Managing Distributed Databases in Partitioned Networks

David D. Wright

83-572
September 1983

Department of Computer Science
Cornell University
Ithaca, New York 14853

Managing Distributed Databases in Partitioned Networks

A Thesis
Presented to the Faculty of the Graduate School
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
David Dixon Wright
January 18, 1984

Biographical Sketch

David D. Wright was born November 29, 1956 in Madison, Wisconsin. In September 1974 he entered Dartmouth College in Hanover, New Hampshire, from which he graduated in June 1978, earning a B.A. degree *magna cum laude* with Distinction in mathematics. He entered Cornell University in the fall of 1978 and received a masters degree in Computer Science in January 1982. He is now working for Prime Computer, Inc., Framingham, Massachusetts, as a Senior Software Engineer in operating systems.

To my parents

Acknowledgements

My greatest thanks are reserved for my chairman, Dale Skeen. His enthusiasm, willingness to talk, and inexhaustible supply of ideas have been invaluable. He has constantly forced me to consider not just what I am saying, but how I am saying it. If not for him, this thesis would not exist, and much in it that is good can be traced to his influence.

I would also like to thank Fred Schneider, whose dogged insistence on precision and clarity in writing has been most helpful, and who supplied me with detailed comments on my first draft. Bob Constable required me to delve more deeply into theoretical issues, and Lee Schruben provided some important comments on simulation. Dick Conway was kind enough to discuss some of my simulation results and suggest improvements. Bengt Aspvall, John Gilbert, and Mark Krentel helped me understand NP-completeness and read early versions of some of my work on optimistic commit. Ken Birman and Tommy Joseph also read early versions of some of this material and provided helpful comments.

Finally, I'd like to thank my many friends in the department for making my years at Cornell pleasant. This research was supported in part by NSF Grant MCS-8103605.

Table of Contents

1. Introduction 1
1.1. Network Partitioning 1
1.2. Formal Specification of the Problem 1
1.2.1. Previous Work 3
2. Concurrency Control and Serializability 6
2.1. Concurrency Control 6
2.2. Serializability Theory 7
2.3. Some Particular Graphs 10
3. Conservative Strategies 14
3.1. Basics of the Conservative Approach 14
3.1.1. Some Properties of CCGs 14
3.2. Description of the Problem 16
3.3. Using Transaction Classes 17
3.3.1. Assigning Classes to Partitions 18
3.3.2. Resolving Conflicting Assignments 19
3.3.3. Finding Acyclic CCGs 20
3.3.4. Maintaining CCGs During Partitioning 24
3.4. Using Two Versions 25
3.4.1. Verifying Protocols 33
3.5. More than Two Versions 34
3.5.1. The Complexity of Precedence Edge Reversal 41
3.6. Comparison of the Rules 43
3.7. Using Pseudo-Partitioning in Concurrency Control 45
3.7.1. Implementing Rule 3 50
3.8. Improving Performance under Pseudo-Partitioning 51
3.9. Comparison with Other Proposals 55
3.9.1. Hierarchical Database Decomposition 55
3.9.2. SDD-1 57
4. Optimistic Strategies 58
4.1. Basics of the Optimistic Approach 58
4.1.1. Using Optimistic Commit 59
4.2. Some Properties of Serialization Graphs 60
4.3. The Computational Complexity of PARTITION MERGE 64
4.4. Heuristic Solutions 68

4.4.1. Approximating PARTITION MERGE 70
4.4.2. Approaches to Backout 71
4.4.3. Hybrid Strategies 74
4.5. Extending the Model 75
4.5.1. Deletion of Data-items 76
4.5.2. Addition of Data-items 76
4.5.3. READSETs containing WRITESETs 77
5. Modeling Optimistic Commit 80
5.1. Simulation of Backout Strategies 80
5.2. The Probabilistic Model 81
5.2.1. Second-level Dependencies 84
5.2.2. Partially Replicated Databases 85
5.2.3. Non-uniform Data References 85
5.2.4. Non-overlapping WRITESETs 88
5.3. Evaluating the Probabilistic Model 89
6. Conclusions 91
6.1. The Proposed Systems 91
6.2. Implementation and Feasibility 92
6.3. Future Work 95

Table of Figures

Figure 1.1 2
Figure 2.1 12
Figure 2.2 13
Figure 3.1 16
Figure 3.2 21
Figure 3.3 23
Figure 3.4 23
Figure 3.5 26
Figure 3.6 26
Figure 3.7 28
Figure 3.8 35
Figure 3.9 35
Figure 3.10 36
Figure 3.11 37
Figure 3.12 42
Figure 3.13 44
Figure 3.14 45
Figure 3.15 47
Figure 3.16 50
Figure 4.1 59
Figure 4.2 62
Figure 4.3 63
Figure 4.4 73

Index of Defined Terms

ITEMS 1
data-item 1
site 1
transaction 2
partition 2
class 2
conservative 3
primary copy 3
token passing 3
voting 3
weighted voting 3
optimistic 4
serial 6
serializable 6
version 9
multi-version serialization graph 9
multi-partition serialization graph 10
class conflict graph 12
mp-cycle 15
class assignment rule 17
conflict resolution rule 18
k-cycle 18
mp-feedback vertex set 20
version assignment rule 25
component graph 27
original version 28
item-closed 32
read quorum 33
write quorum 33
pseudo-partition 46
revised version 48
dependency set 58
dep(v) 58
transaction backout set 59

weight 59

CHAPTER 1

Introduction

1.1. Network Partitioning

A distributed database (DDB) offers potentially improved reliability and data accessibility over a database implemented on a single computer. Obtaining these benefits, however, requires that many difficult technical problems be solved. For example, if data-items are replicated at multiple sites, the database system must ensure that the values at each site agree. Many strategies have been proposed to cope with this problem; however, few of them are capable of managing the problem of partitioning. A ***partitioning*** of a DDB occurs when the DDB is divided into two or more subsets such that no member of one subset can communicate with any member of another. When a system becomes partitioned, there are two possibilities. The system can shut off activity and wait for the connections to be reestablished, or it can adjust its behavior and attempt to continue running. Since a major goal of distributed systems is to be resilient to failures, the first alternative is clearly undesirable. This thesis will explore several approaches to the second alternative.

1.2. Formal Specification of the Problem

A DDB consists of a set *ITEMS* of ***data-items***, $\{d_1, \ldots, d_M\}$ and a set of ***sites***, $\{s_1, \ldots, s_N\}$ that are connected by communication links. The system need not be completely connected, but we do assume that during normal operation any site can send a

message to any other site. Copies of the data-items may be stored at more than one site. The values of data-items are manipulated by ***transactions***, which have the property of causing all data-items whose values they change to be updated atomically as seen by any other transaction. Every transaction T has associated with it two sets, $READSET(T)$ and $WRITESET(T)$, which are, respectively, the sets of items read and written by T. We assume that $WRITESET(T) \subseteq READSET(T)$, and that data are fully replicated, i.e. that there is a copy of every data-item at every site[1].

A ***partition*** of the database is a maximal subset of communicating sites. Thus the entire database, when it is functioning normally, forms a single partition. Site or communication link failures may separate the DDB into more than one partition, whereas site or link recoveries may cause partitions to be merged, possibly requiring processing to maintain consistency. The system is said to be ***partitioned*** at any time that it is composed of more than one partition.

In many systems, transactions do not take arbitrary forms; instead, the activities of the system fall into a few categories. In such systems, transactions can be grouped into ***classes*** in the sense of [BSR80]. Classes can be conveniently described by predicates on their *READSET*s (Fig. 1.1).

$$READSET(X_i)=\{a,b,c, \text{ where } c>10\}$$
$$WRITESET(X_i)=\{a,b\}$$

Figure 1.1. Specification of class X_i.

For any transaction, the function $CLASS$: Transactions $\rightarrow$ Classes gives the class of that transaction.

---

[1] Relaxing these assumptions will be discussed in Chapters 4 and 5.

Note that we do not consider the problem of *detecting* partitioning, only of managing it in cases where it has been detected (say, by the communications subsystem). Detecting partitioning, particularly in systems where components can fail and recover rapidly, is difficult and deserving of further study.

1.2.1. Previous Work

Strategies to allow a partitioned DDB to continue functioning fall into two categories. The first category, which we call *conservative*, covers those strategies that guarantee that the values produced by transactions in each partition will always be compatible with the values produced in all other partitions. This category includes primary copy methods [AlDa76, Ston79], token passing [Elli77, LeLa78], and various voting schemes [Giff79, Thom79, EaSe83]. Under a primary copy system, one copy of a data-item is designated as the *primary copy*. A site is allowed to access an item iff the site is part of the partition that possesses the primary copy of the item. A *token passing* scheme is a generalization of primary copy in which the primary copy is the one at the site that holds the "token" for that object. The token is passed from site to site as requests are submitted. A different generalization of primary copy is *voting*. Under a voting scheme, each copy of an item is assigned a vote. A site may access an object iff its partition has a majority of the votes for that object. [Giff79] proposed *weighted voting* as an extension of voting in which different copies can have different numbers of votes. Token passing and weighted voting could be combined, resulting in a system where votes can move from copy to copy. Once communication is restored, the partitions that were able to do updates can report to the other partitions all changes they made to the database, allowing all copies to be brought up to date.

Currently proposed conservative methods do not allow a data-item to be read in one partition and written in another. Some strategies, e.g. primary copy, are still more restrictive and allow an item to be referenced in at most one partition. Often, such restrictions are unduly severe. One major result of this thesis (Chapter 3) is more powerful schemes that subsume previous proposals and allow a greater variety of transactions to be run during partitioning. Our methods are based on dividing transactions into classes and using multiple versions of data-items. We also show how these concepts can be applied to concurrency control and give methods for increasing their efficiency.

A drawback of conservative schemes is that failures may occur in such a way that no partition can perform updates. For example, in a voting scheme, no partition may have a majority. The second major category of partitioning strategies allows each partition to perform updates that might conflict with updates in another partition. Such strategies are called *optimistic* because they assume that the number of conflicts will be small (and preferably zero). When communication between two or more partitions is restored, the partitions compare their actions during partitioning, detect conflicts, and back out (undo) transactions until no conflicts remain. This approach was proposed by Davidson and Garcia-Molina [DaGa81, Davi82]; a simple form has been implemented in the LOCUS system [Park81]. In Chapter 4, we will analyze the computational complexity of backing out transactions under the optimistic approach and the possibilities for efficient solutions to the problem. Chapter 5 presents a probabilistic model of optimistic commit. This model is used to estimate the number of transactions that would be backed out.

Before presenting our new results, however, we must first discuss some preliminaries. The next chapter is a review of serializability, which forms the theoretical underpinnings of the work in this thesis. We will present some specific applications of serializability to the problems of partitioning. Once this is done, we will be able to turn our attention to new results.

CHAPTER 2

Concurrency Control and Serializability

2.1. Concurrency Control

A goal of database systems is to allow the user to assume that the system executes each of his requests in isolation and one at a time. Such an execution (transactions run one at a time) is called *serial*. Although the database could function this way (only one transaction in execution at any point), this approach makes inefficient use of computing hardware. For this reason, most database systems execute many transactions concurrently, interleaving their operations. It is up to the database system to interleave transactions so that the result is *serializable;* that is, the result is equivalent to running the transactions serially in some order. To illustrate the problems in doing this, suppose two transactions, T1 and T2, are submitted, each of which increments the value of some data-item a. The implementation of such a transaction would be something like the following: READ(a); INCREMENT(a); WRITE(a). Running two such transactions should increment a twice; if the interleavings of the two transactions are not controlled, however, this may not be the result. Suppose T1 and T2 are initiated simultaneously and run in lock-step. This execution will only increment a once, since both T1 and T2 will read the initial value of a. Such results are called *non-serializable*, since they do not correspond to any serial execution of T1 and T2.

Many protocols have been proposed for guaranteeing serializable behavior in distributed systems; surveys can be found in [BeGo81, Kohl81]. For our purposes, it does not

matter what concurrency control protocol is used within a partition as long as it is correct. Our concern is with merging the actions of multiple partitions, each of which is running a correct concurrency control protocol. (It is not necessary to assume that the same protocol is being used in all partitions.)

Under partitioning, simply running a correct concurrency control protocol in each partition does not guarantee that the actions of all partitions can be combined to give a serializable result. Suppose in the above example that transaction T1 were run in one partition and T2 in another. Although each partition in isolation is correct, neither transaction's output is a value consistent with running both transactions in sequence. Again, the result is not serializable.

Formally specifying and verifying correctness of concurrency control protocols is the domain of ***serializability theory***, to which we now turn.

2.2. Serializability Theory

Serializability theory typically involves describing an execution of a system by a directed graph in which nodes are transactions and edges describe transaction interaction. For example, if transaction Tb reads a value written by transaction Ta, there would be an arc (Ta, Tb) in the digraph. If the graph is acyclic, the execution is serializable. We shall consider two methods of analyzing serializability, and, in the next section, specialize them to obtain the particular graphs we need. We describe transactions by their $READSET$s and $WRITESET$s alone; that is, we do not consider the ***values*** of the items involved.

One general model of transaction execution has been proposed by Papadimitriou [Papa79]; our next definition is based on his. This definition is more general than we will require; for instance, it allows histories to be specified with the read action of a transaction not immediately followed by its write action.

Definition 2.1: A ***history*** is a quintuple $h=(n,\pi,ITEMS,READSET,WRITESET)$, where n is a positive integer; π is a permutation of the set $\Sigma_n=\{R_1,W_1,\cdots,R_n,W_n\}$ —that is, a one-to-one function $\pi:\Sigma_n\rightarrow\{1,2,...,2n\}$— such that $\pi(R_i)<\pi(W_i)$ for $i=1,2,...,n$; $READSET$ is a mapping from $\{R_1,...,R_n\}$ to 2^{ITEMS}; and $WRITESET$ is a mapping from $\{W_1,...,W_n\}$ to 2^{ITEMS}. Each pair (R_i,W_i) will be called a ***transaction*** T_i. For transaction T_i, define $READSET(T_i)\equiv READSET(R_i)$ and $WRITESET(T_i)\equiv WRITESET(W_i)$. A history is ***serial*** if $(\forall i\in[1..n]$: $\pi(R_i)=\pi(W_i)-1$. ❐

A history $h=(n,\pi,V,READSET,WRITESET)$ is used to construct a digraph $D(h)$. The nodes of $D(h)$ are the transactions T_i of h, and the pair (T_i,T_j) is an arc in $D(h)$ iff one of the following holds:

(a) $READSET(T_i)\cap WRITESET(T_j)\neq\phi$ and $\pi(R_i)<\pi(W_j)$

(b) $WRITESET(T_i)\cap READSET(T_j)\neq\phi$ and $\pi(W_i)<\pi(R_j)$

(c) $WRITESET(T_i)\cap WRITESET(T_j)\neq\phi$ and $\pi(W_i)<\pi(W_j)$.

[Papa79] shows that if $D(h)$ is acyclic, the history is serializable. The converse does not hold unless restrictions are placed on transactions; one such restriction is a requirement that $WRITESET(T_i)\subseteq READSET(R_i)$, which we have said we are observing.

To adapt Papadimitriou's model for use in multi-partition environments, we must show how it can be applied to transactions from more than one partition. We will assume that each partition maintains serializability within the partition, so the histories of each partition are assumed to be equivalent to serial schedules. If these histories are compatible, it must be possible to find a combined history such that transactions read the same values in the combined history that they did in the individual histories. A model that enforces this restriction is presented in the next section.

To the user of a database system, a data-item is an object having a single value. To the system, however, the item has a series of values. If older values are retained, the system has a choice of which value to give to a transaction in response to a request for the item. These different values are known as ***versions*** ([Reed78]; the following discussion is based on [BeGo82]. See also [StRo81].)

A version of an item is labeled (subscripted) by the index of the transaction that wrote it. Suppose that for each item, we totally order all its versions. Let $\ll$ be the union of all these total orders. Then given a log L and $\ll$, the ***multi-version serialization graph***, denoted $MVSG(L,\ll)$, is the digraph (V,E), where V is the set of transactions T_i, and there is an arc $(T_i,T_j)\in E$ if any of the conditions (a)—(c) for the single-version graph hold. Edges are also added for the following reason.

(d) For each $x \in READSET(T_k) \cap WRITESET(T_i)$, if $x_j \in READSET(T_k)$ and $x_i \in WRITESET(T_i)$, if $x_i \ll x_j$, then include (T_i,T_j) in E; otherwise (*i.e.* $x_j \ll x_i$), include (T_k,T_i).

[BeGo82] contains a proof that the system is serializable iff there is a total order $\ll$ such that $MVSG(L,\ll)$ is acyclic. However, it also contains a proof that deciding

whether such an order exists is NP-complete. Fortunately, we will frequently be able to construct the graph in such a fashion that we will have the desired total order on hand.

2.3. Some Particular Graphs

Two types of graphs will be of particular importance in our discussion of concurrency control: multi-partition serialization graphs and class conflict graphs (CCGs). They are similar in construction, although not identical, and play different roles in our systems. The next type of graphs we will define, multi-partition serialization graphs, are a specialized form of Papadimitriou's graphs, extended to include transactions from two partitions.[1] They are used to analyze the results of multi-partition executions.

Definition 2.2: [DaGa81, Davi82] Let $H_1=T_{11},T_{12},\ldots,T_{1n}$ and $H_2=T_{21},T_{22},\ldots,T_{2m}$ be serial schedules of transactions from partitions P_1 and P_2 respectively. The ***multi-partition serialization graph*** (or simply ***serialization graph***) $G(H_1,H_2)=(V,E)$ is the digraph defined by:

(1) $V = \{T_{11},\ldots,T_{1n}\} \cup \{T_{21},\ldots,T_{2m}\}$.

(2) E = {Dependency Edges} ∪ {Precedence Edges} ∪ {Interference Edges}, where

(a) Dependency Edges[2] —there is a dependency edge $T_{ai} \rightarrow T_{ak}$ iff $i<k$ and there is a data-item d such that $d \in WRITESET(T_{ai}) \cap READSET(T_{ak})$ and $(\forall j: i<j<k: d \notin WRITESET(T_{aj}))$.

