ in KAPPA such that there exists a $\bar{c}$ in KAPPA and an alpha and beta in DELTA such that $r^{\bar{c}}_{\text{alpha}}$ and $r^{\bar{c}}_{\text{beta}}$ lie on a nonredundant cycle in CG(D) in a subpath $(w^{\bar{b}}_{\text{beta}}, r^{\bar{c}}_{\text{beta}}, e^{\bar{c}}, r^{\bar{c}}_{\text{alpha}}, w^{\bar{a}}_{\text{alpha}})\}$.

$\mathrm{EDGES}_{\mathrm{aug-P2}} = \{(e^{\bar{a}}, e^{\bar{b}})$: for all classes $\bar{a}$, $\bar{b}$ in KAPPA such that there exists a $\bar{c}$ in KAPPA and an alpha in DELTA such that $r^{\bar{c}}_{\mathrm{alpha}}$ lies on a nonredundant cycle in CG(D) in a subpath $(w^{\bar{a}}_{\mathrm{alpha}}, r^{\bar{c}}_{\mathrm{alpha}}, w^{\bar{b}}_{\mathrm{alpha}})$.

We reiterate that the augmented conflicts are required only to retain certain ordering information between E's in a log, so that this information is not destroyed while trying to serialize a log.

4.3.3 The Serialization Procedure

The serialization procedure takes a non-serial log and tries to serialize it by switching adjacent atoms that are not in augmented conflict. The actual serialization is done by the procedures MOVELEFT and MOVERIGHT which scan the sublog that separates the two atoms to be serialized and tries to remove atoms from that sublog, thereby bringing the two atoms closer together. The procedure SP repeatedly calls MOVELEFT and MOVERIGHT until the two atoms have been serialized or until the two atoms cannot be brought closer together. The choice of which atoms to serialize is made by SERIALIZE, which quits if either the given log has been completely serialized or there are two atoms which cannot be serialized.

SERIALIZE: PROCEDURE (L_{in}, L_{out}, LEFTATOM, RIGHTATOM) RETURNS (BOOLEAN);

/*The procedure takes L_{in} as input. If L_{in} is successfully serialized, it returns TRUE. If not, it returns FALSE, and the log L_{out} is the partially serialized log where LEFTATOM and RIGHTATOM is the pair of atoms that could not be serialized.*/

```
        $L_{out}$ := $L_{in}$;

        DO FOREVER;
          Select the leftmost atom in $L_{out}$ that is either
             i.   an atom $A^a$ and there is an alpha with
                  $R^a_{alpha}$ in $L_{out}$ but $R^a_{alpha}$ is
                  not in $A^a$;  or

             ii.  an atom $W^a_{alpha}$ in $L_{out}$ but $W^a_{alpha}$ is
                  not in $A^a$;

          IF there is no (i) or (ii) THEN RETURN (TRUE);

          IF (i) is the case satisfied above
          THEN BEGIN LEFTATOM := rightmost $R^a$ in $L_{out}$ but
```

```
                    not in $A^a$;   RIGHTATOM := $A^a$; END;
          ELSE BEGIN LEFTATOM := $A^a$; RIGHTATOM := $W^a_{alpha}$; END;
          IF NOT SP($L_{out}$, LEFTATOM, RIGHTATOM)
             THEN RETURN (FALSE);
             ELSE MERGE LEFTATOM and RIGHTATOM into
                          a single $A^a$;

        END

        END SERIALIZE;


SP:  PROCEDURE  (LOG, LEFTATOM, RIGHTATOM) RETURNS
             (BOOLEAN);

     TEMP1 := TEMP2 := TRUE;

     DO WHILE ((TEMP1  or  TEMP2)  and  (LEFTATOM  is  not
     adjacent to RIGHTATOM));

        TEMP1 := MOVELEFT (LOG, LEFTATOM, RIGHTATOM);

        TEMP2 := MOVERIGHT (LOG, LEFTATOM, RIGHTATOM);

     END;
     IF (LEFTATOM is adjacent to RIGHTATOM)
                                THEN RETURN (TRUE);
                                ELSE RETURN (FALSE);
END SP;
```

```
MOVELEFT:  PROCEDURE (LOG, LEFTATOM, RIGHTATOM) RETURNS
      (BOOLEAN);

     TEMPLOG := LOG;  TEMPOUT := FALSE;

     DO  FOR  EACH atom, X, between LEFTATOM and RIGHTATOM
     in LOG beginning with the right neighbor of  LEFTATOM
     and moving right;

        DO WHILE ((left neighbor of X in TEMPLOG is not in
        augmented  conflict with X) AND (right neighbor of
        X is not LEFTATOM));

        Switch X with its left neighbor in TEMPLOG;

        END;

     IF (right neighbor of X in TEMPLOG is LEFTATOM)
     THEN TEMPOUT := TRUE;

     END;

     LOG := TEMPLOG;

     RETURN (TEMPOUT);

     END MOVELEFT;
```

```
MOVERIGHT:  PROCEDURE (LOG, LEFTATOM, RIGHTATOM) RETURNS
             (BOOLEAN);

     TEMPLOG := LOG;  TEMPOUT := FALSE;

     DO FOR each atom, X, between RIGHTATOM  and  LEFTATOM
     in  LOG beginning with the left neighbor of RIGHTATOM
     and moving left;

        DO WHILE (right neighbor of X in TEMPLOG is not in
        augmented conflict with X) AND (left neighbor of X
        is not RIGHTATOM));

        Switch X with its right neighbor in TEMPLOG;

        END;

     IF (left neighbor of X is RIGHTATOM)
     THEN TEMPOUT := TRUE;

     END;

     LOG := TEMPLOG;

     RETURN (TEMPOUT);

     END MOVERIGHT;
```

4.3.4 The Serialization Lemma

If the serialization procedure, SERIALIZE, is given a log that is not serially reproducible, then certain properties must be true of the output of SERIALIZE. These properties are summarized in lemma S presented in this section.

First, we require two new definitions. A log, L_1, is a projection of a log, L_2, if L_1 can be obtained from L_2 simply by excising atoms from L_2. A log, L, is blocked if every atom in L is in augmented conflict with both its left and right neighbors in L. Our goal in lemma S will be to construct a blocked projection of the log that SERIALIZE outputs.

Lemma S Let LOG_{given} be a well-formed log defined on the database design D. If LOG_{given} is not serially reproducible, then

I. SERIALIZE (LOG_{given}, LOG_{out}, LA, RA) returns FALSE;

II. every atom of the form W^{a}_{alpha} to the left of RA in LOG_{out} is a member of A^{a};

III. every atom of the form A^a to the left of RA in LOG_{out} has no R^a's in LOG_{out} that are not members of A^a;

IV. there is a blocked projection, $LOG_{blocked}$, of LOG_{out} such that

i. LA and RA are the leftmost and rightmost atoms of $LOG_{blocked}$ respectively;

ii. there is an a in TAU and an alpha in DELTA such that either (LA = R^a_{alpha} and RA = A^a) or (LA = A^a and RA = W^a_{alpha}).

Proof (Part I) Since only equivalence preserving transformations are attempted by SERIALIZE (by corollary C-AUG), if LOG_{given} is not serially reproducible then SERIALIZE must fail to serialize it and therefore returns FALSE.

(PARTS II and III) The last atom selected by SERIALIZE was the leftmost atom that was either a W not in any A or an A with an outstanding R. Hence, there can be no atoms to the left of RA in LOG_{out} with either of these properties.

(PART IV) Construct $LOG_{blocked}$ from LOG_{out} as follows: Excise all atoms to the left of LA and to the right of RA in LOG_{out}. Let X be LA's right neighbor. Let Y be an atom in the log somewhere to the right of X that conflicts with X. There must be such a Y, for otherwise MOVERIGHT would have moved X to the right of RA. Excise all atoms in LOG_{out} between X and Y. If Y ≠ RA, then set X := Y and find a Y to the right of X that conflicts with X as before. Repeat this process until Y = RA. The resulting log, $LOG_{blocked}$, is a projection of LOG_{out} and is blocked (by construction). Furthermore, by the choice of LA or RA in SERIALIZE, IV (ii) must hold. Q.E.D.

While lemma S shows that every non-serially reproducible log will fail to be serialized by SERIALIZE, it does not claim that if a log is serially reproducible then SERIALIZE will succeed. This converse is not in general true, for the transformations we use are not complete, as mentioned in Section 3.3. If we were able to find a more complete set of transformations, then this might permit us to weaken our protocols; for some of the serially reproducible logs that are not serializable under our current transformations may no longer require a strong protocol to guarantee that they will not occur.

4.4 Showing Nonserializable Logs are Impossible

The proof of theorem SR is embodied in two major lemmas. We first present the structure of the proof and then proceed to the lemmas.

Theorem SR If a system is well-formed, satisfies the pipelining rules, and satisfies the protocol selection rules, then all logs that it can generate are serially reproducible.

Proof Assume the theorem is false. Then there is a log, say LOG_{given}, which is well-formed, satisfies the pipelining rules, and satisfies the protocol selection rules, but is not serially reproducible. By lemma S, SERIALIZE (LOG_{given}, LOG_{out}, LA, RA) returns false and, by IV(ii) there is a transaction a in TAU and an alpha in DELTA where either ($LA = R^{a}_{alpha}$ and $RA = A^{a}$) or ($LA = A^{a}$ and $RA = W^{a}_{alpha}$). These possibilities are shown below to be impossible by lemmas RA and AW respectively. Hence, the conclusions of lemma S were false. But this is possible only if the hypothesis of lemma S is false. So, the hypothesis that LOG_{given} was not serially reproducible must be false. Q.E.D.

To prove lemmas RA and AW we will use the following lemmas.

Lemma P Let L be a blocked log over transaction set TAU and database design D such that the leftmost atom is in class $\bar{a}$, the rightmost atom is in class $\bar{b}$, and the log has no atom in class $\bar{c}$. Then there is a path in CG(D) which is not incident with any node in class $\bar{c}$.

Proof If $\bar{a} = \bar{b}$, then the lemma is trivially true. If $\bar{a} \neq \bar{b}$, then since the log is blocked, each atom is in augmented conflict with its neighbors. Each such conflict corresponds to an edge in ACG(D), so there is a path from $\bar{a}$ to $\bar{b}$ in ACG(D) that is not incident with $\bar{c}$. To find a new path in CG(D) we need to replace each edge in the old path that is in $EDGES_{aug}$ by a path in CG(D). Consider some edge, say $(e^{\bar{d}}, e^{\bar{f}})$, in the path in $EDGES_{aug}$. If the edge is in $EDGES_{aug-P3}$, then replace it by the path $(e^{\bar{d}}, w^{\bar{d}}_{alpha}, r^{\bar{f}}_{alpha}, e^{\bar{f}})$ that must exist by definition of $EDGES_{aug-P3}$. If the edge is in $EDGES_{aug-P2f}$, then there is a class, $\bar{g}$, and data modules alpha and beta such that the subpath $(e^{\bar{d}}, w^{\bar{d}}_{alpha}, r^{\bar{g}}_{alpha}, e^{\bar{g}}, r^{\bar{g}}_{beta}, w^{\bar{f}}_{beta}, e^{\bar{f}})$ is in CG(D) and there is a path in CG(D) from $\bar{d}$ to $\bar{f}$ that is not incident with $\bar{g}$. If $\bar{g} \neq \bar{c}$, then replace $(e^{\bar{d}}, e^{\bar{f}})$ by the subpath (which is not incident with $\bar{c}$). If $\bar{g} = \bar{c}$, then replace $(e^{\bar{d}}, e^{\bar{f}})$ by the other $\bar{d}$ - $\bar{f}$ path. If the edge is in $\text{EDGES}_{\text{aug-P2}}$, then there is a class, $\bar{g}$, and a datamodule, alpha, such that there is a subpath ($e^{\bar{d}}$, $w^{\bar{d}}_{\text{alpha}}$, $r^{\bar{g}}_{\text{alpha}}$, $w^{\bar{f}}_{\text{alpha}}$, $e^{\bar{f}}$) in CG(D) and there is a path in CG(D) from $\bar{d}$ to $\bar{f}$ that is not incident with $\bar{g}$. If $\bar{g} \neq \bar{c}$, then replace ($e^{\bar{d}}$, $e^{\bar{f}}$) by the subpath; else replace it by the other $\bar{d}$ - $\bar{f}$ path. If all edges in $\text{EDGES}_{\text{aug}}$ are replaced in this way by paths in CG(D) that are not incident with class $\bar{c}$, then we have constructed a path in CG(D) with a node in class $\bar{c}$. To make the path nonredundant, simply replace each nontrivial subpath whose endpoints are in the same class by vertical edges. This nonredundant path then satisfies the lemma. Q. E. D.

