Increasing Availability in Partitioned Database Systems

Dale Skeen*
David D. Wright†

TR 83-581
November 1983
(Revised March 1984)

*Department of Computer Science
Cornell University
Ithaca, New York 14853

†Prime Computer Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

Increasing Availability in Partitioned Database Systems

Dale Skeen

Computer Science Department
Cornell University
Ithaca, New York

David D. Wright

Prime Computer Inc.
500 Old Connecticut Path
Framingham, Massachusetts

1. Introduction

A *partitioning* of a distributed database system (DDBS) occurs when the DDBS 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 in the face of failures, the first alternative is clearly undesirable. This paper explores approaches to the second alternative.

Strategies to allow a partitioned DDBS to continue functioning fall into two categories. The first category, which we call *conservative,* comprises 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]. 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) [DaGa81, Davi82, Park81, Wrig83]. Only conservative strategies are considered in this paper.

Currently proposed conservative methods are unduly restrictive. We introduce more powerful schemes that subsume previous proposals and allow a greater variety of transactions to execute during a partitioning. Our methods are based on dividing transactions into classes and using multiple versions of data-items.

Note that we do not consider the problem of *detecting* partitioning, only of managing it when it has been detected (say, by the communications subsystem). Detecting partitioning is a difficult problem, deserving further study.

2. Formal Specification of the Problem

A DDBS consists of a set *ITEMS* of ***data-items,*** $\{d_1, ..., d_M\}$, and a set of ***sites*** that are connected by communication links. We assume that during normal operation any site can send a message to any other 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 $WRITESET(T) \subseteq READSET(T)$, for reasons explained below. We do not assume that the data are fully replicated; any degree of replication is allowed.

A ***partition*** of the database system is a maximal subset of communicating sites. Thus the entire database system, when it is functioning normally, forms a single partition. Site or communication link failures may separate the DDBS 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. The state of the system with respect to partitioning is described by the set Partitions, whose elements are non-intersecting sets of sites. Each site is a member of exactly one element of Partitions.

We assume that transactions can be grouped into ***classes*** in the sense of [BSR80]. The function $CLASS$: Transactions $\rightarrow$ Classes gives the class of each transaction. $READSET$s and $WRITESET$s for classes are defined in the obvious way. When a partitioning occurs, the function $ASSIGN$: Classes $\rightarrow 2^{\text{Partitions}}$ maps each class to 0 or more partitions in which its member transactions can potentially execute. The assignment may be done in any convenient way.

Our concern is with coordinating 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.) Simply running a correct concurrency control protocol in each partition does not guarantee that the combined actions of all partitions will produce a serializable result. We must further restrict transaction processing so that overall serializable behavior can be achieved. The challenge is to restrict processing minimally given that the particular transactions to be executed during the partitioning are not known ***a priori*** and that serializability must be preserved.

Deciding whether a set of transaction executions is serializable typically involves describing the execution (in a nonpartitioned system) by a directed graph in which nodes are transactions and edges describe transaction interaction [Papa79, BSW79]. A particularly simple graph model and serializability test can be used whenever transactions' $READSET$s contain their $WRITESET$s (which is why this assumption is made). Transactions violating this assumption can still be allowed if, for purposes of checking serializability, we pretend that such transactions read the items they write.

Definition 2.1: The *serialization graph* describing the execution of a set of transactions $\{T_1, \cdots, T_n\}$ is the digraph $S=(V,E)$ defined by:

(1) $V = \{T_1, \cdots, T_n\}$

(2) E = {Dependency Edges} $\cup$ {Precedence Edges}

(a) Dependency Edges —there is a dependency edge $T_i \rightarrow T_j$ iff T_j reads the value of a data item d written by T_i (hence, $d \in WRITESET(T_i) \cap READSET(T_j)$).

(b) Precedence Edges—there is a precedence edge $T_i \rightarrow T_j$ iff T_i reads the value of a data item d that is overwritten by T_j (hence, $d \in READSET(T_i) \cap WRITESET(T_j)$). $\square$

Transaction execution is serializable if and only if the corresponding serialization graph is acyclic.[1]

This model can be extended to partitioned DDBSs by representing transaction processing in each partition by a serialization graph and adding edges between graphs when partitions execute conflicting actions on the same items. (For simplicity, we consider the 2-partition case here.)