---

[1]These graphs are called ***precedence graphs*** in [DaGa81, Davi82]; we have chosen our term as both more descriptive and more in line with current usage.

[2]Called ***ripple edges*** in [DaGa81, Davi82].

(b) Precedence Edges—there is a precedence edge $T_{ai} \rightarrow T_{ak}$ iff $i<k$, there is a data-item d such that $d \in READSET(T_{ai}) \cap WRITESET(T_{ak})$ and $(\forall j: i<j<k: d \notin WRITESET(T_{aj}))$, and there is no dependency edge $T_{ai} \rightarrow T_{ak}$.

(c) Interference Edges—there is an interference edge $T_{1i} \rightarrow T_{2j}$ iff there is a data-item d such that $d \in READSET(T_{1i}) \cap WRITESET(T_{2j})$. (Similarly for $T_{2i} \rightarrow T_{1j}$.) □

A dependency edge $T \rightarrow T'$ indicates that T' reads a value written by by T (these edges are sometimes known as "reads-from" edges). A precedence edge $T \rightarrow T'$ indicates that T read the value of a data-item later changed by T'. Interference edges indicate that a transaction in one partition must precede a transaction in the other because the former transaction read a data-item written by the latter. Interference edges illustrate the requirement that any read of an item in one partition must precede all writes of that item in the other partition in any serialization. In this respect, interference edges are similar to precedence edges. Note that interference edges only run between partitions, while dependency and precedence edges only run within partitions and are the same as those defined by Papadimitriou. Fig. 2.1 is a serialization graph for the execution of four transactions. Within the nodes, the *READSET* appears above the line, the *WRITESET* below. In all graphs, smooth lines denote precedence or interference edges; wavy lines denote dependency edges.

[Davi82] contains a proof that G is acyclic iff $H_1 \cup H_2$ is serializable; some other properties of serialization graphs will be considered in Chapter 4.

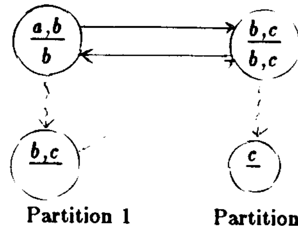


Figure 2.1. Sample serialization graph.

Serialization graphs are valuable in analyzing the results of an execution; frequently, however, we want to know what the results of an execution could be, so that we can decide what to do. We now define a class of graphs that will be used extensively in Chapter 3 for this purpose. These graphs are used for static analyses of particular allocations of classes to partitions. They show all the interactions that *could* happen if transactions were run in the correct order.

Definition 2.3: Let $A_1=\{C_{11},C_{12},...,C_{1n}\}$ and $A_2=\{C_{21},C_{22},...,C_{2m}\}$ be assignments of transaction classes to partitions P_1 and P_2 respectively. The *class conflict graph* $G(A_1,A_2)=(V,E)$ is the directed multigraph defined by:

(1) $V = \{C_{11},...,C_{1n}\} \cup \{C_{21},...,C_{2m}\}$.

(2) $E =$ {Dependency Edges} ∪ {Precedence Edges} ∪ {Interference Edges}, where

(a) There is a dependency edge $C_{ai} \rightarrow C_{ak}$ iff there is a data-item d such that $d \in WRITESET(C_{ai}) \cap READSET(C_{ak})$.

(b) There is a precedence edge $C_{ai} \rightarrow C_{ak}$ iff there is a data-item d such that $d \in READSET(C_{ai}) \cap WRITESET(C_{ak})$.

(c) There is an interference edge $C_{1i} \rightarrow C_{2j}$ iff there is a data-item d such that

$d \in READSET(C_{1i}) \cap WRITESET(C_{2j})$. (Similarly for $C_{2i} \rightarrow C_{1j}$.) □

Each edge in the CCG is labeled with the items associated with that edge (those items that cause the edge to exist). The function $LABEL: E \rightarrow 2^{ITEMS}$ maps an edge to the items labeling the edge. Fig. 2.2 is a CCG for four classes. As with serialization graphs, the *READSET* appears above the line, the *WRITESET* below. Throughout this thesis, we will use circles to represent individual transactions and rectangles to represent classes.

We will subscript G and E with d, i, or p to indicate restriction to dependency, interference, or precedence edges (or some combination) in both kinds of graphs.

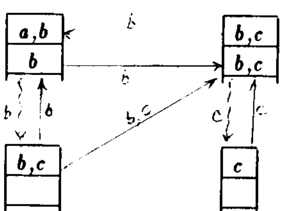


Figure 2.2. Sample class conflict graph.

CHAPTER 3

Conservative Strategies

3.1. Basics of the Conservative Approach

Conservative strategies are primarily intended for environments where backing out transactions is unacceptable. Such an environment can be found in banking, where an automatic teller machine is dispensing cash. The machine has no way to recover money if a transaction is backed out, so a conservative approach would be desirable here.

Conservative strategies are also important in environments where the level of conflict between partitions is likely to be so high that an optimistic approach would require backing out nearly all the work done in one partition. This would be the case in a small database with heavy update activity. Most items in the database would be updated in both partitions, so almost all transactions would be involved in conflicts and have to be backed out.

Before presenting the new results of this section, which are based on class conflict graphs, we must first establish several important properties of these graphs.

3.1.1. Some Properties of CCGs

If a CCG contains an interference edge (C_a, C_b), any execution that runs $x \in C_a$ and $y \in C_b$ will have a serialization graph containing an interference edge (x, y), since the definition of interference edge is the same in each case. The situation with respect to precedence and dependency edges is more complex. Since we require that

$WRITESET(C) \subseteq READSET(C)$ for any class C, if there is a dependency edge (C_a, C_b) in a CCG, there will also be a precedence edge (C_b, C_a). During execution, if a partition runs $x \in C_a$ and $y \in C_b$, there will be a path between x and y, but its nature will depend on the order in which the transactions were run and the classes of any intervening transactions.

A CCG may contain cycles of precedence and dependency edges (this is not possible in a serialization graph). Unlike serialization graphs, cycles in CCGs are not always significant; it is those involving members of both partitions, called *mp-cycles*, that are dangerous, as the following lemmas show.

Lemma 3.1: Let $G=(V,E)$ be a CCG containing an mp-cycle $CYC=(C_1, C_2, \ldots, C_m)$ of nodes connected by interference edges only. If a transaction from each member of CYC is run in its respective partition, the resulting serialization graph will contain a cycle.

Proof: The serialization graph resulting from the execution would contain a cycle isomorphic to CYC, as the rules for interference edge construction are the same in both cases. □

Lemma 3.2: Let $G=(V,E)$ be a CCG containing no mp-cycles. Then no sequence of transaction executions can yield a cyclic serialization graph.

Proof: Suppose there is an execution of the system that produces a serialization graph P containing an mp-cycle $CYC=(T_1, T_2, \ldots, T_k, T_1)$. Each transaction is a member of a class represented in G, and these classes would be connected in G by the same type of edges that connect the members of CYC. But then G would contain an mp-cycle, a contradiction. □

3.2. Description of the Problem

Several strategies that allow a partitioned DDB to continue functioning and maintain serializability have been published. They include voting schemes, token passing, primary copy, and so on. The restrictions imposed by these schemes are often stronger than necessary. As an example, consider Fig. 3.1 (recall that the *READSET* is above the line, the *WRITESET* below). No previously proposed scheme would allow C2 to be assigned to either partition. Running the transactions as shown cannot lead to nonserializability, however. This is because all transactions from C2 will use an old version of *a* and thus be serialized before all transactions from C1.

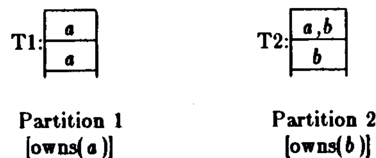


Figure 3.1. Serializable Classes.

Consider the serialization graph produced by, say, a voting scheme for some system. Since no item can be read in one partition and written in another, the serialization graph resulting from any execution in such a system will contain no interference edges, and thus trivially contain no cycles. Note that the graph remains acyclic even if we add edges from transactions in one designated partition to transactions in other partitions. This suggests a way to extend previous approaches while guaranteeing serializability. We do this by allowing one "privileged" partition P to write the same set of items as before, but to read ***any*** item. The serialization graph resulting from any execution of this algorithm may contain interference edges from partition P to other partitions.

However, since there will be no interference edges into P, or between other partitions, we are guaranteed that the serialization graph will be acyclic. The privileged partition can be chosen in any convenient way, such as those used to assign individual items (e.g. voting).

If no further information is available about the structure of transactions, or if transactions may take any form, we conjecture that the preceding method (or some variant of it) is about all we can do. In the next section, we will impose additional structure on a DDB to gain concurrency.

3.3. Using Transaction Classes

We can obtain still greater levels of concurrency if the transactions to be carried out by the DDB are divided into classes (which are known before partitioning). In this section, we will consider methods of finding sets of runnable transaction classes that are strictly larger than those allowed by previously proposed methods. In our approach, when the DDB becomes partitioned, the members of each partition P_i compute a preliminary assignment of transaction classes to P_i using a ***class assignment rule***. By making assumptions about the state of the sites that are not members of it, P_i will be able to divide the classes into three sets:

$SELF_i$ The classes that P_i can run.

$NONE_i$ The classes that no partition can run.

$OTHER_i$ The remaining classes.

The use of $NONE_i$ is not strictly necessary, as it could be merged with $OTHER_i$. However, knowing that certain classes cannot be run anywhere may allow a better

assignment of the remaining classes.

We impose two requirements on the sets to ensure that different partitions will produce compatible solutions.

$(\forall i,j \in \textit{sites}: i \neq j: SELF_i \subseteq OTHER_j)$

$(\forall i,j \in \textit{sites}: SELF_i \cap NONE_j = \phi)$

That is, we require that if $SELF_i$ contains a class, every other partition P_j will include that class in its $OTHER_j$, and if $NONE_i$ contains a class, no partition P_j will include that class in $SELF_j$. For safety, P_i must assume that the members of $OTHER_i$ *can* be run in another partition under the class assignment rule.

An assignment rule may allocate classes in such a way that it is necessary to use a ***conflict resolution rule*** to avoid nonserializable executions. There are two general forms of this idea. One version requires that P_i compute, at partition time, a subset of $SELF_i$ such that running any members of the subset in any order cannot result in nonserializable behavior. P_i's activity is then restricted to this subset. Alternatively, the system can accept or reject requests based on the transactions run since the database was partitioned. Mixed approaches that disallow some classes at partition time and others at run time are also possible.

3.3.1. Assigning Classes to Partitions

Allowing update transactions from the same class to run in two different partitions would generate a 2-cycle[1] in the serialization graph for that execution, since each tran-

---

[1] A *k-cycle* is a cycle of length k.

saction would need to precede the other. Therefore, we must guarantee that at most one partition will be able to run any given class of update transactions. Previous rules have required a partition to "possess" every item in a class if it is to run transactions from that class. (By "possess" we mean, e.g., to own the primary copy.) Such approaches can result in certain classes being assigned to no partition.

An example of a more general rule is as follows: each transaction class is assigned initially to the partition that possesses the lowest-numbered item in the *WRITESET* of the class, using some global numbering for items. (Read-only classes can be assigned to every partition that has a copy of every item in the *READSET*.) This method can never result in an update class being assigned to more than one partition as long as only one partition can possess a given item. In addition, as long as every partition has a copy of every item, and every item is owned by some partition, all classes will be assigned. (This would be the case for network failures in voting or primary copy schemes.) Unlike older conservative schemes, this rule can yield assignments that would permit nonserializable behavior. If this happens, some of the classes in the initial assignment must be discarded. We discuss this further in the next section.

3.3.2. Resolving Conflicting Assignments

Since some class assignments will not ensure serializability, we must have a way of detecting such assignments. To analyze the results of an assignment of classes, we use class conflict graphs. Our approach to partition management is as follows: each partition P_i first uses the system's class assignment rule to compute $SELF_i$, $NONE_i$, and $OTHER_i$. It then builds a CCG, using its preliminary assignment $SELF_i$ as one parti-

tion, and $OTHER_i$ as the other. (The system may actually be divided into more than two partitions; however, the 2-partition case contains the maximum number of edges and is therefore the most conservative estimate. A system of three or more partitions will have a CCG isomorphic to a subgraph of the 2-partition CCG. In fact, all G_{i_p} are isomorphic for any number of partitions; only the edge types change.) If the CCG contains no mp-cycles, the algorithm terminates and P_i can run the members of $SELF_i$ in any order. If the CCG does contain mp-cycles, there are several possibilities:

(1) Revert to a more restrictive assignment rule that guarantees serializability.

(2) Maintain the effective serialization graph since partitioning and allow transactions to run as long as it remains mp-acyclic.

(3) Iteratively remove classes from $SELF_i$ and $OTHER_i$ until the CCG is mp-acyclic. (Note that we must have a rule that will cause the partitions to make compatible removals!)

The latter two approaches will be discussed in the following sections.

3.3.3. Finding Acyclic CCGs

Let G be an mp-cyclic CCG for which we wish to find a set $S \subseteq V$ such that removing S from V leaves G mp-acyclic. Such a set S is called an ***mp-feedback vertex set***. Clearly, the smallest such set is to be preferred. We pose the question as "given a constant K, is there a set $S \subseteq V$ with $|S| \leq K$ such that S is an mp-feedback vertex set of G?" and call this problem CLASS COMPATIBILITY. Our next result suggests that finding an answer to it is not always feasible:

Theorem 3.3: CLASS COMPATIBILITY is NP-complete.

Proof: We will show that an instance of FEEDBACK VERTEX SET (does an arbitrary digraph G have a feedback vertex set of size $\leq K$?) is polynomially transformable to an instance of CLASS COMPATIBILITY. (For a proof that FEEDBACK VERTEX SET is NP-complete, see e.g. [AHU74].) Let $G=(V,E)$ be an arbitrary directed graph. It is obvious that we can determine in polynomial time if a set $S \subseteq V$ is a feedback vertex set of G. We will show how to construct in polynomial time a graph $G'=(V',E')$ such that G' is a CCG with an mp-feedback vertex set of weight $\leq K$ iff G has a feedback vertex set of size $\leq K$.

Define G' as follows: let $V' = V \cup \{v_{ab}: (a,b) \in E\}$, and $E' = \{(a,v_{ab}), (v_{ab},b): (a,b) \in E\}$. That is, replace each edge $(a,b) \in E$ by two edges, (a,v_{ab}) and (v_{ab},b). (See Fig. 3.2.)

G' is the CCG for a system in which transaction a writes item a and reads all items b such that $(a,b) \in E'$. Let $A_1=V$ and $A_2=(V'-V)$, and thus all $e \in E'$ are interference edges. Note that the cycles of G and G' are in a 1-to-1 correspondence, so any feedback vertex set S of G is an mp-feedback vertex set of G'.

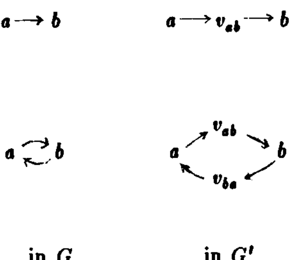


Figure 3.2. Mapping G to G'.

Now suppose that S' is an mp-feedback vertex set of G'. Then there is a feedback vertex set S of G that is no larger than S' and contains only members of V. To see this, notice that if there were some $v_{ab} \in S$, we could replace it in S' by b, since any cycle that includes v_{ab} must include b. Let S be the set derived from S' by doing all such replacements. Clearly, $|S| \leq K$ and S is a feedback vertex set of G. □

The above discussion makes the implicit assumption that the mp-feedback vertex set with the fewest members is to be preferred. This is not necessarily the case, unless the frequency of invocation of members of each class is about the same. If frequencies of invocation differ significantly, the set with the fewest members will not necessarily be the one whose removal results in the largest fraction of submitted transactions being allowed to run. In such a case, the preferred set will be the one with the smallest total frequency of invocation.

The proof of the theorem works only if the number of classes and data-items is unbounded. If the number of classes is constant, the problem can be solved by exhaustive search in constant time, which may be acceptable if the number of classes is not too large. Unfortunately, this is not as straightforward as it might seem. Problems arise because partition P_i must "guess" the classes runnable in other partitions. Although P_i will not overlook any classes as long as the assignment rule is correct, it may have to assume that the other partition can run classes that actually cannot be run anywhere. As an example, consider the situation in Figs. 3.3 and 3.4. The former is the actual CCG; the latter shows the situation as estimated by the two partitions. Transaction classes C7 and C8 can be run by neither partition. However, the pessimistic assumptions of P_1 and P_2 cause them to see the graphs as shown. P_1 would choose C3 as a

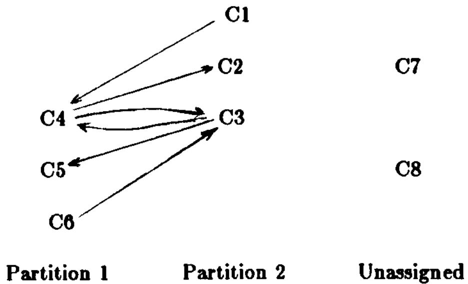


Figure 3.3. Actual class assignment.

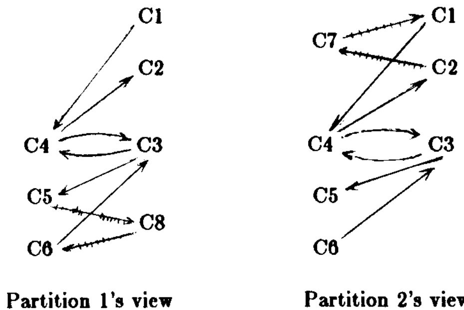


Figure 3.4. As computed by the partitions.

minimum-cost solution, while P_2 would choose C4. Both classes would thus remain in the system, and a cyclic serialization graph could result. The conclusion we reach is that removal of classes should in general be restricted to $SELF_i$.

This method is particularly well-suited to systems where a good estimate of the transaction mix is available. The classes can be weighted according to expected frequency, and an mp-acyclic subgraph of maximum weight sought.