<u>Lemma B</u> Let L be a log defined over transaction set TAU and database design D. Let L_{out} be a log obtained from L by the serialization procedure, and let L'_{out} be a projection of L_{out}. Let X^a and Y^b be symbols (i.e., not atoms) that are in augmented conflict such that X^a precedes Y^b in L'_{out}. Then X^a precedes Y^b in L.

<u>Proof</u> Since the serialization procedure never switches atoms that are in augmented conflict, X^a and Y^b must have appeared in the same order in L and L_{out}. The same must hold in L'_{out}, since the latter is a projection of L_{out}. Q. E. D.

Lemma RA Let LOG_{given} be a log defined over database design D and transaction set TAU such that it is well-formed, satisfies the pipelining rules, and satisfies the protocol selection rules. Then it is not possible that SERIALIZE(LOG_{given}, LOG_{out}, LA, RA) returns FALSE with LA = R^{a}_{alpha} and RA = A^{a} ior some a in TAU and alpha in DELTA.

Proof Assume that the lemma is false. Then, SERIALIZE returns FALSE and, by lemma S, there is a blocked projection of LOG_{out}, say $LOG_{blocked}$, whose leftmost and rightmost atoms are R^{a}_{alpha} and A^{a} respectively. That is, $LOG_{blocked}$ is of the form R^{a}_{alpha} ... A^{a}.

Beginning with A^{a}, scan left in $LOG_{blocked}$ until the first occurrence is found of an $R^{a'}_{beta}$ where $R^{a'}_{beta}$ is not in its $A^{a'}$ and transclass(a') = transclass(a). (Note: possibly alpha = beta, and possibly $R^{a'}_{beta}$ = R^{a}_{alpha}.) Now, starting with $R^{a'}_{beta}$, scan right in $LOG_{blocked}$ until the first $A^{a''}$ is found with transclass(a") = transclass(a).

We want to show that $A^{a''}$ is actually A^{a}. So suppose not, i.e., a" $\neq$ a. Since $R^{a'}_{beta}$ is not in $A^{a'}$ (by construction), by lemma S part III, E^{a} must precede $E^{a'}$ in LOG_{out}. Since E^{a} and $E^{a'}$ conflict, by lemma B E^{a} preceded $E^{a'}$ in LOG_{given} (or E^{a} = $E^{a'}$). Since transclass(a") = transclass(a), and since $E^{a''}$ and E^{a} conflict in

$LOG_{blocked}$, by lemma B $E^{a''}$ preceded E^{a} in LOG_{given}. By lemma S part III, $A^{a''}$ must contain all of its R's, including $R^{a''}_{beta}$. Since $R^{a'}_{beta}$ precedes $R^{a''}_{beta}$ in $LOG_{blocked}$ and $R^{a'}_{beta}$ conflicts with $R^{a''}_{beta}$, by lemma B $R^{a'}_{beta}$ preceded $R^{a''}_{beta}$ in LOG_{given}. But since $E^{a''}$ preceded $E^{a'}$, this is a violation of R-R pipelining. So, we have a" = a, as desired. That is, $LOG_{blocked}$ is of the form R^{a}_{alpha} ... $R^{a'}_{beta}$ ... A^{a} ... $A^{a'}$, such that no $A^{a''}$ with transclass(a") = transclass(a) appears between $R^{a'}_{beta}$ and A^{a}.

We create a new log, $LOG'_{blocked}$, by excising from $LOG_{blocked}$ those atoms to the left of $R^{a'}_{beta}$ and those between A^{a} and $A^{a'}$. Since A^{a} - $A^{a'}$ conflict, $LOG'_{blocked}$ is indeed blocked. So, $LOG'_{blocked}$ is of the form $R^{a'}_{beta}$ ... A^{a} $A^{a'}$ (where possibly a' = a).

Consider $R^{a'}_{beta}$. There are only two kinds of atoms that can be $R^{a'}_{beta}$'s conflicting right neighbor: either an $R^{a''}_{beta}$ where transclass(a") = transclass(a'); or a W^{b}_{beta} where transclass(a') ≠ transclass(b) and the three-way intersection of matzn-of-class(transclass(a'))(transreadset(a')) and $logical^{-1}$(transwriteset(b)) and $stored\text{-}data^{-1}$(beta) is nonempty. By choice of $R^{a'}_{beta}$, $R^{a''}_{beta}$ is not possible. So, $R^{a'}_{beta}$'s right neighbor must be W^{b}_{beta}. (Note: possibly alpha = beta). By lemma S (part II), W^{b}_{beta} is a member of A^{b}.

Let X^c be the left neighbor of A^a in $LOG'_{blocked}$. That is, $LOG'_{blocked}$ is of the form $R^{a'}_{beta}A^b[\ldots W^b_{beta}\ldots]\ldots X^cA^{a'}$. Since $LOG'_{blocked}$ is blocked, X^c and A^a are in augmented conflict. (Note: possibly c=b, $A^b=X^c$). By the above argument regarding $A^{a''}$, transclass(a) $\neq$ transclass(c).

In the remainder of this proof, let $\bar{a}$ = transclass(a), $\bar{b}$ = transclass(b), and $\bar{c}$ = transclass(c).

<u>Claim RA-path</u> There is a nonredundant path in CG(D) from a node labelled $\bar{b}$ to a node labelled $\bar{c}$ such that the path passes through no other node labelled $\bar{a}$.

This claim, which follows directly from lemma P, will be applied repeatedly in the remainder of the proof.

In the remainder of the proof, we analyze the ways in which X^c can be in augmented conflict with A^a, and show each possible conflict to be impossible. Since the only assumption made so far is that the lemma is false, the contradiction that X^c does not conflict with A^a will prove the lemma.

X^c can only be in augmented conflict with $A^{a'}$ due to one of NTR_1' - NTR_8', $ANTR_{P2}$, $ANTR_{P2f}$, or $ANTR_{P3}$. Since $\bar{c} \neq \bar{a}$ (by construction), NTR_1', NTR_2', NTR_4', NTR_5', NTR_6'(i), and NTR_7'(ii) cannot be the cause of the conflict. NTR_3' trivially does not apply, since it does not apply to an A.

NTR_8' cannot apply because by lemma S, X^c cannot be a W^c that is not a member of A^c. The remaining cases are NTR_6'(ii), NTR_6'(iii), NTR_6'(iv), NTR_7'(i), $ANTR_{P2}$, $ANTR_{P2f}$, and $ANTR_{P3}$; they are subsumed by the following cases:

I. $X^c = A^c$; there is a gamma in DELTA with W^c_{gamma} in A^c and R^a_{gamma} in A^a; and the three-way intersection of matzn-of-class($\bar{a}$)(transreadset(a)) and $logical^{-1}$(transwriteset(c)) and $stored\text{-}data^{-1}$(gamma) is nonempty.

II. there is a gamma in DELTA such that either $X^c = R^c_{gamma}$ or ($X^c = A^c$ and R^c_{gamma} is in A^c); W^a_{gamma} is in A^a; and the three-way intersection of matzn-of-class($\bar{c}$)(transreadset(c)) and $logical^{-1}$(transwriteset(a)) and $stored\text{-}data^{-1}$(gamma) is nonempty.

III. $X^c = A^c$ and the intersection of transwriteset(c) and transwriteset(a) is nonempty.

IV. $X^c = A^c$ and A^c-A^a are in augmented conflict by $ANTR_{P2}$, $ANTR_{P2f}$, or $ANTR_{P3}$.

We analyze each of the four cases in detail.

Case I (X^c = A^c and contains W^c_{gamma}; A^a contains R^a_{gamma}; R^a_{gamma} and W^c_{gamma} conflict)

$LOG'_{blocked}$ is of the form $R^{a'}_{beta}$ $A^b[\ldots W^b_{beta}\ldots]\ldots$ $A^c[\ldots W^c_{gamma}]$ $A^a[\ldots R^a_{gamma}\ldots]$ $A^{a'}$.

There are two subcases to consider: beta ≠ gamma and beta = gamma.

Subcase beta ≠ gamma

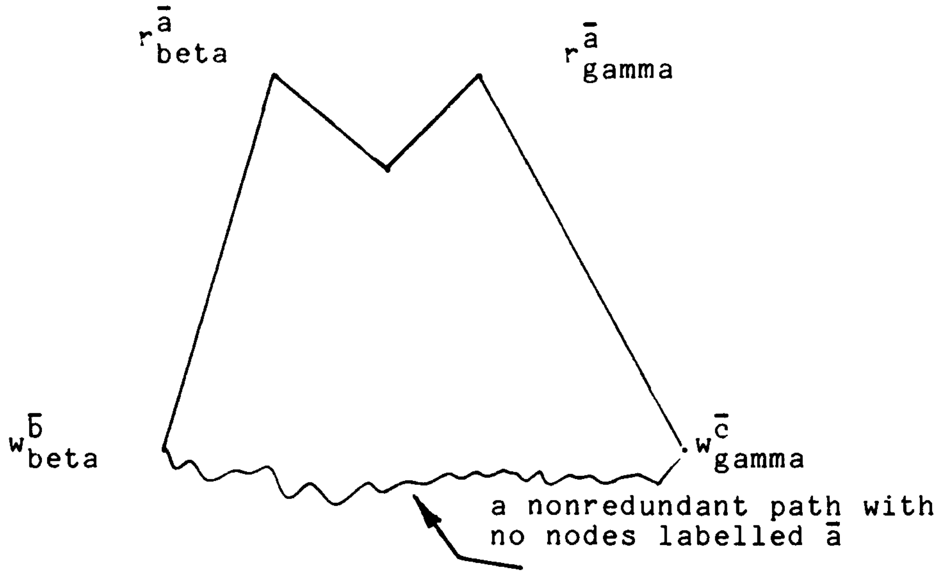
From the a' - b and c - a conflict in $LOG'_{blocked}$, the edges ($r^{\bar{a}}_{beta}$, $w^{\bar{b}}_{beta}$) and ($w^{\bar{c}}_{gamma}$, $r^{\bar{a}}_{gamma}$) are in CG(D).

By definition of $EDGES_{vert}$, the edges ($r^{\bar{a}}_{beta}$, $e^{\bar{a}}$) and ($r^{\bar{a}}_{gamma}$, $e^{\bar{a}}$) are in CG(D).

By claim RA-path, there is a nonredundant path in CG(D) from $w^{\bar{b}}_{beta}$ to $w^{\bar{c}}_{gamma}$ that does not pass through any nodes in class $\bar{a}$. Graphically, we have the cycle noted in figure 4.1.

This cycle and the protocol selection rules imply $R^{a'}_{gamma}$ and $R^{a'}_{beta}$ must satisfy protocol P2f against $\bar{c}$ and $\bar{b}$ (respectively) at gamma and beta (respectively). The following sequence of inferences leads to a contradiction.