Definition 2.2: [DaGa81] Let $S_1=(V_1,E_1)$ be the serialization graph for transaction processing in partition 1, and let $S_2=(V_2,E_2)$ be the serialization graph for transaction processing in partition 2. The *multi-partition serialization graph* $S=(V,E)$ is the digraph defined by:

(1) $V = V_1 \cup V_2$ (i.e., transactions from both partitions)

[1]This holds only if *READSETs* contain *WRITESETs*; otherwise, a more elaborate graph model is needed [Papa79].

(2) $E = E_1 \cup E_2 \cup$ {Interference Edges},

where there is an *interference edge* $T_i \rightarrow T_j$ iff $T_i \in V_k$ reads a data-item that is written by $T_j \in V_m$, where $k \neq m$. □

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. They 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.

Theorem 2.1: ([Davi82]) A multi-partition serialization graph S is acyclic iff transaction execution is serializable across all partitions. □

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 for this purpose. These graphs are used for static analyses of particular allocations of classes to partitions. They encode all the interactions that *could* happen if transactions were run in a suitable 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) $V = A_1 \cup A_2$.

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

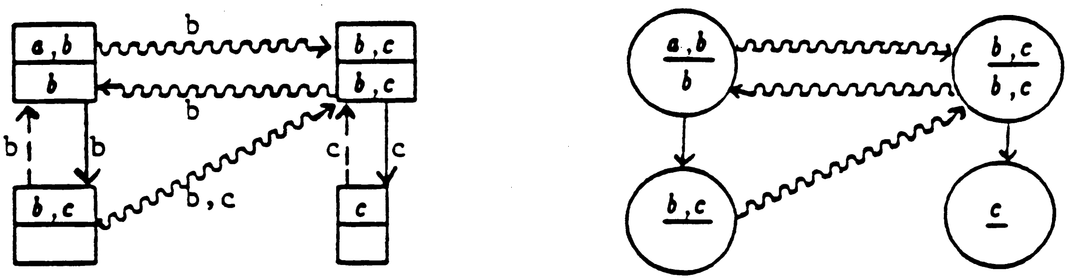
(a) There is a dependency edge $C_{ai} \rightarrow C_{ak}$ iff there is a data-item d such that $d \in WRITESET(C_{ai}) \cap READSET(C_{ak})$.

(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 $C_{2i} \rightarrow C_{1j}$.) □

The notion of class conflict graphs is borrowed from SDD-1 [BSR80], although our definition differs significantly from theirs.

Each edge in the class conflict graph (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. 2.1(a) is a CCG for four classes. Fig. 2.1(b) illustrates a serialization graph characterizing one possible execution of a transaction from each class. The $READSET$ appears above the line, the $WRITESET$ below. Precedence edges (- - ->), dependency edges (———>), and interference edges (~~>) are illustrated. Throughout, we will use circles to represent individual transactions and rectangles to represent classes.



Partition 1 Partition 2 Partition 1 Partition 2

(a) Class conflict graph (b) Serialization graph

Figure 2.1. Sample class conflict graph (a) and serialization graph (b).

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

3. Basics of the Conservative Approach

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 $CYC=(C_1, C_2, ..., C_m)$ 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. □

One way to create an mp-cycle of interference edges is to assign an "update" class, one with a nonempty WRITESET, to two different partitions. In this case, the cycle is of length two, joining the two instances of the class in the CCG. The above lemma confirms that the result of executing this class of transactions in both partitions will always be nonserializable.

Theorem 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 S 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. □

When the assignment of classes to partitions results in an mp-cyclic graph, some set X of classes must be removed until the remaining graph is mp-acyclic.[2]

Previously proposed schemes such as weighted voting ensure that the CCG will be mp-acyclic by their restriction that an item cannot be read in one partition and written in another. Thus, there will be no interference edges at all in the CCG. This suggests a way to extend previous approaches while guaranteeing serializability. We allow 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.

[2]To determine the size of a minimum such X is NP-complete [Wrig83].