Because CLASS.COMPATIBILITY is NP-complete, if the CCG is large, it may be infeasible to find an optimum solution. Heuristics for finding approximate solutions have been considered, but the best error bound known is $O(n/log(n))$ for a graph of n nodes (Chapter 4). However, if the CCG contains only a hundred nodes or so, it may be possible to solve the problem exactly using, say, branch-and-bound techniques.

3.3.4. Maintaining CCGs During Partitioning

Disallowing transaction classes at partition time is undesirable if it is not known what requests are likely to be submitted. A poor choice might result in most requests being rejected although they could have been accepted had a different choice been made. To maintain flexibility, the decision to accept or reject requests can be based on those accepted since partitioning occurred. In the most general form of this solution, each partition computes $SELF_i$, $NONE_i$, and $OTHER_i$ as before. It assumes that all possible interconnections will take place in the other partition, and maintains a serialization graph accordingly. When a transaction request R is submitted to the system, the partition computes whether running R would induce a cycle in the serialization graph. If so, the request is rejected; otherwise, it is accepted.

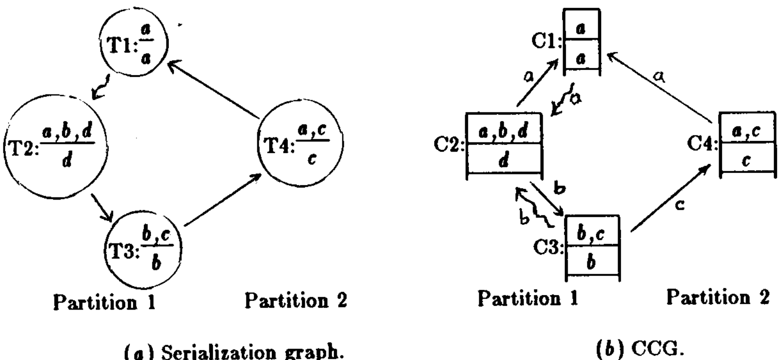
Some optimizations are possible. For example, once a class has been disallowed, it will never be allowed again. The system can maintain a list of disallowed classes and look up incoming requests in constant time to see if they are in the list. Also, if a request is received that is in the same class as the most recently granted request, it can be accepted without further testing.

This approach is an alternative to finding an mp-feedback vertex set at partition time. It is not strictly more powerful, as for each strategy it is possible to find examples where it outperforms the other. The greatest drawback to this method is that if the CCG is large, determining whether a class can be run may be too costly in CPU time. If so, a cheaper but less general method must be used. We will return to this topic in a later section in conjunction with the use of multiple versions.

3.4. Using Two Versions

The user view of a database is typically of data-items having a single value. However, it is possible for the system to view an item as consisting of several versions, a new version being created every time the item is written. This view of items has been studied in [PaKa81], [Reed78], [Thom79], [BeGo82], *et al.* All these schemes have in common that any transaction will use the most recent version of an item existing at the time the transaction is run. In this section, we will show that additional concurrency can be obtained by using older versions. That is, the class assignment and conflict resolution rules are augmented with a ***version assignment rule,*** which specifies the version of each item read by a transaction.

As an example of how using older versions can help, consider the cyclic serialization graph in Fig. 3.5. The CCG for the system appears on the right.

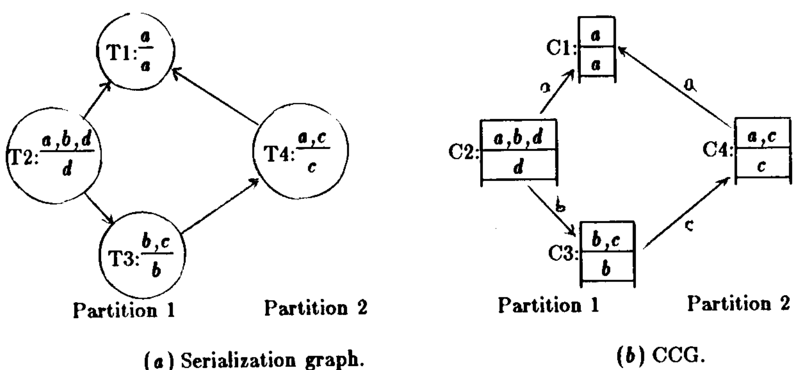


(*a*) Serialization graph. (*b*) CCG.

Figure 3.5. Serialization graph and CCG for a system.

Since the serialization graph is cyclic, the execution is not serializable. The execution could have been serialized, however, if T2 had used an older version of *a*, leading to the serialization graph of Fig. 3.6, which is acyclic. This serialization graph is isomorphic to G_{ip} (right side of figure), where G is the CCG for the system.

If G_{ip} is mp-acyclic, it defines a partial order on its strongly-connected components. By using old versions, we will serialize transactions from classes in different components



(*a*) Serialization graph. (*b*) CCG.

Figure 3.6. Serializable execution and CCG G_{ip}.

in an order consistent with the partial order on G_{ip}. (Within components, we can use any convenient concurrency control method.) The following theorem formalizes this notion.

Theorem 3.4: If G_{ip} is mp-acyclic, there is a 2-version scheme such that any sequence of transactions in G_{ip} can be serialized.

To prove this theorem, we must establish several preliminary results. We will first define a class of graphs that will allow us to assign versions to transactions. We then show that using these versions will give a serializable schedule within a partition. Finally, we show that the overall schedule must be serializable.

Definition 3.1: Let $G=(V,E)$ be a CCG. The ***component graph*** of G, $COMPGRAPH(G)=(V',E')$ is the digraph whose nodes are the strongly connected components of G. The function $COMPONENT: V \rightarrow V'$ maps v to the strongly connected component containing v. There is an edge $(x,y) \in E'$ iff there is an edge in G between two members of the strongly connected components x and y. We extend the function $LABEL$ to component graphs by saying that an edge (a,b) in $COMPGRAPH(G)$ is labeled by all items labeling the edges in G that connect any two elements of the components a and b. Formally, if G is a CCG and (a,b) is an edge in $COMPGRAPH(G)$, $LABEL((a,b)) = \cup(LABEL((x,y) \in E): COMPONENT(x)=a \wedge COMPONENT(y)=b)$.

□

The component graph of Fig. 3.5(b) would be a single node, since all the members of the CCG lie on one cycle, while the component graph of Fig. 3.6(b) is isomorphic to the CCG, since the CCG contains no cycles. As a more complex example, Fig. 3.7 is the component graph of the CCG in Fig. 3.3.

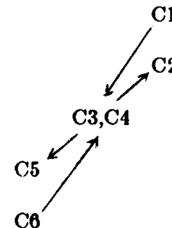


Figure 3.7. Component graph of Fig. 3.3.

$COMPGRAPH(G)$ must be acyclic. If it were not, there would be a cycle of strongly connected components, which would be a larger strongly connected component and hence should have been a node in $COMPGRAPH(G)$.

The system will maintain two versions of each item: the ***original version*** (the one existing at partition time) and the most recently written version. Versions are assigned to transactions by the following rule.

Rule 1: Let C be a transaction class with $READSET(C) = \{d_1,d_2,\ldots,d_j\}$ and $WRITESET(C) = \{d_1,d_2,\ldots,d_k\}$ $(k \leq j)$. When running a transaction T such that $CLASS(T)=C$, for an item $d \in \{d_1, \cdots, d_j\}$, if $COMPONENT(C)$ has an out-edge in $COMPGRAPH(G_{ip})$ labeled with d, use the original version; otherwise, use the most recent version. □

One implication of this rule is that transactions will use the most recent versions of items in their $WRITESET$s. This is because any pair of classes that write the same item will form a 2-cycle of precedence edges and thus be in the same component.

Before proceeding with the proof that Rule 1 has the desired properties, let's take an informal look at why it works. Within a component, we are guaranteeing serializability by ensuring that each time a member of the component is run, it will be serialized

after all earlier members of the component, since it uses the latest versions of all items written within the component. Between components, if there is an edge (a,b) in *COMPGRAPH*, all transactions from a will read original versions of items written in b and will be serialized before members of b. Since component graphs are acyclic, the result will be acyclic and therefore serializable.

Lemma 3.5: Rule 1 will produce a serializable schedule within a partition for an mp-acyclic CCG.

Proof: Our approach is based on multi-version serialization graphs. For simplicity, we define the special transaction T_{init}, which writes the original values of all data-items.

We proceed by induction on the number of transactions. In the base case, the graph resulting from running one transaction T_{new} must be acyclic, containing only the edge $T_{init} \rightarrow T_{new}$.

Induction step. Induction hypothesis: The MVSG G produced by running any set of n transactions is acyclic. Suppose we run another transaction T_{new}. We proceed by cases:

1) T_{new} references only original versions (i.e. is read-only). Adding T_{new} to the set of transactions will result in an edge $T_{init} \rightarrow T_{new}$, and edges $T_{new} \rightarrow T_{first}$, where T_{first} is any transaction (other than T_{init}) that first writes anything T_{new} reads. There must already be edges from T_{init} to those transactions, so adding T_{new} will introduce no cycles.

2) T_{new} references only current versions. The added edges will be of the form $T_{mostrec} \rightarrow T_{new}$, where the $T_{mostrec}$ are the transactions that performed the most recent writes of the items referenced by T_{new}. Since T_{new} will have no out-edges, no cycles will

be introduced.

3) T_{new} references both current and original versions. It can have only one type of out-edge, the precedence edge $T_{new} \rightarrow T_{first}$. Suppose running T_{new} induces a cycle CYC into the graph. T_{new} must be in a different component from the T_{first}, else it would be reading what they wrote (or some later version). Those precedence edges on CYC that do not connect members of the same component will have counterparts in the component graph of the system. However, this means such edges are forming a cycle of components, a contradiction since a component graph is acyclic. □

The remainder of Theorem 3.4 is proved in the following lemma.

Lemma 3.6: Let G be an mp-acyclic CCG for a system. The serialization graph for any execution in this system using Rule 1 is acyclic.

Proof: Suppose not. Then there would be an execution of the system resulting in a serialization graph containing a cycle CYC. Since G_{ip} is mp-acyclic, CYC cannot be made up completely of interference and precedence edges (by reasoning similar to Lemma 3.2). Thus, CYC must contain a dependency edge $T_a \rightarrow T_b$. Under Rule 1, T_b can read a value written by T_a iff

$$COMPONENT(CLASS(T_a)) = COMPONENT(CLASS(T_b)).$$

This is because the CCG for the system will have a precedence edge $CLASS(T_b) \rightarrow CLASS(T_a)$, which would result in T_i reading the original versions of items written by T_j unless their classes were in the same component. By induction, any sequence of transactions connected by dependency edges must be in the same component. Those edges on CYC that do not connect members of the same component must therefore be precedence and interference edges that have counterparts in

$COMPGRAPH(G_{ip})$. But this implies that $COMPGRAPH(G_{ip})$ would contain a cycle, a contradiction since a component graph must be acyclic. □

Using serializability as the correctness criterion for a system can have disconcerting side effects. For example, suppose a user runs an update transaction U and, being of a suspicious nature, runs a read-only transaction R to make sure that the update really took place. Rule 1 will cause R to read all original versions, making it appear to the user that the update was not done. Since the actions of R are a subset of the actions of U, and since we could run another transaction U' of the same class as U after R and let U' read current versions, R could have been allowed to read current versions.

Under Rule 1, R reads current versions of those items written in $COMPONENT(CLASS(R))$. Read-only classes, however, form their own components in a CCG. Our problem is thus to determine whether $CLASS(R)$ can safely be merged with the update classes that write the items in its $READSET$. (The complexity of merging classes into the smallest number of components that will leave the CCG mp-acyclic is an open problem; doing this, however, introduces a performance penalty, since more frequent synchronization will be necessary in a component containing more classes.)

Suppose we want to extend a system to allow transactions from class C to read current versions of some item a, but Rule 1 requires members of C to read the original version of a. If members of C read current versions of a, this has the effect of adding dependency edges (X,C) to G_{ip}, where X is any class that writes a. (G_{ip} must contain precedence edges (C,X), of course.) It may be that one such class X_1 writes some other item b that is also read by members of C. When any member T of C is run, T will have to read current versions of a and b to maintain serializability. Thus we must add

all dependency edges (Y,C), where Y is a class that writes b. This process repeats until no more items can be added. The underlying set of precedence edges for which we were adding dependency edges is called ***item-closed*** when no more edges need be added; we now define the term formally.

Definition 3.2: Let $G=(V,E)$ be a CCG, and let $E' \subseteq E_{ip}$. E' is ***item-closed*** if for every edge $(a,b) \in E'$, E' includes all edges $(a,x) \in E_{ip}$ such that $LABEL((a,b)) \cap LABEL((a,x)) \neq \phi$. □

Once we have an item-closed set PC of precedence edges, if we can add the corresponding set of dependency edges (i.e. the members of PC reversed) to G_{ip} and leave the graph mp-acyclic, we can apply Rule 1 to the extended graph without affecting serializability.

Lemma 3.7: Let $G=(V,E)$ be a CCG such that G_{ip} is mp-acyclic, and let $PC \subseteq E_p$ be an item-closed set of precedence edges. Let $PCR=\{(b,a): (a,b) \in PC\}$. If $G'=(V,E_{ip} \cup PCR)$ is mp-acyclic, then using Rule 1 on G' will yield a serializable schedule.

Proof: Consider an edge $(A,B) \in PC$ such that $(B,A) \notin E_p$. There will be some set of items I associated with this edge, items read by A and written by B. Introducing edge (B,A) is equivalent to replacing class A with a new class A' that writes one or more items in I. Since G' is mp-acyclic, Rule 1 can run these modified classes in any order, producing an acyclic serialization graph. Suppose that in such a schedule we replace class C' with the original class C, but have the members of C use the same version selection (original/current) as C'. Running a member T of C instead of a member T' of C' will still result in an acyclic serialization graph, which we can see as follows.

The dependency edges into T will be the same as those into T'; the precedence and interference edges into T will be a subset of those into T'; the dependency edges out of T will be a subset of those out of T'; the precedence and interference edges out of T will be the same as those out of T'. The only change in the graph will be that some different dependency and precedence edges will be induced (because T does not make some of the updates that T' did); however, these edges will run from ancestors of T' to descendants of T', thus inducing no new cycles and leaving the serialization graph acyclic. □

Corollary 3.8: If C is a read-only class, C' is an update class, and $READSET(C) \subseteq WRITESET(C')$, then members of C can read current versions. □

That is, the "anomalous" transactions mentioned earlier can be safely run using current versions.

3.4.1. Verifying Protocols

Armed with the results of this section, we now turn to providing a proof of a previously proposed partitioning management scheme, weighted voting.

Weighted voting has been proposed by Gifford [Giff79] as both a concurrency control and partitioning management scheme. It provides a powerful and resilient alternative to primary copy. Under weighted voting, each copy of an item is assigned some number (possibly zero) of votes. Associated with each item is a *read quorum* RQ, and a *write quorum* WQ, where RQ+ WQ > total votes. For a transaction to read an item, its partition must have copies of the item whose total weight is at least RQ, and similarly for writing. (Primary copy is thus a case where RQ=WQ=1, one designated copy

has the only vote, and all other copies have zero votes.) Transactions always use the current version of an item.

To model this system, we must have a class assignment rule and a conflict resolution rule. Even if a system does not make explicit use of transaction classes, we can model its actions by listing all possible combinations of items that a transaction could access (i.e. for which the partition possesses a quorum). Construct the CCG $G=(V,E)$ for the system. Since RQ+ WQ > total votes, it is not possible that an item is read in one partition and written in another, so all edges of G_{ip} are precedence edges. That is, the conflict resolution rule is null, as there are no conflicts to resolve. Let $PC=E_p$. Since $E_i=\phi$, G_{ip} must be mp-acyclic, and $(V,E_{ip} \cup PCR)$ is also mp-acyclic. Applying Rule 1 to $(V,E_{ip} \cup PCR)$ will cause all transactions to use current versions of all items accessed. Thus the system will be following the weighted voting strategy, and by Lemma 3.7, the result will be serializable.

3.5. More than Two Versions

The method described in the previous section works well if G_{ip} is mp-acyclic. If G_{ip} is not mp-acyclic, it can be beneficial to have more than two versions. Consider the CCG subgraph G_{ip} in Fig. 3.8, which is not mp-acyclic, although G_i is. Now consider the execution shown in Fig. 3.9 (transactions are run in numerical order within partitions; subscripts on the items indicate versions). Transaction T3 can run serializably only by using the value of a written by transaction T1, assuming a member of each class in partition 2 is also run. Thus, three versions of item a are needed: the original version, the version written by T1, and the most recent version.

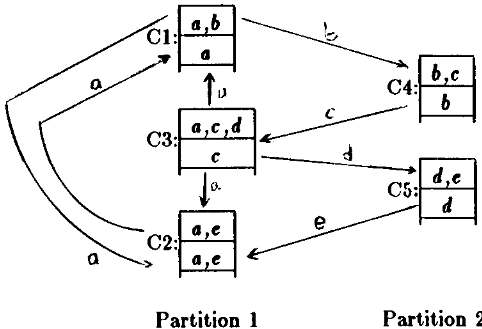


Figure 3.8. Mp-cyclic CCG subgraph G_{ip}.

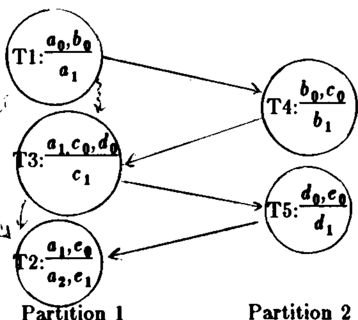


Figure 3.9. An execution requiring three versions of a.

