Optimism and Consistency In Partitioned Distributed Database Systems

SUSAN B. DAVIDSON
University of Pennsylvania

A protocol for transaction processing during partition failures is presented which guarantees mutual consistency between copies of data-items after repair is completed. The protocol is "optimistic" in that transactions are processed without restrictions during failure; conflicts are then detected at repair time using a *precedence graph*, and are resolved by backing out transactions according to some *backout strategy*. The resulting database state then corresponds to a serial execution of some subset of transactions run during the failure. Results from simulation and probabilistic modeling show that the optimistic protocol is a reasonable alternative in many cases. Conditions under which the protocol performs well are noted, and suggestions are made as to how performance can be improved. In particular, a backout strategy is presented which takes into account individual transaction costs and attempts to minimize total backout cost. Although the problem of choosing transactions to minimize total backout cost is, in general, NP-complete, the backout strategy is efficient and produces very good results.

Categories and Subject Descriptors: H.2.2. [**Database Management**]: Physical Design—*recovery and restart*; H.2.4 [**Database Management**]: Systems—*distributed systems, transaction processing*

General Terms: Performance, Reliability

Additional Key Words and Phrases: Serializability, network partitioning, consistency

1. INTRODUCTION: DESCRIPTION OF THE PROBLEM

Partition failures are a major threat to the reliability of distributed database systems and to the availability of replicated data. A partition failure is said to occur when subsets of the database sites can no longer communicate due to a failure in the communication subsystem. However, in many systems (notably SDD-1), it is impossible to differentiate a failure in the communication subsystem from site failure. In such systems, partition failures are caused by either communication subsystem or site failures, and thus occur more frequently. Since there may be replicated data in distributed database systems, transaction processing protocols must guarantee that mutual consistency is maintained between

This paper is based upon work supported in part by the National Science Foundation under grant ECS-8019393, done at Princeton University.
Author's address: Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104.

copies of data-items. During partition failures, however, loss of communication between sites may allow copies of data-items to diverge, unless restrictions are imposed on the transaction processing protocol. Most of the protocols which have been suggested for transaction processing during partition failures assume that transactions cannot be backed out once they have been committed; for example, a transaction which hands out cash to a customer is irreversible. These protocols, therefore, avoid executing conflicting transactions, and guarantee mutual consistency throughout the partition failure by limiting the availability of replicated data. Rules are given which guarantee that each (replicated) data-item is accessible in at most one partition group; updates are simply forwarded at recovery. Examples of such protocols are voting [11, 27], tokens [19], and primary site [2, 22]. Such restrictions are quite severe, and there is no guarantee that every data-item is accessible in at least one partition group: a majority vote may not be obtainable in any partition group, tokens may get lost, and the primary site may crash without a backup. The reliability of the system is severely degraded in terms of users being able to run *what* jobs they need to get done, *when* they need to do them, and *where* they want to run them (see [5] for a survey of these and other partition failure protocols).

The protocol advocated in this paper is "optimistic" in the sense that, during partition failures, no restrictions are imposed on the users of sites which are up. The system attempts to process all transactions, but must delay commitment until recovery is completed, at which point conflicting transactions are backed out to regain mutual consistency. Note that this assumes that transactions can be backed out. This protocol is used by default in many systems (e.g., SDD-1); that is, the system does not have a protocol for transaction processing during partition failures, so conflict detection and transaction backout are performed manually at recovery. Parker, et al. [22] have proposed an automatic conflict detection scheme for file systems, and have extended it to transactions which access more than one file [23]. However, resolving the inconsistencies is not straightforward and is essentially left up to the user. In this paper, a method for automatic conflict detection and transaction backout is proposed which can be used in a general distributed system with replicated data. A model of conflicts between transactions with partitioned data is developed, called a *precedence graph*. Conflicts are detected from cycles in the graph and are resolved by a transaction *backout strategy* which makes inconsistent databases consistent at repair time. Backed-out transactions can then be automatically rerun by the system, or referred back to the user.

This approach is attractive since the brunt of the failure is felt more by the system than by the user. Availability has not been compromised. In some applications this is very important. For example, in military command and control applications, partitions may occur because of an enemy attack, and it is precisely at this time that we do not want transaction processing halted. In an airline reservation system it may be too expensive to have a high-connectivity network, and partitions may occur periodically. Many transactions are executed each second, and each transaction that is not processed may represent the loss of a customer. The airline may therefore be willing to take the risk of temporarily overbooking a flight, and allow later cancellations to rectify the situation. In

other situations, few conflicts may actually occur, either because conditions in the database minimize the probability of conflict or because semantic knowledge or a knowledge of data-reference patterns makes conflicts highly unlikely. For example, in a banking system where each site is a branch office containing account information for all customers, transactions on accounts probably will not conflict; it is very unlikely that a single customer will attempt to access his account at two different branches during a single partition failure. The personal nature of accounts and geographic restrictions of customers render conflict highly unlikely. To further minimize conflict, yet maintain customer satisfaction, routine functions such as clearing checks and crediting deposits could be delayed by the bank until the failure is repaired. The protocol is also resilient in the face of multiple partition failures (see Section 5). The advantages of automating conflict detection and transaction backout are obvious: they can be performed very quickly, thus decreasing the temporary delay in transaction processing while the system recovers from the partition failure, and the cost of backing out transactions can be kept relatively low if a smart backout strategy is used.

Section 2 describes automatic conflict detection and backout in detail. Performance results from simulation and probabilistic modeling are discussed in Section 3, and conditions under which the protocol performs well are noted. Results from the performance evaluation are then used in Section 4 to develop a backout strategy which, given individual transaction costs, attempts to minimize the total backout cost. The strategy is shown to be efficient and to produce good results. Finally, suggestions as to when and how this approach could be useful are made in Section 5, along with a discussion of extensions to the protocol and directions for future research.

2. THE OPTIMISTIC PROTOCOL

Conflicts and interactions between transactions in centralized and distributed systems are often modeled by graphs ([3, 4, 20]; see [1] for fundamental concepts of graph theory). A graph theoretic approach is useful for analyzing local or global histories in proving "correctness" (i.e., serializability). Typically, nodes represent transactions and conflicts are represented by cycles; acyclic graphs can be used to produce an equivalent serial history by a topological sort of the nodes. In this section, these graph techniques will be applied to partitioned distributed systems; when recovery occurs and two partition groups P_1 and P_2 discover that they can communicate, a graph will be created from the global histories of P_1 and P_2. Cycles in this graph will represent conflicts between transactions in P_1 and P_2. By backing out certain transactions, we can make the graph acyclic, which means that the global histories of P_1 and P_2 can be merged to form a single, serial global history for $P_1 \cup P_2$, thus mutual consistency is regained.

2.1 Assumptions and Basic Definitions

A fully replicated distributed database is the context in which partition failures will be studied. This special case of a distributed database has been chosen because it is a simplified model and is the kernel of the partition problem; mutual consistency is threatened during a partition only when multiple copies exist. The database consists of a collection of *data-items* $\{d_1, \ldots, d_M\}$ which is stored

at every *site* in the system. Data-items are operated on by *transactions* consisting of *READ* and *WRITE* actions, and local computation. The set of all data-items read (written) by a transaction T will be denoted *READSET*(T)(*WRITESET*(T)). Transactions have the property that the value which is written for a data-item d functionally depends on the values read by that transaction and on the previous value of d. Note that this implies that *WRITESET*(T) $\subseteq$ *READSET*(T). At some point in time (time used intuitively), according to some *transaction protocol*, transaction T is executed at a site in the system by performing the *READ* and *WRITE* actions and sending update messages to other sites in the system to inform them of the new values which have been written. It is assumed that every message sent is eventually delivered.

A *partition group* is a maximal subset of sites in the system which can communicate. New partition groups are created when failure occurs, either node or communication subsystem failure, and also when *recovery* occurs (two previously separate partition groups discover that they can communicate). A system is *partitioned* as long as there is more than one partition group present. Note that although we only discuss the case of merging two partition groups, recovery may not always be pairwise (this extension to the protocol is discussed in Section 5).

2.2 A Graph Theoretic Approach to Automatic Conflict Detection and Automatic Back Out: An Overview

It is assumed that in each partition group there is exactly one site designated as coordinator, and that update messages are sent only to reachable sites. When a site in one partition group P_1 discovers that it can communicate with a site in another partition group P_2, the coordinators in P_1 and P_2 are notified, and local processing in each group ceases. The coordinator in P_i then derives a total ordering of the transactions performed in its partition group during the failure, called the *global history* H_i [3, 28], and the *READSET* and *WRITESET* of each transaction (for a discussion of how to derive this global history see [6]). Note that this assumes that the transaction protocol executing within each partition group produces a serializable global history for that group (i.e., that such an ordering can be obtained). Using the global histories of P_1 and P_2, H_1 and H_2, a *precedence graph* G (see Definition 2.2.1) is constructed. Conflicting transactions are detected from the cycles of G, and transactions are backed out until G is acyclic. We assume that any transaction can be backed out. *Backing-out* a transaction T involves setting the value of each data-item in *WRITESET*(T) to the value it had when it was read by T. The databases of P_1 and P_2 are then merged by sending update messages of nonbacked-out transactions in P_1 to P_2 (and vice versa), and a new coordinator is elected for the new partition group $P_1 \cup P_2$ (see [9] for a discussion of elections in distributed systems, and [3, 4, 20] for a description of conflict graphs in unpartitioned systems).