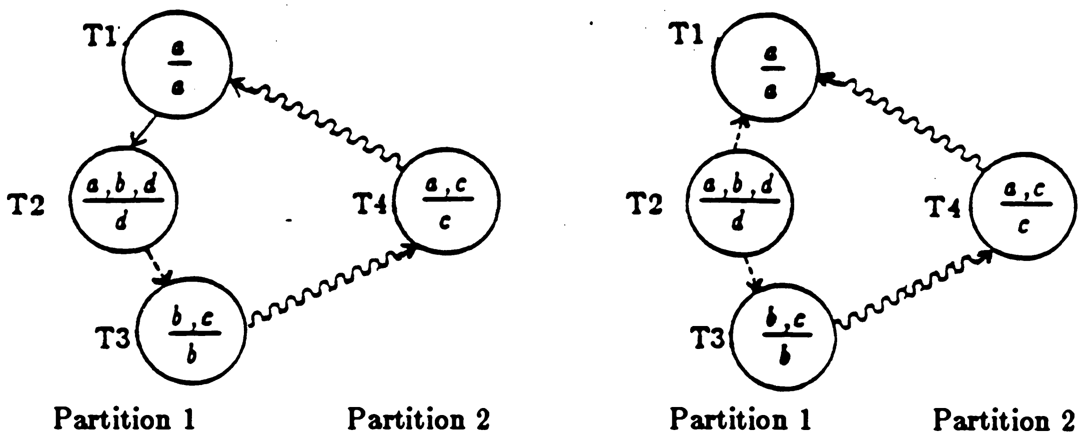
4. Using Multiple Versions

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 [BeGo82], [HsMa83], [PaKa82], [Reed78], [Thom79], *et al.* Multiversion schemes normally are employed to increase throughput. In this section, we investigate multiversion schemes for a different reason – to enrich the variety of transaction classes that can be accommodated in a partitioned DDBS.

As an example of how using older versions of data items can help, consider the cyclic serialization graph in Fig. 4.1(*a*). Since the serialization graph (*a*) is cyclic, the execution is not serializable. The execution would have been serializable, however, if T2 had used an older version of a, leading to the serialization graph (*b*).

4.1. Using Two Versions

G_{ip}, the class conflict graph restricted to interference and precedence edges, defines a partial order on its strongly-connected components. By using old versions, we will serialize



(*a*) Cyclic serialization graph. (*b*) Acyclic serialization graph.

Figure 4.1. Serialization graphs for two executions of a system.

transactions from classes in different components 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 4.1: Let G be a CCG and G_{ip} be the subgraph of G containing only interference and precedence edges. If G_{ip} is mp-acyclic, there is a 2-version scheme such that any sequence of transactions from the classes 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 4.1: 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 $COMPGRAPH(G)$,

$$LABEL((a,b)) = \bigcup (LABEL((x,y) \in E) : COMPONENT(x)=a \wedge COMPONENT(y)=b)$$

□

$COMPGRAPH(G)$ 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 G be a CCG such that G_{ip} is mp-acyclic, and let C be a transaction class with $READSET(C) = \{d_1, d_2, ..., d_j\}$ and $WRITESET(C) = \{d_1, d_2, ..., d_k\}$ $(k \leq j)$. 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 4.2: Rule 1 will produce a serializable schedule within a partition for an mp-acyclic G_{ip}.

Proof: See Appendix. □

Lemma 4.3: Let G_{ip} be an mp-acyclic. The serialization graph for any execution in this system using Rule 1 is acyclic.

Proof: See Appendix. □

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.

Lemma 4.4: 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 above can be safely run using current versions. A generalization of this lemma can be found in [Wrig83].