As mentioned earlier, maintaining an entire serialization graph at run-time is infeasible in some systems. We will compress the information in the CCG so that *some* of it can be efficiently maintained at runtime. Our general approach will be to take G_{ip} and "reverse" some of its edges. The edges will be among those that lie on cycles, and we will reverse edges until the graph becomes mp-acyclic. In practice, this means that the actions of the partition will be restricted so that those edges we choose to reverse will never exist in any serialization graph between members of the affected classes. That is, if (A,B) is a precedence edge in G_{ip}, and we reverse it to form (B,A), we will ensure

there can never be a transaction from class B that is serialized after a transaction from class A. Suppose that in Fig. 3.8, which we will view from the perspective of Partition 1, we reverse edges C2 → C1 and C3 → C1; the resulting graph is mp-acyclic (Fig. 3.10). What we are requiring is that if any T with $CLASS(T) \in C2 \cup C3$ runs, no T' with $CLASS(T') \in C1$ may run later. (This approach is not meaningful for interference edges, since the partition owning the class at the head of the edge cannot communicate with the partition owning the tail.) The set of reversed edges is denoted E_r; when an edge (a,b) is reversed, that edge is removed from E and replaced by edge (b,a), which is a member of E_r. Throughout this chapter we assume that the set E_r of reversed precedence edges is minimal. In this section, "CCG" will refer to modified CCGs made up of dependency, precedence, interference and reversed precedence edges, which will be indicated by the subscripts d, p, i, and r, respectively.

The advantage of reversing edges is that it can provide greater flexibility than simply deleting classes. For example, if we have an edge, one endpoint of which must be deleted and we are unsure which endpoint to delete, we can buy flexibility by reversing an edge. In this way, some of both classes can be run. As another example, consider

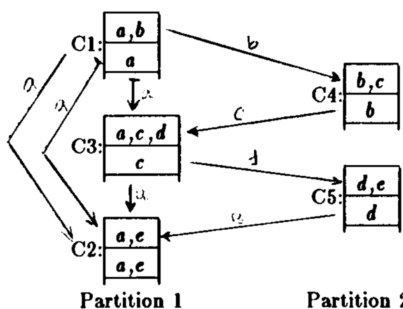


Figure 3.10. Mp-acyclic CCG subgraph G_{ipr}.

Fig. 3.11. Here, class V is one for which many transactions are submitted. Class P is rare, but of of high priority; a transaction from class P must always be accepted if possible. If we are restricted to deleting classes, we must delete class V so that we can accept requests for class P, even though we are unlikely to receive one. Using edge reversals, however, we can accept requests for class V until a class P request is submitted, after which no more requests for V can be accepted. Because P is rarely used, however, this is not apt to be a serious restriction.

Since reversal is not meaningful for interference edges, our approach will be to have each partition find an mp-feedback vertex set S for G_i, delete the classes in S, then apply the reversal policy to the remaining graph G_{ip} if it is not mp-acyclic.

To show the feasibility of this method, we must first prove that it is always possible to establish an mp-acyclic G_{ipr}:

Lemma 3.9: Let $G=(V,E)$ be a CCG such that G_i is mp-acyclic. The precedence edges of G can be oriented in such a way that G_{ip} is mp-acyclic.

Proof: Topologically sort the nodes of G_i to produce an ordering A. For any edge $(a,b) \in E_p$, if $a<b$ in A, return (a,b) to the graph; if $b>a$, delete (a,b) from E_p and add (b,a) to E_r. □

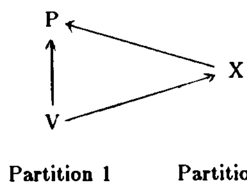


Figure 3.11. CCG G_{ipr}; edge (P,V) is reversed.

Once G_{ipr} has been rendered mp-acyclic, we must find a version assignment rule that will guarantee serializability. In addition, a run-time rule is needed to ensure that transaction classes will be run only if they can be done so serializably. For the former problem, for each item in the database, we topologically sort the components of G_{ipr} containing any class that writes that item. When a transaction T is run, for each item in $READSET(T)$, T reads the version written by the component with the largest index no greater than $COMPONENT(CLASS(T))$'s. For the latter problem, we guarantee that ordering will be maintained by requiring that when a transaction T from component C is run, all classes whose $WRITESET$s intersect $WRITESET(T)$ and are in components with indices less than C are disallowed. We now specify our rules formally.

Rule 2: Let $G=(V,E)$ be a CCG such that G_{ipr} is mp-acyclic. Topologically sort $COMPGRAPH(G)$, producing an ordering π of the components. We add a minimum element, a pseudo-component that wrote all the original versions, to π. Each time a new version of an item is written by transaction T, the version is tagged by $\pi(COMPONENT(CLASS(T)))$. (The previous version with this tag is destroyed.) When a request for transaction T with $CLASS(T)=C$ and $COMPONENT(C)=X$ is submitted, the following rules are used.

(1) T is executable iff $(\forall d \in WRITESET(T)$: $tag(most_recent(d) \leq \pi(X))$.

(2) The version assignment rule is: for an item $d \in READSET(T)$, T reads the version of d with the greatest tag $\leq \pi(X)$. □

The first point reflects the requirement that running a transaction from a class at the head of a reversed edge means we can run no more classes from the tail. It provides an upper bound on the number of versions needed for any item d: maintain the original

version and, for each component that writes d, the most recent version written by any member of that component. Under the version assignment rule, transactions will normally read either original versions or versions written within their components. The exception is the case where an item has not yet been written within a component $C1$ but *has* been written in another component $C2$ with a smaller index in π. This implies the existence of an edge $(C2,C1)\in E_r$ in G_{ipr}, so reading a version written by a component with an index $<\pi(C2)$ could lead to nonserializability.

Lemma 3.10: Let $G=(V,E)$ be a CCG such that G_{ipr} is mp-acyclic. Then no edge in E_r lies on a cycle in G_{ipr}.

Proof: Suppose there is some $(a,b)\in E_r$ such that (a,b) lies on a cycle. In the original (before reversals) G_{ip}, (b,a) must lie on an mp-cycle (this is why it was reversed). Since (a,b) lies on a cycle in G_{ipr}, there must be a path P in G_{ipr} from b to a. But then in the original G_{ip}, P would also have been on an mp-cycle. Since G_{ipr} is mp-acyclic, the cycle including P must have been broken by reversing some edge not on P. But this would also break the cycle involving (b,a), so (b,a) need not have been reversed, contradicting our assumption that E_r is minimal. □

Corollary 3.11: Members of E_r run only between components in G_{ipr}.

Proof: If not, such edges would lie on a cycle. □

We now turn to considering the effects of running Rule 2 within a partition.

Lemma 3.12: Let M be the MVSG resulting from the execution within one partition of a system using Rule 2. No edge in M connecting transactions whose classes are in different components of G_{ip} can lie on a cycle.

Proof: Transactions from different components can be connected in M only by precedence edges or by dependency edges that are due to members of E_r in the CCG. These edges have counterparts in $COMPGRAPH(G_{ipr})$. Assuming they could lie on a cycle would imply that a cycle could exist in $COMPGRAPH(G_{ipr})$, which is acyclic. □

We now show that within a single component, serializability is maintained.

Lemma 3.13: Rule 2 maintains serializability among members of a component of G_{ipr}.

Proof: We analyze the MVSG of an execution, proceeding by induction on n, the number of transactions of the component executed. Terminology is the same as in the preceding section.

Basis ($n=0$): Trivial.

Induction step ($n=k>0$): Induction hypothesis: Rule 2 will run the first $k-1$ transactions serializably. Let T_{new} be the k-th transaction. T_{new} can have only two types of out-edges in the MVSG:

$T_{new} \rightarrow T_{first}$. These precedence edges cannot lie on a cycle by Lemma 3.12, since they must connect members of different components.

$T_{new} \rightarrow T_{srev}$, where T_{srev} is the subsequent write of an item for which T_{new} is reading a non-current, non-original version. $CLASS(T_{srev})$ cannot lie in the same component as $CLASS(T_{new})$, for if it did, Rule 2 would require T_{new} to read the output of T_{srev} or one of its descendants. If $CLASS(T_{srev})$ is in a different component from $CLASS(T_{new})$, the precedence edge cannot lie on a cycle by Lemma 3.12. □

We can now establish the central result of this section:

Theorem 3.14: Rule 2 guarantees serializability.

Proof: The preceding lemmas show that the transactions within a partition must be serializable. We are left only to show that the serialization graph P generated by combining these executions is acyclic. The graphs for the individual partitions must be acyclic, so if P contains a cycle, it must involve interference edges. Interference edges, however, can only connect members of different components, and by reasoning similar to Lemma 3.12, these edges cannot lie on a cycle. Hence the serialization graph is acyclic and the transactions must be serializable. □

3.5.1. The Complexity of Precedence Edge Reversal

Once G_i is mp-acyclic, we are faced with the problem of orienting the precedence edges of G so that G_{ipr} will be mp-acyclic. Lemma 3.9 gives a method for doing this, but the results of the method may be unsatisfactory if many precedence edges are reversed (thus rendering many classes vulnerable to removal). To minimize such problems, we would like to reverse as small a set of precedence edges as possible. Unfortunately, even discovering the size of a minimum such set may be impractical:

Lemma 3.15: Given a constant K and a CCG $G=(V,E)$ such that G_i is mp-acyclic, finding an answer to the question "is there a set $S \subseteq E$ with $|S| \leq K$ such that reversing the members of S leaves G acyclic?" is NP-complete.

Proof: We reduce FEEDBACK ARC SET (does an arbitrary digraph $G=(V,E)$ have a feedback arc set of size $\leq K$?) to the stated problem. Let K be a constant and $G=(V,E)$ be an arbitrary digraph. We will show how to transform G to a CCG

$G'=(V',E')$ such that G has a feedback arc set of size $\leq K$ iff G' has a reversal set of size $\leq K$. We proceed by replacing each $(a,b) \in E$ by three edges (a,v_{ab1}), (v_{ab1},v_{ab2}), and (v_{ab2},b) (see Fig. 3.12). The members of V form one partition; the v_{ab1} and v_{ab2} form the other. The edges (a,v_{ab1}) and (v_{ab2},b) are interference edges, and the (v_{ab1},v_{ab2}) are precedence edges. There is a 1-to-1 correspondence between the cycles of G and G', and all cycles in G' are mp-cycles. Any feedback arc set of G can be converted to a feedback arc set of G' by replacing edges (a,b) by (v_{ab1},v_{ab2}); similarly, a feedback arc set of G' can be converted to a feedback arc set of G that is no larger by replacing any edge of the form (a,v_{ab1}), (v_{ab1},v_{ab2}), or (v_{ab2},b) by (a,b). Next, we note that in any feedback arc set of G', any edges (a,v_{ab1}) or (v_{ab2},b) can be replaced by (v_{ab1},v_{ab2}) (i.e., precedence edges only). The final necessary observation is that if S is a feedback arc set of G' that contains only precedence edges, it is also a reversal set of G'. Combining these observations, we have a way to convert an arbitrary digraph G to a CCG G' such that G has a feedback arc set of size $\leq K$ iff G' has a reversal set of size $\leq K$. □

The above lemma actually proves a slightly stronger result than required. Since all the precedence edges connect members of the same partition, we have demonstrated that

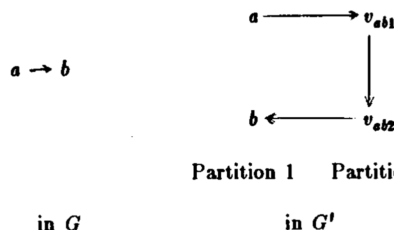


Figure 3.12. Mapping G to G'.

the problem remains NP-complete even if the precedence edges lie in only one partition.

As was the case with CLASS COMPATIBILITY, finding a set with the fewest members is not necessarily the optimum solution. This is because the tails of the reversed precedence edges may be heavily used classes, so their removal from $ALLOWED_i$ would seriously degrade system performance. If we assume that the system will remain partitioned long enough for many members of every class to be submitted, the optimum solution will be the one with the smallest total frequency of use of the tails of the reversed edges.

3.6. Comparison of the Rules

Although Rule 2 is more complicated than Rule 1, when $E_r = \phi$, the two rules are the same. If $E_r = \phi$, all classes that write an item must be in the same component, so under Rule 2, for a class C, either an item in *READSET*(C) is written in C's component and members of C will read the current version, or the item is written in a different component and members of C will read the original version. This is simply Rule 1. In this section, we explore more carefully the relationship between the two rules.

Let G be a CCG such that G_i is mp-acyclic and G_{ip} is not. Suppose we are prepared to use Rule 2, but instead decide to use Rule 1. Is there any easy way to convert a Rule 2 solution to a Rule 1 solution? The obvious answer is to take either the head or the tail of each edge $e \in E_r$ and add it to S, the multi-partition (mp) feedback vertex set of G_i. Since all mp-cycles in the original G_{ip} were broken by reversing the members of E_r, this enlarged S will be an mp-feedback vertex set of G_{ip}. Even if E_r is minimal, however, this does not guarantee that S will be minimal, as the example in

Fig. 3.13 shows. The two obvious rules for mapping E_r into S are to take the head of every edge in E_r or the tail of every such edge. In the figure, the only solution under Rule 2 is $E_r = \{C5 \rightarrow C3,\ C3 \rightarrow C1\}$. Using the tails of the edges gives the solution {C5,C3}, while using the heads gives {C3,C1}. Neither of these is minimal, since {C3} alone will serve. We conclude that such approaches can be used to find a solution for Rule 1, but not a minimal solution. Finding the size of a minimum such solution is NP-complete. (***Sketch of proof***: reduction from VERTEX COVER. For an arbitrary graph G, construct a CCG G' by numbering the vertices of G and orienting the edges to point from a lower- to a higher-numbered vertex. All these edges will be reversed precedence edges; we guarantee this by introducing two interference edges (a, v_{ab}) and (v_{ab}, b) for each directed edge (a,b) in G'. Any vertex cover for the original graph will be a minimum set of endpoints of reversed precedence edges in the constructed graph, and vice-versa.)

We now consider the question from the opposite direction. Rule 2 does have the advantage that a member of a class at the tail of an edge in E_r can be run until a member of the class at the head is run. It would be useful to be able to convert a Rule

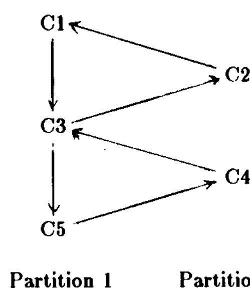
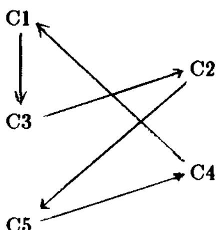


Figure 3.13. Minimal E_r does not produce minimal S.

1 solution to a Rule 2 solution that is at least as powerful by putting the members of S at the tails of members of E_r. Unfortunately, this is not always possible. Even reversing all precedence edges incident on members of S need not give a solution for Rule 2. In Fig. 3.14, {C5} is a minimal mp-feedback vertex set, but has no precedence edges incident on it.



Partition 1 Partition 2

Figure 3.14. Solution to Rule 1 does not map to Rule 2.

We conclude that a solution for Rule 1 gives no easy method for finding a solution for Rule 2, and certainly not a method for finding a minimal solution. This is not surprising, however, since Rule 2 subsumes Rule 1.

3.7. Using Pseudo-Partitioning in Concurrency Control

The techniques described in previous sections are intended for use in systems where the sites have become physically disconnected from each other. However, variants of these methods can be used for concurrency control in non-partitioned distributed systems to reduce communication overhead. The methods can be used even if the CCG for the system is mp-cyclic.

As an example of the type of system where such an approach might be fruitful, consider a DDB with several sites in New York and several others in Los Angeles. Suppose that the sites within a city are connected by high-speed links, but the cross-country link is relatively slow. The system replicates data in both cities; however, the transactions submitted in New York tend to use different data than those submitted in Los Angeles. Because of this, most of the overhead incurred in copying writes across the country is wasted, since most of the values copied are not used. In addition, the low cross-country bandwidth makes this a potential bottleneck if the volume of activity is large. What is needed is some way to run transactions within the city where they are submitted, exchanging data with the other city only if this is required to maintain serializability.

We apply our partitioning techniques to general DDBs as follows. Divide the sites up into ***pseudo-partitions*** such that every site belongs to exactly one pseudo-partition. (It is possible to consider a site as belonging to more than one partition, i.e. one physical processor could run several virtual processors, which could be assigned to different partitions. This is useful in balancing the computing load among partitions. We assume that each virtual machine belongs to exactly one pseudo-partition.) Assign every class to one or more pseudo-partitions and build the CCG, $G=(V,E)$ for this class assignment. If G_{ip} is mp-acyclic, the system simply uses Rule 1.

Suppose G_{ip} is mp-cyclic, and let S be a minimal mp-feedback vertex set of G. Transactions can be run using Rule 1 as long as no transaction T such that $CLASS(T) \in S$ is submitted. For example, in Fig. 3.15, the system is serializable under Rule 1 as long as no transaction T in class C is run. Since attempting to run T using

Figure 3.15. Mp-cyclic CCG with S=C

Rule 1 could lead to nonserializability, a different approach is called for.

One simple solution would be to block incoming transactions, let the system complete all transactions in progress, then update all copies of all items to the current value. T could then be run and its writes installed in all copies of the affected items. At this point, the system would resume activity.

This simple approach will certainly maintain serializability, but it would seriously degrade system performance if many such transactions were submitted. Indeed, the volume of updating required would probably prohibit the use of the technique in systems where real-time response is necessary, even if members of S were very rare. In addition, this method often does more work than necessary, updating original versions that could safely have been left alone. What is needed is a method of restricting the simple rule to the minimum amount of work needed. The key observation to be made at this point is that updating of original versions can be restricted to those read by members of the component containing T.

Rule 3: Let $G=(V,E)$ be the CCG for a pseudo-partitioning, and let S be a minimal mp-feedback vertex set for G_{ip}. Within a partition, run transactions using Rule 1 on $(V-S,E_{ip})$ until a transaction T such that $CLASS(T) \in S$ is submitted. At this point, block further transactions from $COMPONENT(CLASS(T))$ and let currently

executing transactions complete. Let C1 = $COMPONENT(CLASS(T))$, and define I = $\{x$: x is a data item written by a member of C1$\}$. Run T using Rule 1 applied to G_{ip}. After running T, set the copies of all original versions of items in I that are read by members of C1 to the value of the current version. (Copies read by nonmembers of C1 are assumed to be distinct and are not affected.) Once T has been run and its updates installed, the incoming transactions can be unblocked. □

If G_{ip} is mp-acyclic, S is empty and Rule 3 degenerates to Rule 1. Strongly-connected components of G_{ip} that are mp-acyclic will also use Rule 1. In practice, updating items in I not written by T can run in parallel with T. (This version of Rule 3 is too restrictive; the entire component need not always be blocked when a member of a class in S is run. In Fig. 3.15, for example, only class D really needs to be blocked while a transaction T with $CLASS(T)$=C is running; classes A and B can continue to run until the updating phase of T begins. Finding improved versions of Rule 3 remains an open problem.)