The precedence graph. We say that the global histories of the partition groups H_1 and H_2 are serializable if and only if there exists some H, a total ordering of the transactions in both H_1 and H_2, to which they are *equivalent*. That is, given any set of initial values for the data-items in the database and any interpretation of the transactions in H_1 and H_2, if H were executed in $P_1 \cup P_2$ then each

transaction in H would read or write the same values as it read or wrote in the execution of H_1 in P_1 and H_2 in P_2. Executing history H in partition P involves (1) serially executing the transactions of H in the order defined by H, and (2) forwarding updates from one transaction to all sites in P before the next transaction is executed. To achieve mutual consistency with serializable H_1 and H_2, all we need to do is to forward updates in H_2 to P_1 and in H_1 to P_2 (in the order that the updates were generated). When such an H exists, it will be called the *merged history* of H_1 and H_2.

Example. Let $H_1 = T_{11}, T_{12}, T_{13}$ where

$$READSET(T_{11}) = WRITESET(T_{11}) = \{d_1, d_2\},$$
$$READSET(T_{12}) = \{d_2, d_3\},\ WRITESET(T_{12}) = \{d_3\},$$
$$READSET(T_{13}) = \{d_3, d_4, d_5\},\ WRITESET(T_{13}) = \{d_4\},$$

and $H_2 = T_{21}, T_{22}$ where

$$READSET(T_{21}) = WRITESET(T_{21}) = \{d_5\},$$
$$READSET(T_{22}) = \{d_1, d_5\},\ WRITESET(T_{22}) = \{\ \}.$$

Clearly T_{22} must precede T_{11} since it reads d_1, which forces the merged history to be $H = T_{21}, T_{22}, T_{12}, T_{13}$. But this is incorrect, since T_{13} then reads d_5 after T_{21}, which has altered the value for d_5. Hence, there is a conflict between T_{11}, T_{12}, T_{13} and T_{21}, T_{22}, and H_1 and H_2 are not serializable.

Definition 2.2.1. Given $H_1 = T_{11}, \ldots, T_{1N_1}$ and $H_2 = T_{21}, \ldots, T_{2N_2}$, the *precedence graph* $G(H_1, H_2) = (V, E)$ is the directed graph defined by

(1) V = the set of all transactions in H_1 and H_2,
$\{T_{11}, \ldots, T_{1N_1}\} \cup \{T_{21}, \ldots, T_{2N_2}\}$
(2) E = {*Ripple Edges*} ∪ {*Precedence Edges*} ∪ {*Interference Edges*}

(a) *Ripple Edges*—represent the fact that one transaction read a value produced by another transaction in the same partition.
$T_{ij} \rightarrow T_{ik}$ if and only if $j < k$ and there exists some d such that d is in $WRITESET(T_{ij}) \cap READSET(T_{ik})$ and there is no $l, j < l < k$ such that d is in $WRITESET(T_{il})$.

(b) *Precedence Edges*—represent the fact that a transaction read a value which was later changed by a second transaction in the same partition.
$T_{ij} \rightarrow T_{ik}$ if and only if $j < k$, there is no $T_{ij} \rightarrow T_{ik}$ ripple edge; there exists some d such that d is in $READSET(T_{ij}) \cap WRITESET(T_{ik})$ and there is no $l, j < l < k$ such that d is in $WRITESET(T_{il})$.

(c) *Interference Edges*—represent the fact that if a transaction in one partition reads a data-item d, it must precede any transaction which writes a new value for d in the other partition.
$T_{1i} \rightarrow T_{2j} (T_{2i} \rightarrow T_{1j})$ if and only if there exists some d such that d is in $READSET(T_{1i}) \cap WRITESET(T_{2j}) (READSET(T_{2i}) \cap WRITESET(T_{1j}))$.

THEOREM 2.2.2. *Given H_1 and H_2, the precedence graph $G(H_1, H_2)$ is acyclic if and only if H_1 and H_2 are serializable (i.e., equivalent to some merged history H).*

PROOF. ($\Rightarrow$) Since $G(H_1, H_2)$ is acyclic, we can perform a topological sort of its nodes to produce a serial ordering of transactions H. The claim is that H_1 and

H_2 are equivalent to H. That is, given any initial state, the values in the final state after executing H in $P_1 \cup P_2$ are the same as the values in the final state after executing H_1 followed by updates from H_2 in P_1, and H_2 followed by updates from H_1 in P_2. Throughout this proof, G will be used for $G(H_1, H_2)$. We will also assume that there is a fictitious first transaction in each history H, H_1, and H_2 that reads and writes each data-item without changing its value. This avoids the problem of values being read from the initial state.

Claim 1. Each transaction executed in H reads and writes the same values as it reads and writes in H_1 or H_2.

(1) The values read by transactions are correct.

Suppose not. Then there exists some transaction T and data-item d such that T reads value y in H_i for d and value x in H for d. Let $T^{\wedge}$ be the transaction that writes the value of d read by T in H.

Case 1. (T and $T^{\wedge}$ are in different partitions): An interference edge from T to $T^{\wedge}$ forces T to precede $T^{\wedge}$ in H. Contradiction.

Case 2. (T and $T^{\wedge}$ are in the same partition H_i, and $T^{\wedge}$ follows T in H_i): A precedence edge from T to $T^{\wedge}$ forces T to precede $T^{\wedge}$ in H. Contradiction.

Case 3. (T and $T^{\wedge}$ are in the same partition H_i, and $T^{\wedge}$ precedes T in H_i): There must be some transaction T^* that writes x, the value of d read by T in H_i. Therefore, the path of ripple edges, $T^{\wedge} \rightarrow \cdots \rightarrow T^* \rightarrow T$ must occur in H_i. This forces the relative order of these transactions in H. So $T^{\wedge}$ does not write the value of d read by T in H. Contradiction.

(2) The values that would be written by transactions in H are correct. This follows from the definition of functional dependence and by the fact that the values read are correct (for a given set of values read, any transaction will always write the same values).

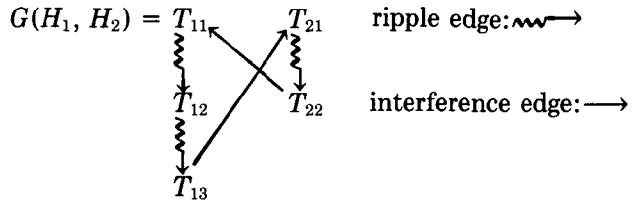
Claim 2. Each data-item is updated by transactions in at most one partition. Otherwise a cycle results. Contradiction.

Claim 3. The final value for each data-item d in H is written by the same transaction as it was in H_1 and H_2. Since each data-item d is updated in at most one of H_1 and H_2, there is at most one "last" transaction in H_1 and H_2 to update d. H is a topological sort of G, so the "write" ordering of transactions is preserved (by ripple edges). Hence, the last transaction to write d in H_1 and H_2 is the last transaction to write d in H.

So for each H_i, any data-item d updated within P_i has the same final value as in H since it is written by the same transaction T (Claim 3) in H_i as in H, T writes the same value in H_i as in H (Claim 1), and the fact that updates from the other partition will not include a value for d (Claim 2). Any data-item d that is not updated in P_i, but is updated in the other partition, will receive the modified correct value when updates are exchanged. Finally, any item d not updated in any partition will not be modified by H either. So H is equivalent to H_1 and H_2, and hence H_1 and H_2 are serializable.

($\Leftarrow$) Suppose not. Let H be a merged global history equivalent to H_1 and H_2. If G contains a cycle, then there must be transactions T_a and T_b such that T_a precedes T_b in H, but there is an edge from T_b to T_a in G. This implies that T_a reads a different value in H than in H_1 (or H_2, depending on where T_a was executed). Therefore, H_1 and H_2 are not equivalent to H. Contradiction. □

Example



$G(H_1, H_2)$ contains a cycle, and so H_1 and H_2 are not serializable.

Theorem 2.2.2 tells us that if $G(H_1, H_2)$ is acyclic, then all sites in P_1 can be updated to reflect the transactions that occurred in P_2, all sites in P_2 can be updated to reflect the transactions that occurred in P_1, and mutual consistency is again achieved. If $G(H_1, H_2)$ is not acyclic, transactions must be backed-out to achieve mutual consistency. This amounts to finding some acyclic subgraph of $G(H_1, H_2)$. However, note that if transaction T is chosen to be backed-out, then every transaction to which there is a directed path of ripple edges from T must also be backed-out, since they functionally depend upon T. In the above example, if T_{11} is chosen to be backed-out, then T_{12} and T_{13} must also be chosen to be backed-out, due to the path of ripple edges $T_{11} \rightarrow T_{12} \rightarrow T_{13}$. Suppose then that

(A1) $H_1^{\wedge}$ and $H_2^{\wedge}$ are subsequences of H_1 and H_2 which represent the transactions that have not been chosen to be backed-out.

(A2) There is no path of ripple edges from a transaction in $H_i - H_i^{\wedge}$ to a transaction in $H_i^{\wedge}$ (i.e., no transaction in $H_i^{\wedge}$ functionally depends on a transaction which has been backed-out).

(A3) The precedence graph $G(H_1^{\wedge}, H_2^{\wedge})$ is acyclic.

Then what we need to prove is that there exists a merged history which accurately models the behavior of the system during the failure and through recovery (i.e., that there exists an H which is equivalent to executing H_1 in P_1 and H_2 in P_2), backing-out the transactions in $(H_1 - H_1^{\wedge})$ and $(H_2 - H_2^{\wedge})$, and forwarding updates from transactions in $H_1^{\wedge}$ to P_2 and $H_2^{\wedge}$ to P_1. Note that, for each data-item d modified by a transaction in H_i, we only need to forward the update generated by the last transaction in H_i that modifies d; that is, we only need the final value of d to merge the two partitions. Also note that transactions must be backed-out in reverse order; that is, for each data-item modified by a transaction in $H_i - H_i^{\wedge}$ we need the value read by the earliest transaction in $H_i - H_i^{\wedge}$.[1]

[1] Graph reduction techniques can also be used to substantially reduce the size of the precedence graph (see [29]).