Nonredundant Cycle, for Case I of Lemma RA   Figure 4.1



i. Since E^{a} and $E^{a'}$ are in augmented conflict and E^{a} precedes $E^{a'}$ in $LOG_{blocked}$, by lemma B E^{a} precedes $E^{a'}$ in LOG_{given} By R-R pipelining, R^{a}_{gamma} precedes $R^{a'}_{gamma}$ in LOG_{given}.

ii. R^{a}_{gamma} conflicts with W^{c}_{gamma}, so by lemma B R^{a}_{gamma} followed W^{c}_{gamma} in LOG_{given}.

iii. By (i), (ii) and transitivity, W^{c}_{gamma} precedes $R^{a'}_{gamma}$ in LOG_{given}.

iv. By definition of NTR_{P2f}, E^{b} and E^{c} are in augmented conflict. So by lemma B, E^{b} precedes E^{c} in LOG_{given}.

v. Since $R^{a'}_{beta}$ and W^{b}_{beta} conflict and $R^{a'}_{beta}$ precedes ${}^{b}_{beta}$ in $LOG'_{blocked}$, by lemma B $R^{a'}_{beta}$ precedes W^{b}_{beta} in LOG_{given}.

vi. But (iii), (iv), and (v) constitute a violation of the partitioned writes property for $R^{a'}_{beta}$ and $R^{a'}_{gamma}$ with respect to $\bar{b}$ and $\bar{c}$ respectively. So, a' violated protocol P2f, a contradiction. This proves case I, subcase beta ≠ gamma.

<u>Subcase beta = gamma</u>

In this case, $LOG'_{blocked}$ is of the form:

$$R^{a'}_{beta}A^{b}[\ldots W^{b}_{beta}\ldots]\ldots A^{c}[\ldots W^{c}_{beta}\ldots]A^{a}[\ldots R^{a}_{beta}\ldots]A^{a'}.$$

If a = a' then $R^{a'}_{beta}$ isn't unique in the log, a contradiction. If a ≠ a'; then since $R^{a'}_{beta}$ and R^{a}_{beta} conflict, by lemma B $R^{a'}_{beta}$ precedes R^{a}_{beta} in LOG_{given}. Since E^{a} and $E^{a'}$ conflict, by lemma B E^{a} precedes $E^{a'}$ in LOG_{given}. This is a violation of R-R pipelining. Contradiction!

Case II (either $R^{c}_{gamma} = X^{c}$ or R^{c}_{gamma} is in $A^{c} = X^{c}$; W^{a}_{gamma} is in A^{a}; and R^{c}_{gamma} and W^{a}_{gamma} conflict)

$LOG'_{blocked}$ is either of the form

$R^{a'}_{beta}A^{b}[\ldots W^{b}_{beta}\ldots]\ldots R^{c}_{gamma}A^{a}[\ldots W^{a}_{gamma}\ldots]A^{a'}$

or

$R^{a'}_{beta}A^{b}[\ldots W^{b}_{beta}\ldots]\ldots A^{c}[\ldots R^{c}_{gamma}\ldots]A^{a}[\ldots W^{a}_{gamma}\ldots]A^{a'}$.

From the conflicts in $LOG'_{blocked}$, the edges $(r^{\bar{a}}_{beta}, w^{\bar{b}}_{beta})$ and $(r^{\bar{c}}_{gamma}, w^{\bar{a}}_{gamma})$ are in CG(D).

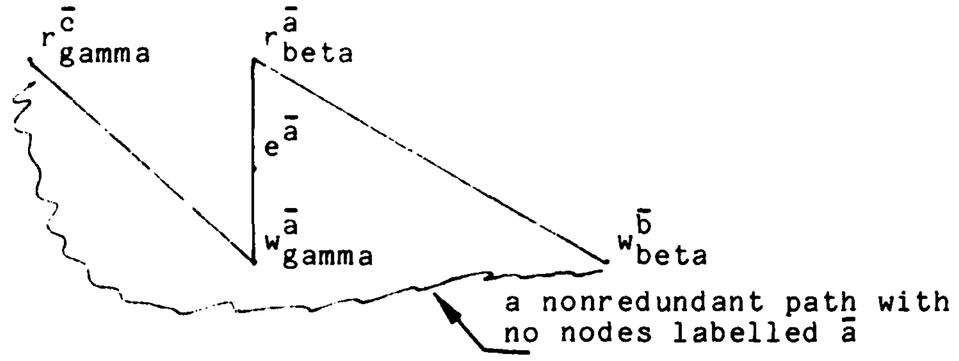
By definition of $EDGES_{vert}$, the edges $(r^{\bar{a}}_{beta}, e^{\bar{a}})$ and $(e^{\bar{a}}, w^{\bar{a}}_{gamma})$ are in CG(D).

By claim RA-path, there is a nonredundant path from $w^{\bar{b}}_{beta}$ to $r^{\bar{c}}_{gamma}$ that does not pass through any node in class $\bar{a}$. So we have the nonredundant cycle shown in figure 4.2.

This graph and the protocol selection rules implies that $R^{a'}_{beta}$ must satisfy protocol P3 with respect to $\bar{b}$ at beta. By $ANTR_{P3}$, E^{b} and E^{a} are in augmented conflict. Since E^{b} precedes E^{a} in $LOG'_{blocked}$, by lemma B E^{b} precedes E^{a} in LOG_{given}. By the same argument, we deduce that E^{a} precedes $E^{a'}$ in LOG_{given}. By transitivity, E^{b} precedes $E^{a'}$ in LOG_{given}. Since $R^{a'}_{beta}$ and W^{b}_{beta} conflict, by lemma B $R^{a'}_{beta}$ precedes W^{b}_{beta} in LOG_{given}. This is a violation of P3, a contradiction, thereby proving case II.

<u>Case III</u> ($X^{c} = A^{c}$ and the intersection of transwriteset(c) and transwriteset(a) is nonempty.

Nonredundant Cycle, for Case II of Lemma RA Figure 4.2



$LOG'_{blocked}$ is of the form $R^{a'}_{beta}A^{b}[...W^{b}_{beta}...]...A^{c}A^{a}A^{a'}$.

(This argument is essentially the same as Case II.)

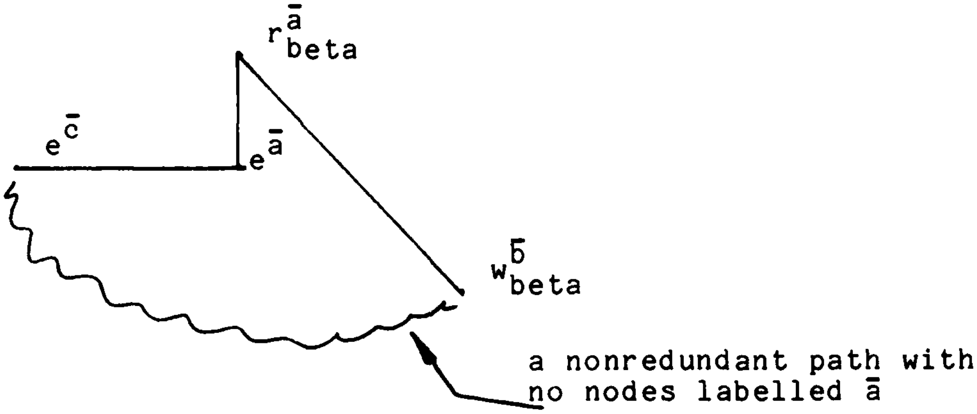
From the conflicts in $LOG'_{blocked}$, the edges $(r^{\bar{a}}_{beta}, w^{\bar{b}}_{beta})$ and $(e^{\bar{a}}, e^{\bar{c}})$ are in CG(D).

By definition of $EDGES_{vert}$, the edge $(r^{\bar{a}}_{beta}, e^{\bar{a}})$ is in CG(D).

By claim RA-path, there is a nonredundant path from $w^{\bar{b}}_{beta}$ to $e^{\bar{c}}$ that does not pass through any node in class $\bar{a}$. So, we have the nonredundant cycle shown in figure 4.3.

This graph and the protocol selection rules imply $R^{a'}_{beta}$ must satisfy P3 with respect to $\bar{c}$ at beta. By $ANTR_{P3}$, E^{b} and E^{a} are in augmented conflict. Since E^{b} precedes E^{a} in $LOG'_{blocked}$, by lemma B E^{b} precedes E^{a} in LOG_{given}. By the same argument, we deduce that E^{a} precedes $E^{a'}$ in LOG_{given}. By transitivity, E^{b} precedes $E^{a'}$ in LOG_{given}. Since $R^{a'}_{beta}$ and W^{b}_{beta} conflict, by lemma B, $R^{a'}_{beta}$ precedes W^{b}_{beta} in LOG_{given}. This violates P3, a contradiction, thereby proving case III.

Nonredundant cycle, for Case III of Lemma RA   Figure 4.3



CASE IV ($X^{c} = A^{c}$ and A^{c} - A^{a} are in augmented conflict by $ANTR_{P2}$, $ANTR_{P2f}$, or $ANTR_{P3}$.)

$LOG'_{blocked}$ is of the form

$R^{a'}_{beta}A^{b}[\ldots W^{b}_{beta}\ldots]\ldots A^{c}A^{a}A^{a'}$.

From the log, the edge ($r^{\bar{a}}_{beta}$, $w^{\bar{b}}_{beta}$) is in CG(D). From claim RA-path, there is a nonredundant path in CG(D) from a node in $\bar{c}$ to a node in $\bar{b}$ that does not pass through any node in $\bar{a}$. There are now three subcases to consider for each of $ANTR_{P2}$, $ANTR_{P2f}$ and $ANTR_{P3}$ -- the only ways that $A^{c} - A^{a}$ can be in augmented conflict.

Subcase IV - $ANTR_{P2}$

By $ANTR_{P2}$, there is a class, $\bar{d}$, and a data module, gamma, such that there is a nonredundant cycle in CG(D) with the subpath

$$(w^{\bar{c}}_{gamma}, r^{\bar{d}}_{gamma}, w^{\bar{a}}_{gamma}).$$

So, we can deduce that the edges $(w^{\bar{c}}_{gamma}, r^{\bar{d}}_{gamma})$ and $(r^{\bar{d}}_{gamma}, w^{\bar{a}}_{gamma})$ are in CG(D). By definition of $EDGES_{vert}$, the edges $(r^{\bar{a}}_{beta}, e^{\bar{a}})$ and $(e^{\bar{a}}, w^{\bar{a}}_{gamma})$ are in CG(D). Hence, given $(r^{\bar{a}}_{beta}, w^{\bar{b}}_{beta})$ above, we have the nonredundant path in CG(D) of

$$(w^{\bar{b}}_{beta}, r^{\bar{a}}_{beta}, e^{\bar{a}}, w^{\bar{a}}_{gamma}).$$

To complete a nonredundant cycle, we need an independent nonredundant path from $w^{\bar{a}}_{gamma}$ to $w^{\bar{b}}_{beta}$. If $\bar{d} = \bar{b}$, then we are done since the edges

$$(r^{\bar{d}}_{gamma}, w^{\bar{a}}_{gamma}), \quad (r^{\bar{b}}_{gamma}, \quad e^{\bar{b}}), \quad (e^{\bar{b}}, \quad w^{\bar{b}}_{beta})$$

suffice.

If $\bar{d} \neq \bar{b}$, then the edges $(w^{\bar{a}}_{gamma}, r^{\bar{d}}_{gamma})$ and $(r^{\bar{d}}_{gamma}, w^{\bar{c}}_{gamma})$ together with the known nonredundant path from $\bar{c}$ to $\bar{b}$ suffices. (If the path intersects $\bar{d}$, then the

$(r^{\bar{d}}_{gamma}, w^{\bar{c}}_{gamma})$ edge can be removed and replaced by vertical edge(s) connecting $r^{\bar{d}}_{gamma}$ to the $\bar{c}$ - $\bar{b}$ path.) So, we have a nonredundant cycle (see figure 4.4).

By the protocol selection rules, $R^{a'}_{beta}$ must satisfy P3 with respect to $\bar{b}$ at beta. However, P3 is violated in the following way. By ANTR$_{P3}$ and the cycle, E^{a} and E^{b} are in augmented conflict. So, since E^{b} precedes E^{a} in $LOG'_{blocked}$, by lemma B, E^{b} precedes E^{a} in LOG_{given}. As deduced earlier, E^{a} precedes $E^{a'}$ in LOG_{given}. By transitivity, E^{b} precedes $E^{a'}$ in LOG_{given}. Since $R^{a'}_{beta}$ conflicts with W^{b}_{beta}, by lemma B $R^{a'}_{beta}$ precedes W^{b}_{beta} in LOG_{given}. This violates P3, a contradiction, thereby proving the subcase.