Rule 3 is a 3-version scheme, using the original version, the current version, and the updated original versions within a component, which we will call *revised versions*. In cases where the CCG has only one component, the scheme will use only two versions, current and revised. Revised versions and current versions are not read outside their components (members of other components read the original version). This means that in a serialization graph describing the execution of the system, members of different components will only be connected by precedence or interference edges, which have counterparts in the CCG and the component graph, and which cannot form a cycle, since the component graph is acyclic. Thus, we need only show that members of a given

component are serializable under Rule 3 to show that the entire system will be serializable.

Theorem 3.16: Rule 3 guarantees a serializable schedule.

We prove this result as several lemmas.

Lemma 3.17: Rule 3 guarantees a serializable schedule within a partition.

Proof: We use an MVSG analysis. Assume that G_{ip} is not mp-acyclic. (Otherwise, $S=\phi$ and the system just uses Rule 1.) Rule 3 will work as long as no member of S is run (this is simply using Rule 1). Suppose a transaction T such that $CLASS(T) \in S$ is submitted. Since Rule 1 is correct, the MVSG for the system up to this point must be acyclic. The edges out of T in the MVSG will be precedence edges of the form $T \rightarrow T_{first}$, where T_{first} is the first transaction to write some item for which T reads the original version. Such edges have counterparts in the component graph and thus cannot lie on a cycle.

Finally, we must consider the updating rule and show that it maintains serializability. Suppose we run a transaction T1 after running a member of S. If $T1 \in S$, the proof is simply as in the preceding paragraph, so consider a transaction T1 with $CLASS(T1) \notin S$. T1 in the MVSG will have out-edges of the form $T1 \rightarrow T_{first}$, which again have counterparts in the component graph and cannot lie on a cycle. □

We conclude the proof of the theorem with the following lemma.

Lemma 3.18: Rule 3 produces a serializable schedule over all partitions.

Proof: Same as Lemma 3.6. □

Rule 3 is minimal in the sense that attempting to update fewer items than the rule requires can lead to a nonserializable schedule. For an example, see Fig. 3.16, which depicts a CCG of three classes, with $S=\{C2\}$. If we attempted to update only items a and c, running the transactions T3, T2, and T1 in that order (where $CLASS(Ti)=Ci$) would produce a nonserializable result.

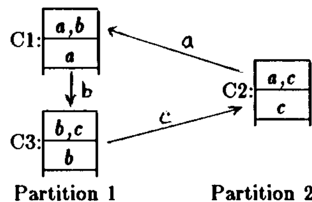


Figure 3.16. Mp-cyclic CCG of three classes.

3.7.1. Implementing Rule 3

If Rule 3 is to be implemented, there are two main considerations. First, there must be a method of assigning sites to partitions. Second, there must be a method of assigning classes to partitions.

Solutions to the first problem are likely to be dictated by hardware considerations; in our example earlier, the geographic distribution of the sites and requests made the partition assignment obvious.

The second problem poses two difficulties. On one hand, transaction classes should be assigned so that the computing load on each partition is equal. On the other hand, some assignments will result in larger mp-feedback vertex sets than others, leading to higher loads on the system due to more frequent synchronization. The first problem is NP-complete (it is simply PARTITION [GaJo79]); however, good approximation

algorithms for it are known. Since finding a minimum feedback vertex set is NP-complete even if the assignment is given, and no good algorithms for the problem are known (see Chapter 4), this aspect will probably be the more difficult of the two to solve. Sometimes, e.g. our NY/LA example above, the *READSET*s and *WRITESET*s of the transaction classes may determine their partition assignments, leaving few or no decisions to be made in this area. On the other hand, in a homogeneous system, there may be no *a priori* reason to assign classes to given partitions, complicating the problem.

If a given solution for S begins to result in too much synchronization, the members of S for any component in G_{ip} can be recomputed during a synchronization for that component.

One final note: a "minimum" mp-feedback vertex set is not necessarily the one with the fewest members. A better measure of the size of an mp-feedback vertex set is the amount of work it is expected to impose on the system (i.e. the product of its fraction of use and the amount of data that must be transferred for a member of this class to be run).

3.8. Improving Performance under Pseudo-Partitioning

Even though Rule 3 can provide better performance than the simple rule proposed earlier, the synchronization overhead incurred when a member of S is run can be significant. This is true particularly when the "data-items" used in the CCG represent many items rather than just one (e.g. all employee records for employees earning over $50,000 per year). We would like to reduce the time involved in this synchronization by allowing updating of revised versions to go on in the background as the system is running. (We require, of course, that serializability be maintained.) Such an updating policy also has the property of keeping the values read by transactions current. As a peripheral but related issue, we would also like to propagate updates to different components to keep the information there up-to-date.

The problem can be decomposed into two subproblems. We are faced with the problem of updating items within components but across pseudo-partitions. We also wish to pass updates between components.

Some updates can be made relatively simply. In the discussion of partitioning control, it was shown in Lemma 3.7 how certain classes of transactions could read current versions, even though Rule 1 would require them to read original versions. The same observation applies here. However, our orientation is now towards items, not classes; that is, we are interested in updating revised versions. If such updating can be done frequently, or the current versions are rarely changed, time will be saved when a member of S is run, since less data will need to be transferred.

The "best" type of item updates are those that can be done without interfering with or blocking running transactions. This is, transactions can continue to use a revised or original version as new values for the version are being distributed. When all the new values have been distributed, the updating takes place and subsequent transactions use the new value. (It is possible that transactions would be blocked during this updating phase.) We now turn to identifying items to which this approach is applicable. Our approach will start from Lemma 3.7, upon which we will expand in various ways. Our first expansion follows.

Lemma 3.19: Let $G=(V,E)$ be a serialization graph such that G_{ip} is mp-acyclic. Let $RP \subseteq E_p$ be an item-closed set of precedence edges, and let $RPR=\{(b,a): (a,b) \in RP\}$ be such that $G'=(V,E_{ip} \cup RPR)$ is mp-acyclic. If the system described by G is run under Rule 1 on G_{ip}, then run further under Rule 1 on G', the result will be serializable.

Proof: Let T be the most recent transaction run under Rule 1 on G'. If we build the MVSG for the system, all out-edges of T will have the form $T \rightarrow T_{first}$. These are precedence edges that run only to members of different components and, as usual, cannot lie on a cycle in the MVSG. The proof that the overall result is serializable is the same as Lemma 3.6. □

Going from G' to G_{ip} in the above is a bit trickier. Doing a simple-minded reversion to G_{ip} would cause transactions to go back to reading original versions even though their ancestors did not do so; this would often lead to nonserializability. Those classes that were reading current versions and would go back to original versions must use revised versions (those current at the time the system resumes using G_{ip}).

Lemma 3.20: Let G, G', and RPR be as in the previous lemma. If the system described by G is run under Rule 1 on G', then under Rule 1 on G_{ip} (using revised versions as described above), the result will be serializable.

Proof: We must show that the results will be serializable within a partition, then over all partitions. Suppose the system is run using G', then reverts to G_{ip}. Let T be a transaction run under G_{ip}. If we build the MVSG for the system, T can have two types of out-edges (both of them precedence edges): $T_{new} \rightarrow T_{first}$, and $T_{new} \rightarrow T_{srev}$, where T_{srev} is the first transaction to rewrite a revised version. (If no member of RPR is incident on

$CLASS(T)$, only the first type of edge will be present.) Either type of edge must run to a member of a different component, and thus cannot lie on a cycle, as this would imply a cycle of components, a contradiction. The proof that the entire system is serializable is the same as Lemma 3.6. □

Given these two lemmas, it is straightforward to see that any sequence of using G_{ip} and G' will give a serializable schedule.

The above lemmas apply only to items referenced in one partition, as they use precedence edges only. Under pseudo-partitioning, however, there are no true interference edges. Since the network is not really partitioned, those edges that we have called interference edges are more accurately viewed as a special class of precedence edges that do not lie on cycles. We can thus apply the above lemmas to the system as a whole and maintain serializability. Since we are primarily concerned with items that are referenced in more than one pseudo-partition, we would like to be able in one partition to update revised versions of items that are written in the other pseudo-partition. We will assume for now that we only wish to keep one version of the items (that is, within a pseudo-partition, every transaction reading a revised version should be reading the same thing). The propagation of values across pseudo-partitions takes place unpredictably, so the method should work no matter what class most recently wrote the item in question. We arrange this as follows. Let I be a set of items read in partition P1 and written in P2 for which we wish to provide updated revised versions in P1. (We assume here that G_{ip} is mp-acyclic; if it is not, the discussion applies to $(V-S,E_{ip})$, where S is an mp-feedback vertex set of G_{ip}.) Let CC = {all classes in P1 that read some member of I}, and let E' be the item-closure of all edges incident on members of CC that are labeled with

some member of I. Let $ER' = \{(b,a): (a,b) \in E'\}$, and $I' = \bigcup_{e \in E'} LABEL(e)$. To update the revised values, we could run the system using Rule 1 on $(V, E_{ip} \cup ER')$ and revert to G_{ip} once all items in I' have been referenced in P1. However, there is another approach that has the desirable "non-interfering" property mentioned earlier. We pick a cutoff point Y and the the most recent versions of items in I' that were written no later than Y. These versions are propagated to P1 and installed simultaneously. The effect is of running the system under Rule 1 on $(V, E_{ip} \cup ER')$ up to time Y and then reverting to G_{ip}.

3.9. Comparison with Other Proposals

The use of multiple versions in concurrency control has been considered by various other researchers. We will contrast our proposal with a recent such effort, that of Hsu and Madnick [HsMa83]. Class conflict analysis in non-partitioned DDBs has been used in the SDD-1 system; we will compare our solution with theirs.

3.9.1. Hierarchical Database Decomposition

Using old versions for read-only transactions has been proposed in the past as a form of concurrency control. The concept has been extended to include update transactions by Hsu and Madnick [HsMa83] in their Hierarchical Database Decomposition (HDD) technique, which is primarily intended for centralized systems. Under HDD, data-items are grouped into segments, which form the nodes of a directed graph called a ***data hierarchy graph*** (DHG). There is an edge (a,b) in the graph if there is some transaction that writes an item in segment a and reads an item in segment b. All items written by a transaction must be in the same segment. If the DHG is a transitive semi-

tree (a directed acyclic graph whose transitive reduction, when undirected, forms a spanning tree), the technique can be used. Once the DHG is built, transactions are divided into classes, where a transaction is in class T_i if it writes into segment D_i. The DHG is in turn converted into a ***transaction hierarchy graph*** (THG) by relabeling the nodes as classes. [HsMa83] presents a method of finding the most recent usable version of items that are read but not modified by a transaction; items being modified are managed under a timestamp ordering protocol. A similar, slightly more complex, technique is used for read-only transactions, allowing them to find the most recent usable version of the items they read. The technique has the advantage that read-only transactions are never delayed or aborted, since they can use old versions.

The HDD approach can require a significant amount of computation to find the correct versions to be read. This would result in considerable communication overhead in a distributed system. (The overhead is acknowledged by Hsu and Madnick, who give a cost-reducing protocol for read-only transactions that only recomputes which version to use periodically, rather than for each transaction.) In addition, HDD cannot be used if the THG is not a transitive semi-tree. (There is always a way to construct segments so that the THG will be a semi-tree; if necessary, all data-items can be combined into one segment. The complexity of finding an optimum solution is an open problem.) If the data-items are extensively inter-related, there may be only one data class, and the system will simply use a timestamp-based protocol, with the extensions for read-only transactions. The HDD approach can be simplified if the users of the system are willing to use data that may not be quite as up-to-date as that provided by "pure" HDD. This is the approach taken by the background updating method of the previous section, which picks one cutoff point that applies to all affected items.

3.9.2. SDD-1

The concurrency control in the SDD-1 system [BSR80] is based on transaction classes. Most of the mechanisms described are not relevant to our discussion, since they are intended for intra-partition synchronization. However, SDD-1 does face the problem of cycles in class conflict graphs. In particular, SDD-1 systems will generally include one "superclass" that is intended to contain transactions that do not fall into any other class. This is done by specifying the *READSET* and *WRITESET* of the class to be the entire database. The drawback to such a superclass, of course, is that in the CCG it will lie on a 2-cycle with any other class that does updates. This induces concurrency requirements that will in general be unduly severe, particularly if the superclass is rarely used. To reduce this problem, SDD-1 defines a special protocol, P4, that is intended for use with such high-conflict, rarely-used classes. P4 would be used on the mp-feedback vertex set S as described in previous sections; it is essentially the same as Rule 3.

The protocols define for concurrency control in SDD-1 systems allow the possibility of using older versions, e.g. in a read-only transaction (protocol P2). Such a transaction T can be processed if every update transaction U with $WRITESET(U) \cap READSET(T) \neq \phi$ and $timestamp(U) < timestamp(T)$ has been processed. However, the authors of SDD-1 also require that no conflicting update transactions with timestamps greater than T have been processed. This will ensure that T reads the most current data, but can lead to T being rejected if its timestamp is too small.

CHAPTER 4

Optimistic Strategies

4.1. Basics of the Optimistic Approach

Optimistic partitioning management strategies are intended for environments where there are few conflicts between partitions and conflicts can be resolved by backing out (undoing) conflicting transactions until the transactions remaining are serializable. To simplify the discussion, we shall assume that only two partitions are to be reconnected at a time. However, the method described generalizes to any number of partitions.

The approach used here requires that whenever the DDB is partitioned, each partition P_i must maintain a history of the *READSET* and *WRITESET* of all transactions T_{ij} that have committed in that partition since it was formed. When two partitions are reconnected, each P_i derives a serial schedule H_i for its partition using the history it has maintained. The serial schedules are then used to construct a serialization graph $G(H_1,H_2)$.

For each vertex $v \in V$, the ***dependency set*** of v, $dep(v)$, is the set of all vertices v' in V s.t. there is a path of dependency edges from v to v'. For example, in Fig. 4.1, the dependency set of a is $\{a,d\}$, and $dep(c)=\{a,b,c,d\}$. Note that if v is backed out, then $dep(v)$ must also be backed out, since the values read by the transactions in $dep(v)$ depend on the values written by v. We will use the convention that if S is a set of vertices, $dep(S)$ is the union of the dependency sets of the members of S.

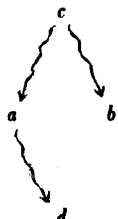


Figure 4.1. Transactions connected by dependency edges.

$G(H_1,H_2)$ is acyclic iff the combined set of transactions is serializable [Davi82]. Thus, if G contains cycles and we wish to find a serializable subset of $T_1 \cup T_2$, then some transactions (and their dependency sets) must be backed out until the graph is acyclic.

We can now state PARTITION MERGE: given a serialization graph $G=(V,E)$ and an integer $K>0$, is there a set $S \subseteq V$ such that $dep(S)$ is a feedback vertex set of G and $|dep(S)| \leq K$? Such a set S is called a ***transaction backout set*** or simply a ***backout set***. The quantity $|dep(S)|$ is called the ***weight*** of the set S. The remainder of this chapter will be concerned with the problem of finding transaction backout sets. We will consider both the complexity of the general problem and some of its subproblems, and the feasibility of finding efficient heuristics to solve it.

4.1.1. Using Optimistic Commit

For optimistic commit to be worthwhile in a DDB, the costs of the protocol must be outweighed by the costs of delaying transactions. We can express this relationship for n transactions as follows:

$$f(n)Dn > A(n) + (B+C)b(n)n,$$

where

$A(n)$ is the cost of running the backout selection algorithm

B is the average cost of backing out a transaction (may include a penalty cost)

C is the average cost of re-executing a transaction

D is the average cost of delaying a transaction

$b(n)$ is the fraction of transactions backed out

$f(n)$ is the fraction of transactions rejected by a conservative method

The $f(n)$ term on the left side of the inequality reflects that an ordinary DDB, when partitioned, would be able to run some of the transactions submitted under a conservative method. In the strategies used in our simulations, $A(n)=An$, and if we assume that transactions are small and range uniformly over the database, $f(n)=1/2$. If so, we can rearrange this equation to $\frac{D-2A}{2(B+C)} > b(n)$. For values of n satisfying this inequality, it will be advantageous to use optimistic commit. It should not be difficult to estimate the parameters involved (except D) for a given database system.

4.2. Some Properties of Serialization Graphs

This section contains some important lemmas about the properties of serialization graphs. The point of these lemmas is that certain edges may be added or deleted without affecting the solution to an instance of PARTITION MERGE. This will be important in our discussion of heuristic solutions for the problem, since we will typically be able to reduce serialization graphs to smaller equivalent graphs and a smaller graph

will usually be easier and faster to work with.

Our next lemma demonstrates that we can safely remove edges that do not lie on a cycle.

Lemma 4.1: Suppose $G=(V,E)$ is a serialization graph, and let $e \in E$ be an edge that does not lie on a cycle in G. Then $S \subseteq V$ is a backout set of $G \Leftrightarrow S$ is a backout set of $G'=(V,E-\{e\})$.

Proof: ($\Rightarrow$) Trivial, since G' is a subgraph of G.

($\Leftarrow$) Suppose not. Then S would fail to break some cycle in G. Since G and G' differ only in e, the cycle in G would have to involve e. But this is impossible, since e does not lie on a cycle. □

This lemma can be used efficiently by finding the strongly connected components of G and deleting all edges that do not connect members of the same component. The resulting graph is not completely equivalent in the sense that if dependency edges are removed, it may not be possible to determine the exact cost of a solution. Instead, the solution S found will be a basis for the exact solution, which is found by closing S under dependency edges in G.

We now turn to identifying interference edges that can be removed without affecting the solutions to an instance of PARTITION MERGE. First, we show that if G contains edges (a,b) and (a,c) (both edges of the same type), with $c \in dep(b)$, the latter edge is redundant.