THEOREM 2.2.3. *Given* $H_1, H_1^\wedge$ *and* $H_2, H_2^\wedge$, *which follow the assumptions* (A1), (A2), *and* (A3), *then the following are serializable*:

(1) *In* P_1: *executing* H_1, *followed by the backing-out of all transactions in* $(H_1 - H_1^\wedge)$, *followed by all updates from transactions in* $H_2^\wedge$ *in the order in which they were generated*; *and*
(2) *in* P_2: *executing* H_2, *followed by the backing-out of all transactions in* $(H_2 - H_2^\wedge)$, *followed by all updates from transactions in* $H_1^\wedge$ *in the order in which they were generated.*

PROOF. Since $G(H_1^\wedge, H_2^\wedge)$ is acyclic (assumption (A3)), then $H_1^\wedge$ and $H_2^\wedge$ are serializable by Theorem 2.2.2, and are equivalent to the history produced by a topological sort of $G(H_1^\wedge, H_2^\wedge)$. It is enough to show that executing $H_i^\wedge$ in P_i is equivalent to executing H_i in P_i, followed by the backing-out of all transactions in $(H_i - H_i^\wedge)$.

Claim 1. Each transaction in $H_i^\wedge$ would read (and hence write by the definition of functional dependence) the same values as in H_i. This follows from the fact that the relative order of transactions is the same in $H_i^\wedge$ as in H_i ($H_i^\wedge$ is a subsequence of H_i, assumption (A1)), and that every transaction in $H_i^\wedge$ functionally depends on the same transactions as it did in H_i (assumption (A2)).

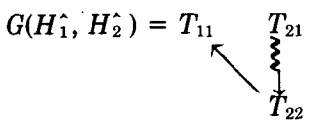
Claim 2. Each data-item d has the same final value in $H_i^\wedge$ as it does after executing H_i, followed by the backing-out of transactions in $H_i - H_i^\wedge$. Since transactions read and write the same values in $H_i^\wedge$ as in H_i (Claim 1), it is enough to show that the same transaction writes the final value for d in $H_i^\wedge$, as in H_i.

Case 1. d is not updated in H_i. Then it is not updated in $H_i^\wedge$, and the value of d is the initial value.

Case 2. d is updated in both H_i and $H_i^\wedge$. Consider the last transaction to update d in $H_i^\wedge$, T_{d1}. All subsequent transactions in H_i which update d have been backed-out by the definition of $H_i^\wedge$ (assumption (A2)). Let T_{d2} be the next transaction after T_{d1} to update d in H_i. By assumption (A2), T_{d2} is the earliest transaction in $(H_i - H_i^\wedge)$ to update d. So the value of d after backing-out $(H_i - H_i^\wedge)$ is the value read by T_{d2} (i.e., the value written by T_{d1}).

Case 3. d is updated in H_i but not in $H_i^\wedge$. Then the earliest transaction to update d in $(H_i - H_i^\wedge)$ is the earliest transaction to update d in H_i. The value read by this transaction is the initial value of d. □

Example. If we choose to backout T_{12}, then T_{13} must be backed-out, as it functionally depends on T_{12}. Then $H_1^\wedge = T_{11}$, $H_2^\wedge = T_{21}, T_{22}$,



G is acyclic and so $H_1^\wedge$ and $H_2^\wedge$ are serializable. Specifically, they are equivalent to the merged history $H = T_{21}, T_{22}, T_{11}$, produced by a topological sort of the nodes of G.

Note that this procedure graph definition differs from that of augmented ancestor graphs and concurrency control graphs [4, 7, 26] in several respects. First, the precedence graph represents two execution sequences which must be merged. It is static; the transactions have already executed. Second, the edge semantics are different. Precedence and ripple edges represent a required ordering of transactions in the same partition, with ripple edges dictating the effect of backing-out a transaction. Interference edges represent potential conflicts between transactions in different partitions, which may lead to violations of mutual consistency. Last of all, there is more freedom in choosing which transaction to back-out to break cycles. Concurrency control graphs are generated on the fly, and are used to prevent committing transactions, which would cause a violation of mutual consistency. With partitions, everything is static and transactions have already run to completion. Any transaction may be chosen to break cycles. The next section discusses this backout selection problem.

2.3 Backout Selection Strategies

Given the global histories of the partitions to be merged, it would be nice to minimize the number of transactions that are backed-out. Even better would be to assign a weight or backout cost to each transaction, and then minimize the total backout cost. Unfortunately, just minimizing the number of transactions backed-out (assigning each transaction a weight of one) is NP-complete, hence minimizing total backout cost (assigning transactions different weights) is also NP-complete. (For a discussion of the theory of NP-completeness, see [8].)

Transaction backout minimization problem. Given the precedence graph $G = (V, E)$, find a subset $V\hat{} \subseteq V$ such that: (1) there are no nodes in $(V - V\hat{})$ reachable via ripple edges from nodes in $V\hat{}$, (2) the subgraph $G\hat{}$ of G induced by the nodes of $V\hat{}$ is acyclic, and (3) $| V\hat{} |$ is minimized.

The associated decision problem is the transaction backout problem.

Transaction backout problem. Given the precedence graph $G = (V, E)$ and integer K, is there a subset $V\hat{} \subseteq V$ such that: (1) there are no nodes in $(V - V\hat{})$ reachable via ripple edges from nodes in $V\hat{}$, (2) the subgraph $G\hat{}$ of G induced by the nodes of $V\hat{}$ is acyclic, and (3) $| V\hat{} | \leq K$?

Since the decision problem can be no harder than the associated minimization problem, it suffices to show that the transaction backout problem is NP-complete.

THEOREM 2.3.1. *The transaction backout problem with a fixed maximum number of partitions N ($N \geq 2$) and an arbitrarily large number of data-items in the database is NP-complete.*

PROOF. (in NP) Guess a subset $V\hat{} \subseteq V$. Checking that $| V\hat{} | \leq K$, that $V\hat{}$ is closed with respect to ripple edges, and that $G\hat{}$ is acyclic can all be done in polynomial time.

(NP-hard) Reduce the known NP-complete feedback vertex set problem (FVS) to the transaction backout problem (TB). The feedback vertex set problem [15] is: given a graph $G = (V, E)$ and integer K, is there a subset $V\hat{} \subseteq V$ such that $V\hat{}$ contains at least one vertex from every directed cycle in G, $| V\hat{} | \leq K$? So,

given input $G = (V, E)$, K for FVS, insert a new node v_{ij} into every edge $v_i \rightarrow v_j$ to create the precedence graph $G_1 = (V_1, E_1)$. That is, V_1 consists of the old nodes from V together with the new nodes inserted in every edge; E_1 consists of the edges $v_i \rightarrow v_{ij}$, $v_{ij} \rightarrow v_j$ for every old edge $v_i \rightarrow v_j$. Label all edges as *interference edges*. Note that this corresponds to a valid precedence graph of two partitions, since:

(1) Interference edges always connect nodes in different partitions; here, the new nodes represent one partition and the old nodes another partition. Each edge $e_k: v_i \rightarrow v_j$ forces a $READ(d_k)$ in the transaction represented by node v_i and a $WRITE(d_k)$ in the transaction represented by node v_j, where d_k is a data-item unique for the edge e_k.

(2) The subgraphs of new nodes and old nodes are acyclic since they contain no edges. Call this new graph $G_1 = (V_1, E_1)$ and submit G_1 and K as input to TB.

Solution to FVS $\Rightarrow$ *Solution to TB.* Suppose there is a subset $\hat{V} \subseteq V$, $| \hat{V} | \leq$ K and $\hat{V}$ contains at least one vertex from every directed cycle of G. Then the removal of $\hat{V}$ from G would break all cycles, making this subgraph of G acyclic. Since $V \subseteq V_1$, it follows that $\hat{V} \subseteq V_1$. Removal of $\hat{V}$ from V_1 would also break all cycles of G_1 since, in the creation of E_1, the removal of e from E and replacement by e_1, e_2 does not create any new cycles. Since there are no ripple edges in G_1, $\hat{V}$ is closed with respect to ripple edges.

Solution to TB $\Rightarrow$ *Solution to FVS.* Suppose there is a solution $\hat{V_1}$ to TB. Then $| \hat{V_1} | \leq K$ and the removal of $\hat{V_1}$ from V_1 breaks all cycles in G_1. $\hat{V_1}$ may contain nodes that are not in V, specifically some new nodes inserted into edges of E. Note that, by construction, each new node v has exactly one incoming and one outgoing edge: $v_i \rightarrow v \rightarrow v_j$, where $v_i, v_j \in V$. Let V_{FVS}, the solution to FVS, be constructed from $\hat{V_1}$ as follows: each old node v in $\hat{V_1}$ ($v \in V$) is added to V_{FVS}. For each new node v ($v \in V$), add to V_{FVS} the node incident on the incoming edge v_i. Since $v_i \in V$, it follows that V_{FVS} contains only nodes in V. Furthermore, $| V_{\text{FVS}} | \leq | \hat{V_1} | \leq K$, since at most one node is added to V_{FVS} for each node in $\hat{V_1}$. We now claim that removing V_{FVS} breaks all cycles in the original graph G. Suppose not. Then there is a cycle in G none of whose nodes are in V_{FVS}. Since the corresponding cycle in G_1 was broken, one of the new nodes in the cycle in G_1 was chosen. Then the (unique) old node incident on the incoming edge to v, v_i, was put in V_{FVS}, and therefore the cycle in G was broken. Contradiction. □

Although this result discourages attempts to minimize the total backout cost, there are well-known algorithms for enumerating all cycles in a directed graph, which can be used for a polynomial time (albeit nonoptimal) backout strategy (see, for example, [14, 24]). However, since the total number of cycles in a precedence graph can be exponential in the number of nodes in the graph, a backout strategy should break cycles as they are detected rather than enumerating all cycles and then attempting to break them. The next section discusses performance results from simulations of the optimistic protocol using several different backout strategies; details can be found in Appendix 1.