Subcase IV ANTR$_{P2f}$

By ANTR$_{P2f}$, there is a class, $\bar{d}$, and two distinct data modules, gamma and delta, such that there is a nonredundant cycle in CG(D) with the subpath

$$(w^{\bar{c}}_{gamma}, r^{\bar{d}}_{gamma}, e^{\bar{d}}, r^{\bar{d}}_{delta}, w^{\bar{a}}_{delta}).$$

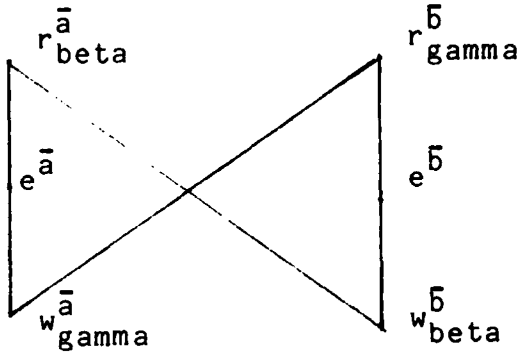
We can now continue exactly as in subcase IV - ANTR$_{P2}$, yielding the same P3 violation (see figure 4.5).

Subcase IV - ANTR$_{P3}$

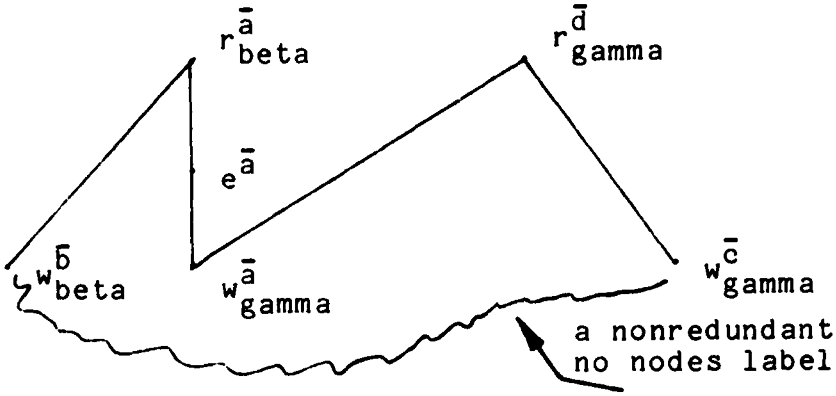
By ANTR$_{P3}$, there is a data module, gamma, such that there

Nonredundant Cycles, for Case IV-ANTR$_{P2}$ of Lemma RA   Figure 4.4

Subcase $\bar{b} = \bar{d}$



Subcase $\bar{b} \neq \bar{d}$



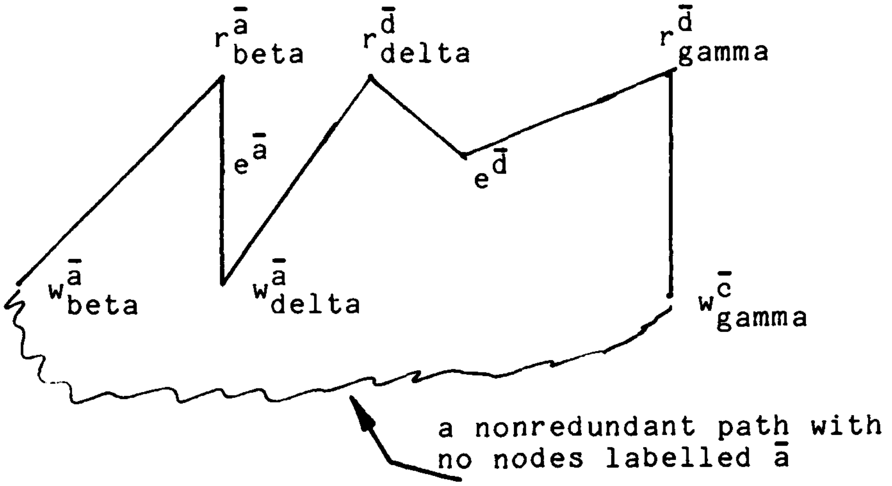
a nonredundant path with no nodes labelled $\bar{a}$

If the $\bar{b}$ - $\bar{c}$ path intersects $\bar{d}$, then we have vertical edge(s) from $r^{\bar{d}}_{gamma}$ to the path, thereby completing the cycle in a slightly different way.

---

is a nonredundant cycle in CG(D) with either the subpath

Nonredundant Cycle, for Case IV - ANTR$_{P2f}$ Figure 4.5



$(e^{\bar{c}}, r^{\bar{c}}_{gamma}, w^{\bar{a}}_{gamma}, e^{\bar{a}})$

or $(e^{\bar{a}}, r^{\bar{a}}_{gamma}, w^{\bar{c}}_{gamma}, e^{\bar{c}})$.

We treat each subpath as a separate case.

Subcase IV - ANTR$_{P3}$ - $(e^{\bar{c}}, r^{\bar{c}}_{gamma}, w^{\bar{a}}_{gamma}, e^{\bar{a}})$

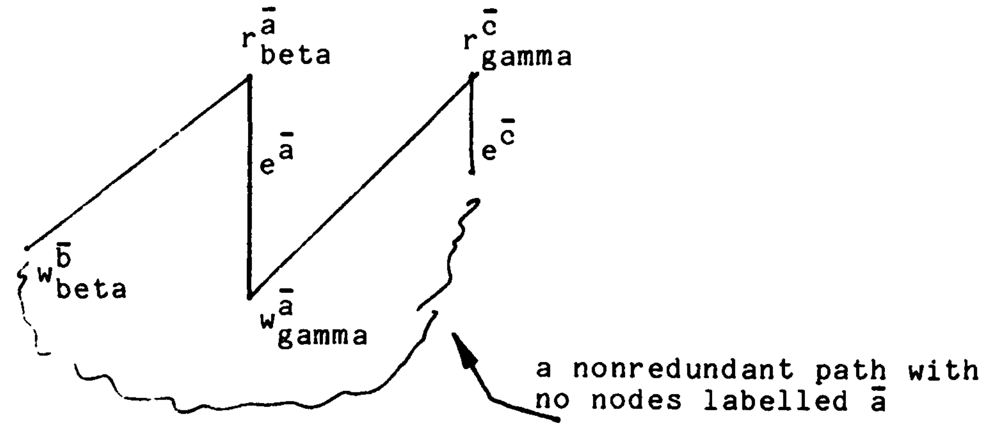
We can deduce that the edge $(r^{\bar{c}}_{gamma}, w^{\bar{a}}_{gamma})$ is in CG(D). We can now continue exactly as in Subcase IV - ANTR$_{P2}$, yielding the same P3 violation. (See figure 4.6.)

Subcase IV - ANTR$_{P3}$ - $(e^{\bar{a}}, r^{\bar{a}}_{gamma}, w^{\bar{c}}_{gamma}, e^{\bar{c}})$

Since $e^{\bar{a}}$- $e^{\bar{c}}$ are in EDGES$_{aug-P3}$, there is a nonredundant path from $e^{\bar{a}}$ to $e^{\bar{c}}$ that does not pass through any $r^{\bar{a}}$ (including $r^{\bar{a}}_{beta}$). From claim RA-path, there is a nonredundant path from $\bar{c}$ to $\bar{b}$ that does not pass through

Nonredundant Cycle for Case IV-ANTR$_{P3}$ -- Figure 4.6

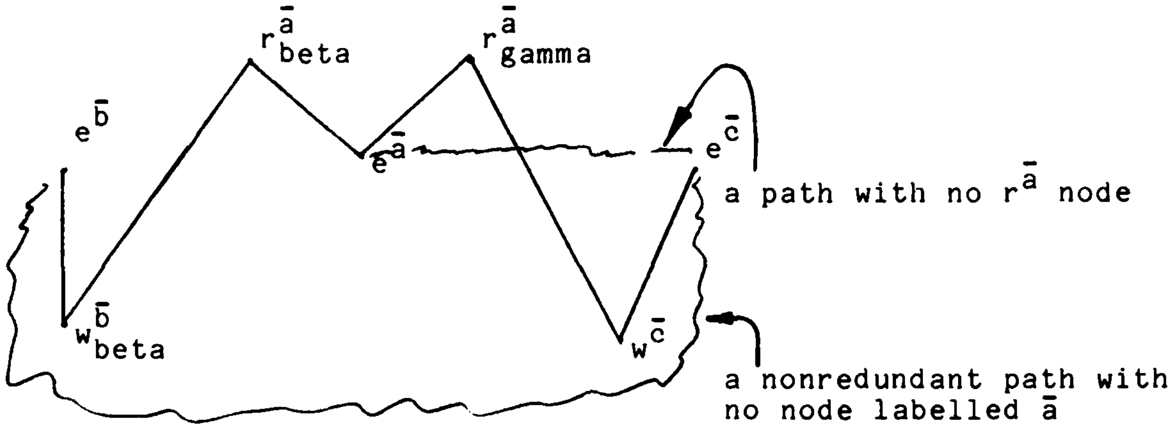
($e^{\bar{c}}$, $r^{\bar{c}}_{gamma}$, $w^{\bar{a}}_{gamma}$, $e^{\bar{a}}$) of Lemma RA



any $\bar{a}$ node. By concatenating the $\bar{a}$ - $\bar{c}$ and $\bar{c}$ - $\bar{b}$ paths and eliminating any redundant subpaths, we obtain a nonredundant path from $e^{\bar{a}}$ to $e^{\bar{b}}$ containing no $r^{\bar{a}}$ node. This path does not pass through the edge ($r^{\bar{a}}_{beta}$, $w^{\bar{b}}_{beta}$), which therefore completes a nonredundant cycle containing ($r^{\bar{a}}_{beta}$, $w^{\bar{b}}_{beta}$) (see figure 4.7). So, by the protocol selection rules, $R^{a'}_{beta}$ must satisfy P3 with respect to $\bar{b}$ at beta. We now continue as in subcase IV ANTR$_{P2}$, yielding the same P3 violation. This completes case IV, and the proof of lemma RA. Q. E. D.

<u>Lemma AW</u> Let LOG$_{given}$ be a log defined over database design DELTA and transaction set TAU such that it is well-formed, satisfies the pipelining rules, and satisfies the protocol selection rules. Then it is not possible that

Nonredundant Cycle for Case IV - ANTR_{P3} - Figure 4.7
($e^{\bar{a}}$, $r^{\bar{a}}_{gamma}$, $w^{\bar{c}}_{gamma}$, $e^{\bar{c}}$) in Lemma RA



SERIALIZE (LOG_{given}, LOG_{out}, LA, RA) returns FALSE with

LA = A^a and RA = W^a_{alpha} for some a in TAU and alpha in DELTA.

<u>Proof</u> Assume that the lemma is false. Then, SERIALIZE returns FALSE and, by lemma S, there is a blocked projection of LOG_{out}, say $\text{LOG}_{blocked}$, whose leftmost and rightmost atoms are A^a and W^a_{alpha} respectively. That is, $\text{LOG}_{blocked}$ is of the form A^a... W^a_{alpha}.

Consider W^a_{alpha}. There are only two kinds of atoms that can be W^a_{alpha}'s conflicting left neighbor: either $W^{a'}_{alpha}$ (which by lemma S part II must be contained in $A^{a'}$) where transclass(a') = transclass(a), or R^b_{alpha} (which may or may not be contained in A^b) such that the three-way intersection of matzn-of-class(transclass(b))(transreadset(b)) and logical^{-1}(transwriteset(a)) and stored-data^{-1}(alpha) is nonempty.