Lemma 4.2: Suppose $G=(V,E)$ is a serialization graph, $a,b,c \in V$, $(a,b) \in E$ is any type of edge, $(b,c) \in E_d$, and $(a,c) \notin E$. Let $G'=(V,E \cup \{(a,c)\})$, where (a,c) is of the same type as (a,b). Then $S \subseteq V$ is a backout set of $G \Leftrightarrow S$ is a backout set of G'.

Proof: ($\Leftarrow$) Trivial, since G is a subgraph of G'.

($\Rightarrow$) Suppose S is a backout set of G. Any cycle that includes (a,c) has a corresponding cycle in which (a,c) is replaced by (a,b) and (b,c). All the latter are broken by S. This could be done by removing a, b, c, or some other nodes on the cycles involving (a,b) and (b,c). However, *any* of those removals will also break all cycles involving (a,c). Thus, S must be a backout set of G'. □

A sample application of this result is shown in Fig. 4.2. One of the implications of this lemma is that an interference edge need only be drawn from a transaction T to the *first* transaction in the other partition that writes each item read by T.

Employing the above lemmas never increases the number of components in the graph. We will now show how to find interference edges that can be removed in such a way that the number of components will increase; in particular, individual nodes may become disconnected and can thus be removed from the graph.

Lemma 4.3: Let $G=(V,E)$ be a serialization graph, where $(a,b) \in E_i$ and $(b,a) \in E_i$, $c \in dep(a)$, $d \in dep(b)$, $(c,d) \in E_i$, and $(c,d) \neq (a,b)$. Then $S \subseteq V$ is a backout set of $G \Leftrightarrow S$ is a backout set of $G'=(V,E-\{(c,d)\})$.

Figure 4.2. Equivalent serialization graphs.

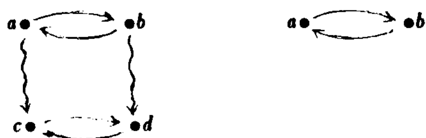
Proof: (⇒) Trivial, since G' is a subgraph of G.

(⇐) Suppose S is a backout set of G' but not of G. Any cycle not broken in G must include (c,d), since all other edges of G are in G'. But the 2-cycle involving a and b is in G' and can only be broken by deleting a or b. Deleting either of these will result in removing (c,d), so that edge cannot be involved in a cycle in G, a contradiction. Hence, S must be a backout set of G. □

An application of this result is shown in Fig. 4.3. This lemma implies that interference edges due to a data-item d need only be drawn from transactions that read the initial version of d. This is because transactions reading versions of d written by some T in their partition are in the dependency set of T, which is in a 2-cycle with the transaction that writes d in the other partition.

Some other interference edges can be deleted; in particular, if transaction T' is a member of $dep(T)$ and both have interference edges to some node W, we can sometimes delete the edge (T',W).

Lemma 4.4: Let $G=(V,E)$ be a serialization graph, where $(a,b)\in E_i$ and $(c,b)\in E_i$, $c \in dep(a)$, $c \neq b$. Suppose that all cycles involving (c,b) pass through a. Then $S \subseteq V$ is a backout set of $G \Leftrightarrow S$ is a backout set of $G'=(V,E-\{(c,b)\})$.



Partition 1 Partition 2 Partition 1 Partition 2

Figure 4.3. Equivalent serialization graphs.

Proof: (⇒) Trivial, since G' is a subgraph of G.

(⇐) Suppose that S is a backout set of G'. Note that every cycle CYC involving (c,b) has a corresponding cycle in which the section of CYC from a to c and (c,b) is replaced by (a,b). This latter cycle is broken by S. This could be done by removing a, b, or some other transaction on the path from b to a; however, any of these removals will also break CYC. □

This lemma is apt to be more expensive to use than the others. There are some easily-recognizable subcases, however; for example, if node c in the above has (a,c) as its only in-edge, the conditions for the lemma are obviously satisfied.

The greatest drawback of methods that remove nodes is that in the resulting smaller graph, it is not always possible to determine the actual cost of a given solution, even if the weight of each transaction is known. This is because a node may have more than one parent via dependency edges, and if such a node is removed in constructing a smaller equivalent serialization graph, the weights alone will not reflect it. If it is necessary to calculate the exact cost of a solution, the algorithm that removes edges can avoid removing dependency edges to nodes that have more than one dependency edge entering them.

4.3. The Computational Complexity of PARTITION MERGE

This section contains several theoretical results about PARTITION MERGE. The first is

Theorem 4.5: PARTITION MERGE is NP-complete.[1]

[1] [Davi82] contains a similar, independently developed, proof.

Proof: Almost identical to the proof in Chapter 3 that CLASS COMPATIBILITY is NP-complete. The only difference is that here we are using histories rather than class assignments. □

The proof holds only if the number of items in the database is unbounded, since the number of items needed is of the order of the number of edges in the graph G. If the number of items is a constant, the problem can be solved in time polynomial in the number of transactions (but doubly exponential in the number of items) [Kren]; however, it seems likely that in those environments for which the optimistic protocol is suited, the number of data items will be much larger than the number of transactions.

Since there are no numerical values (other than K) as part of a problem instance, this problem is NP-complete in the strong sense. This will be of importance when discussing heuristic solutions.

In a real database system, one partition might be "privileged" in that its transactions would never be backed out. This restriction does not help, however; since the construction used in the previous proof only removes nodes from one partition, we can conclude

Corollary 4.6: PARTITION MERGE with one partition ineligible for backout is NP-complete. □

Even if a problem is NP-complete, it may have subproblems that can be solved in polynomial time. An example is:

Theorem 4.7: [Davi82]. Let $G=(V,E)$ be a serialization graph for two partitions such that all edges are either dependency edges or are interference edges that lie on 2-cycles. Then a solution to PARTITION MERGE can be found in polynomial time. □

Davidson's proof actually allows a more general conclusion. Let $P \subseteq E$ be the set of precedence edges in E. Then any $S \subseteq E$ is a backout set for G iff it is a backout set for $G'=(V,E-P)$. To see this, suppose S is a backout set of G'. Then it must break all 2-cycles in G' (all of which are also in G). The only way to do this is to remove one or both vertices lying on each 2-cycle. This has the effect of removing *all* interference edges in G' (hence G). Since a graph with no interference edges can have no cycles, S is a backout set of G.

The above theorem is particularly useful since for most environments we would expect that very few cycles will be left after all 2-cycles are broken.

Unfortunately, Theorem 4.7 does not extend to more than 2 partitions. We will show this using a restricted form of the problem "does G have a vertex cover[2] of size $\leq K$?" (VERTEX COVER), which is NP-complete, even for cubic planar graphs[3] [GaJo79]. We will first show that a restricted version of VERTEX COVER on cubic planar graphs is NP-complete. Next, we will show how to convert an instance of this problem to an instance of PARTITION MERGE that has only three partitions and all interference edges lying on 2-cycles.

Lemma 4.8: VERTEX COVER is NP-complete for cubic planar graphs that do not have K_4, the complete graph on 4 vertices, as a component.

Proof: Let G be a cubic planar graph. We can easily find the K_4 components of G in polynomial time. (They are simply those components containing exactly 4 vertices.) Assume that there are m such components, and let G' be the subgraph of G

---

[2]A subset S of V such that every edge in E is incident on some member of S.

[3]A graph is *cubic* if all its vertices have degree 3.

formed by deleting all those components. Since K_4 has a vertex cover consisting of any 3 of its vertices, G' has a vertex cover of size K iff G has a cover of size $K+3m$. □

Lemma 4.9: Every cubic planar graph $G=(V,E)$ that does not have K_4 as a component can be 3-colored in polynomial time.

Proof: see [Lovász75].

Theorem 4.10: PARTITION MERGE is NP-complete for serialization graphs of 3 or more partitions even if all interference edges lie on 2-cycles.

Proof: Let $G=(V,E)$ be a cubic planar graph that does not contain K_4. We will construct in polynomial time a serialization graph $G'=(V,E')$ for 3 partitions with the property that G has a vertex cover of size $\leq K$ iff G' has a backout set of size $\leq K$. Define G' as follows: replace each edge $\{a,b\} \in E$ in G with two directed edges (a,b) and (b,a) in E'. Let each edge in E' be an interference edge. Construct a 3-coloring for G, and use the colors as names for the partitions in G'. Then there will be no interference edges connecting two members of the same partition. Suppose $S \subseteq V$ is a vertex cover of G but not a backout set of G'. But every edge in G (hence G') is incident on a member of S and hence no cycle can exist. Contrariwise, suppose $S \subseteq V$ is a backout set of G'. In particular, this means that all 2-cycles of G' are broken by S. But every edge in G' is on a 2-cycle (corresponding to an edge in G) and hence S must be a vertex cover of G. □

Finally, we note in the following lemma that Theorem 4.7 also does not extend to graphs with longer cycles.

Lemma 4.11: PARTITION MERGE on graphs with no 2-cycles and for which every interference edge lies on a 3-cycle is NP-complete.

Proof: We reduce VERTEX COVER to this restricted form of PARTITION MERGE. Let $G=(V,E)$ be an undirected graph. We will construct a serialization graph $G'=(V',E')$ such that G' has a transaction backout set of size $\leq K$ iff G has a vertex cover of size $\leq K$. First, number the vertices of V. For each edge $\{a,b\} \in V$, where $a<b$ in the numbering, add a precedence edge (a,b) to E'. In addition, add vertices a, b, and v_{ba} to V', and add interference edges (b,v_{ba}) and (v_{ba},a) to E'. G' is then a serialization graph, with the vertices from V in one partition, and the vertices v_{ba} in the other. Any vertex cover of G will be a transaction backout set of G', and any transaction backout set of G' can be converted to a vertex cover of G by replacing any v_{ba} vertices with b. □

4.4. Heuristic Solutions

Since PARTITION MERGE is NP-complete, it seems unlikely that a polynomial-time method for finding an optimum solution for it will be discovered. Thus, if we wish to use this form of optimistic commit, we are faced with the problem of developing an efficient algorithm to find an approximation that is "close to" the optimum solution. Exactly what this will mean in practical terms depends at least in part on the application, but a most desirable algorithm would produce a solution that is always within some fixed constant K of the optimum.

We can describe this more formally as follows: Let A be an algorithm for PARTITION MERGE and OPT an optimization algorithm for the same problem. The *size* of a solution to an instance of PARTITION MERGE is the number of nodes removed by that solution. Given an instance I of the problem, let $OPT(I)$ be the size of the answer

produced by OPT, and $A(I)$ the size of the answer produced by A. The *error* in a solution to the problem is the difference in the size of the solutions produced by A and OPT. We would like to find an A with the property that for all I, $A(I)-OPT(I) < K$. However, unless $P=NP$, no such algorithm can exist:

Lemma 4.12: If $P \neq NP$, there can be no polynomial-time approximation algorithm A to solve PARTITION MERGE with the property that $A(I)-OPT(I) < K$.

Proof: Suppose such an algorithm did exist for some constant K. We will show how to use such an algorithm to construct a polynomial-time exact solution to PARTITION MERGE. Given any serialization graph G, the algorithm constructs a new graph G' consisting of $K+1$ isomorphic copies of G. Such an operation can be done in polynomial time since K is a constant. Clearly, $OPT(G')=(K+1)OPT(G)$. We can construct a backout set for G of weight at most $\lceil A(G')/(K+1) \rceil$ by determining the smallest-weight set chosen by A for any copy of G. Since

$$[A(G')-OPT(G')]=[A(G')-(K+1)OPT(G)] \leq K,$$

$A(G')/(K+1) - OPT(G) \leq K/(K+1)$. Since A and OPT are both integer-valued, it must be that $A(G')/(K+1)=OPT(G)$. Thus, we have a polynomial-time optimization algorithm for PARTITION MERGE, contradicting the assumption that $P \neq NP$. □

Given that our first hope is unrealizable, we must consider weaker approximations. For example, we might hope to find an algorithm that produces a solution whose size is at most a small multiple of the optimum. Although no such algorithm is known, one may exist, as there are some NP-complete problems for which such approximate solutions have been obtained. An example is VERTEX COVER (see [GaJo79] for a discussion). Since VERTEX COVER is transformable to PARTITION MERGE, it would

seem to be a possible line of attack. Unfortunately, although a vertex cover is also a backout set, it can be arbitrarily larger than an optimum transaction backout set. In addition, a vertex cover need not contain an optimum backout set.

Some NP-complete problems, e.g. PARTITION (can a set of integers be divided into two subsets, such that the sum each is half the sum of the entire set?), can be solved in "pseudo-polynomial time." That is, they can be solved in time essentially polynomial in the size of the problem as long as the values involved are small. Unfortunately, [GaJo79] contains a proof that for problems that are NP-complete in the strong sense, such as PARTITION MERGE, no such algorithm is possible unless $P=NP$.

4.4.1. Approximating PARTITION MERGE

For PARTITION MERGE, straightforward methods such as iteratively backing out the transaction of lowest weight until the graph is acyclic have worst-case errors of $O(n)$ for a graph of n nodes. In this section, we first show how to reduce an instance of PARTITION MERGE to FEEDBACK VERTEX SET. We will then present an approximation for FEEDBACK VERTEX SET with a worst-case error of $O(n/log(n))$, which is the best bound known.

Lemma 4.13: Let I be an instance of PARTITION MERGE on graph G. Then we can convert I to an instance I' of FEEDBACK VERTEX SET such that I has a solution of size $\leq K$ iff I' does.

Proof: We convert I to I' by adding edges as follows. For each node v such that there are interference edges of the form (v,x) in G, add an edge (v',x) for each

$v' \in dep(v)$. In addition, for each edge of the form (x,v), add an edge (x,v') for each $v' \in dep(v)$. These changes have the effect of involving each member of $dep(v)$ with a cycle corresponding to any cycle involving v. Thus, if v is chosen as a non-useless part of a solution to I, it will be necessary to also choose all members of $dep(v)$. □

Now consider an arbitrary digraph G. By constructing the strongly-connected components of G, we can determine in polynomial time if G contains a cycle. If it does not, then finding a minimum feedback vertex set is trivial. Suppose G does contain one or more cycles. Divide the vertices of G up into $log(n)$ subsets of size $n/log(n)$. Since $2^{log(n)}=n$, in polynomial time we can consider all possible combinations of these subsets and find the minimum combination that is a feedback vertex set. Suppose a minimum feedback vertex set M of G contains k nodes, where $k \leq log(n)$. The nodes in M can lie in at most k different subsets, so the maximum error from using the subsets is $n/log(n)$. If $k \geq log(n)$, then simply picking $n-1$ vertices will give a feedback vertex set of error $< n/log(n)$. We have thus proved

Theorem 4.14: Let I be an instance of FEEDBACK VERTEX SET on an n-vertex graph G and let $OPT(I)$ be the size of the smallest feedback vertex set for G. There is an approximation that finds a feedback vertex set S of G such that $|S|/OPT(I) \leq O(n/log(n))$. □

4.4.2. Approaches to Backout

If we have no algorithm of guaranteed acceptable performance, we are faced with either dropping PARTITION MERGE altogether, or finding heuristics that behave well in practice. In this section, we will discuss some observations that allow simplification of

the problem.

As a first step, it is important to realize that often it is not necessary to work with the entire serialization graph. The lemmas of Section 4.2 provide several methods for removing edges and nodes to produce a smaller graph with the same minimum backout sets. (The next chapter contains some simulation results that suggest the amount of reduction possible.) Even after unneeded nodes and edges have been removed, there are other possibilities. Some nodes should not be backed out; for example, a node that has weight greater than the combined weight of all its parents. If a strategy elects to remove such a node, it should modify its decision and remove the node's parents instead. Likewise, if a node has weight greater than the set of its children, the children should be removed instead of the node, if possible.

It is never necessary to back out more than half of the transactions, since at worst it will be necessary to back out all the transactions in the partition that contains fewer of them.

One possibility is to restrict attention to subclasses of the problem that have efficient solution algorithms. This approach is taken in [Davi82], using her polynomial-time solution for serialization graphs in which all interference edges lie on 2-cycles. Although such graphs may appear contrived, they arise when every transaction T has the property that $READSET(T) = WRITESET(T)$. Such transactions are common in banking, where a single account is credited or debited. On the other hand, this strategy is not useful in environments where 2-cycles are rare.

It should be noted that 2-cycles are likely to be common in serialization graphs for many environments, as they will exist (for example) between any two transactions in

different partitions that write the same item. Hence, breaking all 2-cycles optimally should often allow the system to come close to an optimal solution for the entire graph.

Strategies like deleting high-numbered vertices in the graph first, or those with the lowest weights first, immediately come to mind. Unfortunately, for both of these it is possible to construct examples for which their performance is arbitrarily bad. Indeed, we have constructed a graph for a database of six items on which the "good" strategies we have tried all have arbitrarily bad performance (Fig. 4.4).

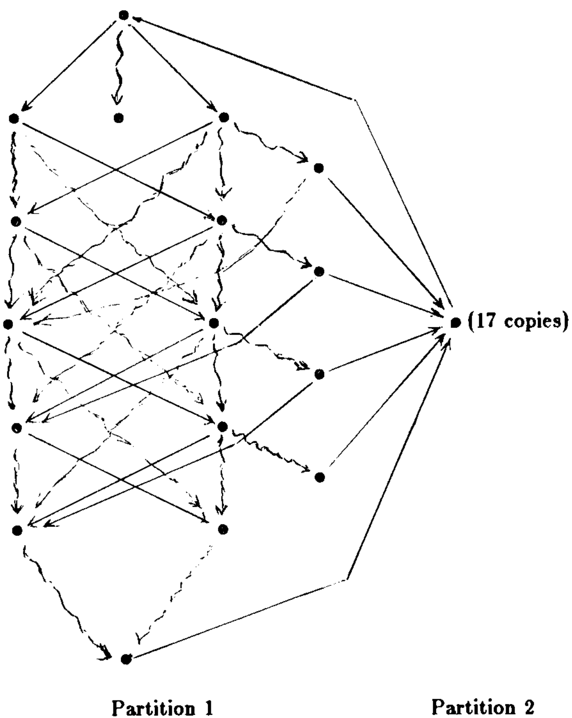


Figure 4.4. Serialization graph on which several heuristics fail.