3. PERFORMANCE EVALUATION

Simulations were run which generated transactions for each partition group, computed the precedence graphs, and determined the percentage of transactions which had to be backed-out to make the graph acyclic, using a given backout strategy. They were used to measure the performance and feasibility of the optimistic protocol as a whole, as well as to study the effects of different backout strategies (see Appendix 1; [29] contains more extensive testing results). The results indicated conditions under which the optimistic protocol performs well and gave insight into how cycles should be detected, leading to an improved backout strategy, presented in the next section.

The following input parameters and assumptions were used in the simulation:

(1) there were M data-items in the (completely replicated) database;
(2) $N_1(N_2)$ transactions were processed in $P_1(P_2)$;
(3) references to data-items were uniformly distributed over the M data-items;
(4) the number of items referenced by each transaction $T(|READSET(T)|)$ was based on a truncated exponential distribution with mean I;
(5) the mean percentage of read-only transactions in each partition was RO;
(6) for the read–write transactions, the mean percentage of updated items was U;

For each set of parameters N_1, N_2, RO, U, M, and I, 50 to 100 trials were made. In each of the graphs in Appendix 1, points correspond to the mean of these trials.

The choice of backout strategy had a noticeable effect on the performance of the protocol. One that performed particularly well detected and broke two-cycles (cycles involving only two nodes, one from P_1 and one from P_2) before looking for longer cycles: not only were fewer transactions backed out on the average, but the simulation ran much more quickly since two-cycles are easier to detect than longer cycles. In fact, it was observed that very few long cycles remained after all two-cycles were broken (see Figure 3.1).

The simulation was also run changing the distribution of references to data-items. New parameters were A, p_1, and $p_2 : A$ of the M data-items were referenced with probability p_1 and the remaining $(M - A)$ data-items were referenced with probability p_2,

$$A * p_1 + (M - A) * p_2 = 1.$$

Input to the simulation was modified to specify $(M - A):A$ and $p_2:p_1$. Figure 3.2 compares results using a uniform distribution of reference to results using the 80–20 rule: 20 percent of the data-items were referenced 80 percent of the time. Performance deteriorated since the effective number of data-items decreased; that is, most of the transactions referenced a very small portion of the database, creating a lot of conflicts. This argues that references should be uniformly distributed over a large number of data-items (assuming that references from different partitions intersect at all).

In general, the protocol seems to perform best when

(1) the total number of transactions is small,
(2) there is a large percentage of read–only transactions,

Total number of transactions	M	% times of long cycles	Average number of long cycles
10	100	5	1.0
20	100	34	1.2
10	500	0	0.0
20	500	6	1.0
30	500	10	1.0
40	500	14	1.0
60	500	30	1.3
10	750	0	0.0
20	750	0	0.0
30	750	10	1.0
40	750	6	1.0
60	750	14	1.1
80	750	40	1.4
10	1000	0	0.0
20	1000	0	0.0
30	1000	3	1.0
40	1000	4	1.0
60	1000	6	1.0

Fig. 3.1. Simulation results. Percentage of times long cycles remain after breaking all two-cycles; and the average number of long cycles remaining when resulting graph is not acyclic.
($I = 5$, $RO = 0.5$, $U = 0.5$, Backout Strategy 2.)

(3) the percentage of updated items within read–write transactions is small,
(4) there is a large number of data-items in the database.

Distributed database systems using this protocol during partition failures could encourage the above conditions by increasing the granularity of data-item reference when failures occur, thus effectively increasing the number of items in the database. A warning message could also be given to users of the system, explaining that there is a partition failure and that transactions cannot be committed until recovery. It could also suggest that read–write transactions should only read the data-items relevant to the results written (that is, values in $WRITESET(T)$ should indeed be functionally dependent on values in $READSET(T)$) and that nonurgent read–write transactions should be delayed until repair is completed. Users then have the freedom to use their knowledge of the system to temper their actions during failure: users who know that they will be the only ones assessing the portion of the database they are interested in could continue to run transactions freely.

These performance results were verified by probabilistic analysis, and a formula for the expected number of transactions backed-out when a strategy which breaks two-cycles first is used was derived. These results can be found in [6].

The optimistic protocol can perform very poorly, and then it can perform very well. It is clearly *not* feasible in situations where a very large percentage of transactions (say 40 percent or more) are on the average backed-out; it would be better to temporarily disable all of the sites in one of the partitions, or adopt some other restrictive transaction protocol, than to have such a large backout

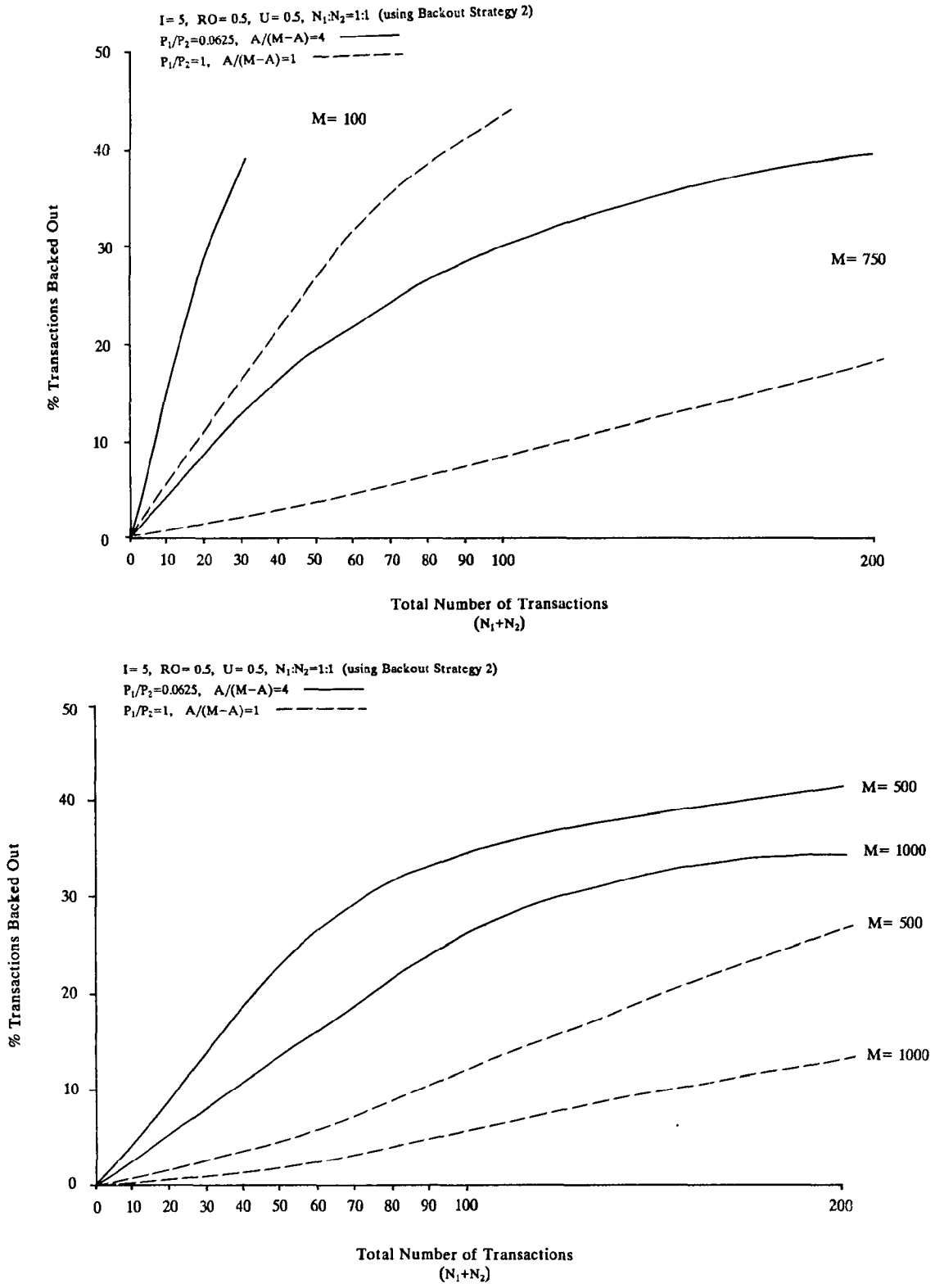


Fig. 3.2. Simulation results. Percentage of transactions backed-out versus total number of transactions (using Backout Strategy 2).

rate. Each application must decide what is an acceptable backout rate. A maximal backout rate of 5 or 10 percent may be quite acceptable to applications whose primary purpose is to keep users or customers happy. As in the example used in the Introduction, if branches of a bank represented the sites in a completely replicated distributed database system and a partition failure occurred, the bank would certainly lose a customer if it refused to process a withdrawal or deposit a transaction on the customer's account. The bank should be willing to accept responsibility for the failure of its own system and automatically rerun backed-out transactions without inconveniencing the customer. Of course, the foxy customer who knows of (or causes) the partition failure could swindle the bank by withdrawing all the money from his account at two different branches during the failure. However, this requires quite an effort on the customer's part; stores take a greater risk every day when they accept personal checks, and generally know less about the customer than the bank does. Banks may be willing to take such risks to maintain their clientele.