Suppose $W^{a'}_{alpha}$ is the conflicting neighbor. Since $W^{a'}_{alpha}$ precedes and conflicts with W^{a}_{alpha}, by lemma B $W^{a'}_{alpha}$ precedes W^{a}_{alpha} in LOG_{given}. Since E^{a} precedes and conflicts with $E^{a'}$ in $LOG_{blocked}$, by lemma B E^{a} precedes $E^{a'}$ in LOG_{given}. But since transclass(a) = transclass(a'), this violates W-W pipelining. So, $W^{a'}_{alpha}$ cannot be W^{a}_{alpha}'s left conflicting neighbor. Therefore, $LOG_{blocked}$ is either of the form

$$A^{a}\ldots R^{b}_{alpha}\ W^{a}_{alpha} \text{ or}$$
$$A^{a}\ldots A^{b}[\ldots R^{b}_{alpha}\ldots]\ W^{a}_{alpha}.$$

We now show that transclass(a) ≠ transclass(b). Assume not. Since E^{b} follows R^{b}_{alpha} in LOG_{out}, E^{b} follows E^{a} in LOG_{out}. Since transclass(a) = transclass(b) and E^{a} precedes E^{b} in LOG_{out}, by lemma B E^{a} precedes E^{b} in LOG_{given}. Since transclass(a) = transclass(b), R^{b}_{alpha} and W^{a}_{alpha} conflict; since R^{b}_{alpha} precedes W^{a}_{alpha} in $LOG_{blocked}$, by lemma B R^{b}_{alpha} precedes W^{a}_{alpha} in LOG_{given}. This violates W-R pipelining, a contradiction. So, transclass(a) ≠ transclass(b).

Beginning at R^b_{alpha}, scan to the left through $LOG_{blocked}$ until the first atom with a superscript of a" where transclass(a) = transclass(a") is found. Say this is $X^{a''}$. (Note: possibly $X^{a''} = A^a$.) Now, beginning with $X^{a''}$, scan to the right in $LOG_{blocked}$ until the first atom with a superscript of b' where transclass(b') = transclass(b) is found. Say this is $Y^{b'}$. (Note: possibly $Y^{b'} = A^b$ or $Y^{b'} = R^b_{alpha}$.) Thus, $LOG_{blocked}$ is either of the form

$$A^a \ldots X^{a''} \ldots Y^{b'} \ldots R^b_{alpha} W^a_{alpha}$$

or

$$A^a \ldots X^{a''} \ldots Y^{b'} \ldots A^b [\ldots R^b_{alpha} \ldots]\ W^a_{alpha}.$$

Consider the left neighbor of $Y^{b'}$, say Z^c. (Note: possibly $Z^c = X^{a''}$.) By choice of $Y^{b'}$, transclass(b') $\neq$ transclass(c). In the remainder of this proof, let $\bar{a}$ = transclass(a), $\bar{b}$ = transclass(b), and $\bar{c}$ = transclass(c). Recall $\bar{a} \neq \bar{b}$, and $\bar{c} \neq \bar{b}$ by construction.

We now construct a new log, $LOG'_{blocked}$, by excising from $LOG'_{blocked}$ those atoms (if any) separating A^a from $X^{a''}$ and those atoms (if any) separating $Y^{b'}$ from A^b (or R^b_{alpha}). Clearly, the resulting log, $LOG'_{blocked}$, is a blocked projection of $LOG_{blocked}$. $LOG'_{blocked}$ is of the form

$$A^a X^{a''} \ldots Z^c Y^{b'} R^b_{alpha} W^a_{alpha} \text{ or}$$

$$A^a X^{a''} \ldots Z^c Y^{b'} A^b [\ldots R^b_{alpha} \ldots] W^a_{alpha}.$$

Claim AW-path There is a nonredundant path in CG(D) from a node with a label superscripted such that the path passes through no other node labelled $\bar{b}$.

This claim, which follows directly from lemma P, will be applied repeatedly in the remainder of the proof.

We analyze the ways in which Z^c can be in augmented conflict with $Y^{b'}$ and show each possible conflict to be impossible. Since the only assumption made so far is that the lemma is false, the contradiction that Z^c does not conflict with $Y^{b'}$ will prove the lemma.

Z^c can only be in augmented conflict with $Y^{b'}$ due to one of NTR'_1 - NTR'_8, $ANTR_{P2}$, $ANTR_{P2f}$, or $ANTR_{P3}$. Since $\bar{c} \neq \bar{b}$, NTR'_1, NTR'_2, NTR'_3, NTR'_4, NTR'_5, NTR'_6(i), and NTR'_7(ii) cannot be the cause of the conflict. NTR'_3 and NTR'_8 do not apply, because no W can appear in the sublog unless it is contained in an A, by lemma S, part II. The remaining cases are NTR'_6(ii), NTR'_6(iii), NTR'_6(iv), NTR'_7(i), $ANTR_{P2}$, $ANTR_{P2f}$, and $ANTR_{P3}$; They are subsumed by the following cases:

I. $Z^c = A^c$; there is a beta in DELTA such that W^c_{beta} is in A^c and either $Y^{b'} = R^{b'}_{beta}$ or $R^{b'}_{beta}$ is in $A^{b'} = Y^{b'}$ and the three-way intersection of matzn-of-class($\bar{b}$)(transreadset(b')) and logical^{-1}(transwriteset(c)) and stored-data^{-1}(beta) is nonempty.

II. there is a beta in DELTA such that either $Z^c = R^c_{beta}$ or R^c_{beta} is in $A^c = Z^c$; $Y^{b'} = A^{b'}$ and $W^{b'}_{beta}$ is in $A^{b'}$; and the three way intersection of matzn-of-class($\bar{c}$)(transreadset(c)) and logical^{-1}(transwriteset(b')) and stored-data^{-1}(beta) is nonempty.

III. $Z^c = A^c$; $Y^{b'} = A^{b'}$; and the intersection of transwriteset(c) and transwriteset(b') in nonempty.

IV. $Z^c = A^c$ and A^c-$A^{b'}$ are in augmented conflict by $ANTR_{P2}$, $ANTR_{P2f}$, or $ANTR_{P3}$. We analyze each of the four cases in detail.

CASE I ($Z^c = A^c$ contains W^c_{beta}; either $Y^{b'} = R^{b'}_{beta}$ or $R^{b'}_{beta}$ is in $A^{b'} = Y^{b'}$; and $R^{b'}_{beta}$ and W^c_{beta} conflict.)

$LOG'_{blocked}$ is of the form:

$$A^a X^{a'} \ldots A^c[\ldots W^c_{beta} \ldots]\ R^{b'}_{beta} R^b_{alpha} W^a_{alpha}$$

where possibly $R^{b'}_{beta}$ is in $A^{b'}$ and possibly R^b_{alpha} is in A^b. There are two subcases to consider: alpha = beta and alpha $\neq$ beta.

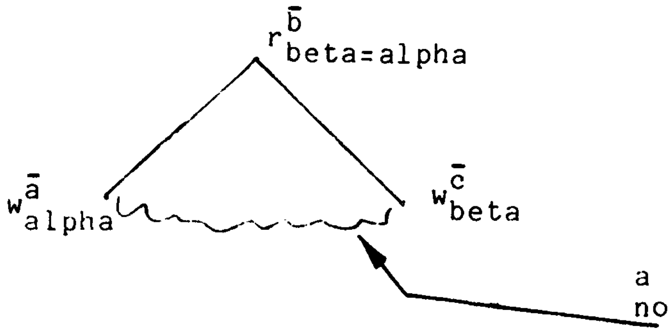
Subcase alpha = beta

From conflicts in $LOG'_{blocked}$, the edges $(r^{\bar{b}}_{beta}, w^{\bar{c}}_{beta})$ and $(r^{\bar{b}}_{alpha}, w^{\bar{a}}_{alpha})$ exist in CG(D). Since alpha = beta,

$(r^{\bar{b}}_{beta}, w^{\bar{c}}_{beta}) = (r^{\bar{b}}_{alpha}, w^{\bar{c}}_{alpha})$.

By claim AW-path, there is a nonredundant path from $w^{\bar{c}}_{alpha}$ to $w^{\bar{a}}_{alpha}$ that does not pass through any node in class $\bar{b}$.

Nonredundant Cycle for Case I (alpha = beta) of Lemma AW   Figure 4.8


$r^{\bar{b}}_{beta=alpha}$
$w^{\bar{a}}_{alpha}$
$w^{\bar{c}}_{beta}$
a nonredundant path with no nodes labelled $\bar{b}$

So, we have the nonredundant cycle noted in figure 4.8.

If c = a, then W^{a}_{alpha} is not unique in the log, a contradiction. So, $c \neq a$.

Either $\bar{c} = \bar{a}$ or $\bar{c} \neq \bar{a}$. Suppose $\bar{c} = \bar{a}$. Then E^{c} and E^{a} conflict and by lemma B, E^{a} precedes E^{c} in LOG_{given}. Since W^{c}_{alpha} and W^{a}_{alpha} conflict, by lemma B W^{c}_{alpha} precedes W^{a}_{alpha} in LOG_{given}. This violates W-W pipelining, a contradiction. Hence $\bar{c} \neq \bar{a}$.

The cycle and the protocol selection rules imply $R^{b'}_{beta}$ must satisfy P2 with respect to $\bar{a}$ and $\bar{c}$ at alpha. By ANTR_{P2}, E^{a} and E^{c} are in augmented conflict and so, by lemma B, E^{a} precedes E^{c} in LOG_{given}. Since W^{c}_{alpha} and $R^{b'}_{beta}$ conflict, by lemma B, W^{c}_{alpha} precedes $R^{a'}_{alpha}$ in LOG_{given}. Since beta = alpha, $R^{b'}_{beta}$ conflicts with R^{b}_{alpha}, so by lemma B, $R^{b'}_{alpha}$ precedes R^{b}_{alpha} in LOG_{given}. Similarly, R^{b}_{alpha} precedes W^{a}_{alpha} in LOG_{given}, so by transitivity, $R^{b'}_{alpha}$ precedes W^{a}_{alpha} in LOG_{given}. But this says that $R^{b'}_{alpha}$ violates P2. Contradiction!

Subcase alpha ≠ beta

From conflicts in $\mathrm{LOG}'_{blocked}$, the edges $(r^{\bar{b}}_{beta}, w^{\bar{c}}_{beta})$ and $(r^{\bar{b}}_{alpha}, w^{\bar{a}}_{alpha})$ exist in CG(D).

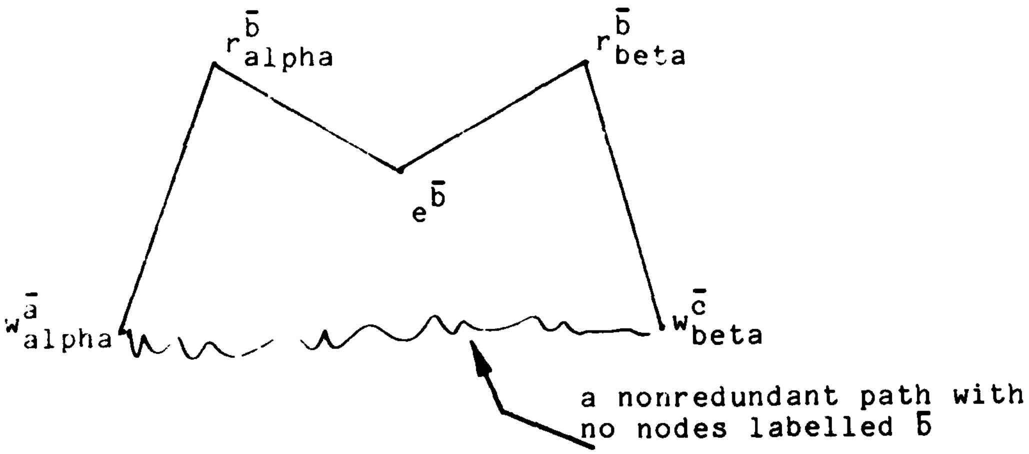
By definition of EDGES_{vert}, the edges $(r^{\bar{b}}_{beta}, e^{\bar{b}})$ and $(r^{\bar{b}}_{alpha}, e^{\bar{b}})$ exist in CG(D).