4.2. 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 advantageous to have more than two versions. Consider the CCG subgraph G_{ip} in Fig. 4.2, which is not mp-acyclic, although G_i is. Now consider the execution shown in the figure (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

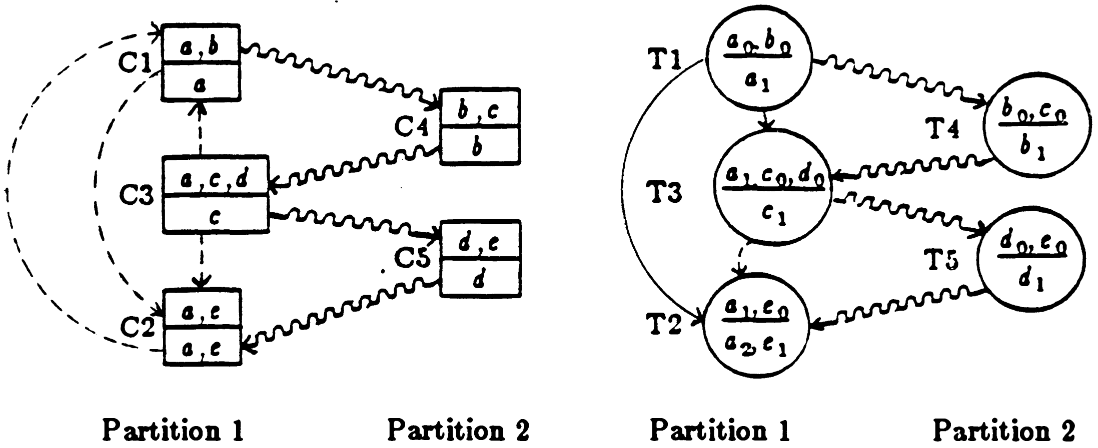


Figure 4.2. Mp-cyclic CCG G_{i_p} and execution requiring three versions of a.

most recent version.

Our general approach will be to take G_{i_p} 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 between members of the affected classes in any serialization graph. That is, if (A,B) is a precedence edge in G_{i_p}, 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. 4.2, which we will view from the perspective of Partition 1, we reverse edges C2 $\rightarrow$ C1 and C3 $\rightarrow$ C1; the resulting graph is mp-acyclic (Fig. 4.3). What we are requiring is that if any T with $CLASS(T) \in \mathrm{C2} \cup \mathrm{C3}$ runs, no T' with $CLASS(T') \in \mathrm{C1}$ 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 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

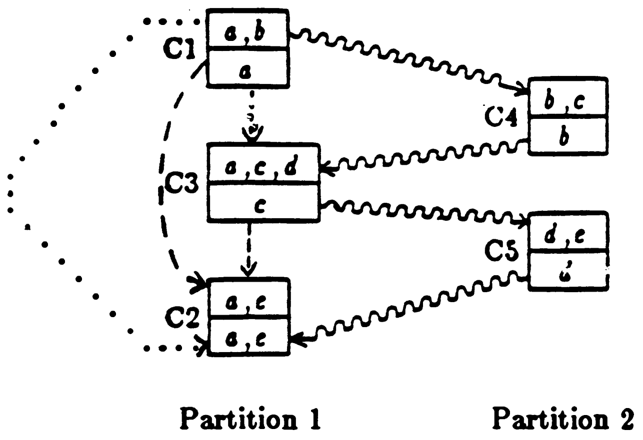


Figure 4.3. Mp-acyclic CCG subgraph G_{ipr}.

edge. In this way, some of both classes can be run.

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. We assume throughout 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.

If G_i is mp-acyclic, it is always possible to establish an mp-acyclic G_{ipr} by topologically sorting 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.

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 this can be done serializably. For the former problem, for each item in the database, we use ordering A (above) on 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

$(\forall d \in WRITESET(T)\text{: } tag(most\_recent(d)) \leq \pi(X))$.

(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. The proof of correctness of Rule 2 is a series of lemmas found in the appendix.

5. Discussion

Class conflict graphs are used to identify bad interactions between transactions that may occur when the network partitions—interactions that lead to nonserializable executions. Although the graphs have been defined assuming that the network partitions into two groups, the extension to arbitrary partitioning is straightforward.

Once bad interactions have been identified, three techniques can be used to preclude them: dependency-edge removal, edge reversal, and class removal. Dependency-edge removal (the technique of Section 4.1) is the best of the three in the sense that it always allows the affected classes to execute. To realize the full benefits of the technique, transactions are sometimes forced to read the original versions of data-items. Hence, two versions—the original and the current—of some items must be maintained.