4. MINIMAL BACKOUT OF TWO-CYCLES

The problem of selecting the minimum number of transactions whose deletion breaks all cycles in the precedence graph is, in general, NP-complete. However, suppose we concentrate on breaking all two-cyles optimally, and forget about longer cycles. Since longer cycles in practice often do not exist after all two-cycles have been broken, there is a very good chance that this will break *all* cycles optimally. It turns out that this problem has a polynomial-time solution.[2] That is, finding the minimum number of nodes to break all cycles in a precedence graph consisting only of two-cycles and ripple edges has a solution which is polynomial in the number of nodes in a partition.

Strategy 4. *Breaking two-cycles optimally.* (Strategies 1, 2, and 3, in Appendix 1)

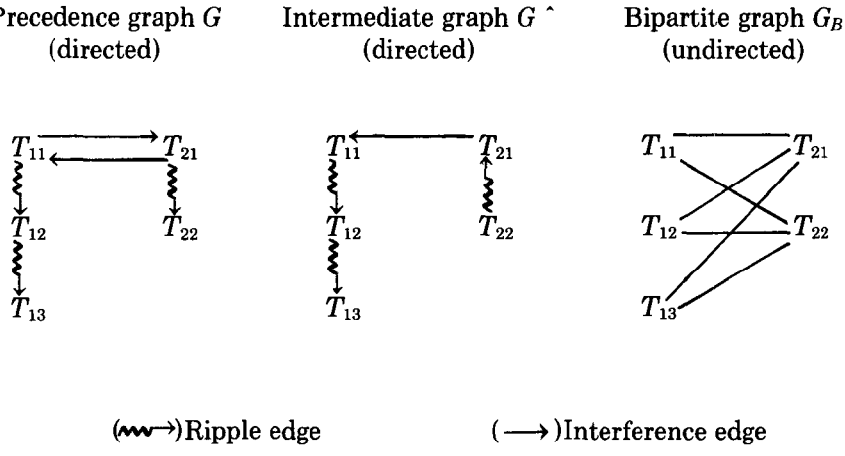
Step 1. Transform the directed precedence graph $G = (V, E)$ to an (undirected) bipartite graph ([21]) $G_B = (V_1, V_2, E_B)$ as follows:

(1) V_1 consists of all nodes in V corresponding to transactions in P_1; V_2 consists of all nodes in V corresponding to transactions in P_2.
(2) To compute E_B: Create the intermediate directed graph $G^{\wedge} = (V^{\wedge}, E^{\wedge})$ where $V^{\wedge} = V$ and $E^{\wedge}$ consists of all ripple edges in E between nodes in P_1, the reverse of all ripple edges in E betweeen nodes in P_2, and all interference edges from a node in P_2 to a node in P_1 that are part of a two-cycle in G. Compute the transitive closure of $G^{\wedge}$; add to E_B all (undirected) edges $v_i - v_j$ such that $v_i \in V_2$, $v_j \in V_1$, and $v_i \rightarrow v_j$ is a (directed) edge in the transitive closure of $G^{\wedge}$.

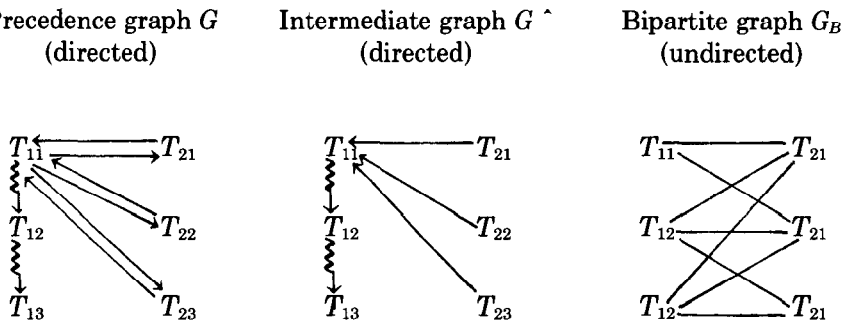
Step 2. Find the minimum cardinality covering all edges by nodes in G_B. This is also the minimum cardinality set that breaks all two-cycles in G, taking ripple

[2] This does not extend to merging three or more partition groups; see [29] for the proof of its NP-completeness.

effects into account. For example,



Minimal solution: $\{T_{21}, T_{22}\}$



Minimal solutions: $\{T_{11}, T_{12}, T_{13}\}$, $\{T_{21}, T_{22}, T_{23}\}$

THEOREM 4.1 *Given a precedence graph G whose only cycles are two-cycles, Strategy* 4 *produces a minimum cardinality backout set.*

PROOF. Every minimal solution to G_B is a solution to G.

Suppose not. Then there is a minimal solution S to G_B that is not a solution to G.

Case 1. S does not break all cycles in G. By definition, S must cover every edge in the bipartite graph G_B. But since edges in G_B correspond exactly to cycles in G, deleting S in G must break all cycles in G. Contradiction.

Case 2. S violates a ripple edge $T_i \rightarrow T_j$ in G. That is, T_i was chosen but T_j was not. But since T_j in G_B has an arc to every T_k connected to T_i in G_B and S is

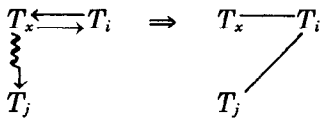
a solution to G_B, then every such T_k must have been chosen. Then T_i need not have been chosen, and S is not minimal for G_B.

Every solution to G is a solution to G_B.
Suppose not. Then there is a solution S to G that is not a solution to G_B. We know two facts about S:

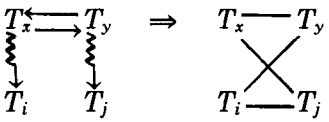
Fact 1. S breaks all two-cycles in G.
Fact 2. Whenever $T_i \in S$ and there is a ripple edge $T_i \rightarrow T_j$, then $T_j \in S$.

If S is not a solution to G_B, it is because some arc (T_i, T_j) has neither T_i nor T_j in S. This arc could have been obtained from G in one of three ways:

(1) An arc added in Step 2 due to a two-cycle involving T_i and T_j. But we noted in Fact 1 that S breaks all two-cyles, so one of T_i, T_j must have been chosen.
(2) An arc added in Step 2 due to the following construct:



But since we noted in Fact 1 that S breaks all cycles, one of T_x and T_i must be chosen. Furthermore, if T_x were chosen, then so would T_j be, by Fact 2.
(3) An arc added in Step 2 due to the following construct:



From Fact 1, either T_x or T_y has been chosen. From Fact 2, either T_i or T_j has been chosen.

Every minimal solution to G_B (G) is a minimal solution to G (G_B).
Suppose not. Then there is some S which is a minimal solution to G_B, hence is a solution to G, but not minimal. Then there is a smaller solution $S\hat{}$ to G which is also a solution to G_B. Therefore S cannot be minimal. (The other direction is similar.) □

Analysis of running time. The transformation of G to G_B is bounded by the cost of computing the transitive closure of the intermediate graph $G\hat{}$, which is equivalent to multiplying two Boolean matrices (see [1] for an $O(N^{2.81})$ solution). The asymptotically fastest algorithm known for the bipartite matching problem is $O(|V|^{0.5}|E|)$ [13]. Since the minimum cardinality covering of edges by nodes problem is the dual of the maximum cardinality matching problem in bipartite graphs, this bipartite matching algorithm can be used for an $O(N^{2.5})$ algorithm solving the minimum cardinality covering problem.

This backout policy was used in simulation runs with excellent results; in most cases all cycles were broken, and when there were longer cycles remaining, an optimal solution could often be obtained by deleting one or two extra nodes (see [5] for examples of successful and unsuccessful runs).

If the transactions are given a positive integer backout cost that is less than or equal to some fixed upper bound C, then Strategy 4 can be used to minimize

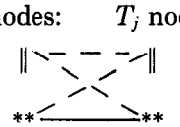
the total backout cost of breaking all two-cycles (note that until this point, each transaction has essentially had a backout cost of 1).

Strategy 5. *Breaking two-cycles optimally with weighted transactions*
Transform $G_B = (V_1, V_2, E_B)$ to bipartite graph $G_B^{\wedge} = (V_1^{\wedge}, V_2^{\wedge}, E_B^{\wedge})$ as follows. For each node T_i with cost c_i in V_1 or V_2, add c_i nodes $T_{i1}, T_{i2}, \ldots, T_{ic_i}$ to $V_1^{\wedge}$ or $V_2^{\wedge}$. For each arc (T_i, T_j) in E, add to $E^{\wedge}$ the $c_i * c_j$ edges forming the complete bipartite graph of nodes $T_{i1}, \ldots, T_{ic_i}$ and $T_{j1}, \ldots, T_{jc_j}$. Then finding a minimum *cardinality* covering of edges by nodes in $G_B^{\wedge}$ corresponds to finding a minimum *cost* covering of edges by nodes in G_B.