By claim AW-path, there is a nonredundant path from $w^{\bar{c}}_{beta}$ to $w^{\bar{a}}_{beta}$ that does not pass through any nodes in class $\bar{b}$. So, we have the nonredundant cycle noted in figure 4.9.

This cycle and the protocol selection rules imply that R^{b}_{alpha}, R^{b}_{beta} and $R^{b'}_{alpha}$, $R^{b'}_{beta}$ both have to satisfy P2f with respect to $\bar{a}$ at alpha and $\bar{c}$ at beta.

Nonredundant Cycle for Case I (alpha $\neq$ beta) of Lemma AW

Figure 4.9



We first show that E^b precedes $E^{b'}$ in LOG_{given}. By $ANTR_{P2f}$, E^a and E^c are in augmented conflict, so by lemma B, E^a precedes E^c in LOG_{given}. Since $R^{b'}_{beta}$ conflicts with W^c_{beta}, by lemma B $R^{b'}_{beta}$ follows W^c_{beta} in LOG_{given}. Since E^a precedes E^c, by P2f $R^{b'}_{alpha}$ follows W^a_{alpha} in LOG_{given}. But since R^b_{alpha} precedes W^a_{alpha} in LOG_{given} (by lemma B), R^b_{alpha} precedes $R^{b'}_{beta}$ in LOG_{given}. Hence, by R-R pipelining, E^b precedes $E^{b'}$ in LOG_{given}.

We now need to show $E^{b'}$ precedes E^b in LOG_{given}, to establish a contradiction. To prove this, we first show each of the following properties of $LOG'_{blocked}$:

i. $b \neq b'$;

ii. $R^{b'}_{beta}$ is not in $A^{b'}$;

iii. R^b_{alpha} is not in A^b;

iv. there is no $A^{b''}$ in LOG_{out} between $R^{b'}_{beta}$ and R^{b}_{alpha} with transclass(b") = $\bar{b}$.

This sufficiently restricts the form of $LOG'_{blocked}$ so that we will be able to obtain a contradiction.

i. Suppose b = b'. Then $R^{b'}_{beta} = R^{b}_{beta}$ and R^{b}_{alpha} must satisfy P2f with respect to $\bar{c}$ at beta and $\bar{a}$ at alpha (respectively). By $ANTR_{P2f}$, E^{a} and E^{c} are in augmented conflict, so by lemma B, E^{a} precedes E^{c} in LOG_{given}. Since W^{c}_{beta} conflicts with R^{b}_{beta} and R^{b}_{alpha} conflicts with W^{a}_{alpha}, by lemma B, W^{c}_{beta} precedes R^{b}_{beta} and R^{b}_{alpha} precedes W^{a}_{alpha} in LOG_{given}. But this violates P2f, contradiction! So, b = b'.

ii. Suppose R^{b}_{alpha} is in A^{b}. By part III of lemma S, R^{b}_{beta} is also in A^{b}. Since R^{b}_{beta} conflicts with W^{c}_{beta} and R^{b}_{alpha} conflicts with W^{a}_{alpha}, we obtain the same P2f violation as (i). So, R^{b}_{alpha} is not in A^{b}.

iii. $R^{b'}_{beta}$ is not in $A^{b'}$ by the same argument as (ii).

iv. There is no other $A^{b''}$ in between $R^{b'}_{beta}$ and R^{b}_{alpha} by the same argument as (ii).

From (iv) and part II of lemma S, we conclude that every atom in class $\bar{b}$ in between $R^{b'}_{beta}$ and R^{b}_{alpha} in LOG_{out} is an R that is not contained in an A. Consider one such $R^{b''}_{gamma}$ in this sublog. Each neighbor of $R^{b''}_{gamma}$ must be either another $R^{b'''}_{gamma}$ in $\bar{b}$ or a W^{d}_{gamma} whose writeset intersects the readset of b" as per NTR_3. By lemma S part II, W^d_{gamma} must be in A^d. Hence, the sublog between $R^{b'}_{beta}$ and R^b_{alpha} is of the form:

$A^c R^{b'}_{beta} R^{b1}_{beta} \ldots R^{b2}_{beta} A^d[\ldots W^d_{beta}] \ldots A^e[\ldots W^e_{gamma}] R^{b3}_{gamma} \ldots$

$R^{b4}_{gamma} A^f[\ldots W^f_{gamma}] \ldots$ etc. $\ldots R^{bn}_{alpha} \ldots R^b_{alpha} W^a_{alpha}$.

Consider one pair of R^b's in this sublog that have no R's in class $\bar{b}$ in between them. For example, consider

$R^{b4}_{gamma} A^f[\ldots W^f_{gamma} \ldots] \ldots A^g[\ldots W^g_{delta} \ldots] R^{b5}_{delta}$.

We want to show that E^{b4} precedes E^{b5} in LOG_{given}. Suppose *gamma* = delta. Since R^{b4}_{gamma} conflicts with and precedes R^{b5}_{gamma} in $LOG'_{blocked}$, by lemma b, R^{b4}_{gamma} precedes R^{b5}_{gamma} in LOG_{given}. Suppose gamma ≠ delta. By lemma P, there is a path in CG(D) from $\bar{f}$ = transclass(f) to $\bar{g}$ = transclass(g) that does not pass through any node in $\bar{b}$. From the log, the edges $(r^{\bar{b}}_{gamma}, w^{\bar{f}}_{gamma})$ and $(r^{\bar{b}}_{delta}, w^{\bar{g}}_{delta})$ are in CG(D). By definition of $EDGES_{vert}$, the edges $(r^{\bar{b}}_{delta}, e^{\bar{b}})$ and $(e^{\bar{b}}, r^{\bar{b}}_{gamma})$ are in CG(D). So, we have a nonredundant (P2f) cycle. Since R^{b4}_{gamma} conflicts with W^f_{gamma} and R^{b5}_{gamma} conflicts with W^g_{delta}, by lemma B, R^{b4}_{gamma} precedes W^f_{gamma} and W^g_{delta} precedes R^{b5}_{gamma} in LOG_{given}. By $ANTR_{P2}$ or $ANTR_{P2f}$ (depending on whether or not $\bar{f} = \bar{g}$), E^g and E^f are in augmented conflict, so by lemma B, E^f precedes E^g in LOG_{given}. By P2f, since R^{b5}_{gamma}

follows W^{g}_{delta}, then $R^{b_5}_{gamma}$ must follow W^{f}_{gamma}. Since $R^{b_4}_{gamma}$ precedes W^{f}_{gamma}, $R^{b_4}_{gamma}$ precedes $R^{b_5}_{gamma}$ (by lemma B). Hence, by R-R pipelining $E^{b}4$ precedes $E^{b}5$.

Recall that the log between $R^{b'}_{beta}$ and R^{b}_{alpha} is of the form:

$A^{c}R^{b'}_{beta}R^{b_1}_{beta}\ldots R^{b_2}_{beta}A^{d}[\ldots W^{d}_{beta}\ldots]\ldots A^{e}[\ldots W^{e}_{gamma}\ldots]$
$R^{b_3}_{gamma}\ldots R^{b_4}_{gamma}A^{f}[\ldots W^{f}_{gamma}\ldots]\ldots$etc.

By R-R pipelining $E^{b'}$ precedes $E^{b}2$ in LOG_{given}. By the above argument, $E^{b}2$ precedes $E^{b}3$, so by the transitivity $E^{b'}$ precedes $E^{b}3$. By R-R pipelining and transitivity, $E^{b'}$ precedes $E^{b}4$ in LOG_{given}. By continuing the induction on the number of R's in $\bar{b}$ in between $R^{b'}_{beta}$ and R^{b}_{alpha}, we have that $E^{b'}$ precedes E^{b} in LOG_{given}. This establishes a contradiction, thereby completing case 1 for alpha ≠ beta.

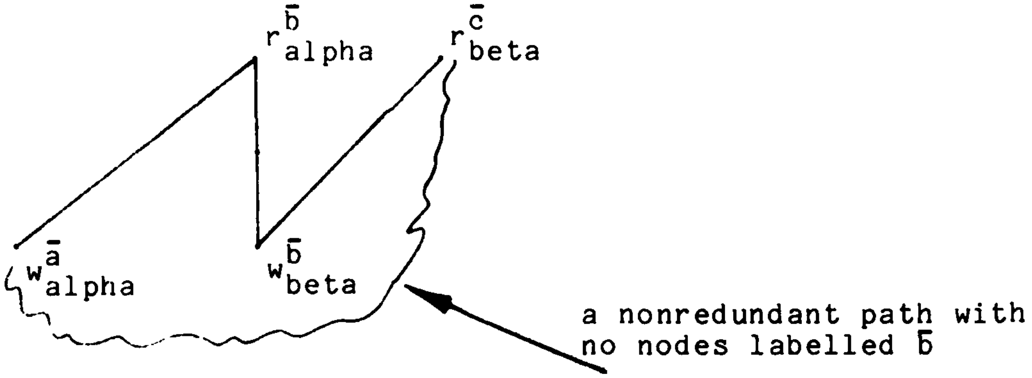
<u>Case II</u> (either $R^{c}_{beta} = Z^{c}$ or R^{c}_{beta} is in $Z^{c} = A^{c}$; $W^{b'}_{beta}$ is in $A^{b'} = Y^{b'}$; $W^{b'}_{beta}$ and R^{c}_{beta} conflict)

$LOG'_{blocked}$ is of the form $A^{a}X^{a'}\ldots R^{c}_{beta}A^{b'}[W^{b'}_{beta}]R^{b}_{alpha}W^{a}_{alpha}$. where possibly R^{c}_{beta} is in A^{c}.

From conflicts in $LOG'_{blocked}$, the edges $(r^{\bar{c}}_{beta}, w^{\bar{b}}_{beta})$ and $(r^{\bar{b}}_{alpha}, w^{\bar{a}}_{alpha})$ are in CG(D).

By claim AW-path, there is a nonredundant path from $w^{\bar{a}}_{alpha}$ to $r^{\bar{c}}_{beta}$ that does not pass through any node in class $\bar{b}$.

Nonredundant cycle for Case II of Lemma AW   Figure 4.10



So, we have the nonredundant cycle noted in figure 4.10.

The cycle and the protocol selection rules imply R^b_{alpha} must satisfy P3 with respect to $\bar{a}$ at alpha. By $ANTR_{P3}$, E^a is in augmented conflict with E^b. Since E^b follows E^a in $LOG'_{blocked}$ (because R^b_{alpha} follows E^a in $LOG'_{blocked}$), by lemma B, E^a precedes E^b in LOG_{given}. Since R^b_{alpha} conflicts with W^a_{alpha}, by lemma b, R^b_{alpha} precedes W^a_{alpha} in LOG_{given}. But this means that R^b_{alpha} violates P3 with respect to $\bar{a}$ at alpha. Contradiction!

Case III ($Z^c = A^c$; $Y^{b'} = A^{b'}$; the intersection of transwriteset(c) and transwriteset(b') is nonempty)

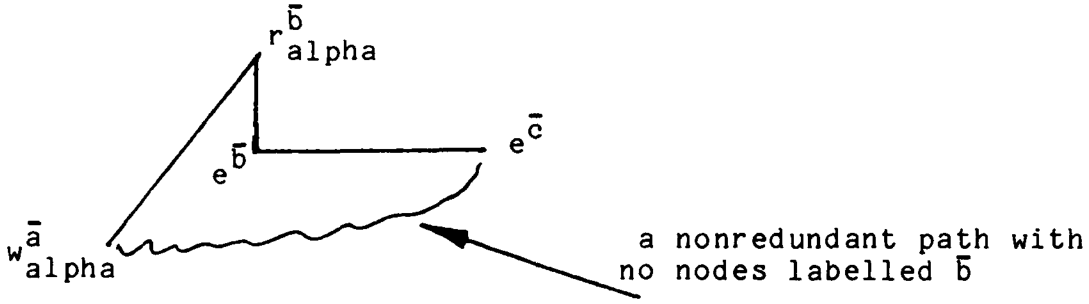
$LOG'_{blocked}$ is of the form

$A^{a}X^{a'}\ldots A^{c}A^{b'}R^{b}_{alpha}W^{a}_{alpha}$

From conflicts in $LOG'_{blocked}$, the edges $(e^{\bar{c}}, e^{\bar{b}})$ and $(r^{\bar{b}}_{alpha}, w^{\bar{a}}_{alpha})$ are in CG(D). By definition of $EDGES_{vert}$ $(e^{\bar{b}}, r^{\bar{b}}_{alpha})$ is in CG(D).