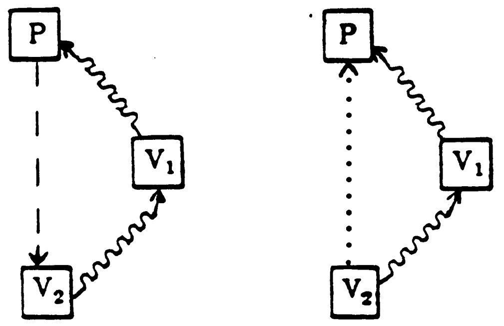
The purpose of edge reversal is to prohibit certain potentially dangerous precedence edges. A "reversed edge" from Ci to Cj means that transactions from class Ci can execute until a transaction from class Cj executes; thereafter, Ci is non-executable. The implementation of edge reversal requires in general multiple versions (>2) of some items.

Class removal is the most straightforward and most easily implemented of the three techniques. As the name implies, it does not allow any executions of transactions from the deleted classes. Not surprisingly, it is the least desirable technique. There are times, however, when it is the only recourse; namely, when cycles of interference edges exist.

Although the full benefits of dependence edge removal and edge reversal are reaped only if multiple versions are maintained, the techniques can still be used in systems retaining only a single version. In this case, a transaction can execute only if the versions prescribed in rules 1 and 2 are available. Even in a one-version system, these two techniques provide more flexibility than class removal. Consider the graph in Fig. 5.1(a). If class deletion is used, either the high priority class (P) or one of the the high volume classes (V_1 or V_2) must be removed. If edge reversal is used instead (see Fig. 5.1(b)), both high volume classes are executable until a high priority transaction is actually submitted. Thereafter, V_2 is no longer executable.

Critics levy two arguments against class-based schemes. The first is that in general-purpose systems the types of transactions are diverse and not known *a priori*; therefore, any classification is, by necessity, arbitrary. The second is that even in specialized systems (e.g. VISI-CAD systems, operating systems) where the types of transactions are readily known, interactions between types are involuted, making classification difficult.

We counter these criticisms by proposing these schemes for use only under extraordinary circumstances—specifically, when communication failures occur. Under such circumstances, even a coarse classification can significantly improve availability. In general purpose databases, it may be acceptable to restrict processing during a partitioning to high volume and high priority transactions, so only these need to be classified. Items not referenced in any high volume or high priority class can be made available by one of the traditional methods (such as



(a) G_{ip} (b) G_{ir}

Figure 5.1. An mp-cycle involving a high priority class (P) and high volume classes (V_1 and V_2) is resolved by edge reversal.

primary copy or weighted voting), which are simple strategies for classifying transactions. Even when a suitable classification is lacking, the notion of a "privileged partition" can always be applied.

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.

[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).

[BSW79] Bernstein, P.A., Shipman, D.W., and Wong, W.S. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Trans. Software Engineering*, v. SE-5, no. 3 (May 1979).

[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.

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

[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.

[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.

[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 1982.

[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.

[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).

[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.

[Wrig83] Wright, D.D. Managing Distributed Databases in Partitioned Networks (PhD Thesis). Technical Report TR 83-572, Dept. of Computer Science, Cornell University, September 1983.

APPENDIX

A.1. Correctness of Rule 1

Our approach is based on multi-version serialization graphs. 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 an execution I and order $\ll$, the *multi-version serialization graph,* denoted $MVSG(I,\ll)$, 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)—(b) for the single-version graph hold (see section 2). Edges are also added for the following reason.

(c) 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 $MVSG(I,\ll)$ is acyclic. However, it also contains a proof that deciding whether such an order exists is in general NP-complete. Fortunately, if readsets contain writesets, deciding is easy.

Proof: (Lemma 4.2) 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. □

Proof: (**Lemma 4.3**) 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. □

A.2. Correctness of Rule 2

Lemma A.1: 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 A.2: 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 A.3: 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 correspond 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 A.4: 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:

(1) $T_{new} \rightarrow T_{first}$. These precedence edges cannot lie on a cycle by Lemma A.3, since they must connect members of different components.

(2) $T_{new} \rightarrow T_{srev}$, 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 A.3. □

We can now establish the central result:

Theorem A.5: 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 A.3, these edges cannot lie on a cycle. Hence the serialization graph is acyclic and the transactions must be serializable. □