THEOREM 4.2. *Given a precedence graph G with weighted nodes whose only cycles are two-cycles, Strategy* 5 *produces a minimum cost backout set.*

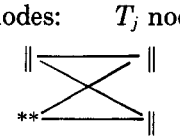
PROOF. The claim is that finding a minimum *cardinality* covering of edges by nodes in $G_B^{\wedge}$ corresponds to finding a minimum *cost* covering of edges by nodes in G_B. First note that any solution $S^{\wedge}$ to $G_B^{\wedge}$ either picks all of $T_{i1}, \ldots, T_{ic_i}$ or none at all. This corresponds to choosing node T_i in G_B. Suppose on the contrary that some of $T_{i1}, \ldots, T_{ic_i}$ were chosen, but others were not.

Case 1. Let $\|$ represent the subset of $T_{i1}, \ldots, T_{ic_i}$ that is in the solution $S^{\wedge}$, and ** represent the subset that is not in $S^{\wedge}$. Consider the following edges in $E^{\wedge}$ representing the edge (T_i, T_j) in E.



Since $S^{\wedge}$ is a solution, all of the above edges must be covered by a node in $S^{\wedge}$. However, none of the edges marked by a solid line are covered, a contradiction.

Case 2. Consider the following edges in $E^{\wedge}$.



Since $S^{\wedge}$ is a solution, all of the above edges must be covered by a node in $S^{\wedge}$ and $S^{\wedge}$ must be minimal. All of the above edges are covered, but there is a smaller solution with none of the T_i nodes chosen. For otherwise there must be another group of nodes T_k with (T_i, T_k) in E and some of $T_{k1}, \ldots, T_{kc_k}$ not in $S^{\wedge}$; that is, the chosen T_i nodes must be covering some edges in $E^{\wedge}$ that would not otherwise be covered. But the (T_i, T_k) edges are an example of Case 1 which has been shown not to exist. So $S^{\wedge}$ is not minimal, a contradiction.

Given solution $S^{\wedge}$ to $G_B^{\wedge}$ with either all of $T_{i1}, \ldots, T_{ic_i}$ chosen or none chosen, the corresponding solution S to G_B is also minimal. Otherwise, there is a smaller solution S_1 to G_B, which implies a smaller solution to $S_1^{\wedge}$ to $G_B^{\wedge}$, a contradiction. □

Analysis of running time. The transformation of G_B to $G_B^{\wedge}$ involves a blowup of nodes in V_1 (V_2) bounded by $C * |V_1|$ ($C * |V_2|$) and a blowup of edges bounded by $C^2 * |E|$.

This section has shown that when no long cycles remain after breaking all two-cycles optimally, both the problem of finding the smallest set of transactions to back-out to regain mutual consistency and of minimizing total backout cost have polynomial-time solutions. Thus, in practice, a minimal solution for the transaction backout problem can often be obtained using Strategies 4 and 5. In the cases where the precedence graph is not acyclic after using these algorithms, a "good" solution can usually be obtained by adding a few more nodes to the preliminary solution [6]. However, the results of Section 2.3 discourage attempts to find an algorithm to find the optimal solution when the algorithm of this section fails to break all cycles in the precedence graph.

5. DISCUSSION AND CONCLUSIONS

5.1 Extensions to the Procotol

Transaction processing during recovery. Since high data-accessibility is one of the advantages of the optimistic protocol, it is desirable to minimize the time during which transactions are forbidden to access data-items during recovery. Initially, however, all transaction processing must be temporarily suspended since neither partition knows what data-items were updated by the other partition. After the coordinator has received all the transactions, data-items which were updated in either partition group must be locked at all sites, but transactions accessing other data-items can safely execute. Additional locks may be released after the coordinator has determined BS (the set of transactions backed out) and KS (the set of transactions kept); data-item d must only be locked at sites in P_i if $d \in WRITESET(T)$ where either

(1) T was executed in P_i and $T \in$ BS, or
(2) T was *not* executed in P_i and $T \in$ KS.

That is, a data-item can safely be unlocked at a site when its value is known to be correct. Until then, the value must be assumed to be incorrect and the data-item must not be accessed.

Multiple failures. The comment was made in Section 1 that the optimistic protocol is resilient in the face of multiple failures. Since in a reliable distributed database system, partitions should be the exception rather than the rule, this should not be a major concern. However, this protocol can be extended to handle various types of multiple failures. (For a discussion of detecting mutual inconsistency in the face of multiple failures in file systems, see [22].)

(1) *K-partitions.* The communication subsystem could fail simultaneously in several places, creating more than two partition groups. If, when recovery occurred, K partition groups needed to be merged, they could either merge pairwise until a single partition group was formed, or the precedence graph definition could be extended. That is, $G(H_1, H_2, \ldots, H_k)$ would model conflicts

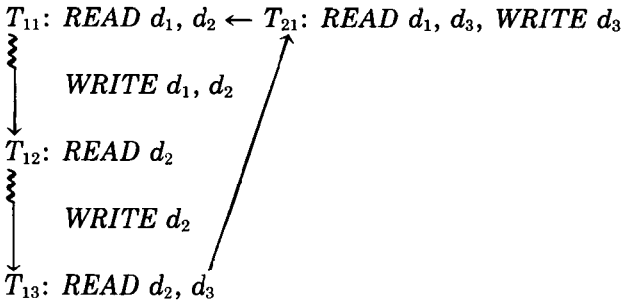
between transactions in all the partitions. In both cases, the problem of minimizing total backout cost becomes more complex.

(2) *Failures and recovery are tree-structured.* Suppose the system contains sites S_1, S_2, S_3, and S_4, and that failure and recovery occur as follows: $\{S_1, S_2, S_3, S_4\} \rightarrow \{S_1, S_2\}\{S_3, S_4\} \rightarrow \{S_1, S_2\}\{S_3\}\{S_4\} \rightarrow \{S_1, S_2\}\{S_3, S_4\} \rightarrow \{S_1, S_2, S_3, S_4\}$. When $\{S_3\}$ and $\{S_4\}$ are merged, the precedence graph need only contain transactions processed since the failure that separated S_3 and S_4 occurred. When $\{S_1, S_2\}$ and $\{S_3, S_4\}$ are merged, the precedence graph need only contain transactions that remain from the merge of $\{S_3\}$ and $\{S_4\}$; transactions backed-out in that merge can be ignored.

(3) *Nontree structured failure and recovery.* Suppose again that the system contains sites S_1, S_2, S_3, and S_4, but that recovery does not happen neatly as in the previous example. Thus $\{S_1, S_2, S_3, S_4\} \rightarrow \{S_1, S_2\}\{S_3, S_4\} \rightarrow \{S_1, S_2\}\{S_3\}\{S_4\} \rightarrow \{S_1, S_2, S_3\}\{S_4\} \rightarrow \{S_1, S_2, S_3, S_4\}$. When $\{S_1, S_2\}$ and $\{S_3\}$ merge, all transactions since the first failure must be contained in the precedence graph. Furthermore, the group $\{S_1, S_2, S_3\}$ must remember which transactions were backed-out of $\{S_3\}$ from the time that S_3 and S_4 formed the partition group $\{S_3, S_4\}$. When $\{S_1, S_2, S_3\}$ and $\{S_4\}$ are merged, transactions backed-out from the first merge must be deleted from the global history of $\{S_4\}$, and any transactions performed since S_3 and S_4 split, which functionally depend on deleted transactions, must automatically be backed-out. The precedence graph can then be constructed from the remaining transactions in the global histories of $\{S_1, S_2, S_3\}$ and $\{S_4\}$.

Nonreplicated data. The protocol can already handle nonreplicated data-items. There is obviously no way that data-items can be accessed if they are not present within the partition group; transactions attempting to read inaccessible data-items will not be able to execute until the failure is repaired. Write–write conflicts will not occur over nonreplicated data. However, transactions accessing only nonreplicated data-items may still be involved in cycles and must still be present in the precedence graph. Consider T_{12} in the following example.

Example 5.1.1. Let d_2 be a data-item accessible only in P_1, whereas d_1 and d_3 are accessible in both partitions.



So, although transactions accessing only nonreplicated data-items cannot be simply ignored, they do not present any new problems in the precedence graph definition.

5.2 Directions for Future Research

The author is currently studying how to use semantic information to minimize or resolve conflict. There are quite a few ways in which this can be done, for example:

(1) Use semantic knowledge at failure to split shared data-items and create new data-items which are unique to each partition. These can then be automatically recombined at recovery. An example of this was given in [12]:

An airline reservation system. Let SEATS represent the number of seats still available on a particular flight, and suppose a partition occurs. Based on local information, such as how many sites are in the partition, P_1 creates SEATS1 containing 40 percent of the value of SEATS, and P_2 creates SEATS2 containing 60 percent of the value of SEATS. No conflicts would occur since neither partition would be selling seats belonging to the other partition. SEATS = SEATS1 + SEATS2 at recovery would restore SEATS to its actual value.

(2) Use semantic knowledge to merge values of shared data-items at recovery (i.e., to resolve conflict). Suppose that the distributed database had incomplete information [17]. In such a system, the values of data-items can take on any subset of the domain for the data-item. For example, the age of John Doe may be recorded as "less than 30." If, during a failure, the system continues to gather information about John Doe's age and, at recovery, partition P_1 has his age as "between 20 and 30," while partition P_2 has his age as "between 15 and 25," it would be reasonable to take the intersection of these values and conclude that his age is "between 20 and 25."

The preceding examples are interesting but not easy to generalize. Another approach is to work with the commutativity of transactions or classes of transactions [10, 18] to minimize the number of edges in the precedence graph.

Other areas deserving of further research are mechanisms for detecting partition failures, and how to handle failures that occur while recovery is being performed.