On this graph, the "remove minimum weight", "remove highest degree/weight ratio", and "remove in decreasing numerical order" heuristics all perform with $O(n)$ error. (The minimum solution is to remove the uppermost node in partition 1.)

Describing transactions as [readset/writeset], the graph was created from the transaction histories

H_1 = [a,c,f/f], [f/], [c,d/c], [a,b/a], [a,e/], [b,c/b], [a,d/d], [d,e/], [a,b/a], [c,d/c], [c,e/], [a,d/d], [b,c/b], [b,e/], [c,d/c], [a,b/a], [a,c,e/]

H_2 = 17 copies of [e,f/e]

This example can be "pumped" to any size desired by repeating the subsequence from $T_{1,3}$ to $T_{1,16}$ as many times as necessary, increasing the number of nodes in partition 2 to equal the number of nodes in partition 1.

4.4.3. Hybrid Strategies

Although optimistic commit is the least restrictive strategy to use during partitioning, it does have drawbacks. In particular, except for initial read-only activity, there is no guarantee that any particular transaction will not be backed out. This will not always be acceptable. Given this, a method of ensuring that certain transactions cannot be backed out would be needed.

A simple solution would seem to be "tagging" some transactions as non-reversible. Unfortunately, this will not work in general, as different partitions could tag all members of a cycle. Even using a conservative protocol like majority consensus for each item referenced by a transaction T is not enough to allow it to be tagged, as T could be in the dependency set of a transaction that *is* eligible for backout.

We can use our knowledge of conservative strategies from Chapter 3 to aid us here. Partitions that must ensure that none of their transactions will be backed out use a conservative strategy. Since the results of such strategies are serializable, tagging all transactions run in these partitions cannot cause irreconcilable conflict. Instead, backout is simply limited to transactions run in partitions using the optimistic protocol.

4.5. Extending the Model

A model is an abstraction of a system. In developing a model, it is necessary to abstract those features of the system that are relevant, and to make reasonable assumptions where more than one possible abstraction exists. Since the more general our model is, the more systems we can study, we would like to have as few restrictions as possible. Some of the limitations of the model we have been using thus far are:

(1) Data-items may not be deleted

(2) Data-items may not be created

(3) *READSET*s must contain *WRITESET*s

(4) Transactions are atomic actions

(5) Transactions must be serializable

The last two of these restrictions, that transactions must be atomic and serializable, are unlikely to change, since this view of the world would typically be held by the DDB users. The other restrictions, however, do not reflect any properties that would necessarily be assumed by the user of a DDB. Thus, it is desirable to try to extend our model to incorporate them.

4.5.1. Deletion of Data-items

Extending the model to encompass deleting a data-item from the database poses no conceptual difficulties. Clearly, no transaction can reference an item after that item has been deleted. Thus, if transaction T deletes an item in one partition, all transactions in the other partition that reference that item must be serialized before T. We can modify the model so that transactions consist of a *READSET*, a *WRITESET*, and a *DELETESET*, with $WRITESET(T) \subseteq READSET(T)$, $DELETESET(T) \subseteq READSET(T)$, and $DELETESET(T) \cap WRITESET(T) = \phi$. We extend the rules for construction of a serialization graph as follows.

(d) Interference Edges--there is an interference edge $T_{1i} \rightarrow T_{2j}$ iff there is a data-item d such that $d \in READSET(T_{1i}) \cap DELETESET(T_{2j})$. (Similarly for $T_{2i} \rightarrow T_{1j}$.)

One effect of this definition is to introduce a 2-cycle if both partitions delete the same item. Although this may sometimes be unduly restrictive, it's important to bear in mind that if a transaction that would have deleted an item found it already deleted, its behavior could have been completely different. Rule (d) above can be relaxed if the parallel deletions do not reflect a conflict under the semantics of a particular system.

4.5.2. Addition of Data-items

Trying to add a data-item to the DDB while it is partitioned is a much thornier problem. This is because we describe transactions only by the items they affect, not by how those items are chosen.

Suppose transaction T1 in partition P1 creates item a. If transactions act on all items satisfying some predicate, it is possible that partition P2 runs a transaction that would access a if it were present in P2. Worse, if transactions are general programs, it is undecidable whether a transaction would indeed access a given item. Thus, the only way to guarantee serializability in the above example is to require that *all* transactions in P2 precede all transactions in P1. If there is much creation of items, there is almost certain to be a high degree of conflict, making this an unsatisfactory solution.

We can improve this to a limited extent if the system is using transaction classes and the item being created is a new *kind* of item (not the addition of a new item of a known type). A new item can only be referenced by a new class, so the partition would also have to create this class. As long as both partitions do not try to create the same class and items, no conflict will arise from the creation.

4.5.3. READSETs containing WRITESETs

If the function of a transaction T is to copy item a into item b, regardless of the previous value of item b, it is not realistic to require T to read the old value of b. Such transactions can certainly exist in a system; unfortunately, they cause difficulties when we try to model them. Problems arise because given a set of transactions whose only interaction is in writing an item that they do not read, there is no reason to prefer any particular order on the transactions (as long as the final value written is always the same). If we allow such transactions, we no longer know how to decide serializability in polynomial time, as the problem becomes NP-complete. We will not establish this result formally, since the proof is long and the result is included here only for the sake of completeness. The following is a sketch of the approach, which is based on [Papa79].

Histories are described by a graph known as a ***polygraph***. A polygraph (V,E,B) is a directed graph (V,E), augmented by a set B of ***bipaths***, where a bipath is a pair of arcs $(x,y),(y,z)$ (not necessarily in E) such that $(z,x) \in E$. A polygraph is *acyclic* if it contains an acyclic subgraph that includes V, E, and one arc from every bipath.

We use a history h for a partition to construct a polygraph $P(h)=(V,E,B)$ as follows. First, we augment the history with two transactions, T_{n+1}, which writes all the initial values, and T_{n+2}, which reads all the final values. V is the set of live transactions (those whose actions affect the values read by T_{n+2}) in the history. E contains the arcs $\{(T_{n+1},v): v \in V-\{T_{n+1}\}\}$ and the arcs $\{(v,T_{n+2}): v \in V-\{T_{n+2}\}\}$. In addition, if transaction c reads the value of an item x written by transaction d, add (d,c) to E. Finally, if some third transaction f also writes x, add bipath $((c,f),(f,d))$ to B. Two histories are equivalent iff they have the same polygraph.

[Papa79] contains a proof that $P(h)$ is acyclic iff the set of transactions is serializable. However, it also contains a proof that deciding serializability is NP-complete. We can adapt this result to our partitioned environment by a slight modification of the proofs and definitions in [Papa79]. For each partition's history, we construct a polygraph as described above. In addition, we must include bipaths between partitions for the same reason that bipaths are included within partitions. Finally, we must merge the T_{n+1}'s from each partition, and also merge the T_{n+2}'s. The resulting polygraph is acyclic iff the combined histories are serializable.

In showing that determining serializability in this environment is NP-complete, we can use much of the construction of Papadimitriou's Theorem 1 unaltered. The proof shows how to reduce a restricted form of 3-CNF to this problem. We must give rules for

assigning the transactions to partitions and for constructing *READSET*s and *WRITESET*s. For transaction assignment, we do the following:

(1) Assign a_j and b_j to Partition 1, with c_j in Partition 2.

(2) If there is an arc (b_j, z_{ik}), put z_{ik} in Partition 1 and y_{ik} in Partition 2.

(3) If there is an arc (z_{ik}, c_j), put z_{ik} in Partition 2 and y_{ik} in Partition 1.

For *READSET* and *WRITESET* construction, we do the following for nodes other than T_0 and T_f:

(1) For each node a, put x_a in $READSET(T_a)$ and $WRITESET(T_a)$.

(2) For each arc (a,b) connecting two items in the same partition, put x_{ab} in $WRITESET(T_a)$ and $READSET(T_b)$.

(3) For each arc (a,b) connecting two items in different partitions, put x_{ab} in the $READSET(T_a)$ and in both $READSET(T_b)$ and $WRITESET(T_b)$.

(4) For each bipath $((a,b),(b,c))$, put x_{ca} in $WRITESET(T_b)$.

The construction of the history corresponding to this graph is similar to that in Theorem 1.

The NP-completeness of deciding serializability in this environment even though the histories for the individual partitions are serial appears to weigh strongly against attempting to use the optimistic approach in systems where *READSET*s need not contain *WRITESET*s. However, the approach is still usable if the implementation assumes that *READSET*s contain *WRITESET*s even when they do not. This can result in the system erroneously declaring histories nonserializable, but cannot result in declaring nonserializable histories serializable.

CHAPTER 5

Modeling Optimistic Commit

5.1. Simulation of Backout Strategies

In the previous chapter, we discussed the complexity of backing out transactions and showed that doing so optimally can be difficult. Such results, however, give no information on how optimistic commit is apt to perform in a particular database. In an effort to analyze the behavior of optimistic strategies, a probabilistic model of a 2-partition system was developed. This model provides an estimate of the number of transactions that would have to be backed out in a typical execution as a function of the parameters described below.

In modeling a transaction management system, it is necessary to include those parameters that affect the system behavior. The parameters used here were

N1, N2	number of transactions in partition 1 and 2, respectively
M	number of data-items in the database
I	average number of data-items read by a transaction
RO	average ratio of read-only transactions to total transactions
U	average *WRITESET* size as a fraction of *READSET* size in update transactions

5.2. The Probabilistic Model

This section presents a probabilistic model that can be used to estimate the average number of transactions that would have to be backed out as a function of the parameters listed in the previous section. We consider two types of conflict — 2-cycles, and direct dependency effects of 2-cycles — and estimate how many nodes are involved in such conflicts,[1] assuming that references to the database are uniform and independent.

First, we estimate NTC, the number of nodes in one partition that are involved in write-write 2-cycles, which should be the most common kind. (For purposes of exposition, our formulas will work from the perspective of Partition 1.) Read-only transactions cannot be involved in 2-cycles, so we need only consider update transactions, of which there are $N1(1-RO)=N1_{update}$. The average number of items updated by each such transaction is $I*U$. Let TIU be the total number of items updated in Partition 2 (we will estimate this in a moment). Then for an update transaction T in Partition 1, the probability that a given item T writes is *not* written in Partition 2 is $\left(\frac{M-TIU}{M}\right)$, and the probability that *no* item T writes is written in Partition 2 is $\left(\frac{M-TIU}{M}\right)^{I*U}$. Thus, the probability that *some* item written by T is also written in Partition 2 (i.e. that T is in a 2-cycle) is $1-\left(\frac{M-TIU}{M}\right)^{I*U}=PTC$. (This is not exact, since individual transactions do not use selection with replacement; however, it is adequate as long as $M \gg TIU$ and $I*U$ is small. If either of these does not hold, the optimistic protocol is not likely to do well in any case. For the sake of completeness, we note that the exact

[1][Davi82] considers the same effects, but using a restricted model. The model of this section degenerates to Davidson's model when $RO=0$.

formula for PTC is $1-\frac{\binom{M-TIU}{I*U}}{\binom{M}{I*U}}$. Similar corrections can be made to the other formulas in this section; however, the error is very small and exact values are more difficult to compute, particularly if TIU or $I*U$ are not integers.) Although this model considers only write-write conflicts, if I is small, the chance of a 2-cycle being caused strictly by simultaneous read-write conflicts is low and can safely be disregarded ([Davi82] contains similar conclusions). Once PTC is known, the expected number of transactions in Partition 1 involved in 2-cycles is simply $PTC*N1_{update}=NTC$.

We now consider the problem of estimating TIU. First, the total number of writes made by transactions in Partition 2 is $N2*I*(1-RO)*U=TNW$. For a given data-item, the chance that it will not be updated by a particular write is $1-\frac{1}{M}$, so the chance that it will not be updated at all is $\left(1-\frac{1}{M}\right)^{TNW}$. The chance that it will be updated is then $1-\left(1-\frac{1}{M}\right)^{TNW}$, and the expected number of items updated in Partition 2 is $M\left(1-\left(1-\frac{1}{M}\right)^{TNW}\right)=TIU$.

Finally, we estimate NDD, the number of nodes backed out because they are dependency children of nodes in NTC. The relevant nodes are those that are not involved in 2-cycles, but have dependency parents that are. First, let XTC be the number of nodes not involved in 2-cycles; this is just $N1-NTC$. Thus, the probability that a given node in Partition 1 is not involved in a 2-cycle is $\frac{XTC}{N1}$. Only update transactions can have dependency descendants or be involved in 2-cycles, and the expected

number of update transactions preceding T_k, where T_k is the kth node in a serial-equivalent schedule for Partition 1, is $(k-1)(1-RO)$. The probability that a given predecessor of T_k is both in a 2-cycle and a dependency parent of T_k is $\frac{NTC}{N1}\left(1-\left(\frac{M-I}{M}\right)^{I*U}\right)$, so the probability that a predecessor is either not in a 2-cycle or not a dependency parent of T_k is $1-\frac{NTC}{N1}\left(1-\left(\frac{M-I}{M}\right)^{I*U}\right)$, and the probability that *all* predecessors are either not in a 2-cycle or not dependency parents of T_k is $\left(1-\frac{NTC}{N1}\left(1-\left(\frac{M-I}{M}\right)^{I*U}\right)\right)^{(k-1)(1-RO)}$. From this, we see that the probability that one or more of the predecessors of T_k is in a 2-cycle and is a dependency parent is

$$(*)\ 1-\left(1-\frac{NTC}{N1}\left(1-\left(\frac{M-I}{M}\right)^{I*U}\right)\right)^{(k-1)(1-RO)}.$$

To find the expected number of nodes removed strictly through dependency effects, we must sum (*) from $k=1$ to $k=N1$ and multiply by the expected fraction of non-2-cycle nodes $(XTC/N1)$. First, define $\alpha=\left(1-\frac{NTC}{N1}\left(1-\left(\frac{M-I}{M}\right)^{I*U}\right)\right)^{(1-RO)}$. Then the expected number of nodes removed due to dependency effects is $\frac{XTC}{N1}\sum_{k=1}^{N1}1-\alpha^{k-1}$, or $\frac{XTC}{N1}\left(N1-\sum_{k=1}^{N1}\alpha^{k-1}\right)$. As long as the problem is non-trivial (i.e. $U > 0$, $RO < 100\%$, etc.) $0<\alpha<1$, and we can use the identity $\sum_{k=1}^{N1}\alpha^{k-1}=\frac{1-\alpha^{N1}}{1-\alpha}$ to rewrite the formula as $\left(\frac{XTC}{N1}\right)\left(N1-\frac{1-\alpha^{N1}}{1-\alpha}\right)=NDD$. The total number of nodes involved in conflicts is $NTC+NDD$.

5.2.1. Second-level Dependencies

We now consider estimating "second-level" dependency effects; that is, we wish to estimate the number of nodes removed strictly due to having dependency parents in NDD. First, we note that there are $XTCDD=N1-(NTC+NDD)$ nodes that could be backed out in this fashion. Suppose T_k is the kth node in the serial schedule for Partition 1; the probability that T_k is not enumerated in NTC or NDD is $\frac{XTCDD}{N1}$. The probability that a specific predecessor of T_k is an update transaction in NDD is $NDDU=\frac{NDD}{N1}\frac{N1_{update}-NTC}{N1-NTC}$. The probability that an update predecessor is a dependency ancestor of T_k is $1-\left(\frac{M-I}{M}\right)^{I*U}$, so the probability that a predecessor is both in NDD and a dependency ancestor of T_k is $NDDU\left(1-\left(\frac{M-I}{M}\right)^{I*U}\right)$. The probability that all predecessors of T_k are not in NDD or not dependency ancestors is then $1-\left(1-NDDU\left(1-\left(\frac{M-I}{M}\right)^{I*U}\right)\right)^{(k-1)(1-RO)}$. If we let $\beta=\left(1-NDDU\left(1-\left(\frac{M-I}{M}\right)^{I*U}\right)\right)^{(1-RO)}$ then by a process similar to that used in the last section to compute NDD, the number of second-level dependency descendants, $NDD_2=\frac{XTCDD}{N1}\left(N1-\frac{1-\beta^{N1}}{1-\beta}\right)$.

This value, although nonzero, is small (typically a few percent of $NTC+NDD$) and contributes little to the result. We therefore feel justified in not computing the number of conflicts in further levels of dependencies.

5.2.2. Partially Replicated Databases

Not all DDBs have fully-replicated data. We now estimate the backout rates in partitioned databases where each partition has access to only part of the data. Suppose Partition 1 has access to M_1 items, Partition 2 has access to M_2 items, and between Partition 1 and Partition 2 they have access to the entire database. In addition, let $M_{overlap}$ be the number of items in common (this is just $M_1 + M_2 - M$). We can modify our formulas to reflect these changes as follows:

$$PTC = 1-\left(\frac{M_1 - \frac{TIU}{M_2} M_{overlap}}{M_1}\right)^{I*U}, \quad TIU = M_2\left(1-\left(1-\frac{1}{M_2}\right)^{TNW}\right)$$

and in α, replace M by M_1. (The scale factor on TIU in the revised formula for PTC reflects the fraction of M_2 visible in Partition 1.)

5.2.3. Non-uniform Data References

In a real database, some items are almost never referenced and others are referenced frequently. This has two implications. One, the database size M used in these simulations should be the *effective* size; that is, M should be the number of items that have a reasonable chance of being accessed during a partitioning; items only accessed, say, during end-of-month runs would be excluded. Two, our assumption of uniform data references over the effective database is not correct. A common assumption is that references will be "90/10"; that is, that 90% of the references will be to 10% of the items. In this section, we will modify our basic model to use such skewed distributions. (There are actually two parameters present: the size of the heavily-referenced region and its frequency of reference. The rule of thumb we are applying here simply has them

adding to 100.)

Before proceeding to present these calculations, we note that there are two forms of 90/10 distribution. One possibility is that both partitions are referencing the *same* 10% of the database frequently (we would expect this to increase the number of conflicts, as indeed it does). On the other hand, since one motivation for constructing a DDB is to put data where it is used, often different partitions will be making non-uniform data references, but to ***different*** sets of items. We will first consider the former case, which is slightly simpler.