By claim AW-path, there is a nonredundant path from $w^{\bar{a}}_{alpha}$ to $e^{\bar{c}}$ that does not pass through any nodes in class $\bar{b}$.

Nonredundant Cycle for Case III of Lemma AW Figure 4.11



So, we have the nonredundant cycle noted in figure 4.11.

The cycle and the protocol selection rules imply R^{b}_{alpha} must satisfy P3 with respect to $\bar{a}$ at alpha. The remainder of the argument is identical to case II.

<u>Case IV</u> ($Z^{c} = A^{c}$ and A^{c}-$A^{b'}$ are in augmented conflict by $ANTR_{P2}$, $ANTR_{P2f}$, or $ANTR_{P3}$). $LOG'_{blocked}$ is of the form $A^{a}X^{a'}\ldots Z^{c}Y^{b'}R^{b}_{beta}W^{a}_{alpha}$ where possibly R^{c}_{beta} is in A^{b}.

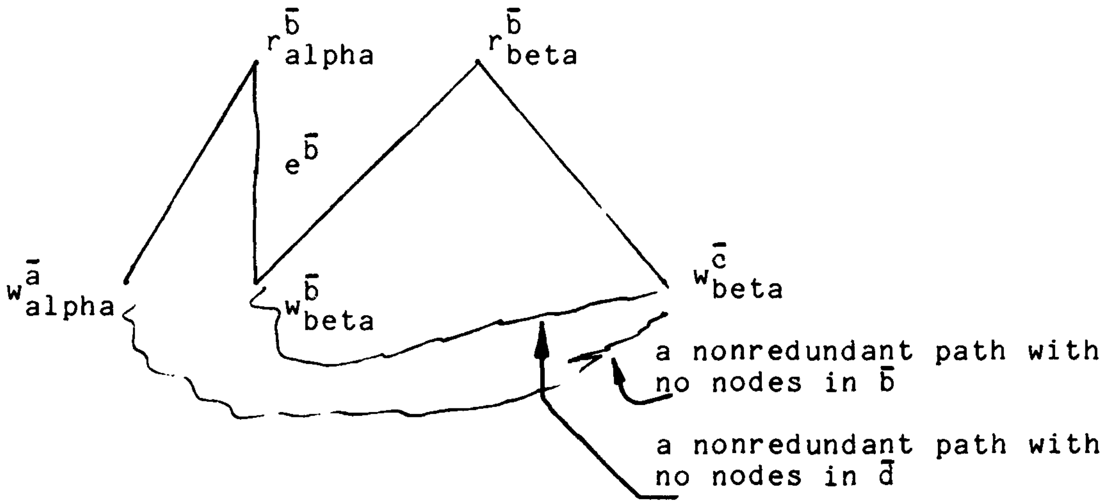
From the log, the edge $(r^{\bar{b}}_{alpha}, w^{\bar{a}}_{alpha})$ is in CG(D). From claim AW-path and the sublog $X^{a'} \ldots Z^{c}$, there is a nonredundant path in CG(D) from a node in $\bar{c}$ to a node in $\bar{a}$ that does not pass through any node in $\bar{b}$. There are now three subcases to consider for each of $ANTR_{P2}$, $ANTR_{P2f}$, and $ANTR_{P3}$ -- the only ways that A^{c}-$A^{b'}$ can be in conflict.

Subcase IV - $ANTR_{P2}$

By $ANTR_{P2}$, there is a class, $\bar{d}$, and a data module, beta, such that there is a nonredundant cycle in CG(D) with the subpath $(w^{\bar{c}}_{beta}, r^{\bar{d}}_{beta}, w^{\bar{b}}_{beta})$. We want to show a subpath

$(w^{\bar{b}}_{beta}, e^{\bar{b}}, r^{\bar{b}}_{alpha}, w^{\bar{a}}_{alpha})$ in a cycle in CG(D). By definition of $EDGES_{vert}$, the edges $(w^{\bar{b}}_{beta}, e^{\bar{b}})$ and $(e^{\bar{b}}, r^{\bar{b}}_{alpha})$ are in CG(D). If $\bar{d} \neq \bar{a}$, then the subpath $(w^{\bar{c}}_{beta}, r^{\bar{d}}_{beta}, w^{\bar{b}}_{beta})$ and the nonredundant path from $\bar{c}$ to $\bar{a}$ that does not pass through $\bar{b}$ are sufficient to complete the cycle (see figure 4.12). If $\bar{d} = \bar{a}$, then the edge $(r^{\bar{a}}_{beta}, w^{\bar{b}}_{beta})$ and the edges $(r^{\bar{a}}_{beta}, e^{\bar{a}})$ and $(e^{\bar{a}}, w^{\bar{a}}_{alpha})$ from $EDGES_{vert}$ are sufficient to complete the cycle (see figure 4.12). So, we have a nonredundant cycle in CG(D) with the subpath $(w^{\bar{b}}_{beta}, e^{\bar{b}}, r^{\bar{b}}_{alpha}, w^{\bar{a}}_{alpha})$. Hence, by the protocol selection rules, $r^{\bar{b}}_{alpha}$ must satisfy P3 with respect to $\bar{a}$ at alpha. However P3 is violated by R^{b}_{alpha} in the following way: By $ANTR_{P3}$ and the cycle, E^{a} and E^{b}

Nonredundant Cycles for Subcase IV -- ANTR$_{P2}$ of Lemma AW Figure 4.12



are in augmented conflict. So, since E^a precedes R^b_{alpha} which precedes E^b in $LOG'_{blocked}$, by lemma B, E^a precedes E^b in LOG_{given}. Since R^b_{alpha} conflicts with W^a_{alpha}, R^b_{alpha} precedes W^a_{alpha} in LOG_{given}. This violates P3, a contradiction.

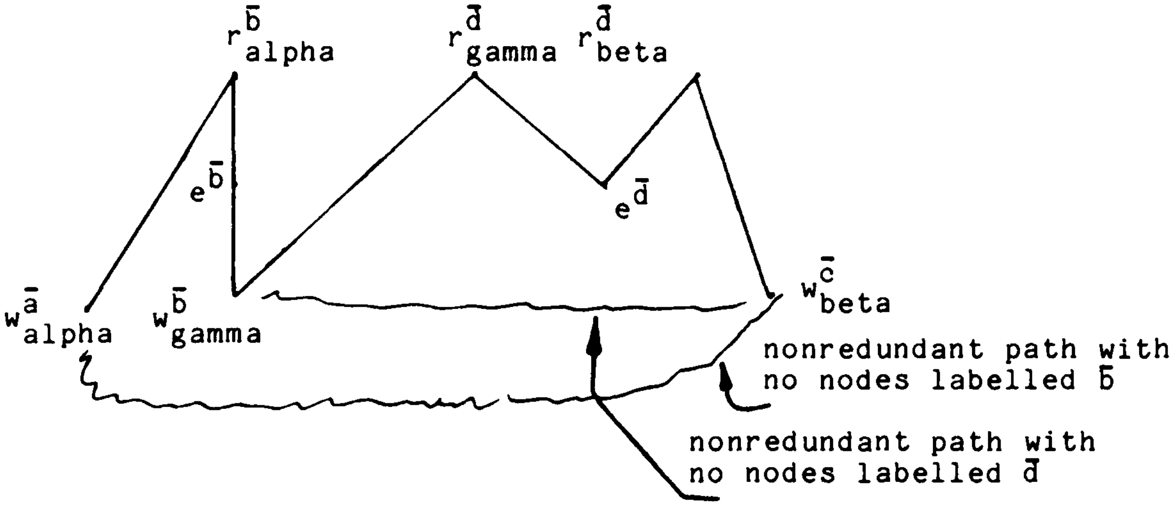
Subcase IV - ANTR$_{P2f}$

By ANTR$_{P2f}$, there is a class, $\bar{d}$, and two distinct data modules, beta and gamma, such that there is a cycle in CG(D) with a subpath

$(w^{\bar{c}}_{beta}, r^{\bar{d}}_{beta}, e^{\bar{d}}, r^{\bar{d}}_{gamma}, w^{\bar{b}}_{gamma})$.

We can proceed exactly as in Subcase IV - ANTR$_{P2}$ yielding the same P3 violation (see figure 4.13).

Nonredundant Cycles for Subcase IV -- ANTR$_{P2f}$ of Lemma AW Figure 4.13



Subcase IV - ANTR$_{P3}$

By ANTR$_{P3}$, there is a data module beta, such that there is a cycle in CG(D) with either the subpath

$(e^{\bar{c}}, w^{\bar{c}}_{beta}, r^{\bar{b}}_{beta}, e^{\bar{b}})$

or

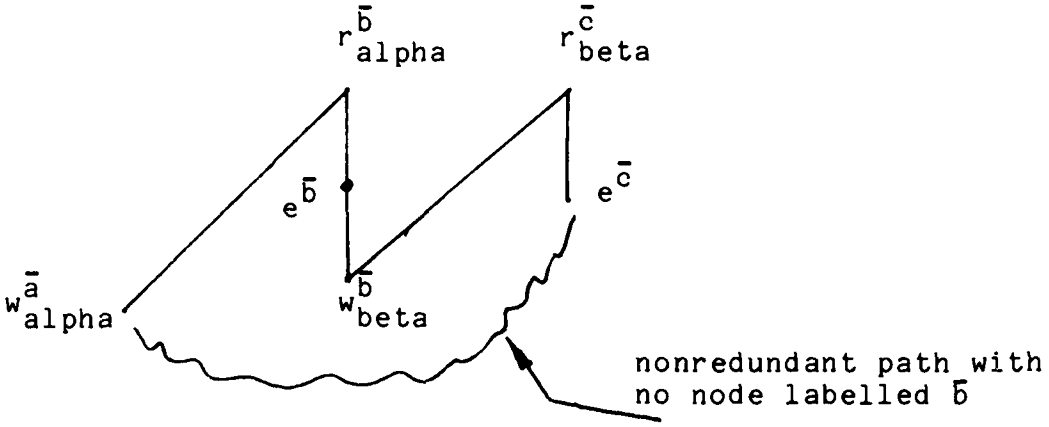
$(e^{\bar{c}}, r^{\bar{c}}_{beta}, w^{\bar{b}}_{beta}, e^{\bar{b}})$.

We treat each subpath as a separate case.

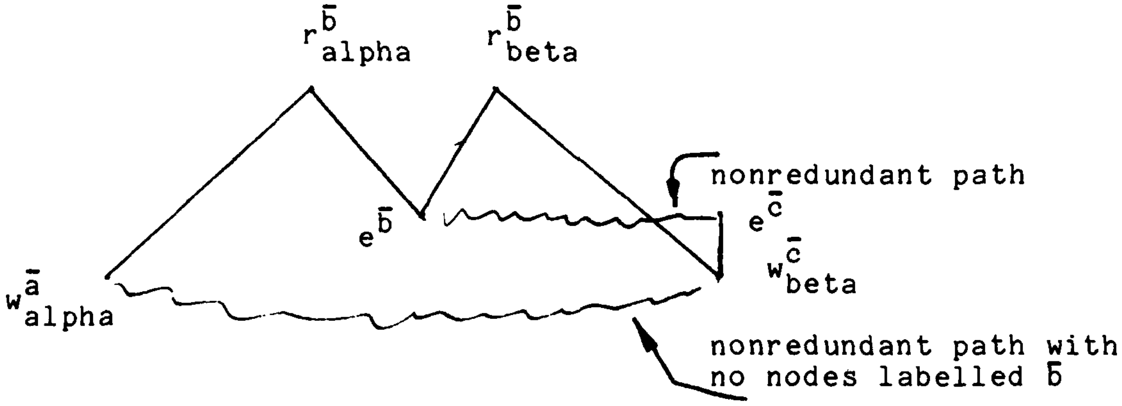
Subcase IV - ANTR$_{P3}$ - $(e^{\bar{c}}, r^{\bar{c}}_{beta}, w^{\bar{b}}_{beta}, e^{\bar{b}})$

We can deduce that the edge $(r^{\bar{c}}_{beta}, w^{\bar{b}}_{beta})$ is in CG(D), and by definition of EDGES$_{vert}$ $(w^{\bar{b}}_{beta}, e^{\bar{b}})$ and $(e^{\bar{b}}, r^{\bar{b}}_{alpha})$ are in CG(D) (see figure 4.14a). As in subcase IV

Nonredundant Cycles for Subcase IV -- Figure 4.14
$ANTR_{P3}$ of Lemma AW



(a)



(b)

- $ANTR_{P2}$, R^{b}_{alpha} must satisfy P3, but violates P3 in LOG_{given}, a contradiction.