5.3 Conclusions

Summarizing the results from Section 3, the optimistic protocol performs "well" when the number of write–write conflicts is small. This occurs when

(1) there is a relatively small number of transactions. The protocol should be used when the partition failure can be repaired in a short period of time, or when updates are infrequent. The system could also decrease the number of transactions submitted during the partition failure by requesting users to delay nonurgent transactions until repair is completed.

(2) there is a relatively large number of data-items in the database. The system could effectively increase the number of data-items by increasing granularity when a partition failure occurs. The distribution of data-item reference should also be more or less uniform; better yet is if the overlap of reference between partitions is known to be small (as in the bank account example).

(3) there is a large percentage of read-only transactions; within read–write transactions the percentage of updated items should be small. Transactions

should read only the data-items necessary to compute values for the updated items.

(4) the size of transactions is small. Users should be encouraged to break their transactions into small, unrelated units whenever possible, and submit as separate transactions.

The performance of the optimistic protocol can also be improved by using a backout strategy which breaks two-cycles intelligently, and then intelligently breaks longer cycles, if any remain. In Section 4 an algorithm was presented which optimally breaks two-cycles, minimizing total backout cost when no cycles remain.

The optimistic protocol could prove worthwhile for many applications. If the expected failure rate is sufficiently small, the inconvenience of occasionally having to resubmit a transaction is offset by the increased reliability and availability of the system. Examples of such applications follow.

Banking system. This example has already been mentioned. The assumption that few, if any, customers will access their accounts at two different branches during a single partition failure minimizes the probability of write–write conflicts. Since customers would not be irritated by possibly being denied access to their accounts, the bank would benefit by using an unrestrictive protocol during partition failures.

Airline reservation systems. The underlying database of flight information is fairly large, and references to flights are probably pretty evenly distributed. Many transactions are read-only: "Is there a flight connecting with flight #207 in Chicago?" "Are there seats available on flight #309 to Los Angeles?" "Is flight #327 arriving at 3:09 in Newark on time ?" etc. People wishing to book seats could be told that their reservations were conditional; backed-out transactions could be automatically rerun and confirmed, unless the flight became overbooked, in which case the customer could be consulted.

Periodic updating system. Consider a system whose primary application is to disseminate information (read-only transactions) and to periodically receive new information at the site closest to where the information is generated (read–write transactions). An example of this could be a weather predicting system, with sites located at major cities throughout the country. Information about weather conditions within each city would be received at that city, or at the nearest available site if the city site were unreachable or down. Information would then eventually be sent throughout the country. Weather predictions for each city are made based on the global information at each site. Should a partition failure occur, write–write conflicts would be very unlikely, since updates for the same data-item would occur at most a few times during a partition failure, and would probably be sent to the same site, hence be made within the same partition group. Predictions made during a partition failure could be labeled as "uncertain" if heavily based on old information (information available at the time of the failure that is being updated in the other partition group), or delayed until recovery. Since predictions are usually subject to error, "backing-out" a transaction would only amount to a "revised" forecast.

ACKNOWLEDGMENTS

The author is grateful to IBM for their support of this research; to the referees for their suggestions; and to Hector Garcia-Molina for his corrections, insights, and improvements at all stages of this paper.

APPENDIX 1

Backout Strategies and Simulation Results

Examples of three "on-line" backout strategies follow. They were chosen for the ways in which they differ. Strategy 1 differs from Strategy 2 only in the cycle detection phase, and tends to find long cycles before short cycles since interference edges are considered last. Strategy 2 differs from Strategy 3 only in the cycle breaking phase, and tries to break cycles intelligently.

Strategy 1

(1) *Detection.* Cycles are detected from the remaining (nonbacked-out) nodes as follows (assuming the ordering $T_{11} < T_{12} < \cdots < T_{1N_1} < T_{21} < \cdots T_{2N_2}$):

for $LOW = T_{11}$ **to** T_{2N_2} **do**

Detect all cycles whose "smallest" node is *LOW* (unless *LOW* has already been deleted). In building the cycles, consider *ripple*, then *precedence*, then *interference* edges in sorted order.
od

(2) *Cycle breaking.* Each node is given a weight, defined as the number of nodes connected to it via ripple edges. This weight is static and does not change to reflect the number of existing nodes connected via ripple edges as nodes are deleted. Cycles are broken by deleting the node with lowest weight, together with its closure with respect to ripple edges. If at any point the total number of nodes deleted exceed N_1, then all nodes in P_1 are chosen instead, and the algorithm terminates.

Strategy 2

(1) *Detection.* Two-cycles (cycles involving only two nodes, one from P_1 and one from P_2) are detected from remaining nodes as follows:

for $T = T_{11}$ **to** T_{1N_1} **do**

Detect all two-cycles involving T (unless T has already been deleted), considering interference edges in sorted order.
od

If the precedence graph of remaining nodes still contains cycles after this step, use the detection strategy in the first example to break remaining cycles.

(2) *Cycle breaking.* Same as in Strategy 1.

Strategy 3

(1) *Detection.* Same as in Strategy 2.

(2) *Cycle breaking.* Delete node from P_1 to break two-cycles. Longer cycles are broken by deleting the minimum weight node from P_1.

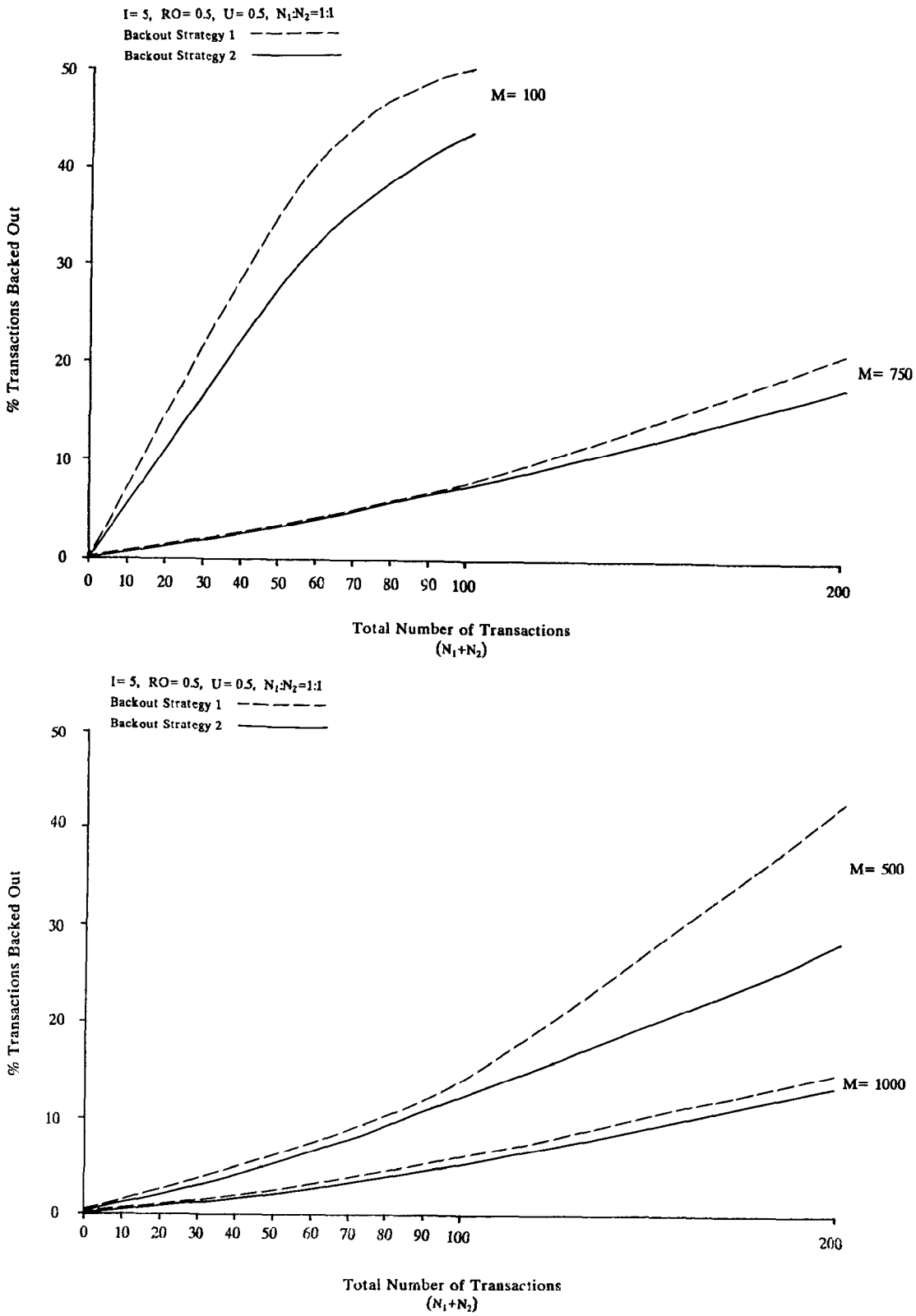


Fig. 6.1. Simulation results. Percentage of transactions backed-out versus total number of transactions (using Backout Strategies 1 and 2).