First, we need to change our calculation of PTC to reflect these new distributions. We describe the distributions as $(1-q)/q$ (e.g. for 90/10, $q=.1$); we refer to the smaller, heavily referenced fraction of the database as qM (its size), and the larger, less-frequently referenced fraction as $(1-q)M$. The first value we recompute is TIU, the total number of items updated in the other partition.

For an item in qM, the chance it will be updated by a given write is $\frac{(1-q)}{qM}$, so the chance that it won't be referenced at all is $\left(1-\frac{1-q}{qM}\right)^{TNW}$, and the chance it will is $1-\left(1-\frac{1-q}{qM}\right)^{TNW}$. For an item in $(1-q)M$, the value is $1-\left(1-\frac{q}{(1-q)M}\right)^{TNW}$. Thus, we expect $TIU_{small}=q\,M\left(1-\left(1-\frac{1-q}{qM}\right)^{TNW}\right)$ updates in qM and $TIU_{large}=(1-q)M\left(1-\left(1-\frac{q}{(1-q)M}\right)^{TNW}\right)$ updates in $(1-q)M$.

There are $N1_{update}$ update transactions, each of which makes an average of $I*U$ writes. Of these writes, $(1-q)I*U$ are to qM and $qI*U$ are to $(1-q)M$. The probability

that the former do not conflict with writes in the other partition is $\left(\frac{qM-TIU_{small}}{qM}\right)^{(1-q)I*U}$, the probability that the latter do not is $\left(\frac{(1-q)M-TIU_{large}}{(1-q)M}\right)^{qI*U}$, so the probability that one or both *do* is $PTC=1-\left(\frac{qM-TIU_{small}}{qM}\right)^{(1-q)I*U}\left(\frac{(1-q)M-TIU_{large}}{(1-q)M}\right)^{qI*U}$. By substituting, we can simplify this expression to $PTC=1-\left(1-\frac{1-q}{qM}\right)^{TNW(1-q)I*U}\left(1-\frac{q}{(1-q)M}\right)^{TNWqI*U}$. When $q=.5$ (uniform distribution) this becomes our original expression for PTC.

We must still find a modified expression for NDD, the number of nodes removed strictly due to dependency effects. We need only to recompute the probability that a predecessor of node T_k is a dependency parent of T_k and insert this value in formulas (*) and α in place of $\left(1-\frac{M-I}{M}\right)^{I*U}$. Transaction T_k references qI items in $(1-q)M$. The chance that a given update transaction preceding T_k does not update any of these items is $\left(\frac{(1-q)M-(1-q)I}{(1-q)M}\right)^{qI*U}$; for the other items, the probability is $\left(\frac{qM-qI}{qM}\right)^{(1-q)I*U}$. Thus the probability that one or more of these items *is* updated by a specific predecessor is $1-\left(\frac{(1-q)M-(1-q)I}{(1-q)M}\right)^{qI*U}\left(\frac{qM-qI}{qM}\right)^{(1-q)I*U}$. Then α (simplified) is $\alpha=\left(1-\frac{NTC}{N1}\left(1-\left(1-\frac{I}{M}\right)^{qI*U}\left(1-\frac{I}{M}\right)^{(1-q)I*U}\right)\right)^{(1-RO)}$.

When the partitions are using non-uniform distributions on different (non-overlapping) sets of items, the expression for NDD is the same as in the preceding, but PTC is different. We have divided the M items up into three sets: one heavily referenced by Partition 1, one heavily referenced by Partition 2, and one lightly referenced by

both (although the "light" set of each includes the "heavy" set of the other). Through an analysis similar to that above, we conclude that for this case, $PTC=1-\left(1-\frac{TIU_{large}}{(1-q)M}\right)^{qI*U}\left(1-\frac{TIU_{large}}{(1-q)M}\right)^{(1-2q)I*U}\left(1-\frac{TIU_{small}}{(1-q)M}\right)^{qI*U}$, or $PTC=1-\left(1-\frac{q}{(1-q)M}\right)^{TNWqI*U}\left(1-\frac{q}{(1-q)M}\right)^{TNW(1-2q)I*U}\left(1-\frac{(1-q)}{qM}\right)^{TNWqI*U}$. If $q=.5$, the second term of the product is 1 and again the expression becomes the same as for the uniform case (as it should be).

5.2.4. Non-overlapping WRITESETs

Suppose each partition has read access to the entire database, but each item can only be written in one partition. Such a strategy will eliminate write-write 2-cycles and thus should reduce backout rates. (Of course, it also reduces the variety of transactions that can be run.) We will estimate the probability that two update transactions are involved in a 2-cycle under this restriction. This value for PTC can then be used in the formulas of earlier sections to compute numbers of transactions backed out. Let T_1 and T_2 be update transactions from partitions Partition 1 and Partition 2 respectively, and define

$$WR = P(WRITESET(T_1)\cap READSET(T_2)=\phi) = \left(\frac{M-I*U}{M}\right)^{I},$$

$$RW = P(READSET(T_1)\cap WRITESET(T_2)=\phi) = \left(\frac{M-I}{M}\right)^{I*U}.$$

Then $P(T_1$ and T_2 are in a 2-cycle$) = (1-WR)(1-RW)$, and the probability that T_1 is in a 2-cycle with some transaction in Partition 2 (i.e. PTC) is $1-(WR+RW-WR*RW)^{(1-RO)N2}$. We note that this value is a lower bound, as it does

not take into account the possibility of large update transactions early in the history. Such transactions are likely to be involved in 2-cycles and to have large dependency sets. Thus, the above formulas should be considered conservative estimates.

5.3. Evaluating the Probabilistic Model

Preliminary simulation results suggest that the probabilistic model generally predicts a backout rate that is too low. This is to be expected, since we did not attempt to account for simultaneous read-write conflicts. In this section, we summarize some rules of thumb that we established by plotting values produced by the model. Simulation has produced the same results.

One useful rule is that, all other factors being equal, if the ratios $N1/N2$ and $(N1+N2)/M$ are held constant, the backout rate will be constant. That is, if the database size is doubled and the number of transactions is doubled, the percentage of transactions backed out does not change.

One surprising result is that the backout rate is insensitive to partial database replication unless most of the data is not replicated. The improvement in the backout rate depends on both the amount of replication and the backout rate in the fully-replicated case. Low backout rates are cut significantly, but high backout rates are not unless only a few percent of the items are accessible in both partitions. This is the most desirable outcome, since optimistic commit is intended for low-backout situations.

One other surprise is that as the degree of replication falls, the backout rate can rise slightly (up to a point; eventually, it begins to fall). This seems to be due to the conflicting effects of fewer items in common versus smaller effective databases. When

each partition has access to, say, 60% of the database, there will only be 20% of the database in common, but those items will be referenced more heavily than items in a fully-replicated database.

A factor that should be considered in modeling a real database is the cost of backing out a transaction. Read-only transactions are apt to be cheaper to back out than update transactions, since no disk accesses would be needed to do the backout. Backout for a read-only transaction may simply involve sending a message to the user saying that the results are not correct. Thus, we may need to adjust our cost estimates for such situations.

The probabilistic model also could be used during partitioning to give an estimate of the probability that a transaction will be accepted. When the probability falls below some threshold value, the system could block incoming transactions, or run them with a warning that the probability of a successful commit is low.

Although the values produced by the probabilistic model will often be too low, this is not an insuperable problem. The probabilistic model gives a lower bound on the backout rates expected in practice. If this number is unacceptably large for a given application, then it is unlikely that the optimistic protocol is suitable. We feel that it provides a valuable analytic tool for studying optimistic commit. It is particularly useful since the formulas can be computed quickly; running a simulation is apt to be very expensive in computer time.

CHAPTER 6

Conclusions

6.1. The Proposed Systems

We have proposed and analyzed new scheme for managing partitioned DDBs. In Chapter 3, we studied the conservative strategies, which guarantee that the results they produce will be serializable. Such strategies have been proposed before; however, we have provided a framework for analyzing these strategies and extended these results to more powerful schemes that subsume previous proposals. We have also showed how these results can be applied to non-partitioned systems to reduce the overhead associated with concurrency control.

The results of Chapter 3 make it clear that better performance in partitioned DDBs is possible than is provided by existing proposals. The results have the advantage that although they do require some effort to understand, the implementation effort involved in putting them to work would be small.

In Chapters 4 and 5, we looked at optimistic strategies. We considered the computational complexity of transaction backout in such strategies in detail, and studied the problems of finding efficient and accurate heuristics to manage such a policy. We also provided new analytic models for estimating the level of conflict expected in a system as a function of its size and type of activity.

Optimistic strategies are attractive since they do not restrict the actions of the partitions at all. In a system where the database is large, update activity is low, and the

duration of partitioning is short, optimistic strategies will rarely be required to back out a transaction, thus allowing the system to provide throughput almost as good as that in the non-partitioned system. These advantages must be weighed against the drawbacks that optimistic strategies are not suitable for systems where transaction backout is unacceptable, or where the level of conflict is apt to be so high that nearly all the activity in one partition would have to be backed out.

6.2. Implementation and Feasibility

If the strategies proposed in this thesis are to be used, there are two main considerations. First, the systems must be appropriate targets. Second, it must be possible to implement the strategies efficiently.

To answer the first point, we must know whether, for example, the system can group its activities into classes. If it cannot (i.e. transactions can make arbitrary accesses), then we will be restricted to the limited approach discussed early in Chapter 3, the "privileged partition" strategy. On the other hand, if the system's activities are class-based, then using the more detailed approaches may prove fruitful. It is desirable that the implementation choose a solution that is no worse than the one that would be used by a more restrictive protocol. One way to do this is to have the system start with a set of transactions chosen by a more restrictive method and attempt to expand the set. This is guaranteed to work, since even if P_i makes pessimistic assumptions about the members of $OTHER_i$, this will at worst produce a CCG with may edges into P_i but none out.

If we are considering the use of the optimistic approach, the first point means more than just "can transactions be backed out?" It also means "will the system partition in a suitable way?" By "suitable", we are asking if the system's activities are localized enough that our partitioning strategies will function well. We must also estimate topologies are likely to occur as the result of partitioning. In a DDB connected by a highly redundant network like Arpanet, it is unlikely that the network would fail in such a way that there would be two large partitions. A much more probable result is the failure of an Interface Message Processor (a gateway to the network), which would leave the nodes serviced by the IMP isolated. (The redundancy in the network makes it unlikely that a single IMP failure could disconnect sites not serviced by that IMP.) Each node serviced by the IMP would form a partition, so the system would be divided into one large partition and several small ones. The other type of failure that is likely to be common is the failure of a node's communication hardware, which would disconnect that node from the network. (We consider failed nodes to be a less severe problem, since a failed node processes no transactions, but the system will often not be able to tell a failed node from a disconnected one, thus necessitating some form of partitioning control.)

Implementing our conservative strategies does not affect the system except when partitioning is detected. At that time, the class allocation and conflict resolution rules would go into effect. If there is a version assignment rule, it would also go into effect; however, if the system also uses pseudo-partitioning or some other multi-version scheme, this would not induce any extra overhead.

Implementing optimistic protocols may be more expensive. For example, the system must keep a list of all transactions and their effects during the partitioning. It is

quite possible, however, that much of this information is already being kept for reliability purposes and that little or no extra cost will be induced by this requirement. One change that might be necessary in some systems would be to the form in which information is stored. Some systems merely keep "before" and "after" images of the disk pages being modified by a transaction. This may not be sufficient information to let the system know what logical items are being changed. An intentions list mechanism listing the items being modified and their new values would be needed here.

Building the serialization graph in an optimistic system can be an expensive operation. Time can be saved if the graph is partially constructed during partitioning. Interference edges cannot be filled in, of course, but other edges can be; [Davi82] discusses the problem in some detail. Various operations that improve efficiency are available; for example, read-only transactions that only use data written before the partition was formed can never be involved in a cycle and need not be included in the graph. In a system with heavy read-only activity, this would be a considerable saving. Similarly, if it is known that certain items cannot be written in another partition (because it has no copies of the items), this will also provide information that certain transactions cannot be involved in conflicts.

The greatest amount of time used in constructing the graphs may come from simply reading the intentions lists of the committed transactions. Building the graph can be done quickly once the data are available, and the graph reductions are also fast. We note that serialization graphs are usually sparse, so an adjacency list representation will generally be both quicker and more storage-efficient than an adjacency matrix.

6.3. Future Work

The results of this thesis suggest that the methods proposed can work, and work well, in production systems. However, several issues remain to be examined in detail.

Although we have restricted our attention to simple transactions in database systems, our methods of analysis can be more widely applied. In object-oriented systems, for example, the system operates on *objects*, which are implementations of abstract data types. Objects are manipulated by *operations* with defined semantics. By using this semantic information, we should be able to apply our techniques to these systems, and the extra information available may make it possible to gain additional concurrency over the simple read/write model used in this thesis.

One major difficulty in studying the suitability of optimistic partitioning strategies is in finding studies of the behavior of production systems. Particularly useful would be data on uniformity and breadth of database access, existence of special, very heavily used items, database size, transaction size, and activity rates. Database folklore suggests that transactions would tend to be small (one or two items), frequent (several hundred transactions per second), often perform updates (at least half the time), and range widely but non-uniformly over the database (a rule of thumb: one transaction per second per 20 Megabytes of data in the database) [Gray]. One difficulty is that certain items are likely to be very heavily accessed, although it is possible that their semantics would allow them to be merged despite cycles in the serialization graph (e.g. a log file).

The powerful conservative methods described in this thesis are still in their infancy. Their applicability to real systems will depend greatly on the composition of those systems. Significant factors include network configurations (which influence the number

and size of the partitions that are likely to exist) and the feasibility of assigning transactions to classes. Better class assignment rules should be sought, as should better conflict resolution rules. Even with these limitations, the possibilities posed by our strategies described in this thesis are intriguing.

References

[AlDa76] Alsberg, P.A. and Day, J.D. A Principle for Resilient Sharing of Distributed Resources. *Proc. 2nd Int. Conf. on Software Engineering,*, San Francisco, October 1976, pp. 562-570.

[AHU74] Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.

[BeGo81] Bernstein, P.A., and Goodman, N. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, v. 13, no. 2 (June 1981), pp. 185-222.

[BeGo82] Bernstein, P.A. and Goodman, N. Concurrency Control Algorithms for Multiversion Database Systems. *Proc. 1982 ACM SIGACT-SIGMOD Symp. on Principles of Database Sys.*, March 1982, pp. 209-215.

[BSR80] Bernstein, P.A., Shipman, D.W., and Rothnie, J.B. Concurrency Control in a System for Distributed Databases (SDD-1). *ACM Trans. Database Systems,* Vol. 5, No. 1, pp. 18-51, (March 1980).

[DaGa81] Davidson, S., and Garcia-Molina, H. Protocols for Partitioned Distributed Database Systems. *Proc. Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, Pa., July 1981, pp. 145-149.

[Davi82] Davidson, S. An Optimistic Protocol for Partitioned Distributed Database Systems. PhD Thesis, Princeton University, Dept. of EECS, (1982).

[EaSe83] Eager, D.L. and Sevcik, K.C. Achieving Robustness in Distributed Database Systems. *ACM Trans. Database Systems,* Vol. 8, No. 3, pp. 354-381, (Sep. 1983).

[Elli77] Ellis, C. A Robust Algorithm for Updating Duplicate Databases. *Proc. 2nd Berkeley Workshop on Distributed Data Management and Computer Networks,* Berkeley, Calif., May 1977, pp. 146-158.

[GaJo79] Garey, M.R. and Johnson, D.S. *Computers and Intractability.* W.H. Freeman & Co., San Francisco, 1979.

[Giff79] Gifford, D.K. Weighted Voting for Replicated Data. *Proc. 7th Symposium on Operating System Principles,* Asilomar, Calif., Dec. 1979, pp.150-162.

[Gray] Gray, J., private communication.

[HsMa83] Hsu, M. and Madnick, S. Hierarchical Database Decomposition — A Technique for Database Concurrency Control. *Proc. 1983 ACM SIGACT-SIGMOD Symp. on Principles of Database Sys.*, March 1983, pp. 182-191.

[Kren] Krentel, M., private communication.

[Kohl81] Kohler, W.H. A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems. *ACM Computing Surveys*, v. 13, no. 2 (June 1981), pp. 149-184.

[LeLa78] LeLann, G. Algorithms for Distributed Data-sharing Systems which use Tickets. *Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks,* Berkeley, Calif., Aug. 1978, pp. 259-272.

[Lova75] Lovász, L. Three Short Proofs in Graph Theory. *J. Combinatorial Theory B,* 19, 111-113.

[Papa79] Papadimitriou, C. The Serializability of Concurrent Database Updates. *Journal Assoc. Comp. Mach*, v. 26, no. 4 (October 1979), pp. 631-653.

[PaKa82] Papadimitriou, C. and Kanellakis, P. On Concurrency Control by Multiple Versions. *Proc. 1982 ACM SIGACT-SIGMOD Symp. on Principles of Database Sys.*, March 1983.

[Park81] Parker, R.S. *et al.* Detection of Mutual Inconsistency in Distributed Systems. *Proc. 5th Berkeley Workshop on Distributed Data Management and Computer Networks,* Feb. 1981.

[Reed78] Reed, D. Naming and Synchronization in a Decentralized Computer System. Technical Report MIT/LCS/TR-205, MIT Dept. of Electrical Engineering and Computer Science, Sept. 1978.

[StRo81] Stearns, R.E. and Rosenkrantz, D.J. Distributed Database Concurrency Controls using Before-values. Technical Report 81-1, Computer Science Dept., SUNY Albany, Feb. 1981.

[Ston79] Stonebraker, M. Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES. *IEEE Trans. Software Engineering,* v. SE-5, no.

3 (May 1979), pp. 203-215.

[Thom78] Thomas, R.H. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Trans. Database Syst.*, v. 4, no. 2 (June 1979), pp. 180-209.

[WrSk83] Wright, D.D. and Skeen, D. Merging Partitioned Databases. Technical Report TR 83-547, Dept. of Computer Science, Cornell University, March 1983.