Subcase IV - $ANTR_{P3}$ - $(e^{\bar{c}}, w^{\bar{c}}_{beta}, r^{\bar{b}}_{beta}, e^{\bar{b}})$

By these augmented edges and $ANTR_{P3}$, there is a path from $e^{\bar{b}}$ to $e^{\bar{c}}$ that does not pass through $r^{\bar{b}}_{gamma}$ (including $r^{\bar{b}}_{alpha}$). This completes the cycle (see figure 4.14b) and we can proceed to a P3 violation as in Subcase IV - $ANTR_{P2}$. Q. E. D.

5. Protocol P4, A Cycle-Breaking Protocol

5.1 Motivation for a Cycle-Breaking Protocol

From a logical standpoint, {P1, P2, P2f, P3} are a sufficient set of mechanisms to correctly execute all transactions in all classes. That is, with these protocol schemas alone, serial reproducibility can be guaranteed. However, from an efficiency standpoint, these protocol schemas have a serious problem. The problem is that a single class can cause cycles in the conflict graph and thereby force many classes to run expensive protocols, 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 C_{total}, that has a read-set and write-set that includes the entire logical database. Every conceivable transaction can fit into C_{total}, so this apparently solves the problem. But the cost is enormous, for C_{total} induces a two-class cycle with every other class in the system. So, every class has to run P3 against C_{total}, and C_{total} has to run P3 against every other class. Since P3 is the most expensive protocol schema, this is an unfortunate state of affairs. It is especially unfortunate because transactions will rarely need to execute in C_{total}, since most transactions fit into other less expensive classes. So, C_{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. Clearly, the problem of proliferation of cycles is especially acute in C_{total}. However, other classes with smaller read-sets and write-sets may manifest the same problem.

To alleviate these problems we introduce a new protocol schema, called P4. the purpose of P4 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 (and, therefore, run P1 with respect to that cycle). In other words, the protocol selection rules only apply to cycles that do not contain a class that runs P4.

That we need a P4 cycle-breaking protocol is clear. In the remainder of this section, we discuss how such a protocol can be implemented.

5.2 Overview of P4

One way to implement P4 is to shut off the system so that no new transactions can be introduced. After all outstanding WRITE messages have been processed, then the system has quiesced. Assuming every class was running the correct protocol, the log (up to this point) should be serially reproducible. Now, we run the P4 transaction. After all of this transaction's WRITE messages arrive and are processed, it is safe to start up the system again, allowing new transactions to be run. What we have done is turn off the system, wait until a serially reproducible database state is reached, run the P4 to completion, and then start up the system again. The P4 transaction partitions the log in half, and each half is serially reproducible (since the other transactions are running the correct protocols).

The degradation of performance that results from shutting off the system, even temporarily, is likely to be severe. So, the above P4 algorithm is unacceptable. To weaken it, we observe that the P4 need only synchronize against other classes that lie on the cycle including the P4 class, since only classes on cycles can cause non-serially reproducible logs. Also, we note that even these classes need not quiesce completely before the P4 runs. All that we need is the weaker condition that the log be equivalent to some log in which all of the classes have quiesced before the P4. With these observations in mind, a much weaker P4 can be derived.

5.3 Implementation of P4

Protocol schema P4 differs structurally from the other protocol schemas in two ways: First, P4 requires some direct communication between transaction modules. By this communication, the P4 class requests that certain other transaction modules perform synchronization to avoid conflicting with the P4 transaction. Second, P4 requires an augmented form of read condition. Recall that a standard read condition is a pair <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 received. It does not require that no messages from classes timestamped greater than "timestamp" be received (as in standard read conditions).

To implement P4, we use three additional types of messages that are sent from TM's to TM's (not from TM's to DM's). A P4-ALERT message is sent from a P4 class to some other class. A P4-ALERT message includes the P4 class's name and timestamp as its parameters. A class responds to a P4-ALERT with either a P4-ACC (i.e., an acceptance) or a P4-REJ (i.e., a rejection).

To run a transaction t_{P4} in the P4 class c_{P4}, one performs the following steps:

1. Choose a timestamp for t_{P4}, say TS_{P4}.

2. Send a message P4-ALERT (TS_{P4}) to every class that lies on the cycle in CG(D).

3. Wait for the P4-ACC's to be received from all classes to which a P4-ALERT was sent. If a P4-REJ is received, then restart the protocol from step 1.

4. Construct the READ messages for t_{P4}. For each data module, alpha, to which a read message will be sent, include a condition $<TS_{P4},C_i>$ for each class C_i such that the edge $(r^{cP4}_{alpha}, w^{c_i}_{alpha})$ lies on the cycle.

When a transaction module receives a P4-ALERT(t_{P4}, TS_{P4}) for a particular class, c_i, it performs the following steps:

1. If the class has run or begun running a transaction with a timestamp greater than TS_{P4}, then respond to c_{P4} by sending P4-REJ. Otherwise, send P4-ACC and do not run another transaction in c_i timestamped earlier than TS_{P4}.

2. For the next transaction run in c_i, for each datamodule alpha and each class c_j such that edge $(r^{c_i}_{alpha}, w^{c_j}_{alpha})$ lies on a cycle with C_{P4}, include the condition $<mintime=TS_{P4}, c_j>$, in the READ message to DM_{alpha}. These conditions are in addition to those normally included by c_i in its read messages.

It should be emphasized that (2) is only performed for the first transaction executed in C_i with timestamp greater than TS_{P4}. Later transactions in c_i can run P1 again, with respect to this P4 cycle.

5.4 Proof of Correctness for Protocol P4

A proof of serial reproducibility incorporating protocol P4 has been developed and will appear in a later Technical Report.

A. Update Semantics and Fragment Definition

A.1 Insertion / Deletion Semantics

The basic update operation in SDD-1 is a WRITE message that changes the value of existing data items (see Section 2). To enable insertions and deletions using this write message format, we augment each relation by a special boolean domain named "Existence-bit" (abbr. E-bit). From a logical viewpoint, every TID value is "present" in the sense that it can be referenced. We distinguish between TIDs that label real tuples and those that label an empty slot for a tuple by the E-bit: If E-bit=1 then the tuple exists in the relation; otherwise, the tuple does not exist.

Using this model, we define four operation on relations: RETRIEVE, DELETE, INSERT, and CONDITIONAL INSERT. These are the kinds of operations that we expect users will want to perform on SDD-1 relations, and they essentially correspond to standard query language commands. RETRIEVE selects a portion of a relation to be read; it only reads tuples with the E-bit = 1. DELETE simply sets E-bit = 0 for the tuples to be deleted. INSERT sets E-bit = 1 for the TID values for tuples to be inserted. CONDITIONAL INSERT inserts TIDs provided they do not already exist, by checking that E-bit = 0 before setting E-bit = 1. This latter operation may be needed to avoid overwriting already existing tuples.

The E-bit domain must be used in determining the read-set and write-set for a class of transactions. Insert and delete operations are in conflict precisely insofar as they both use the E-bit domain, and this conflict may require adding some edges to the conflict graph.

A.2 Fragment Updates

Recall the definition of logical fragments. First partition the relation according to a set of restrictions and then define each logical fragment to be a projection of a partition on the TID domain and one other domain.

That fragments are defined logically creates certain problems on updates. If the restriction qualification that defines a fragment uses domain D, say, then updates to a D-value may cause a tuple to "migrate" from one partition (and hence fragment) to another. For example, if the EMPLOYEE relation is partitioned based on the DEPARTMENT domain, then moving an employee to a new department causes a tuple migration to a different partition. Since fragments in different partitions are stored as independant files, often at different data modules, the tuple migration requires WRITE messages to delete the tuple from one fragment and add it to another. When determining the read-sets and write-sets of a transaction class, potential tuple migrations must be considered, since additional WRITE messages may be required to maintain the consistency of the fragment within its definition.

References

[BERNSTEIN et al]

Bernstein, P.A.; Goodman, N; Rothnie, J.B.; and Papadimitriou, C.A. "Analysis of Serializability in SDD-1: A System for Distributed Databases (The Fully Redundant Case)", First International Conference on Computer Software and Applications (COMPSAC 77), IEEE Computer Society, Chicago Illinois, November 1977. (Also available from Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139 as Technical Report No. CCA-77-05).

[CODD]

Codd, E. F., "A Relational Model of Data for Large Shared Data Banks" CACM, 13 (1970), pp. 377-387.

[LAMPSON and STURGIS]

Lampson, B.; and Sturgis, H. "Crash Recovery in a Distributed Data Storage System", unpublished paper, Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto California 94304, 1976.

[ESWARAN et al]

Eswaran, K.P.; Gray, J.N.; Lorie, R.A.; Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database System", CACM, Vol. 19, No. 11, November 1976.

[GRAY et al]

Gray, J.N.; Lorie, R.A.; Putzolu, G.R.; Traiger, I.L. "Granularity of Locks and Degrees of Consistency in a Shared Database", IBM Research Report RJ1654, San Jose California, 1975.

[HAMMER and SHIPMAN]

Hammer, M.M.; and Shipman, D.W. "Resiliency Mechanisms in SDD-1" Technical Report in progress. Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139.

[HEWITT]

Hewitt, C.E. "Protection and Synchronization in Actor Systems", Artificial Intelligence Laboratory Working Paper No. 83, Massachusetts Institute of Technology, November 1974.

[MANNA]

Manna, Z. Mathematical Theory of Computation, McGraw-Hill, New York, 1974.

[MARILL and STERN]

Marill, Thomas; and Stern,D.H. "The Datacomputer: A Network Utility", Proceedings AFIPS National Computer Conference, AFIPS Press, Vol. 44, 1975.

[METCALFE]

Metcalfe, R.M. Packet Communication, Technical Report No. TR-114, Laboratory for Computer Science, M.I.T., Cambridge Massachusetts, December 1973.

[PAPADIMITRIOU et al]

Papadimitriou, C.A.; Bernstein, P.A.; and Rothnie, J.B. "Some Computational Problems Related to Database Concurrency Control", Conference on Theoretical Computer Science, University of Waterloo, Waterloo Ontario, August 1977.

[ROTHNIE and GOODMAN]

Rothnie, J.B.; and Goodman, N. "An Overview of the Preliminary Design of SDD-1: A System for

Distributed Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley California, May 1977. (Also available from Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139, as Technical Report No. CCA-77-04).

[ROTHNIE et al]

Rothnie, J.B.; Goodman, N.; and Bernstein, P.A. "The Redundant Update Algorithm of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", First International Conference on Computer Software and Applications (COMPSAC 77), IEEE Computer Society, Chicago Illinois, November 1977. (Also available from Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139, as Technical Report No. CCA-77-02).

[THOMAS]

Thomas, R.H. "A Solution to the Update Problem for Multiple Copy Databases Which Uses Distributed Control", BBN Report No. 3340, Bolt Beranek and Newman Inc., Cambridge Massachusetts, July 1975.

[WONG]

Wong, E. "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley California, May 1977. (Also available from Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139, as Technical Report No. CCA-77-03).