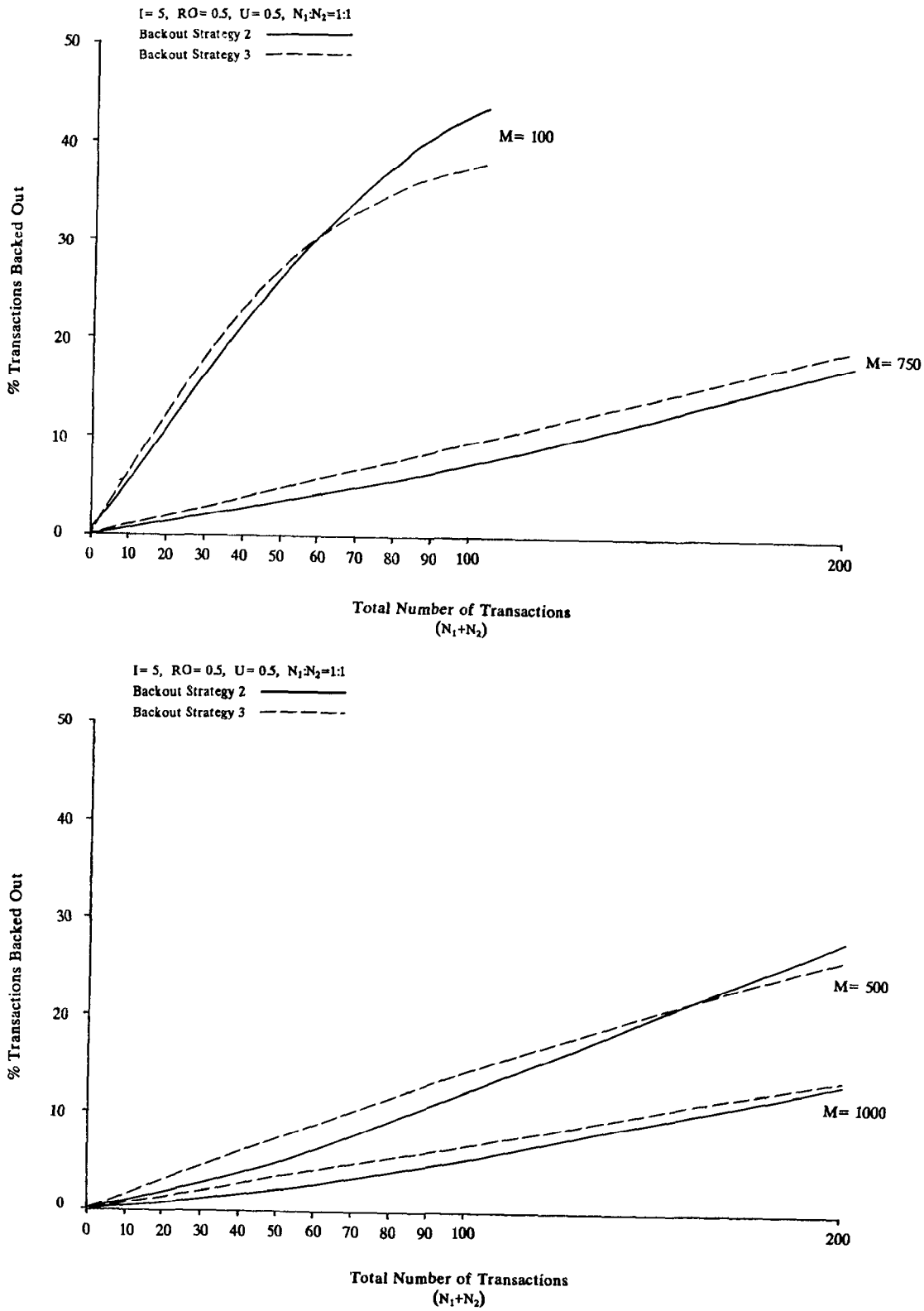


Fig. 6.2. Simulation results. Percentage of transactions backed-out versus total number of transactions (using Strategies 2 and 3).

Figure 6.1 compares Strategy 1 with Strategy 2, and indicates that Strategy 2 performs better. Not only were fewer transactions backed-out on the average, but the simulation ran much more quickly since two-cycles (cycles involving only two nodes) are easier to detect than longer cycles. In fact, it was observed that very few long cycles occurred after all two-cycles were broken (see Figure 3.1).

Figure 6.2 compares Strategy 2 with Strategy 3. When the backout rate (percentage of transactions backed out) is "small," that is, less that about 25 percent, Strategy 2 seems slightly preferable to Strategy 3. The two strategies differ only in how cycles are broken: Strategy 2 tries to be smart and choose the smallest number of nodes to break any cycle, while Strategy 3 settles for nodes in a given partition (P_1). When the backout rate is "large," Strategy 3 outperforms Strategy 2. This is reasonable, since, when a lot of nodes have been deleted, the static weights are more likely to be inaccurate, and Strategy 2 is not as smart in its node selection. Also, when more than N_1 transactions are chosen in Strategy 2, it gives up and deletes all nodes in P_1, reducing to the cycle-breaking method of Strategy 2.

REFERENCES

1. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. ALSBERG, P.A., AND DAY, J.D. A principle for resilient sharing of distributed resources. In *Proceedings 2nd International Conference on Software Engineering* (Oct. 13–15, 1976), 562–570.
3. BERNSTEIN, P.A., ROTHNIE, J.B., GOODMAN, N., AND PAPADIMITRIOU, C.A. The concurrency control mechanism of SDD-1: A system for distributed databases (the fully redundant case). *IEEE Trans. Softw. Eng. 4*, 3 (May 1978), 154–167.
4. BERNSTEIN, P.A., SHIPMAN, D.W., AND WONG, W.S. Formal aspects of serializability in database concurrency control. *IEEE Trans. Softw. Eng. 5*, 3 (May 1979), 203–216.
5. DAVIDSON, S.B., GARCIA-MOLINA, H., AND SKEEN, D. Consistency in the face of partition failures: A survey. Tech. Rep. TR#84-4, Dept. of Computer and Information Sciences, Univ. of Pennsylvania, Feb. 1984.
6. DAVIDSON, S.B. An optimistic protocol for partitioned distributed database systems. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, Princeton Univ. Princeton, N.J., Oct. 1982.
7. ESWARAN, K.P., GARY, J.N., LORIE, R.A., AND TRAIGER, I.L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.
8. GAREY, M.R., AND JOHNSON, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., San Francisco, 1979.
9. GARCIA-MOLINA, H. Elections in a distributed computing system. *IEEE Trans. Comput. C-31*, 1 (Jan. 1982), 48–59.
10. GARCIA-MOLINA, H. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst. 8*, 2 (June 1983), 186–213.
11. GIFFORD, D.K. Weighted voting for replicated data. In *Proceedings 7th Symposium on Operating System Principles* (Pacific Grove, Calif., Dec. 10–12, 1979), ACM, New York, 150–162.
12. HAMMER, M.M., AND SHIPMAN, D.W. The reliability mechanisms of SDD-1: A system for distributed databases. *ACM Trans. Database Syst. 5*, 4 (Dec. 1980), 431–466.
13. HOPCROFT, J.E., AND KARP, R.M. An n^{25} algorithm for maximum matching in bipartite graphs. *J. SIAM Comput 2* (1973), 225–231.
14. JOHNSON, D.B. Finding all the elementary circuits of a directed graph. *SIAM J. Comput. 4* (1975), 77–84.
15. KARP, R.M. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R.E. Miller and J.W. Thatcher, Eds., Plenum Press, New York, 85–103.
16. LAWLER, E.L. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, New York, 1976.

17. LIPSKI, W. On semantic issues connected with incomplete information databases. *ACM Trans. Database Syst. 4*, 3 (Sept. 1979), 262–296.
18. LYNCH, N.A. Multilevel atomicity—a new correctness criterion for distributed databases. *ACM Trans. Database Syst. 8*, 4 (Dec. 1983), 484–502.
19. MINOURA, T., AND WEIDERHOLD, G. Resilient extended true-copy token scheme for a distributed database system. *IEEE Trans. Softw. Eng. SE8*, 3 (May 1982), 173–189.
20. PAPADIMITRIOU, C.H. The serializability of concurrent database updates. *J. ACM 26*, 4 (Oct. 1979), 631–653.
21. PAPADIMITRIOU, C.H., AND STEIGLITZ, K. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, N.J., 1982.
22. PARKER, D.S., POPEK, G.J., RUDISIN, G., STOUGHTON, A., WALKER, B., WALTON, E., CHOW, J., EDWARDS, D., KISER, S., AND KLINE, C. Detection of mutual inconsistency in distributed systems. In *Proceedings 5th Berkeley Workshop on Distributed Data Management and Computer Networks* (Feb. 1981), 172–183.
23. PARKER, D.S., AND RAMOS, R.A. A distributed file system architecture supporting high availability. In *Proceedings 6th Berkeley Workshop on Distributed Data Management and Computer Networks* (1982), 161–183.
24. REINGOLD, E.M., NEIVERGELT, J., AND DEO, N. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, Englewood Cliffs, N.J., 1977, 348–352.
25. ROTHNIE, J.B., AND GOODMAN, N. A survey of research and development in distributed database management. In *Proceedings 3rd International Conference on Very Large Databases* (Tokyo, Oct. 1977), 48–61.
26. STEARNS, R.E., LEWIS, P.M., II, AND ROSENKRANTZ, D.J. Concurrency control for database systems. In *Foundations of Computer Science* (1976), 19–32.
27. THOMAS, R.H. A solution to the concurrency control problem for multiple copy databases. In *IEEE Compcon* (Spring 1978), 56–62.
28. TRAIGER, I.L., GRAY, J.N., GALTIERI, C.A., AND LINDSAY, B.G. Transactions and consistency in distributed database systems. IBM Res. Rep. RJ2555 33155, June 5, 1979.
29. WRIGHT, D.D. Merging partitioned databases. Ph.D. dissertation, TR83-575, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Sept. 1983.